WebSphere Development Studio Client Advanced Edition
for iSeries

EGL Reference Guide for iSeries

Version 6 Release 0

SC31-6838-01

WebSphere Development Studio Client Advanced Edition
for iSeries

EGL Reference Guide for iSeries

Version 6 Release 0

SC31-6838-01

Note
Before using this information and the product it supports, read the information in "Notices," on page 1043.

Second Edition (April 2005)

This edition applies to version 6, release 0, modification 0 of WebSphere Development Studio Advanced Edition for iSeries (product number 5724-D46) and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Overview

Introduction to EGL

Enterprise Generation Language (EGL) is a development environment and programming language that lets you write full-function applications quickly, freeing you to focus on the business problem your code is addressing rather than on software technologies. You can use similar I/O statements to access different types of external data stores, for example, whether those data stores are files, relational databases, or message queues. The details of Java™ and J2EE are hidden from you, too, so you can deliver enterprise data to browsers even if you have minimal experience with Web technologies.

After you code an EGL program, you generate it to create Java or COBOL source; then EGL prepares the output to produce executable objects. EGL also can provide these services:

- Places the source on a deployment platform outside of the development platform
- Prepares the source on the deployment platform
- Sends status information from the deployment platform to the development platform, so you can check the results

EGL even produces output that facilitates the final deployment of the executable objects.

An EGL program written for one target platform can be converted easily for use on another. The benefit is that you can code in response to current platform requirements, and many details of any future migration are handled for you. EGL also can produce multiple parts of an application system from the same source.

Related concepts
"Development process" on page 8
"EGL projects, packages, and files" on page 13
"Generated output" on page 515
"Parts" on page 17
"Run-time configurations" on page 9

Related tasks
"Creating an EGL Web project" on page 117

Related reference
"EGL editor" on page 471
"EGL source format" on page 478

What's new in EGL 6.0.0.1

Version 6.0.0.1 includes the following changes:

- The EGL form editor provides a graphical user interface for creating text and print forms.
- Target environments include HP-UX and Solaris. EGL provides 32- and 64-bit support for those platforms and has added 64-bit support for AIX®.

- The EGL debugger has the following changes:
  - Allows you to debug consoleUI-based applications
  - Allows use of an EBCDIC code page to represent character and numeric data during a debugging session
- The language is more flexible:
  - The system variables **SysVar.sqlCode** and **SysVar.sqlState** are modifiable
  - Array subscripts and substring indexes can include numeric expressions, as long as those expressions do not include functions
  - Any function that returns a value can be invoked from within a numeric, text, or logical expression, if the type of the return value is valid in the expression
  - Any function that returns a value can be used as an argument to a function parameter that has the modifier **in**, if the return value and parameter types are assignment compatible
  - Any EGL system variable can be passed as an argument to any function parameter that has the modifier **in**, if the argument and parameter types are assignment compatible
  - Any modifiable EGL system variable can be passed as an argument to a function parameter that has the modifier **out** (if the argument and parameter types are assignment compatible) or **inOut** (if the argument and parameter types are reference compatible)
- Documentation now identifies the access modifier (**in**, **out**, or **inOut**) for every parameter in every EGL system function; and describes reference and assignment compatibility
- New system functions are available:
  - **MathLib.stringAsDecimal** accepts a character value (like *"98.6"*) and returns the equivalent value of type DECIMAL.
  - **MathLib.stringAsFloat** accepts a character value (like *"98.6"*) and returns the equivalent value of type FLOAT.
  - **MathLib.stringAsInt** accepts a character value (like *"98"*) and returns the equivalent value of type BIGINT.
  - **SysLib.conditionAsInt**accepts a logical expression (like *myVar == 6*) , returning a 1 if the expression is true, a 0 if the expression is false.
  - **SysLib.startLog** opens an error log. Text is written into that log every time your program invokes **SysLib.errorLog**.
  - **SysLib.errorLog** copies text into the error log that was started by the system function **SysLib.startLog**
  - New functions support consoleUI--
    - **ConsoleLib.currentArrayCount** returns the number of elements in the dynamic array that is associated with the current active form
    - **ConsoleLib.setCurrentArrayCount** specifies how many rows exist in a dynamic array that is bound to an on-screen arrayDictionary
    - **ConsoleLib.hideAllMenuItems** hides all menuItems in the currently displayed menu
    - **ConsoleLib.showAllMenuItems** shows all menuItems in the currently displayed menu
- The Informix® 4GL conversion tool is included with the product
- The VAGen migration tool has changes that allow for a more efficient migration

Related concepts
"Sources of additional information on EGL" on page 12

What's new in the EGL 6.0 iFix

Note: EGL provides services to help you convert old code to code that works with the EGL 6.0 iFix:

- If you used a pre-6.0 version of EGL to create a Web application that is based on JavaServer Faces, do as follows in the workbench--
  1. Click **Help** > **Rational Help**
  2. In the **Search** text box of the help system, type at least the initial characters in this string: *Migrating JavaServer Faces resources in a Web project*
  3. Click GO
  4. Click *Migrating JavaServer Faces resources in a Web project* and follow the directions in that topic
- For other details on migrating code from EGL 6.0 or from an earlier version, see *Migrating EGL code to the EGL 6.0 iFix*.
- If you are migrating code from Informix 4GL or from VisualAge® Generator, see *Sources of additional information on EGL*.

The version 6.0 iFix represents a significant upgrade to the EGL language:

- Introduces the EGL report handler, which contains customized functions that are invoked at different times during execution of a JasperReports design file. The data returned from each function is included in your output report, which can be rendered in PDF, XML, text, or HTML format. The technology is an improvement on the reporting capability that was available in Informix 4GL.
- Introduces the EGL console UI, which is a technology for creating a character-based interface that allows an immediate, keystroke-driven interaction between the user and an EGL-generated Java program. The technology is an improvement on the dynamic user interface that was available in Informix 4GL.
- Provides new flexibility for code development--
  - Allows you to declare new types of variables:
    - A reference variable, which does not contain business data but points to such data.
    - A variable that contains or refers to a large quantity of data; specifically, to a binary large object (BLOB) or a character large object (CLOB).
    - A string variable, which refers to a Unicode string whose length varies at run time.
    - An ANY-typed variable, which can contain business data of any primitive type.
  - Allows you to include function invocations in expressions.
  - Allows you to reference a record without having development-time knowledge of the size or other characteristics of the record or of the fields in that record. Each field can itself refer to a record.
  - Expands support for dynamic arrays, which can now have multiple dimensions.
  - Introduces two new kinds of data collections:
    - A dictionary, which is composed of a set of key-and-value entries. You can add, delete, and retrieve entries at runtime, and the value in a given entry can be of any type.

    - An arrayDictionary, which is composed of a set of one-dimensional arrays, each of any type. You access the content of an arrayDictionary by retrieving the same-numbered elements across all the arrays.
  - Expands the number of system functions for various purposes:
    - To improve datetime processing, runtime message handling, and retrieval of user-defined Java runtime properties.
    - To support the new functionality related to reports, console UI, BLOB, and CLOB.
  - Provides better support for exception handling, for data initialization, and for DLL access.
- Provides a new wizard to create EGL report handlers.
- Allows you to customize a Web-page template for use with the Data Parts and Pages wizard, which quickly provides a Web application for accessing a single relational database.
- Allows you to create code that reflects the runtime behavior of Informix 4GL in relation to null processing and database commits.

Related concepts
"EGL-to-EGL migration" on page 96
"Sources of additional information on EGL" on page 12
"What's new in EGL version 6.0"

What's new in EGL version 6.0

Note: If you used an earlier version of EGL to create a Web application that is based on JavaServer Faces, do as follows in the workbench:

1. Click **Help** > **Rational Help**
2. In the **Search** text box of the help system, type at least the initial characters in this string: *Migrating JavaServer Faces resources in a Web project*
3. Click GO
4. Click *Migrating JavaServer Faces resources in a Web project* and follow the directions in that topic

Version 6.0 increases the power of the EGL language:

- Processing of relational databases has improved--
  - New wizards let you quickly do as follows:
    - Create data parts directly from relational database tables
    - Create Web applications that create, read, update, and delete table rows from such tables
  - New system functions are available:
    - **sysLib.loadTable** loads information from a file and inserts it into a relational database table.
    - **sysLib.unloadTable** unloads information from a relational database table and inserts it into a file.
  - If you are generating Java code, you can access SQL database rows in a cursor by navigating to the next row (as was always true); by navigating to the first, last, previous, or current row; or by specifying an absolute or relative position in the cursor.

  - The **forEach** statement allows you to loop easily through the rows of an SQL result set.
  - The **freeSQL** statement frees any resources associated with a dynamically prepared SQL statement, closing any open cursor associated with that SQL statement.
- String processing has improved--
  - You can specify substrings in a text expression, as in the following example:

```
myItem01 = "1234567890";

// myItem02 = "567"
myItem02 = myItem01[5:7];
```

  - You can specify a back space, form feed, or tab in a text literal
  - You can compare strings against either of two pattern types:
    - An SQL-type pattern, which includes the LIKE keyword. An example is as follows:

```
// variable myVar01 is the string expression
// whose contents will be compared to a like criterion
myVar01 = "abcdef";

// the next logical expression evaluates to "true"
if (myVar01 like "a_c%")
  ;
end
```

    - A regular-expression pattern. An example is as follows:

```
// variable myVar01 is the string expression
// whose contents will be compared to a match criterion
myVar01 = "abcdef";

// the next logical expression evaluates to "true"

if (myVar01 matches "a?c*")
  ;
end
```

  - You can use these text-formatting system functions:

    strLib.characterAsInt
    Converts a character string into an integer string

    strLib.clip
    Deletes trailing blank spaces and nulls from the end of returned character strings

    strLib.formatNumber
    Returns a number as a formatted string

    strLib.integerAsChar
    Converts an integer string into a character string

    strLib.lowercase
    Converts all uppercase values in a character string to lowercase values

    strLib.spaces
    Returns a string of a specified length.

    strLib.upperCase
    Converts all lowercase values in a character string to uppercase values.
- You can declare variables and structure items of new types.

  The new numeric types are as follows--

FLOAT
Concerns an 8-byte area that stores a double-precision floating-point numbers with as many as 16 significant digits

MONEY
Concerns a currency amount that is stored as a fixed-point decimal number up to as many as 32 significant digits

SMALLFLOAT
Concerns a 4-byte area that stores a single-precision floating-point number with as many as 8 significant digits

The new datetime types are as follows--

DATE
Concerns a specific calendar date, as represented in 8 single-byte digits

INTERVAL
Concerns a span of time that is represented in 1 to 21 single-byte digits and is associated with a mask such as ″hhmmss″ for hours, minutes, and seconds

TIME
Concerns an instance in time, as represented in 6 single-byte digits

TIMESTAMP
Concerns an instance in time that is represented in 1 to 20 single-byte digits and is associated with a mask such as ″yyyyMMddhh″ for year, month, day, and hour

- The syntax provides additional options--
  - You could always reference an element of a structure-item array as follows, but in light of iFix changes, you are asked to avoid this syntax:

    ```
    mySuperItem.mySubItem.mySubmostItem[4,3,1]
    ```

    The following syntax is strongly recommended--

    ```
    mySuperItem[4].mySubItem[3].mySubmostItem[1]
    ```
  - You can use a comma-delineated list of identifiers when you declare parameters, use-statement entries, set-statement entries, or variables, as in this example:

    ```
    myVariable01, myVariable02 myPart;
    ```
  - In a numeric expression, you can now specify an exponent by preceding a value with a double asterisk (**), so that (for example) 8 cubed is 8**3
  - You can now specify expressions that each resolve to a date, time, timestamp, or interval; and date arithmetic lets you do various tasks such as calculating the number of minutes between two dates
  - The following additions also allow for date and time processing:
    - **DateTimeLib.currentTime** and **DateTimeLib.currentTimeStamp** are system variables that reflect the current time
    - New formatting functions are available for dates (**StrLib.formatDate**), times (**StrLib.formatTime**), and timestamps (**sysLib.TimeStamp**)
    - Each of the following functions let you convert a series of characters to an item of a datetime type so that the item can be used in a datetime expression:
      - **DateTimeLib.dateValue** returns a date
      - **DateTimeLib.timeValue** returns a time
      - **DateTimeLib.timeStampValue** returns a timestamp associated with a particular mask such as ″yyyyMMdd″

    - **DateTimeLib.intervalValue** returns an interval associated with a particular mask such as "yyyyMMdd"
    - **DateTimeLib.extendDateTimeValue** accepts a date, time, or timestamp and extends it to an item associated with a particular mask such as "yyyyMMddmmss"
  - You can use these new, general statements:
    - The **for** statement includes a statement block that runs in a loop for as many times as a test evaluates to true. The test is conducted at the beginning of the loop and indicates whether the value of a counter is within a specified range.
    - The **continue** statement transfers control to the end of a **for**, **forEach**, or **while** statement that itself contains the **continue** statement. Execution of the containing statement continues or ends depending on the logical test that is conducted at the start of the containing statement.
  - You can run a system command synchronously (by issuing the function **sysLib.callCmd**) or asynchronously (by issuing the function **sysLib.startCmd**). These functions are available only when you generate Java output.
  - You can use two new functions that let you access command-line arguments in a loop
    - **sysLib.callCmdLineArgCount** returns the number of arguments
    - **sysLib.callCmdLineArg** returns the argument that resides in a specified position in the list of arguments

    These functions are available only when you generate Java output.
  - You can now specify a **case** statement in which each clause is associated with a different logical expression. If you use this new syntax, the EGL run time executes the statements that are associated with the first true expression:

```
case
  when (myVar01 == myVar02)
     conclusion = "okay";
  when (myVar01 == myVar03)
     conclusion = "need to investigate";
  otherwise
     conclusion = "not okay";
end
```

  - You can control whether a function parameter is used only for input, only for output, or for both; and you can avoid the choice by accepting the default setting, which is the unrestricted "for both".
  - You can now specify a datetime, text, or numeric expression that is more complex than a single item or constant, in these cases:
    - When you specify the value that is provided to the operating system by a **return** statement
    - When you specify an argument that is passed in either a function invocation or a program call; however, the characteristics of the receiving parameter must be known at generation time
  - You can now specify a complex numeric expression when exiting from the program

The development environment has improved as well:

- Two new features give you the ability to access parts quickly, even as your code grows in complexity--

  - The Parts Reference view lets you display a hierarchical list of the EGL parts that are referenced by a program, library, or PageHandler; and from that list, you can access any of the referenced parts
  - The EGL search mechanism lets you specify a search criterion to access a set of parts or variables in your workspace or in a subset of your projects
- Finally, the EGL Web perspective has been eliminated in favor of the widely used Web perspective.

Related concepts
"EGL-to-EGL migration" on page 96
"Sources of additional information on EGL" on page 12
"What's new in the EGL 6.0 iFix" on page 3

Development process

Your work with EGL includes the following steps:

Setup
You set up a work environment; for example, you set preferences and create projects.

Create and open EGL files
You begin to create the source code.

Declaration
You create and specify the details of your code.

Validation
At various times (such as when you save a file), EGL reviews your declarations and indicates whether they are syntactically correct and (to some extent) whether they are internally consistent.

Debugging
You can interact with a built-in debugger to ensure that your code fulfills your requirements.

Generation
EGL validates your declarations and creates output, including source code.

Preparation
EGL prepares the source code to produce executable objects. In some cases this step places the source code on a deployment platform outside of the development platform, prepares the source code on the deployment platform, and sends a results file from the deployment platform to the development platform.

Run through
In some cases, you can run your code immediately in the Workbench just by right-clicking the Java output and clicking **Run** > **Java application**.

Deployment
EGL produces output that makes deployment of the executable objects easier.

Related concepts
"EGL debugger" on page 261
"Generation of Java code into a project" on page 301
"Introduction to EGL" on page 1

Related tasks
"Processing Java code that is generated into a directory" on page 315
"Setting up the J2EE run-time environment for EGL-generated code" on page 333

Related reference
"EGL editor" on page 471
"EGL source format" on page 478

Run-time configurations

EGL provides the following kinds of generated code, among others:

- A Java program can be generated for any of several supported platforms. You can deploy the program outside of J2EE or in the context of any of the following J2EE containers--
  - J2EE application client
  - J2EE Web application
  - EJB container; in this case, you also generate an EJB session bean
- A non-interactive COBOL program also can be generated to run on iSeries™.

In addition, EGL provides a way to define a Web application that has the following characteristics:

- Delivers graphical pages to Web browsers
- Is able to store and retrieve data for a potentially large number of users
- Is embedded in a Java-based framework called JavaServer Faces

For details on this specialized support for Web applications, see *PageHandler part*.

Finally, you can use EGL to generate a Java wrapper, as described in the next section.

Use of a Java wrapper

The EGL-generated Java wrapper is a set of classes that let you invoke an EGL-generated program from non-EGL-generated Java code; for example, from an action class in a Struts- or JSF-based J2EE web application or from a non-J2EE Java program. The Java-to-EGL integration task is as follows:

1. Generate Java wrapper classes, which are specific to a generated program
2. Incorporate those wrapper classes into the non-generated Java code
3. From the non-generated Java code, invoke the wrapper-class methods to make the actual call and to convert data between these two formats:
   - The data-type formats used by Java
   - The primitive-type formats required when passing data to and from the EGL-generated program

Valid calls

The next table shows the valid calls to or from the EGL-generated code.

Calling object	Called object
An EGL-generated Java wrapper in a Java class that is outside of J2EE	An EGL-generated Java program (non-J2EE)
	An EGL-generated Java program in a J2EE application client
	An EGL-generated EJB session bean
	A CICS® COBOL program that was generated by VisualAge Generator
An EGL-generated Java wrapper in a J2EE application client	An EGL-generated Java program (non-J2EE)
	An EGL-generated Java program in a J2EE application client
	An EGL-generated EJB session bean
	A CICS COBOL program that was generated by VisualAge Generator
An EGL-generated Java wrapper in a J2EE Web application	An EGL-generated Java program (non-J2EE)
	An EGL-generated Java program in a J2EE application client
	An EGL-generated Java program in the same J2EE Web application
	An EGL-generated EJB session bean
	A CICS COBOL program that was generated by VisualAge Generator
An EGL-generated Java program that is outside of J2EE	An EGL-generated Java program (non-J2EE)
	An EGL-generated Java program in a J2EE application client
	An EGL-generated EJB session bean
	A CICS COBOL program that was generated by VisualAge Generator
	A non-EGL-generated program that was written in C or C++
	A non-generated program that was written in any language and runs under CICS
An EGL-generated Java program that is in a J2EE application client	An EGL-generated Java program (non-J2EE)
	An EGL-generated Java program in a J2EE application client
	An EGL-generated EJB session bean
	An EGL-generated CICS COBOL program
	A non-generated program that was written in any language and runs under CICS
	A non-generated program that was written in C or C++

Calling object	Called object
An EGL-generated Java program in a J2EE Web application	An EGL-generated Java program (non-J2EE)
	An EGL-generated Java program in a J2EE application client
	An EGL-generated Java program in the same J2EE Web application
	An EGL-generated EJB session bean
	A CICS COBOL program that was generated in VisualAge Generator
	A non-generated program written in C or C++
An EGL-generated EJB session bean	An EGL-generated Java program (non-J2EE)
	An EGL-generated Java program in a J2EE application client
	An EGL-generated EJB session bean
	A CICS COBOL program that was generated by VisualAge Generator
	A non-generated program written in C or C++
An EGL-generated COBOL program on iSeries	An EGL-generated COBOL program on iSeries
	A non-EGL-generated program written in any language and running on iSeries
A non-EGL-generated program written in any language and running on iSeries	An EGL-generated COBOL program on iSeries
	A non-EGL-generated program written in any language and running on iSeries

Valid transfers

The next table shows the valid transfers to or from EGL-generated code.

Transferring object	Receiving object
An EGL-generated Java program that is outside of J2EE	An EGL-generated Java program (non-J2EE)
An EGL-generated Java program that is in a J2EE application client	An EGL-generated Java program in the same J2EE application client
An EGL-generated Java program in a J2EE Web application	An EGL-generated Java program in the same J2EE Web application
An EGL-generated program on iSeries	An EGL-generated COBOL program on iSeries
	A non-EGL-generated program (written in any language and running on iSeries)
An non-EGL-generated program written in any language and running on iSeries	An EGL-generated COBOL program on iSeries
	A non-EGL-generated program written in any language and running on iSeries

Related concepts
"COBOL program" on page 306
"Generated output" on page 515
"Introduction to EGL" on page 1
"Java program, PageHandler, and library" on page 306
"Java wrapper" on page 282
"PageHandler" on page 180

Related tasks
"Setting up the J2EE run-time environment for EGL-generated code" on page 333

Sources of additional information on EGL

The most recent copy of this document is at the following Web site:

http://www.ibm.com/developerworks/rational/library/egldoc.html

For details on migrating source code written in VisualAge Generator, see the *VisualAge Generator to EGL Migration Guide* (file vagenmig.pdf), which is on the Web site mentioned earlier and in the help system section called *Installing and migrating*.

For details on run-time issues when you generate COBOL output, see the *EGL Server Guide for iSeries*, which is on the Web site mentioned earlier and in the help system section called *Installing and migrating*.

However, it is recommended that you access the Web site mentioned earlier.

However, it is recommended that you access the Web site.

Related concepts
"Introduction to EGL" on page 1

EGL language overview

EGL projects, packages, and files

An EGL project includes zero to many source folders, each of which includes zero to many packages, each of which includes zero to many files. Each file contains zero to many parts.

EGL project

An EGL project is characterized by a set of properties, which are described later. In the context of an EGL project, EGL automatically performs validation and resolves part references when you perform certain tasks; for example, when you save an EGL file or build file. In addition, if you are working with PageHandler parts (the output of which is used to debug Web applications in the Websphere test environment), EGL automatically generates output, but only in this case:

- You have set the automatic build process after selecting these options: **Window > Preferences > Workbench > Perform build automatically on resource modification**
- You have established a default build descriptor as a preference or property

An EGL project is formed by selecting **EGL** or **EGL Web** as the project type when you create a new project. You assign properties while working through the steps of project creation. To begin modifying your choices after you have completed those steps, right-click the project name and when a context menu is displayed, click **Properties**.

The EGL properties are as follows:

EGL source folder
One or more project folders that are the roots for the project's packages, each of which is a set of subdirectories. A source folder is useful for keeping EGL source separate from Java files and for keeping EGL source files out of the Web deployment directories. It is recommended that you specify EGL source folders in all cases; but if a source folder is not specified, the only source folder is the project directory.

The value of this property is stored in a file named .eglpath in the project directory and is saved in the repository (if any) that you use to store EGL files.

The EGL project wizards each create one source folder named **EGLSource**.

EGL build path
The list of projects that are searched for any part that is not found in the current project.

The value of this property is stored in a file named .eglpath in the project directory and is saved in the repository (if any) that you use to store EGL files.

In the following example of an .eglpath file, EGLSource is a source folder in the current project, and AnotherProject is a project in the EGL path:

```
<?xml version="1.0" encoding="UTF-8"?>
<eglpath>
  <eglpathentry kind="src" path="EGLSource"/>
  <eglpathentry kind="src" path="\AnotherProject"/>
</eglpath>
```

The source folders for AnotherProject are determined from the .eglpath file in *that* project.

Default build descriptors
The build descriptors that allow you to generate output quickly, as described in *Generation in the workbench*.

Package

A package is a named collection of related source parts. No package is in use when you create build parts.

By convention, you achieve uniqueness in package names by making the initial part of the package name an inversion of your organization's Internet domain name. For example, the IBM® domain name is ibm.com®, and the EGL packages begin with "com.ibm". By using this convention, you gain some assurance that the names of Web programs developed by your organization will not duplicate the names of programs developed by another organization and can be installed on the same server without possibility of a name collision.

The folders of a given package are identified by the package name, which is a sequence of identifiers separated by periods (.), as in this example:

```
com.mycom.mypack
```

Each identifier corresponds to a subfolder under an EGL source folder. The directory structure for com.mycom.mypack, for example, is \com\mycom\mypack, and the source files are stored in the bottom-most folder; in this case, in mypack. If the workspace is *c:\myWorkspace*, if the project is *new.project*, and if the source folder is EGLSource, the path for that package is as follows:

```
c:\myWorkspace\new.project\EGLSource\com\mycom\mypack
```

The parts in an EGL file all belong to the same package. The file's package statement, if any, specifies the name of that package. If you do not specify a package statement, the parts are stored directly in the source folder and are said to be in the *default package*. It is recommended that you always specify a package statement because files in the default package cannot be shared by parts in other packages or projects.

Two parts with the same identifier may not be defined in the same package. *It is strongly recommended that you avoid using the same package name under different projects or different folders.*

The package for generated Java output is the same as the EGL file package.

EGL files

Each EGL file belongs to one of these categories:

Source file
An EGL source file (extension .egl) contains logic, data, and user interface parts and is written in EGL source format.

Each of the following *generatable parts* can be transformed into a compilable unit:

- DataTable
- FormGroup

- Handler (the basis of a report handler)
- Library
- PageHandler
- Program

An EGL source file can include zero to many non-generatable parts but can include no more than one generatable part. The generatable part (if any) must be at the top level of the file and must have the same name as the file.

Build file

An EGL build file (extension .eglbld) contains any number of build parts and is written in Extensible Markup Language (XML), in EGL build-file format. You can review the related DTD, which is in the following directory:

installationDir\egl\eclipse\plugins\
com.ibm.etools.egl_*version*

installationDir
The product installation directory, such as C:\Program Files\IBM\Rational\SPD\6.0. If you installed and kept a Rational® Developer product before installing the product that you are using now, you may need to specify the directory that was used in the earlier install.

version
The installed version of the plugin; for example, 6.0.0

The file name (like egl_wssd_6_0.dtd) begins with the letters *egl* and an underscore. The characters *wssd* refer to Rational Web Developer and Rational Application Developer; the characters *wsed* refer to Rational Application Developer for z/OS®; and the characters *wdsc* refer to Rational Application Developer for iSeries.

After you add parts to files, you can use a repository to maintain a history of changes.

Recommendations

This section gives recommendations for setting up your development projects.

For build descriptors

Team projects should appoint one person as a build-descriptor developer. The tasks for that person are as follows:

- Create the build descriptors for the source-code developers
- Put those build descriptors in a project separate from the source code projects; and make that separate project available in the repository or by some other means
- Ask the source-code developers to set the property **default build descriptors** in their projects, so that the property references the appropriate build descriptors
- If a small subset of the build descriptor options (such as for user ID and password) varies from one source-code developer to the next, ask each source-code developer to do as follows:
  - Code a personal build descriptor that uses the option **nextBuildDescriptor** to point to a group build descriptor

– Ask the source-code developers to set the property **default build descriptors** in their files, folders, or packages, so that the property references the personal build descriptor. They do not specify the property at the project level because the project-level property is under repository control, along with other project information.

For additional information, see *Build descriptor part*.

For packages

For packages, recommendations are as follows:

- Do not use the same package name in different projects or source directories
- Do not use the default package

Part assignment

For parts, many of the recommendations refer to good practices, not hard requirements. Fulfill even the optional recommendations unless you have good reason to do otherwise:

- A *requirement* is that you put JSPs in the same project as their associated PageHandlers.
- If a non-generatable part (like a record part) is used only by one program, library, or PageHandler, place the non-generatable part in the same file as the using part.
- If a part is referenced from different files in the same package, put that part in a separate file in the package.
- If a part is shared across packages in a single project, place that part in a separate package in that project.
- Put code for completely unrelated applications in different projects. The project is the unit for transferring code between your local directory structure and the repository. Design project structure so that developers can minimize the amount of code they have to have loaded into their development system.
- Name projects, packages, and files in a way that reflects the use of the parts they contain.
- If your process emphasizes code ownership by a developer, do not assign parts for different owners to the same file.
- Assign parts to packages with a clear understanding of the purpose of the package; and group those parts by the closeness of the relationship between them.

  The following distinction is important:

  – Moving a part from file to file in the same package does not require that you change import statements in other files.

  – Moving a part from one package to another may require an import statement to be added or changed in every file that references the moved part.

Related concepts

"Build descriptor part" on page 275

"Generation in the workbench" on page 311
"References to parts" on page 20
"Import" on page 30
"Introduction to EGL" on page 1
"Parts" on page 17

Related reference

"EGL build-file format" on page 358
"EGL source format" on page 478
"EGL statements" on page 83

Parts

An EGL file contains a set of *parts*, each of which is a discrete, named unit. Some parts (such as a program) are *generatable parts*; each of these is the basis of a compilable unit. A generatable part must have the same name as the EGL source file that contains the part.

An EGL source file (extension .egl) can include zero or one generatable part and zero to many other parts.

Parts are also categorized in this way:

- *Logic parts* define a run-time sequence that you write in the EGL procedural language--
  - The non-generatable part *function* is the basic unit of logic. Every other kind of logic part can include functions.
  - You can define either of two types of *programs*, which vary by interface type. Each is a generatable part:
    - A *basic program* either avoids interacting with the user or limits that interaction to a particular kind of character-based interface. The interface technology in this case works as follows:
      - Displays output in a command window; and
      - Allows the user to interact with the program in an immediate way, with each keystroke potentially defining a separate event for the program to handle.

      For details on this kind of interface, which is available only if you are generating output in Java, see *Console user interface*.
    - A *textUI program* interacts with the user in this way:
      - Displays a set of fields in a command window or a 3270 screen; and
      - Accepts the user's field input only when the user presses a submit key.

    You can define either type of program to be a *main program*. That kind of program is started in any of these ways:
    - By the user
    - By a program transfer other than a call
    - Directly by an operating-system process

    Also, you can declare either kind of program to be a *called program*, which can be invoked only by a call.

    For other details on the run-time deployment of main and called programs, see *Run-time configurations*.
  - A *PageHandler* is a generatable part that controls the interaction between the user and a Web page.
  - A *Handler* of type JasperReport is a generatable part that contains customized functions which are invoked at different times during execution of a JasperReports design file. The data returned from each function is included in your output report, which can be rendered in PDF, XML, text, or HTML format.

- A *Library* is also a generatable part; a collection of shared functions and variables that can be made available to programs, pageHandlers, and other libraries.

- *Data parts* define the data structures that are available to your program.

  The following kinds of data parts are used as types in variable declarations:

  - *DataItem parts* contain information about the most elementary kind of data. These parts are similar to entries in a system-wide data dictionary, with each part including details on data size, type, formatting rules, input-validation rules, and display suggestions. You define a DataItem part once and can use it as the basis for any number of primitive variables or record fields.

    The DataItem part gives you a convenient way to create a variable from a primitive type. For example, consider the following definition of myStringPart, which is a DataItem part of type String:

    ```
    DataItem
      MyStringPart String { validValues = ["abc", "xyz"] }
    end
    ```

    When you develop a function, you can declare a variable of type MyStringPart:

    ```
    myString MyStringPart;
    ```

    The following declaration has the same effect as the previous one:

    ```
    myString STRING { validValues = ["abc", "xyz"] };
    ```

    As shown, the name of a DataItem part is simply an alias for a primitive type that has specific property settings.

  - *Record parts* are a basis for complex data. A variable whose type is a record part includes fields. Each field can be based on any of these:

    - A primitive type such as STRING
    - A DataItem part
    - A fixed-record part (as described later)
    - Another record part
    - An array of any of the preceding kinds

    Each field also can be a Dictionary or ArrayDictionary (as described later); or an array of Dictionaries or ArrayDictionaries.

    The variable that is based on a record part is called a record, and the length of the data in the record can vary at run time.

    You can use a record part to create variables for general processing or to access a relational database.

  - *Fixed record parts* are a basis for complex data that is of fixed length. A variable whose type is a fixed record part includes fields, and each field can have any of these as a type:

    - A primitive type such as CHAR
    - A dataItem part

    Each field can be substructured. For example, a field that specifies a telephone number can be defined as follows:

    ```
    10 phoneNumber    CHAR(10);
       20 areaCode    CHAR(3);
       20 localNumber CHAR(7);
    ```

    Although you can use fixed record parts for any kind of processing, their best use is for I/O operations on VSAM files, MQSeries® messages queues, and other sequential files.

To some extent, EGL supports fixed record parts to allow compatibility with earlier products such as VisualAge Generator. Although you can use fixed records for accessing relational databases or for general processing, it is recommended that you avoid using fixed records for those purposes.

- A *Dictionary part* is always available; you do not define it. A variable that is based on a dictionary part may include a set of keys and their related values, and you can add and remove key-and-value entries at run time.
- An *ArrayDictionary part* is always available; you do not define it. A variable that is based on an ArrayDictionary part lets you access a series of arrays by retrieving the same-numbered element of every array. A set of elements that is retrieved in this way is itself a dictionary, with each array name treated as a key that is paired with the value in the array element.

  An ArrayDictionary is especially useful in relation to the display technology described in *Console user interface*.

The other data part is a *DataTable*, which is treated as a variable rather than as a type for a variable. The DataTable is a generatable part that can be shared by multiple programs. It contains a series of rows and columns; includes a primitive value in each cell; and is treated as a variable that is (in most cases) global to the run unit.

- *UI (user interface) parts* describe the layout of data presented to the user in fixed-font screen and print forms. UI parts are used in different contexts and are of the following types:
  - A *record part* of subtype *ConsoleForm*is an organization of data that is presented to the user in the context of consoleUI technology. Like other record parts, each is used as a type for one or more variables; but in this case, each variable is called a *console form* rather than a record. The ConsoleUI technology also includes other parts that are defined for you and can be used as the basis of variables; for details, see *Console user interface*
  - A *Form* is also an organization of data that is presented to the user. One kind of form organizes the data sent to a screen in a textUI program, and another organizes the data sent to a printer in any kind of program.

    Each form includes a fixed, internal structure like that of a fixed record; but a form cannot include a substructure.

    A form is made available to a Program, PageHandler, or Library only if the form is included or referenced by a FormGroup, as described next.
  - A *FormGroup part* is a collection of text and print forms and is a generatable part. A program can include only one formGroup for most uses, along with one formGroup for help-related output. The same form can be included in multiple FormGroups.

    The forms in a FormGroup are global to a program, though access must be specified in a program-specific use statement. The forms are referenced as variables.

  You create Web user interfaces with Page Designer, which builds a JSP file and associates it with an EGL pageHandler. The JSP file replaces the role of a UI part for applications that interact with the user by way of the Web.
- *Build parts* are defined in EGL build files (extension .eglbld) and define a variety of processing characteristics:
  - A *build descriptor part* controls the generation process and indicates what other control parts are read during that process.
  - A *linkage options part* gives details on how a generated program transfers to and from other programs. The information in this part is used at generation time, test time, and run time.

– A *resource associations part* relates an EGL record with the information needed to access a file on a particular target platform; the information in this part is used at generation time, test time, and run time.

A fixed record, DataTable, or form (whether text or print) includes a *fixed structure*. The structure is composed of a series of fields, each of which has a size and type that is known at generation time; and in the case of a DataTable or fixed record, the field can be substructured.

Related concepts
"ArrayDictionary" on page 81

"Build descriptor part" on page 275
"Compatibility with VisualAge Generator" on page 428
"Console user interface" on page 165
"DataItem part" on page 123
"Dictionary" on page 77
"EGL projects, packages, and files" on page 13
"Fixed record parts" on page 125
"Function part" on page 132
"Import" on page 30
"Introduction to EGL" on page 1

"Linkage options part" on page 291
"Program part" on page 130
"Record parts" on page 124
"References to parts"
"References to variables in EGL" on page 55
"Resource associations and file types" on page 286
"Run-time configurations" on page 9
"Fixed structure" on page 24
"Typedef" on page 25
"Web support" on page 173

Related reference
"EGL build-file format" on page 358
"EGL editor" on page 471
"EGL source format" on page 478
"EGL statements" on page 83
"Primitive types" on page 31

References to parts

This section describes a set of rules that determine how EGL identifies the part to which a name refers. These rules are important in the following situations:

- One function invokes another
- A non-function part (a dataItem part, for example) refers to a validator function
- A part acts as a typedef (a model of format) in the declaration of a structure item or variable
- One part references another in a use declaration
- One build part references another

A second set of rules determine how EGL resolves variable references. For details, see *References to variables and constants*.

Basic visibility rules

In the simplest case, you define parts one after the next in a single package, without declaring one part within another. The following list omits many details, but shows a series of parts that are at the same hierarchical level:

```
Function:  Function01
Function:  Function02
Function:  Function03
Record:    Record01
```

Parts at the same level are available to one another. Function01, for example, can invoke one or both of the other functions; and Record01 can be used as a typedef for variables in each of the three functions.

In most cases, a part *cannot* nest another part. The exceptions are as follows:

- A program, library, or pageHandler can nest functions, but even then, the inclusion must be direct; a function cannot nest another function
- A form group can nest forms

An example with nested parts is as follows:

```
Program:  Program01
  Function:  Function01
  Function:  Function02
Function:  Function03
Record:    Record01
```

Parts at the top level are available to every other part in the package. However, the nested parts (Function01 and Function02) are available only to a subset of parts in the package:

- They are available to each other.
- They are available to the nesting part and to functions that are used by the nesting part at run time. If Function01 invokes Function03, for example, Function03 can invoke Function02 because Function03 is used in Program01.

Finally, if your code includes text or print forms, a use declaration is necessary to access the form group that includes those forms. A use declaration is also desirable when accessing data tables or libraries. For additional information, see *Use declaration*.

Additional visibility rules

Most development efforts have parts that are shared across more than one package. These rules are in effect:

- Any part in the file can reference parts from other packages, so long as the accessed parts have these characteristics:
  - Are top-level parts
  - Are not declared as *private*
  - Are either in the same project as the referencing part or are in a project listed in the EGL build path of the referencing project

  You can provide access in these ways:
  - You can qualify the part name with the package name, in which case no import statement is needed in your source file. If a package name is *my.package*, for example, and a part name is *myPart*, you can reference the part as follows:

    ```
    my.package.myPart
    ```
  - You can use import statements, which provide the following benefits:

- Import statements make it possible for you to avoid qualifying the names of the imported parts, unless you must use a package name to avoid an ambiguous reference.
- Import statements provide a way to document what packages are used in the source code.

Part-name resolution

To resolve a part reference, EGL conducts a search that includes one to many steps. The following statements apply *at each step*:

- The search ends successfully if a uniquely named part is found
- The search ends with an error if two same-named parts are found.

These situations are possible:

- The part reference is qualified with a package name; in this case, the search always includes only one step
- The part reference is not qualified with a package name and is not a function invocation
- The part reference is not qualified with a package name and is a function invocation

The next statements are rarely important, but could apply in either of the last two situations:

- The property **containerContextDependent** in the referencing function may be set to *yes*. You set that property to extend the name space used to resolve references, as described in *containerContextDependent*.
- If one of your functions is visible to the program or PageHandler and has a name that is identical to the name of an EGL system function, your function is referenced rather than the system function.

Part-name resolution when the package name is specified: As noted earlier, you can specify the name of a package when referencing a part, as in the example `my.package.myPart`. The current project is considered, as are any projects listed in the EGL build path.

If the reference is from a part that is inside the same package, the following statements apply:

- The package name is valid but unnecessary
- The part name is resolved even if the part is declared as private

Part-name resolution (other than function invocation) when the package name is not specified: If a part references a part other than a function and does not specify a package name, the steps in the search order are as follows:

1. Search the parts nested in the same container as the one in which the referencing part is nested.
2. Search the parts that were explicitly imported in the file where the referencing part resides. The current project is considered, as are any projects listed in the EGL build path.

   Each import statement in this case explicitly references a particular part in a particular package. The part named in such an *explicit-type import statement* acts as an override of the same-named part in the current package.

   If you have identically named packages in two different projects, a given explicit-type import statement uses the EGL build path to do a first-found search, stopping when the required part is found. (The part must be unique to

a package in a given project.) The presence of a same-named package in two different projects is not an error, but creates a confusing situation and is not recommended.

An error occurs if you have two explicit-type import statements that name the same part.

3. Search the top-level parts that are in the same package as the referencing part. The current project is considered, as are any projects listed in the EGL build path. Finding two parts of the same name causes an error.
4. Search other imported parts. The current project is considered, as are any projects listed in the EGL build path.

   Each import statement in this case references all parts in a given package and is called a *wild-card import statement*.

   If you have identically named packages in two different projects, a given wild-card import statement uses the EGL build path to do a first-found search, stopping when the required part is found. (The part must be unique to a package in a given project.)

   If more than one wild-card import statement retrieves the same-named part, an error occurs.

Function invocation when the package name is not specified: If a part invokes a function and does not specify a package name, the steps in the search order are as follows:

1. Search the functions nested in the same container as the one in which the invoker is nested.
2. Search the functions residing in the libraries specified in the container's use declarations.
3. Continue the search only with functions that are being included in the container at generation time. (To include functions other than those nested in the same container or residing in a library, set the container property **includeReferencedFunctions** to *yes*.)

   The search of the included functions occurs as follows:

   a. Search the parts that were explicitly imported in the file where the container resides. The current project is considered, as are any projects listed in the EGL build path.

      Each import statement in this case explicitly references a particular part in a particular package. The part named in such an *explicit-type import statement* acts as an override of the same-named part in the current package.

      If you have identically named packages in two different projects, a given explicit-type import statement uses the EGL build path to do a first-found search, stopping when the required function is found. (The function must be unique to a package in a given project.) The presence of a same-named package in two different projects is not an error, but creates a confusing situation and is not recommended.

      An error occurs if you have two explicit-type import statements that name the same part.

   b. Search the top-level functions in the same package as the container. The current project is considered, as are any projects listed in the EGL build path. An error occurs if the search finds two parts of the same name.
   c. Search other imported parts. The current project is considered, as are any projects listed in the EGL build path.

      Each import statement in this case references all parts in a given package and is called a *wild-card import statement*.

If you have identically named packages in two different projects, a given wild-card import statement uses the EGL build path to do a first-found search, stopping when the required part is found. (The part must be unique to a package in a given project.)

If more than one wild-card import statement retrieves the same-named part, an error occurs.

Program invocation

When a program is invoked on a **call** or **transfer** statement, the argument list of the invoker must match the parameter list of the invoked program. A mismatch of argument and parameter causes an error.

Related concepts

"EGL projects, packages, and files" on page 13
"Import" on page 30
"Introduction to EGL" on page 1
"Parts" on page 17
"References to variables in EGL" on page 55

Related reference

"containerContextDependent" on page 453
"EGL build path and eglpath" on page 465
"EGL editor" on page 471
"EGL source format" on page 478
"Use declaration" on page 930

Fixed structure

A *fixed structure* establishes the format of a text form, print form, dataTable, or fixed-record part; and is composed of a series of fields that each describes an elemental memory location or a collection of memory locations, as in this example:

```
10 workAddress;
  20 streetAddress1 CHAR(20);
     30 Line1 CHAR(10);
     30 Line2 CHAR(10);
  20 streetAddress2 CHAR(20);
     30 Line1 CHAR(10);
     30 Line2 CHAR(10);
  20 city CHAR(20);
```

You can define all the fields directly in the definition, as in the preceding example. Alternatively, you can indicate that all or a subset of the structure is equivalent to the structure that is in another fixed record part; for details, see *Typedef*.

Access to a field is based on a variable name, then a series of field names with a dot syntax. If you declare that the record *myRecord* includes the structure shown in the previous example, each of the following identifiers refers to an area of memory:

```
myRecord.workAddress
myRecord.workAddress.streetAddress1
myRecord.workAddress.streetAddress1.Line1
```

An *elementary structure field* has no subordinate structure fields and describes an area of memory in either of these ways:

- By a specification of length and primitive type, as in the previous example; or
- By pointing to the declaration of a dataItem part, as described in *Typedef*.

As shown earlier, a field in a fixed structure can have subordinate fields. Consider the next example:

```
10 topMost;
  20 next01 HEX(4);
  20 next02 HEX(4);
```

When you define a superior structure field (like *topMost*), you have several options:

- If you do not assign a length or primitive type, the superior structure field is of type CHAR, and EGL calculates the length. The primitive type of *topMost* is CHAR, for example, and the length is 4.
- If you assign a primitive type but do not assign a length, EGL calculates the length based on characteristics of the subordinate structure items
- If you assign both a length and primitive type, the length must reflect the space provided for the subordinate structure fields; otherwise, an error occurs

Note: The primitive type of a fixed-structure field determines the number of bytes in each unit of length; for details, see *Primitive types*.

Each elementary structure field has a series of properties, whether by default or as specified in the structure field. (The structure field may refer to a dataItem part that itself has properties.) For details, see *Overview of EGL properties and overrides*.

Related concepts
"DataItem part" on page 123
"Fixed record parts" on page 125
"Overview of EGL properties" on page 60
"Parts" on page 17
"References to variables in EGL" on page 55
"Typedef"

Related reference
"Data initialization" on page 459
"EGL source format" on page 478
"Primitive types" on page 31
"SQL item properties" on page 63

Typedef

A type definition (typedef) is a part that is used as a model of format. You use the typedef mechanism for these reasons:

- To identify the characteristics of a variable
- To reuse part declarations
- To enforce formatting conventions
- To clarify the meaning of data

Often, typedefs identify an abstract grouping. You can declare a record part named *address*, for example, and divide the information into *streetAddress1*, *streetAddress2*, and *city*. If a personnel record includes the structure items *workAddress* and *homeAddress*, each of those structure items can point to the format of the record part named *address*. This use of typedef ensures that the address formats are the same.

Within the set of rules described in this page, you may point to the format of a part either when you declare another part or when you declare a variable.

When you declare a part, you are not required to use a part as a typedef, but you may want to do so, as in the examples that are shown later. Also, you are not required to use a typedef when you declare a variable that has the characteristics of a data item; instead, you can specify all characteristics of the variable, without reference to a part.

A typedef is *always* in effect when you declare a variable that is more complex than a data item. For instance, if you declare a variable named **myRecord** and point to the format of a part named **myRecordPart**, EGL models the declared variable on that part. If you point instead to the format of a part named **myRecordPart02**, the variable is called **myRecord** but has all characteristics of the part named **myRecordPart02**.

The table and sections that follow give details on typedefs in different contexts.

Entry that points to a typedef	Type of part to which the typedef can refer
function parameter or other function variable	a record part or dataItem part
program parameter	dataItem part, form part, record part
program variable (non-parameter)	dataItem part, record part
structure item	dataItem part, record part

DataItem part as a typedef

You can use a dataItem part as a typedef in the following situations:

- When declaring a variable or parameter
- When declaring a structure item, which is a subunit of a record part, form part, or dataTable part

These rules apply:

- If a structure item is a parent to other structure items that are listed in the same declaration, the structure item can point only to the format of a dataItem part, as in this example:

```
DataItem myPart CHAR(20) end

Record myRecordPart type basicRecord
  10 mySI myPart; // myPart acts as a typedef
    20 a  CHAR(10);
    20 b  CHAR(10);
end
```

  The previous record part is equivalent to this declaration:

```
Record myRecordPart type basicRecord
  10 mySI CHAR(20);
    20 a CHAR(10);
    20 b CHAR(10);
end
```

- You cannot use a dataItem part as a typedef and also specify the length or primitive type of the entity that is pointing to the typedef, as in this example:

```
DataItem myPart HEX(20) end

// NOT valid because mySI has a primitive type
// and points to the format of a part (to myPart, in this case)
Record myRecordPart type basicRecord
  10 mySI CHAR(20) myPart;
end
```

- A variable declaration that does not refer to a record part either points to the format of a dataItem part or has primitive characteristics. (A program parameter can refer to a form part, too.) A dataItem part, however, cannot point to the format of another dataItem part or to any other part.
- An SQL record part can use only the following types of parts as typedefs:
  - Another SQL record part
  - A dataItem part

Record part as a typedef

You can use a record part as a typedef in the following situations:

- When declaring a structure item
- When declaring a variable (including a parameter), in which case the variable reflects the typedef in these ways:
  - Format
  - Record type (for example, indexedRecord or serialRecord)
  - Property values (for example, value of the **file** property)

When you declare a structure item that points to the format of another part, you specify whether the typedef adds a level of hierarchy, as illustrated later.

These rules apply:

- A record part can be a typedef when you use a structure item to facilitate reuse--

```
Record address type basicRecord
  10 streetAddress1 CHAR(30);
  10 streetAddress2 CHAR(30);
  10 city CHAR(20);
end

Record record1 type serialRecord
{
  fileName = "myFile"
}
  10 person CHAR(30);
  10 homeAddress address;
end
```

  The second record part is equivalent to this declaration--

```
Record record1 type serialRecord
  { fileName = "myFile" }
  10 person CHAR(30);
  10 homeAddress;
    20 streetAddress1 CHAR(30);
    20 streetAddress2 CHAR(30);
    20 city CHAR(20);
end
```

  If a structure item uses the previous syntax to point to the format of a structure part, EGL adds a hierarchical level to the structure part that includes the structure item. For this reason, the internal structure in the previous example has a structure-item hierarchy, with *person* at a different level from *streetAddress1*.
- In some cases, you prefer a flat arrangement in the structure; and an SQL record that is an I/O object for relational-database access *must* have such an arrangement--
  - In the previous example, if you substitute the word **embed** for a record part's structure item name (in this case, *homeAddress*) and follow that word with the name of the record part that acts as a typedef (in this case, *address*), the part declarations look like this:

```
Record address type basicRecord
  10 streetAddress1 CHAR(30);
  10 streetAddress2 CHAR(30);
  10 city CHAR(20);
end

Record record1 type serialRecord
{
  fileName = "myFile"
}
  10 person CHAR(30);
  10 embed address;
end
```

The internal structure of the record part is now flat:

```
Record record1 type serialRecord
{
  fileName = "myFile"
}
  10 person CHAR(30);
  10 streetAddress1 CHAR(30);
  10 streetAddress2 CHAR(30);
  10 city CHAR(20);
end
```

The only reason to use the word **embed** in place of a structure item name is to avoid adding a level of hierarchy. A structure item identified by the word **embed** has these restrictions:

- Can point to the format of a record part, but not to a dataItem part
- Cannot specify an array or include a primitive-type specification

– Next, consider the case in which a record part is a typedef when you are declaring identical structures in two records--

```
Record common type serialRecord
{
  fileName = "mySerialFile"
}
  10 a BIN(10);
  10 b CHAR(10);
end

Record recordA type indexedRecord
{
  fileName = "myFile",
  keyItem = "a"
}
  embed common; // accepts the structure of common,
                // not the properties
end

Record recordB type relativeRecord
{
  fileName = "myOtherFile",
  keyItem = "a"
}
  embed common;
end
```

The last two record parts are equivalent to these declarations--

```
Record recordA type indexedRecord
{
  fileName = "myFile",
  keyItem = "a"
}
  10 a BIN(10);
  10 b CHAR(10);
```

```
end

Record recordB type relativeRecord
{
  fileName = "myOtherFile",
  keyItem = "a"
}
  10 a BIN(10);
  10 b CHAR(10);
end
```

- You can use a record part multiple times as a typedef when declaring a series of structure items. This reuse makes sense, for example, if you are declaring a personnel record part that includes a home address and a work address. A basic record could provide the same format in two locations in the structure:

```
Record address type basicRecord
  10 streetAddress1 CHAR(30);
  10 streetAddress2 CHAR(30);
  10 city CHAR(20);
end

Record record1 type serialRecord
{
  fileName = "myFile"
}
  10 person CHAR(30);
  10 homeAddress address;
  10 workAddress address;
end
```

  The record part is equivalent to this declaration:

```
Record record1 type serialRecord
{
  fileName = "myFile"
}
  10 person CHAR(30);
  10 homeAddress;
    20 streetAddress1 CHAR(30);
    20 streetAddress2 CHAR(30);
    20 city CHAR(20);
  10 workAddress;
    20 streetAddress1 CHAR(30);
    20 streetAddress2 CHAR(30);
    20 city CHAR(20);
end
```

- You cannot use a record part as a typedef and also specify the length or primitive type of the entity that is pointing to the typedef, as in this example:

```
Record myTypedef type basicRecord
  10 next01 HEX(20);
  10 next02 HEX(20);
end

// not valid because myFirst has a
// primitive type and points to the format of a part
Record myStruct02 type serialRecord
{
  fileName = "myFile"
}
  10 myFirst HEX(40) myTypedef;
end
```

  Consider the following case, however:

```
Record myTypedef type basicRecord
  10 next01 HEX(20);
  10 next02 HEX(20);
end
```

```
Record myStruct02 type basicRecord
  10 myFirst myTypedef;
end
```

The second structure is equivalent to this declaration:

```
Record myStruct02 type basicRecord
  10 myFirst;
    20 next01 HEX(20);
    20 next02 HEX(20);
end
```

The primitive type of any structure item that has subordinate structure items is CHAR by default, and the length of that structure item is the number of bytes represented by the subordinate structure items, regardless of the primitive types of those structure items. For other details, see Structure.

- The following restrictions are in effect in relation to SQL records:
  - If an SQL record part uses another SQL record part as a typedef, each item provided by the typedef includes a four-byte prefix. If a non-SQL record uses an SQL record part as a typedef, however, no prefix is included. For background information, see *SQL record internals*.
  - An SQL record part can use only the following types of parts as typedefs:
    - Another SQL record part
    - A dataItem part
- Finally, neither a structure nor a structure item can *be* a typedef

Form as a typedef

You can use a form part as a typedef only when declaring a program parameter.

Related concepts
"DataItem part" on page 123
"Form part" on page 144
"Introduction to EGL" on page 1
"Record parts" on page 124
"Fixed structure" on page 24

Related tasks
"Creating an EGL program part" on page 129

Related reference
"EGL statements" on page 83
"SQL record internals" on page 726

Import

An import statement identifies a set of parts that are in a specified package (for EGL source files) or in a specified set of files (for EGL build files). The file that holds an import statement can reference the imported parts as if they were in the same package as the file.

Background

If a public part resides in a package other than the current one but is not identified in an import statement, your code needs to qualify the part name (for example, `myPart`) with the package name (for example, `my.pkg`), as in this example:

```
my.pkg.myPart
```

If the part is identified in an import statement, however, your code can drop the package name. In this case, the unqualified part name (like `myPart`) is sufficient.

For a description of the circumstances in which import statements are used to resolve a part name, see *References to Parts*.

Format of the import statement

The syntax is as follows for the import statement in an EGL file:

```
import packageName.partSelection;
```

packageName
Identifies the name of a package in which to search. The name must be complete.

partSelection
Is a part name or an asterisk (*). The asterisk indicates that all parts in the package are selected.

An import statement in a build file identifies other build files whose parts can be referenced by parts in the importing file. The import statements follow the <EGL> tag in the build file, and each statement has the following syntax:

```
<import file=filePath.eglbld>
```

filePath
Identifies the path and name of the file to import. If you specify a path, the following statements apply:

- The file path is in any of the source directories in the same project or in any other project that is in the EGL path
- Each qualifier is separated from the next by a virgule (/)

You may specify an asterisk (*) as the file name or as the last chararacter of the file name. If the asterisk is used, EGL imports all the .eglbld files with these characteristics:

- Are in the specified file path.
- Have names that begin with the characters that precede the asterisk. (If the asterisk has no preceding characters, all build files in the directory path are selected.)

The file extension .eglbld is optional.

Related concepts
"EGL projects, packages, and files" on page 13
"Introduction to EGL" on page 1
"Parts" on page 17
"References to parts" on page 20

Related tasks
"Editing an EGL build path" on page 300

Primitive types

Each EGL primitive type characterizes an area of memory. There are three kinds of primitive types: character, numeric, and datetime.

- The character types are as follows:
  - *CHAR* refers to single-byte characters.

  - *DBCHAR* refers to double-byte characters. dbchar replaces DBCS, which was a primitive type in EGL V5.
  - *MBCHAR* refers to multibyte characters, which are a combination of single-byte and double-byte characters. mbchar replaces MIX, which was a primitive type in EGL V5.
  - *STRING* refers to a field of varying length, where the double-byte characters conform to the UTF-16 encoding standards developed by the Unicode Consortium.
  - *UNICODE* refers to a fixed field, where the double-byte characters conform to the UTF-16 encoding standards developed by the Unicode Consortium.
  - *HEX* refers to hexadecimal characters.
- The datetime types are as follows:
  - *DATE* refers to a specific calendar date that has a fixed length of eight single-byte digits.
  - *INTERVAL* refers to a span of time that has a length ranging from two to twenty-seven single-byte digits.
  - *TIME* refers to an instance in time that has a fixed length of six single-byte digits.
  - *TIMESTAMP* refers to the current time and has a length ranging from two to twenty single-byte digits.
- The large object types are as follows:
  - *BLOB* refers to a binary large object with a length ranging from one byte to two gigabytes.
  - *CLOB* refers to a character large object with a length ranging from one byte to two gigabytes.
- The numeric types are as follows:
  - *BIGINT* refers to an 8-byte area that stores an integer of as many as 18 digits. This type is equivalent to type BIN, length 8, no decimal places.
  - *BIN* refers to a binary number.
  - *DECIMAL* refers to packed decimal characters whose sign is represented by a hexadecimal C (for a positive number) or a hexadecimal D (for a negative number) in the right half of the rightmost byte. DECIMAL replaces PACK, which was a primitive type in EGL version 5.0.
  - *FLOAT* refers to an 8-byte area that stores a double-precision floating-point numbers with up to 16 significant digits.
  - *INT* refers to a 4-byte area that stores an integer of as many as 9 digits. This type is equivalent to type BIN, length 4, no decimal places.
  - *MONEY* refers to currency amounts, which are stored as DECIMAL values.
  - *NUM* refers to numeric characters whose sign is represented by a sign-specific hexadecimal value in the left half of the rightmost byte. For ASCII, that value is 3 (for a positive number) and 7 (for negative); for EBCDIC, that value is F (for a positive number) and D (for negative).
  - *NUMC* refers to numeric characters whose sign is represented by a sign-specific hexadecimal value in the left half of the rightmost byte. For ASCII, that value is 3 (for a positive number) and 7 (for negative); for EBCDIC, that value is F (for a positive number) and C (for negative).
  - *PACF* refers to packed decimal characters whose sign is represented by a hexadecimal F (for a positive number) or a hexadecimal D (for a negative number) in the right half of the rightmost byte.

- *SMALLFLOAT* refers to a 4-byte area that stores a single-precision floating-point number with up to 8 significant digits.
- *SMALLINT* refers to an 2-byte area that stores an integer of as many as 4 digits. This type is equivalent to type BIN, length 2, no decimal places.

The internal representation of a field of any of the fixed-point numeric types is the same as an integer representation, even when you specify a decimal point. The representation of 12.34 is the same as that of 1234, for example. Similarly, currency symbols are not stored with fields of type MONEY.

When you interact with DB2® (directly or by way of JDBC) or when you generate for COBOL, the maximum number of digits in a fixed-point number is 31 at most.

A variable of type ANY receives the type of the value that is assigned to that variable, as described in the topic *ANY*.

At declaration time, you specify the primitive type that characterizes each of these values:

- The value returned by a function
- The value in a field, which is an area of memory that is referenced by name and contains a single value

Other entities also have a primitive type:

- A system variable has a primitive type (usually NUM) that is specific to the field
- A character literal is of one of these types:
  - CHAR if the literal includes only single-byte characters
  - DBCHAR if the literal includes only double-byte characters from the double-byte character set
  - MBCHAR if the literal includes a combination of single-byte and double-byte characters
- Character literals of type UNICODE are not supported.

Each primitive type is described on a separate page; and additional details are available on the pages that cover assignments, logical expressions, function invocations, and the call statement.

The sections that follow cover these subjects:

- Primitive types at declaration time
- Relative efficiency of different numeric types

Primitive types at declaration time

Consider the following declarations:

```
DataItem
  myItem CHAR(4)
end
Record mySerialRecordPart type serialRecord
  {
     fileName="myFile"
  }
  10 name CHAR(20);
  10 address;
    20 street01 CHAR(20);
    20 street02 CHAR(20);
end
```

As shown, you must specify a primitive type when you declare these entities:

- A primitive variable
- A structure field that is not substructured

You may specify the primitive type of a substructured structure field like *address*. If you fail to specify the primitive type of such a structure field but you reference the structure field in your code, the product makes these assumptions:

- The primitive type is assumed to be CHAR, even if the subordinate structure fields are of a different type
- The length is assumed to be the number of bytes in the subordinate structure fields

Relative efficiency of different numeric types

EGL supports the types DECIMAL, NUM, NUMC, and PACF so you can work more easily with files and databases that are used by legacy applications. It is recommended that you use fields of type BIN in new development or that you use an equivalent integer type (BIGINT, INT, or SMALLINT); calculations are most efficient with fields of those types. You get the greatest efficiency by using fields of type BIN, length 2, and no decimal places (the equivalent of type SMALLINT).

In calculations, assignments, and comparisons, fields that are of type NUM and have no decimal places are more efficient than fields that are of type NUM and have decimal places.

For code generated in Java, calculations with fields of types DECIMAL, NUM, NUMC, and PACF are equally efficient. For code generated in COBOL, however, these distinctions apply:

- Calculations with fields of types NUM are more efficient than calculations with fields of type NUMC
- Calculations with fields of types DECIMAL are more efficient than calculations with fields of type PACF

Related concepts

"DataItem part" on page 123
"Record parts" on page 124
"References to variables in EGL" on page 55
"Fixed structure" on page 24

Related reference

"ANY" on page 35
"Assignments" on page 352
"BIN and the integer types" on page 47
"call" on page 547
"CHAR" on page 36
"DATE" on page 38
"DBCHAR" on page 36
"DECIMAL" on page 47
"Exception handling" on page 89
"FLOAT" on page 48
"Function invocations" on page 504
"HEX" on page 36
"INTERVAL" on page 39
"Logical expressions" on page 484
"MBCHAR" on page 37

"MONEY" on page 48
"NUM" on page 48
"NUMC" on page 49
"Numeric expressions" on page 491
"Operators and precedence" on page 653
"PACF" on page 49
"SMALLFLOAT" on page 50
"SQL item properties" on page 63
"STRING" on page 37
"Text expressions" on page 492
"TIME" on page 40
"TIMESTAMP" on page 41
"UNICODE" on page 38

ANY

A variable of type ANY receives the type of the value that is assigned to that variable. The value can be of a primitive type such as INT or can be a variable that is based on a data part used as a type. The value cannot be a form or dataTable.

Consider this example:

```
myInt INT = 1;
myString STRING = "EGL";

myAny01, myAny02 any;

// myAny01 receives the value 1 and the type INT
myAny01 = myInt;

// myAny02 receives the value "EGL" and the type STRING
myAny02 = myString;

// The next statement is
// NOT VALID because a variable of type INT
// is being assigned to a variable of type STRING
myAny02 = myAny01;
```

Actions that combine types in an invalid way are detected only at run time and cause program termination. Those actions include assigning a value to a field of an incompatible type, passing an argument value to a parameter of an incompatible type, or combining incompatible values inside an expression.

The type of a literal is implied by the value of that literal:

- A quoted string is of type STRING
- An integer of 4 digits or less is of type SMALLINT
- An integer of 5 to 8 digits is of type INT
- An integer of 9 to 18 digits is of type BIGINT
- A number that includes a decimal point is of type NUM

When you reference a variable of type ANY, access is always dynamic. You cannot include a field of type ANY in a fixed structure (a dataTable, print form, text form, or fixed record).

Related reference
"Primitive types" on page 31

Character types

CHAR

An item of type CHAR is interpreted as a series of single-byte characters. The length reflects both the number of characters and the number of bytes and ranges from 1 to 32767.

Workstation platforms like Windows® 2000 use the ASCII character set; mainframe platforms like z/OS UNIX® System Services use the EBCDIC character set. Differences in collating sequence generally cause greater-than and less-than comparisons to have different results in the two types of environments.

Related reference
"Primitive types" on page 31

DBCHAR

An item of type DBCHAR is interpreted as a series of double-byte characters. The length reflects the number of characters and ranges from 1 to 16383. To determine the number of bytes, double the length value.

Workstation platforms like Windows 2000 use the ASCII character set; mainframe platforms like z/OS UNIX System Services use the EBCDIC character set. Differences in collating sequence generally cause greater-than and less-than comparisons to have different results in the two types of environments.

DBCS data is ideographic, as is necessary to display Chinese, Japanese, or Korean, for example. Display of such data requires a terminal device with DBCS capability.

Related reference
"Primitive types" on page 31

HEX

An item of type HEX is interpreted as a series of hexadecimal digits (0-9, a-f, and A-F), which are treated as characters. The length reflects the number of digits and ranges from 1 to 65534. To determine the number of bytes, divide by 2.

For an item of length 4, the internal bit representations of example values are as follows:

```
// hexadecimal value 04 D2
00000100 11010010

// hexadecimal value FB 2E
11111011 00101110
```

The primary use of an item of type HEX is to access a file or database field whose data type does not match another EGL primitive type.

You can assign a hexadecimal value by using a literal that is of type CHAR and that includes only characters in the range of hexadecimal digits, as in these examples:

```
myHex01 = "ab02";

myHex02 = "123E";
```

You can include a hexadecimal item as an operand in a logical expression, as in these examples:

```
if (myHex01 = "aBCd")
  myFunction01();
else
  if (myHex > myHex02)
    myFunction02();
  end
end
```

You cannot include a hexadecimal item in an arithmetic expression.

Related reference
"Primitive types" on page 31

MBCHAR

An item of type MBCHAR is interpreted as a combination of single-byte and double-byte characters. The length reflects the number of single-byte characters that the item can contain and also reflects the number of bytes. The length ranges from 1 to 32767.

Workstation platforms like Windows 2000 use the ASCII character set; mainframe platforms like z/OS UNIX System Services use the EBCDIC character set. Differences in collating sequence generally cause greater-than and less-than comparisons to have different results in the two types of environments.

On a mainframe environment, you must include space for shift-out and shift-in characters if double-byte characters are possible in the item:

- A single-byte shift-out character (hex value 0E) indicates the beginning of a series of double-byte characters
- A single-byte shift-in character (hex value 0F) indicates the end of that series

The shift-out and shift-in characters are deleted during an EBCDIC-to-ASCII data conversion and are inserted during an ASCII-to-EBCDIC data conversion. If a variable-length record is being converted, and if the current record end (as indicated by the record length) is within a structure item that is of type MBCHAR, the record length is adjusted to reflect the insertion or deletion of the shift-out and shift-in characters.

Double-byte character data is ideographic, as is necessary to display Chinese, Japanese, or Korean, for example. Display of such data requires a terminal device with double-byte character set capability.

Related reference
"Primitive types" on page 31

STRING

The primitive type STRING is composed of double-byte UNICODE characters.

You can store the value of the field in a file or database. If your code interacts with DB2 UDB, you must ensure that the code page for GRAPHIC data is UNICODE and that the column that stores the data item value is of SQL data type GRAPHIC or VARGRAPHIC.

For details on Unicode, see the web site of the Unicode Consortium (www.unicode.org).

Related reference
"Primitive types" on page 31

UNICODE

The primitive type UNICODE gives you a way to process and store text that may be in any of several human languages; however, the text must have been provided from outside your code. Literals of type UNICODE are not supported.

The following statements are true of an item of type UNICODE:

- The length reflects the number of characters and ranges from 1 to 16383. The number of bytes reserved for such an item is twice the value you specify for length.
- The item can be assigned or compared only to another item of type UNICODE.
- All comparisons compare the bit values in accordance with the order of characters in the UTF-16 encoding standard.
- When necessary, EGL pads the item with Unicode blanks.
- The system string functions treat the item as a string of individual bytes, which include the added Unicode blanks, if any. Any lengths you specify in those functions must be in terms of bytes rather than in terms of characters.
- You can store the value of the item in a file or database. If your code interacts with DB2 UDB, you must ensure that the code page for GRAPHIC data is UNICODE and that the column that stores the data item value is of SQL data type GRAPHIC or VARGRAPHIC.

For details on Unicode, see the web site of the Unicode Consortium (www.unicode.org).

Related reference
"Primitive types" on page 31

DateTime types

DATE

An item of type DATE is a series of eight single-byte numeric digits that reflect a specific calendar date.

The format of type DATE is *yyyyMMdd*:

yyyy
Four digits that represent a year. The range is 0000 to 9999.

MM
Two digits that represent a month. The range is 01 to 12.

dd Two digits that represent a day. The range is 01 to 31, and an error occurs if your code assigns an invalid date such as 20050230.

The internal hexadecimal representation of an example value is as follows if the item is on a host environment which uses EBCDIC:

```
// March 15, 2005
F2 F0 F0 F5 F0 F3 F1 F5
```

The internal hexadecimal representation of an example value is as follows if the item is on a workstation environment like Windows 2000, which uses ASCII:

```
// March 15, 2005
32 30 30 35 30 33 31 35
```

An item of type DATE can receive data from and provide data to a relational database.

Related reference
"EGL library DateTimeLib" on page 768
"Datetime expressions" on page 483
"Primitive types" on page 31
"Date, time, and timestamp format specifiers" on page 42

INTERVAL

An item of type INTERVAL is a series of one to twenty-one single-byte numeric digits that reflect an interval, which is the numeric difference between two points in time. The meaning of each digit is determined by the mask that you specify when declaring the item.

An interval can be positive (as when 1980 is subtracted from 2005) or negative (as when 2005 is subtracted from 1980), and at the beginning of the item is an extra byte that is not reflected in the mask. If an item of type INTERVAL is in a record, you must account for that extra byte when calculating the length of the record as well as the length of the superior item, if any.

You can specify a mask that is in either of two formats:

- Month span, which can include years and months
- Second span, which can include days, hours, minutes, seconds, and fractions of seconds

In either case, each character in the mask represents a digit. In the month-span format, for example, the set of *y*'s indicate how many years are in the item. If you only need three digits to represent the number of years, specify *yyy* in the mask. If you need the maximum number of digits (nine) to represent the number of years, specify *yyyyyyyyy*.

In a given mask, the first character may be used as many as nine times (unless otherwise stated); but the number of each subsequent kind of character is restricted further.

For a mask that is in month-span format, the following characters are available, in order:

y Zero to nine digits that represent the number of years in the interval.

M Zero to nine digits that represent the number of months in the interval. If *M* is not the first character in the mask, only two digits are allowed, at most.

The default mask is *yyyyMM*.

For a mask that is in second-span format, the following characters are available, in order:

d Zero to nine digits that represent the number of days in the interval.

H Zero to nine digits that represent the number of hours in the interval. If *H* is not the first character in the mask, only two digits are allowed, at most.

m Zero to nine digits that represent the number of minutes in the interval. If *m* is not the first character in the mask, only two digits are allowed, at most.

s Zero to nine digits that represent the number of seconds in the interval. If *s* is not the first character in the mask, only two digits are allowed, at most.

f Zero to six digits that each represent a fraction of seconds; the first represents tenths, the second represents hundreds, and so on. Even when *f* is the first character in the mask, only six digits are allowed, at most.

Although you can have zero characters of a given kind at the beginning or end of a mask, you cannot skip intermediate characters. Valid masks include these:

```
yyyyyyMM
yyyyyy
MM

ddHHmmssffffff
HHmmssff
mmss
HHmm
```

The following masks, however, are invalid because intermediate characters are missing:

```
// NOT valid
ddmmssffffff
HHssff
```

The internal hexadecimal representation of an example value is as follows if the default mask (*yyyyMM*) is in effect and if the item is on a host environment which uses EBCDIC:

```
// 100 years, 2 months; the 4E means the value is positive
4E F0 F1 F0 F0 F0 F2

// 100 years, 2 months; the 60 means the value is negative
60 F0 F1 F0 F0 F0 F2
```

The internal hexadecimal representation of an example value is as follows if the default mask (*yyyyMM*) is in effect and if the item is on a workstation environment like Windows 2000, which uses ASCII:

```
// 100 years, 2 months; the 2B means the value is positive
2B 30 31 30 30 30 32

// 100 years, 2 months; the 2D means the value is negative
2D 30 31 30 30 30 32
```

An item of type INTERVAL is strongly typed, so you cannot compare an item of this type with an item of any other type; nor can you assign an item of any other type to or from an item of this type.

Finally, an item of type INTERVAL cannot receive data from or provide data to a relational database.

Related reference
"EGL library DateTimeLib" on page 768
"Datetime expressions" on page 483
"Primitive types" on page 31
"Date, time, and timestamp format specifiers" on page 42

TIME

An item of type TIME is a series of six single-byte numeric digits that reflect a specific moment.

The format of type TIME is *HHmmss*:

HH
Two digits that represent the hour. The range is 00 to 24.

mm
Two digits that represent the minute within the hour. The range is 00 to 59.

ss Two digits that represent the second within the minute. The range is 00 to 59.

The internal hexadecimal representation of an example value is as follows if the item is on a host environment which uses EBCDIC:

```
// 8:40:20 o'clock
F0 F8 F4 F0 F2 F0
```

The internal hexadecimal representation of an example value is as follows if the item is on a workstation environment like Windows 2000, which uses ASCII:

```
// 8:40:20 o'clock
30 38 34 30 32 30
```

An item of type TIME can receive data from and provide data to a relational database.

Related reference
"EGL library DateTimeLib" on page 768
"Datetime expressions" on page 483
"Primitive types" on page 31
"Date, time, and timestamp format specifiers" on page 42

TIMESTAMP

An item of type TIMESTAMP is a series of one to twenty single-byte numeric digits that reflect a specific moment. The meaning of each digit is determined by the mask that you specify when declaring the item.

The following characters are available, in order, when you specify the mask:

yyyy
Four digits that represent the year. The range is 0000 to 9999.

MM
Two digits that represent the month. The range is 01 to 12.

dd Two digits that represent the day. The range is 01 to 31.

HH
Two digits that represent the hour. The range is 00 to 23.

mm
Two digits that represent the minute. The range is 00 to 59.

ss Two digits that represent the second. The range is 00 to 59.

f Zero to six digits that each represent a fraction of seconds; the first represents tenths, the second represents hundreds, and so on.

The default mask is *yyyyMMddHHmmss*.

When you interact with DB2 (directly or by way of JDBC) or when you generate for COBOL, you must specify every component from year (*yyyy*) through seconds (*ss*). In other contexts, the following is true:

- You can have zero characters of a given kind at the beginning or end of a mask, but cannot skip intermediate characters.
- Valid masks include these:

```
yyyyMMddHHmmss
yyyy
MMss
```

- The following masks are invalid because intermediate characters are missing:

```
// NOT valid
ddMMssffffff
HHssff
```

The internal hexadecimal representation of an example value is as follows if the default mask (*yyyyMMddHHmmss*) is in effect and if the item is on a host environment which uses EBCDIC:

```
// 8:05:10 o'clock on 12 January 2005
F2 F0 F0 F5 F0 F1 F1 F2 F0 F8 F0 F5 F1 F0
```

The internal hexadecimal representation of an example value is as follows if the default mask (*yyyyMMddHHmmss*) is in effect and if the item is on a workstation environment like Windows 2000, which uses ASCII:

```
// 8:05:10 o'clock on 12 January 2005
32 30 30 35 30 31 31 32 30 38 30 35 31 30
```

An item of type TIMESTAMP can be compared with (or assigned to or from) an item of type TIMESTAMP or an item of type DATE, TIME, NUM, or CHAR. However, an error occurs at development time if you assign a value that is not valid. An example is as follows:

```
// NOT valid because February 30 is not a valid date
myTS timestamp("yyyymmdd");
myTS = "20050230";
```

If characters at the beginning of a full mask are missing (for example, if the mask is "dd"), EGL assumes that the higher-level characters ("yyyyMM", in this case) represent the current moment, in accordance with the machine clock. The following statements cause a run-time error in February:

```
// NOT valid because February 30 is not a date
myTS timestamp("dd");
myTS = "30";
```

Finally, an item of type TIMESTAMP can receive data from or provide data to a relational database.

Related reference
"Assignments" on page 352
"Date, time, and timestamp format specifiers"
"Datetime expressions" on page 483
"EGL library DateTimeLib" on page 768
"Logical expressions" on page 484
"Primitive types" on page 31

Date, time, and timestamp format specifiers

The formats of dates, times, and timestamps are specified by a pattern of letters, each representing a component of the date or time. These characters are case-sensitive, and all letters from `a` to `z` and from `A` to `Z` parse to a component of the date or time.

To display letters in the date, time, or timestamp without that text being parsed as a component of the date or time, enclose that letter or letters in single quotes. To display a single quote in the date, time, or timestamp, use two single quotes.

The following table lists the letters and their values in a date, time, or timestamp pattern.

Letter	Date or time component	Type	Examples
G	Era designator	Text	AD
y	Year	Year	1996; 96
M	Month in year	Month	July; Jul; 07
w	Week in year	Number	27
W	Week in month	Number	2
D	Day in year	Number	189
d	Day in month	Number	10
F	Day of week in month	Number	2
E	Day in week	Text	Tuesday; Tue
a	AM/PM marker	Text	PM
H	Hour in day (0-23)	Number	0
k	Hour in day (1-24)	Number	24
K	Hour in AM/PM (0-11)	Number	0
h	Hour in AM/PM (1-12)	Number	12
m	Minute in hour	Number	30
s	Second in minute	Number	55
S	Millisecond	Number	978
z	Time zone	General time zone	Pacific Standard Time; PST; GMT-08:00
Z	Time zone	RFC 822 time zone	-800
C	Century	Century	20; 21

The number of each letter used consecutively in the pattern determines how that group of letters is interpreted and parsed. The interpretation depends on the type of letter. Also, the interpretation depends on whether the pattern is being used for formatting or parsing. The following list describes the types of letters and how different numbers of those letters affect the interpretation.

Text For formatting, if the number of letters is less than 4, the full form is used. Otherwise, an abbreviation is used, if available. In parsing, both forms are accepted, independent of the number of pattern letters.

Number
For formatting, the number of pattern letters represents the minimum number of digits. Zeroes are added to shorter numbers to make them the designated length. For parsing, the number of pattern letters is ignored unless it is needed to separate two adjacent fields.

Year For formatting, if the number of pattern letters is 2, the year is truncated to 2 digits. Otherwise, it is interpreted as the Number type.

For parsing, if the number of pattern letters is not 2, the year is interpreted literally, regardless of the number of digits. For example, the pattern `MM/dd/yyyy` assigned the value `01/11/12` parses to January 11, 12 A.D. The same pattern assigned the value `01/02/3` or `01/02/0003` parses to January 2, 3 A.D. In the same way, the same pattern assigned the value `01/02/-3` parses to January 2, 4 B.C.

For parsing, if the pattern is `yy`, the parser determines the full year relative to the current year. The parser assumes that the two-digit year is within 80 years before or 20 years after the time of processing. For example, if the current year is 2004, the pattern `MM/dd/yy` assigned the value `01/11/12` parses to January 11, 2012, while the same pattern assigned the value `05/04/64` parses to May 4, 1964.

Month

If the number of pattern letters is 3 or more, the month is interpreted as the Text type. Otherwise, it is interpreted as the Number type.

General time zone

General time zones are interpreted as the Text type if they have names. For time zones representing a GMT offset value, the following syntax is used:

`GMTOffsetTimeZone = GMT` *`Sign Hours : Minutes`*

Sign Either `+` or `-`

Hours A one-digit or two-digit number from 0 to 23. The format is locale independent and must be taken from the Basic Latin block of the Unicode standard.

Minutes

A two-digit number from 00 to 59. The format is locale independent and must be taken from the Basic Latin block of the Unicode standard.

For parsing, RFC 822 time zones are also accepted.

RFC 822 time zone

For formatting, the RFC 822 4-digit time zone format is used

`RFC822TimeZone =` *`Sign TwoDigitHours : Minutes`*

`TwoDigitHours` must be a two-digit number from 00 to 23. The other definitions are the same as the General time zone type.

For parsing, General time zones are also accepted.

Century

Displayed as a Number type that takes the full year mod by 100.

The following table lists some examples of date and time patterns interpreted in the U.S. locale.

Date and Time Pattern	Result
`yyyy.MM.dd G 'at' HH:mm:ss z`	2001.07.04 AD at 12:08:56 PDT
`EEE, MMM d, ''yy`	Wed, Jul 4, '01
`h:mm a`	12:08 PM
`hh 'o''clock' a, zzzz`	12 o'clock PM, Pacific Daylight Time
`K:mm a, z`	0:08 PM, PDT

Date and Time Pattern	Result
yyyyy.MMMMM.dd GGG hh:mm aaa	02001.July.04 AD 12:08 PM
EEE, d MMM yyyy HH:mm:ss Z	Wed, 4 Jul 2001 12:08:56 -0700
yyMMddHHmmssZ	010704120856-0700

LOB types

CLOB

An item of type CLOB represents a character large object with a length ranging from one byte to two gigabytes.

The following statements are true of an item of type CLOB:

- It can be declared only as an individual item, and is not supported in BasicRecords.
- It can be passed to local function and program calls. Large object parameters and corresponding arguments must both be declared as large objects of the same type.
- It can be assigned only to another Clob variable.
- It can be moved to another Clob variable, which has the same result as being assigned to a Clob variable.
- You can create a reference variable of BLOB.
- It uses SQLlocator (CLOB); that is, CLOB contains a logical pointer to the SQL CLOB data rather than to the data itself.
- When used with SQLRecord,
  - CLOB represents Character Large Object as a column in the database.
  - CLOB is valid for the duration of the translation in which it was created.
- It cannot be passed to calls to remote programs or to non-EGL programs.
- It cannot be referenced as an operand on assignment statements or in expressions.

You may use the following functions with CLOB:

- attachClobToFile
- freeClob
- getClobLen
- getStrFromClob
- getSubStrFromClob
- loadClobFromFile
- setClobFromString
- setClobFromStringAtPosition
- truncateClob
- updateClobToFile

Related reference
"BLOB" on page 46
"EGL library LobLib" on page 805
"attachClobToFile()" on page 807
"freeClob()" on page 808

“getClobLen()” on page 809
“getStrFromClob()” on page 809
“getSubStrFromClob()” on page 809
“loadClobFromFile()” on page 810
“setClobFromString()” on page 811
“setClobFromStringAtPosition()” on page 811
“truncateClob()” on page 812
“updateClobToFile()” on page 812
“Primitive types” on page 31

BLOB

An item of type BLOB represents a binary large object with a length ranging from one byte to two gigabytes.

The following statements are true of an item of type BLOB:

- It can be declared only as an individual item, and is not supported in BasicRecords.
- It can be passed to local function and program calls. Large object parameters and corresponding arguments must both be declared as large objects of the same type.
- It can be assigned only to another Blob variable.
- It can be moved to another Blob variable, which has the same result as being assigned to a Blob variable.
- You can create a reference variable of BLOB.
- It uses SQLlocator (BLOB); that is, BLOB contains a logical pointer to the SQL BLOB data rather than to the data itself.
- When used with SQLRecord,
  - BLOB represents Binary Large Object as a column in the database.
  - BLOB is valid for the duration of the translation in which it was created.
- It cannot be passed to calls to remote programs or to non-EGL programs.
- It cannot be referenced as an operand on assignment statements or in expressions.

You may use the following functions with BLOB:

- attachBlobToFile
- freeBlob
- getBlobLen
- loadBlobFromFile
- truncateBlob
- updateBlobToFile

Related reference

“CLOB” on page 45
“EGL library LobLib” on page 805
“attachBlobToFile()” on page 806
“freeBlob()” on page 808
“getBlobLen()” on page 808
“loadBlobFromFile()” on page 810
“truncateBlob()” on page 811
“updateClobToFile()” on page 812
“Primitive types” on page 31

Numeric types

BIN and the integer types

An item of type BIN is interpreted as a binary value. The length can be 4, 9, or 18 and reflects the number of positive digits in decimal format, including any decimal places. The value -12.34, for example, fits in an item of length 4. A 4-digit number requires 2 bytes; a 9-digit number requires 4 bytes; and an 18-digit number requires 8 bytes.

For an item of length 4, the internal bit representations of example values are as follows:

```
// for decimal 1234, the hexadecimal value is 04 D2:
00000100 11010010

// for decimal -1234, the value is the 2's complement (FB 2E):
11111011 00101110
```

It is recommended that you use items of type BIN instead of other numeric types whenever possible; for example, for arithmetic operands or results, for array subscripts, and for key items in relative records.

The following types are equivalent to type BIN:

- BIGINT is length 18, no decimal places
- INT is length 9, no decimal places
- SMALLINT is length 4, no decimal places

Related reference
"Primitive types" on page 31

DECIMAL

An item of type DECIMAL is a numeric value in which each half-byte is a hexadecimal character, and the sign is represented by a hexadecimal C (for a positive number) or a hexadecimal D (for a negative number) in the right half of the rightmost byte.

The length reflects the number of digits and ranges from 1 to 32; however, the maximum length for COBOL output is 31.

To determine the number of bytes, add 2 to the length value, divide the sum by 2, and truncate any fraction in the result.

For an item of length 4, the internal hexadecimal representations of example values are as follows:

```
// for decimal 123
00 12 3C

// for decimal -123
00 12 3D

// for decimal 1234
01 23 4C

// for decimal -1234
01 23 4D
```

A negative value that is read from a file or database into a field of type DECIMAL may have a hexadecimal B in place of a D; EGL accepts the value but converts the B to D.

The format of a DB2 UDB column of type DECIMAL is equivalent to the format of a DECIMAL-type host variable.

Related reference
"Primitive types" on page 31

FLOAT

An item of type FLOAT is interpreted as a binary value for double-precision floating-point numbers with as many as 16 significant digits. The length is fixed at 8 bytes. In EGL-generated Java programs, the value ranges from 4.9e-324 to 1.7976931348623157e308. In EGL-generated COBOL programs on iSeries, the value ranges from 2.225074e-308 to 1.797693e+308.

FLOAT corresponds to each of these definitions:

- The FLOAT data type in a relational database management system
- The **double** data type in C, C++, or Java
- The **COMP-2** data type in COBOL

For floating-point values, format conversion between Java and host COBOL formats is supported by DB2 but is not supported on calls to host programs.

Related reference
"Primitive types" on page 31

MONEY

An item of type MONEY is a numeric value that is equivalent in most respects to an item of type DECIMAL. In the case of MONEY, the default for length is 16; the default for decimal places is 2; the minimum length is 2; and a currency symbol is displayed in output fields. MONEY corresponds to the IBM Informix 4GL MONEY data type.

The format is based on the variable defaultMoneyFormat.

Related reference
"DECIMAL" on page 47
"Formatting properties" on page 62
"Primitive types" on page 31

NUM

An item of type NUM is a numeric value in which each byte is a digit in character format, and the sign is represented by a sign-specific hexadecimal value in the left half of the rightmost byte. The length reflects both the number of digits and the number of bytes. The length ranges from 1 to 32.

For an item of length 4, the internal hexadecimal representations of example values are as follows if the item is on a host environment which uses EBCDIC:

```
// for decimal 1234
F1 F2 F3 F4

// for decimal -1234
F1 F2 F3 D4
```

The internal hexadecimal representations of example values are as follows if the item is on a workstation environment like Windows 2000, which uses ASCII:

```
// for decimal 1234
31 32 33 34

// for decimal -1234
31 32 33 74
```

Related reference
"Primitive types" on page 31

NUMC

A field of type NUMC is a numeric value in which each byte is a digit in character format, and the sign is represented by a sign-specific hexadecimal value in the left half of the rightmost byte. The length reflects both the number of digits and the number of bytes and ranges from 1 to 18.

For a field of length 4, the internal hexadecimal representations of example values are as follows if the field is on a host environment which uses EBCDIC:

```
// for decimal 1234
F1 F2 F3 C4

// for decimal -1234
F1 F2 F3 D4
```

The internal hexadecimal representations of example values are as follows if the field is on a workstation environment like Windows 2000, which uses ASCII:

```
// for decimal 1234
31 32 33 34

// for decimal -1234
31 32 33 74
```

Related reference
"Primitive types" on page 31

PACF

A field of type PACF is a numeric value in which each half-byte is a hexadecimal character, and the sign is represented by a hexadecimal F (for a positive number) or a hexadecimal D (for a negative number) in the right half of the rightmost byte. The length reflects the number of digits and ranges from 1 to 18. To determine the number of bytes, add 2 to the length value, divide the sum by 2, and truncate any fraction in the result.

For a field of length 4, the internal hexadecimal representations of example values are as follows:

```
// for decimal 123
00 12 3F

// for decimal -123
00 12 3D

// for decimal 1234
```

```
01 23 4F

// for decimal -1234
01 23 4D
```

A negative value that is read from a file or database into a field of type PACF may have a hexadecimal B in place of D; EGL accepts the value but converts the B to D.

Related reference
"Primitive types" on page 31

SMALLFLOAT

An item of type SMALLFLOAT is interpreted as a binary value for single-precision floating-point numbers with as many as 8 significant digits. The length is fixed at 4 bytes of memory storage.

In EGL-generated Java programs, the value ranges from 3.40282347e+38 to 1.40239846e-45. In EGL-generated COBOL programs on iSeries, the value ranges from 1.175494e-38 to 3.402823e+38.

SMALLFLOAT corresponds to each of these definitions:

- The SMALLFLOAT data type in a relational database management system
- The **float** data type in C, C++, or Java
- The **COMP-1** data type in COBOL

For floating-point values, format conversion between Java and host COBOL formats is supported by DB2 but is not supported on calls to host programs.

Related reference
"Primitive types" on page 31

Declaring variables and constants in EGL

You can declare a variable in these ways:

- You can base a variable on one of several primitive types, as in this example--

  ```
  myItem CHAR(10);
  ```

- You can base a variable on a dataItem part, a record part, or a fixed record part, as in this example--

  ```
  myRecord myRecordPart;
  ```

- You can base a variable on the specific configuration of a dictionary or arrayDictionary, as in this example--

  ```
  myVariable Dictionary
  {
    empnum=0005,
    lastName="Twain",
    firstName="Mark",
    birthday="021460"
  };
  ```

- A program or other generatable part can access the fields of a dataTable, which is treated as a variable that is global to either the program or the run unit. You can use a simpler syntax to access those fields if the dataTable is listed in one of the program's use declarations.

- A program can access the fields of a text or print form, which is treated as a variable that is global to the program. The program must include the related formGroup in a use declaration.
- A program or other generatable logic part can access the library variables that are declared outside of any library function. Those variables are global to the run unit. You can use a simpler syntax to access those fields if the library is listed in one of the program's use declarations.

You declare a constant by specifying the symbol CONST followed by the constant name, type, equal sign, and value; and the specified value cannot be changed at run time. Examples are as follows:

```
const myString String = "Great software!";
const myArray  BIN[] = [36, 49, 64];
const myArray02 BIN[][] = [[1,2,3],[5,6,7]];
```

A constant cannot be in a record or other complex structure.

Finally, to declare multiple variables or constants in a single statement, separate one identifier from the next by a comma, as in these examples:

```
const myString01, myString02 STRING = "INITIAL";
myItem01, myItem02, myItem03 CHAR(5);
myRecord01, myRecord02 myRecordPart;
```

Related concepts
"References to parts" on page 20
"Parts" on page 17
"Typedef" on page 25

Related reference
"Primitive types" on page 31
"Use declaration" on page 930

Dynamic and static access

EGL resolves a variable reference by static or dynamic access:

- When *dynamic access* is in effect, the field name and type are known only at run time. Your code determines the name from a value in the code or from runtime input.

  Dynamic access is in effect when your code is referencing any of these:

  – A variable whose primitive type is ANY.
  – A value field in a dictionary; that field is of type ANY.
  – A field in a record, when the chain of relationships that led to that field (from record to field to subfield) is such that a previous reference used dynamic access.
  – A field that is referenced by the EGL bracket syntax. In this case, the field name does not necessarily follow the rules for identifiers, but can be an EGL reserved word or can include spaces and other characters that would not be valid otherwise.

    For details, see *Bracket syntax for dynamic access*.
- When *static access* is in effect, the field name and type are known at generation time, and the name is always consistent with the naming conventions for EGL identifiers. The name is not used at run time.

  Static access is in effect when your code is referencing any of these:

- A variable that is outside of any container and whose type is other than ANY
- A field in a fixed record
- A field in a non-fixed record, when the chain of relationships that led to that field (from variable to field to subfield) is such that every reference used static access

Consider an example in which the values in a dictionary include a fixed record and a non-fixed record:

```
// a fixed record part
Record myFixedRecordPart type=serialRecord
  {
    fileName = "myFile"
  }
  10 ID INT;
  10 Job CHAR(10);
end

// a record part (not fixed)
Record myDynamicRecordPart type=basicRecord
  ID  INT;
  Job CHAR(10);
end

Program myProgram

  dynamicPerson myDynamicRecordPart;
  myFlexID INT;

  fixedPerson myFixedRecordPart;
  myFixedID INT;

  Function main()

    dynamicPerson.ID = 123;
    dynamicPerson.Job = "Student";

    fixedPerson.ID = 456;
    fixedPerson.Job = "Teacher";

    relationship Dictionary
    {
      dynamicRecord=dynamicPerson,
      staticRecord=fixedPerson
    };
    end
  end
end
```

The following rules apply:

- A reference to a dictionary value is dynamic and every subordinate reference is dynamic. Consider the effect if the code included the following statements:

  ```
  myDynamicID INT;
  myDynamicID = relationship.dynamicRecord.ID;
  ```

  The reference to dynamicRecord would be dynamic, and the reference to ID would be dynamic, with the identifier *ID* visible at run time.
- A reference that begins with a fixed structure can reference only the memory internal to that structure. In the current example, a reference that begins with fixedPerson can access the fields ID and JOB in the fixed record but can access no other fields.

- Your code can access a fixed structure dynamically but the same reference statement cannot access the fields of that field. In the current example, the following reference would not be valid because the identifier *ID* is not available at run time:

```
myFixedID INT;

// NOT valid
myFixedID = relationship.fixedRecord.ID;
```

  You could handle the problem by declaring another fixed record and assigning it values from the fixed record that is in the dictionary:

```
myFixedID INT;
myOtherRecord myFixedRecordPart;
myOtherRecord = relationship.staticRecord;
myFixedID = myOtherRecord.ID;
```

Dynamic access is valid in assignments (on the left- or right-hand sides); in logical expressions; and in the statements **set**, **for**, and **openUI**.

Related concepts
"Bracket syntax for dynamic access" on page 57
"Dictionary" on page 77
"Program part" on page 130
"References to variables in EGL" on page 55
"Typedef" on page 25

Related tasks
"Declaring variables and constants in EGL" on page 50

Related reference
"Assignments" on page 352
"Logical expressions" on page 484
"Primitive types" on page 31
"set" on page 617

Scoping rules and "this" in EGL

If an EGL part declares a variable or constant, the identifier used in the declaration is *in scope* (available) throughout the part:

- If the declaration is in a function, the identifier is in the local scope of the function. If the function Function01 declares the variable Var01, for example, any code in Function01 can reference Var01. The identifier is available even in function code that precedes the declaration.

  The variable can be passed as an argument to another function, but the original identifier is not available in that function. The parameter name is available in the receiving function because the parameter name was declared there.
- If the declaration is in a generatable part such as a program but is outside of any function, the identifier is in *program-global scope*, which means that the identifier can be referenced by any function invoked by that part. For example, if a program declares Var01 and invokes Function01 which in turn invokes Function02, Var01 is available throughout both functions.

  The identifiers in a text or print form are global to the generatable part that references the form. Those identifiers are available even in functions that precede the function which presents the form.
- If the declaration is in a library but outside of any function, the identifier is in *run-unit scope*, which means global to all code in the run unit.

- The names of a dataTable and its fields may be in program-global, run-unit, or an even larger scope, depending on the setting of dataTable properties and on the environment in which the dataTable resides.

Identifiers that are identical cannot be in the same scope. However, most identifiers refer to an area of memory that is logically inside a container such as a record; and in those cases your code qualifies an identifier with the name of the enclosing container. If the function variable myString is in a record called myRecord01, for example, your code refers to the variable as a field of the record:

```
myRecord01.myString
```

If the same identifier is in two scopes, any reference to the identifier is a reference to the most local scope, but you can use qualifiers to override that behavior:

- Consider the case of a program that declares variable Var01 and invokes a function that itself declares a variable of the same name. An unqualified reference to Var01 in the function causes access of the locally declared variable.

  To access an identifier that is program-global even when a local identifier takes precedence, qualify the identifier with the keyword *this*, as in the following example:

  ```
  this.Var01
  ```

  In rare cases the keyword *this* is also used to override a behavior of a set value block in an assignment statement. For details, see *Set value blocks*.
- Consider the following case--
  - A program has a use declaration to access a library; and
  - The program and the library each declare a variable named Var01.

  If a function in the program includes an unqualified reference to Var01, the function accesses the program variable.

  To access an identifier in run-unit scope even when another identifier prevents that access, qualify the identifier with the part name, as in the following example (where myLib is the name of a library):

  ```
  myLib.Var01
  ```

  If the library or dataTable is in a different package and you have not referenced the part in an import statement, you must preface the part name with the package name, as in the following example (where myPkg is the package name):

  ```
  myPkg.myLib.Var01
  ```

  The package name always qualifies a part name and cannot immediately precede a variable or constant identifier.

  Finally, a local identifier may be the same as a dataTable or library name if the local identifier is in a different package from where the dataTable or library resides. To reference the dataTable or library name, include the package name.

Related concepts

"Function part" on page 132
"Library part of type basicLibrary" on page 133
"Library part of type basicLibrary" on page 133
"PageHandler" on page 180
"Parts" on page 17
"Program part" on page 130
"References to parts" on page 20
"References to variables in EGL" on page 55
"Overview of EGL properties" on page 60
"Fixed structure" on page 24
"Typedef" on page 25

Related tasks
"Declaring variables and constants in EGL" on page 50

Related reference
"Function invocations" on page 504
"Function part in EGL source format" on page 513

References to variables in EGL

For details on the distinction between two kinds of memory access, see *Dynamic and static access*.

Regardless of which kind of access is in effect, the EGL dotted syntax is usually sufficient. Consider the following part definitions, for example:

```
Record myRecordPart01 type basicRecord
  myString      STRING;
  myRecordVar02 myRecordPart02;
 end

Record myRecordPart02 type basicRecord
  myString02    STRING;
  myRecordVar03 myRecordPart03;
  myDictionary  Dictionary
  {
    empnum=0005,
    lastName="Twain",
    firstName="Mark",
    birthday="021460"
  };
end

Record myRecordPart03 type basicRecord
  myInt INT;
  myDictionary  Dictionary
  {
    customerNum=0005,
    lastName="Clemens"
  };
end
```

Assume that a function uses the record part *myRecordPart01* as the type when declaring a variable named *myRecordVar01*.

To refer to the field myInt, list the following symbols in order:

- The name of the variable; in this case, *myRecordVar01*
- A period (.)
- A list of the fields that lead to the field of interest, with a period separating one identifier from the next; for example, *myRecordVar02.myRecordVar03*
- The field name of interest, preceded by a period; in this case, *.myInt*

The presence of an array causes a straightforward extension of the same syntax. If myRecordVar03 were declared as an array of three records, for example, you could use the following symbols to access the field myInt in the third element of that array:

```
myRecordVar01.myRecordVar02.myRecordVar03[3].myInt
```

The dotted syntax also works when you reference a dictionary field in this example. To access the value "Twain", specify the following characters on the right-hand side of an assignment statement:

```
myRecordVar01.myRecordVar02.myDictionary.lastName
```

The presence of a field named myDictionary in two different record parts does not pose a problem because each same-named field is referenced in relation to its own, enclosing record.

You also can use dotted syntax to refer to a constant (such as myConst) in a library (such as myLib):

```
myLib.myConstant
```

Two other syntaxes are available:

- When using dynamic access, you may wish to specify a field name as a quoted string or as an identifier of type STRING. This capability is used primarily when you are adding or retrieving a dictionary entry (a key-and-value pair), in these cases:
  - The key is an EGL reserved word or includes a character (such as a period or space) that is not valid in an identifier; or
  - You wish to use a string constant to assign or reference the key.

  The syntax requires that you place the variable, constant, or literal inside a pair of hard brackets([ ]). The content-filled brackets are equivalent to a dot followed by a valid identifier, and you can mix the two syntaxes. However, the beginning of a reference must be an identifier.

  For examples, see *Bracket syntax for dynamic access*.
- You may want the convenience of an abbreviated syntax for referencing a field in a fixed structure (a dataTable, text form, print form, or fixed record). It is recommended that you avoid this syntax, however, in favor of the full qualification described earlier.

  An abbreviated syntax can be valid in relation to fixed structures only if you set the property **allowUnqualifiedItemReferences** to *yes*. That property is a characteristic of generatable logic parts like programs, libraries, and pageHandlers; and the default value is *no*.

  For details, see *Abbreviated syntax for static access*.

Related concepts
"Abbreviated syntax for referencing fixed structures" on page 58
"Bracket syntax for dynamic access" on page 57
"Dynamic and static access" on page 51
"Enumerations in EGL" on page 471
"Function part" on page 132
"Parts" on page 17
"Program part" on page 130
"References to parts" on page 20
"Scoping rules and "this" in EGL" on page 53
"Fixed structure" on page 24
"Typedef" on page 25

Related tasks
"Declaring variables and constants in EGL" on page 50

Related reference
"Arrays" on page 69
"Function invocations" on page 504

"Function part in EGL source format" on page 513
"Primitive types" on page 31
"Use declaration" on page 930

Bracket syntax for dynamic access

Wherever dynamic access is valid, you can reference a field by using a string variable, constant, or literal in brackets. Each content-filled pair of brackets is equivalent to a dot followed by a valid identifier.

Although any keys specified in a dictionary declaration must fulfill the rules for EGL identifiers, you can specify a wider range of keys by using bracket syntax in EGL assignment statements. Bracket syntax is required in the next example, where two entries are added to a dictionary and the value in each of those entries is retrieved:

```
row Dictionary { lastname = "Smith" };
category, motto STRING;

row["Record"] ="Reserved word";
row["ibm.com"]="Think!";

category = row["Record"];
motto    = row["ibm.com"]
```

If you reference a value by using an identifier in dotted syntax, you can reference the same value in bracket syntax by using a string that is equivalent to the identifier. The following assignments have the same effect:

```
row.age = 20;
row["age"] = 20;
```

Assume that you declared a record named myRecordVar01, which includes a field named myRecordVar02, and that myRecordVar02 is itself a record that includes the previous dictionary. A valid reference is as follows:

```
myRecordVar01.myRecordVar02.row.lastName
```

Access is static for most of that reference. Dynamic access begins when you access the field in the dictionary. Assume that these constants are in scope, however:

```
const SECOND STRING = "myRecordVar02";
const GROUP  STRING = "row";
const LAST   STRING = "lastName";
```

You can code the previous reference as follows:

```
myRecordVar01[SECOND][GROUP][LAST]
```

The first symbol in a reference must always be a valid identifier, but in this case, dynamic access is in effect after that identifier.

You can mix the dotted and bracket syntaxes. For example, the following reference is equivalent to the previous one:

```
myRecordVar01[SECOND].row[LAST]
```

As a final example, consider a reference with an array index:

```
myRecordVar01.myRecordVar02.myRecordVar03[3][2].myInt
```

Assume that these constants are in scope:

```
const SECOND  STRING = "myRecordVar02";
const THIRD   STRING = "myRecordVar03";
const CONTENT STRING = "myInt";
```

You can code the previous reference in these ways:

```
myRecordVar01[SECOND][THIRD][3][2][CONTENT]

myRecordVar01[SECOND][THIRD][3][2].myInt

myRecordVar01.myRecordVar02.THIRD[3][2][CONTENT]
```

Related concepts
"Abbreviated syntax for referencing fixed structures"
"Dynamic and static access" on page 51
"Function part" on page 132
"Parts" on page 17
"Program part" on page 130
"References to parts" on page 20
"References to variables in EGL" on page 55
"Scoping rules and ″this″ in EGL" on page 53
"Fixed structure" on page 24
"Typedef" on page 25

Related tasks
"Declaring variables and constants in EGL" on page 50

Related reference
"Arrays" on page 69
"Function invocations" on page 504
"Function part in EGL source format" on page 513
"Options records for MQ records" on page 645
"Primitive types" on page 31
"Use declaration" on page 930

Abbreviated syntax for referencing fixed structures

The following rules are in effect for referencing fields in a dataTable, text form, print form, or fixed record:

- If you are referencing a field in a container such as a fixed record, you can use the usual dotted syntax to avoid ambiguity about the area of memory being referenced. Consider the following part declaration, for example:

```
Record myRecordPart type serialRecord
  {
    fileName = "myFile"
  }
  10 myTop;
    20 myNext;
      30 myAlmost;
        40 myChar CHAR(10);
        40 myChar02 CHAR(10);
end
```

  Assume that a function uses the record part *myRecordPart* as the type when declaring a variable named *myRecordVar*.

  A valid reference to myChar in myRecordVar is as follows:

```
myRecordVar.myTop.myNext.myAlmost.myChar
```

  That reference is considered to be *fully qualified*.

- If you want to refer to a field whose name is unique within a structure, you can specify the variable name, followed by a period, followed by the field name. Valid references for the earlier example include this symbol:

  ```
  myRecordVar.myChar
  ```

  That reference is considered to be *partially qualified*.

  You cannot partially qualify a field name in any other way. You cannot include only some of the field names that are between the variable name and the field name of interest, for example, nor can you eliminate the variable name while keeping any of the names of structure field that are superior to the field of interest. The following references are *not* valid for the earlier example:

  ```
  // NOT valid
  myRecordVar.myNext.myChar
  myRecordVar.myAlmost.myChar
  myNext.myChar
  myAlmost.myChar
  ```

- You can refer to a field without preceding the name with any qualifiers. Valid references for the earlier example include these symbols:

  ```
  myChar
  myChar02
  ```

  Those references are considered to be *unqualified*.

- You must qualify any reference to a structure field to the extent necessary to avoid ambiguity.
- The name of a structure field can be an asterisk (*) if the related memory area is a *filler*, which is an area whose name is of no importance. You cannot include an asterisk in a reference. Consider this example:

  ```
  record myRecordPart type serialRecord
  {
    fileName = "myFile"
  }
    10 person;
      20 *;
        30 streetAddress1 CHAR(30);
        30 streetAddress2 CHAR(30);
        30 nation CHAR(20);
  end
  ```

  If you use that part as a type when declaring the variable *myRecordVar*, you can refer to *myRecordVar.nation* or *nation*, but the following references are not valid:

  ```
  // NOT valid
  myRecordVar.*.streetAddress1
  myRecordVar.*.streetAddress2
  myRecordVar.*.nation
  ```

- When EGL tries to resolve a reference, names of local variables are searched first, then names of structure fields in the records used for I/O in the same function, then names of other local structure fields, then names that are program-global.

  Consider the case in which a function declares both a primitive variable called *nation* and a variable that points to the following basic record:

  ```
  record myRecordPart
    10 myTop;
      20 myNext;
        30 nation CHAR(20);
  end
  ```

  An unqualified reference to *nation* refers to the primitive variable, not to the structure field.

- A name search shows no preference for program-global primitive variables over program-global structure fields. Consider the case in which a program declares both a primitive variable called *nation* and a variable that points to the format of the following basic record:

```
record myRecordPart
  10 myTop;
    20 myNext;
      30 nation CHAR(20);
end
```

  An unqualified reference to *nation* fails because *nation* could refer either to the primitive variable or to the structure field. You can reference the structure field, but only by qualifying the reference.

For additional rules, see *Arrays* and *Use declaration*.

Related concepts
"Function part" on page 132
"Parts" on page 17
"Program part" on page 130
"References to parts" on page 20
"References to variables in EGL" on page 55
"Scoping rules and "this" in EGL" on page 53
"Fixed structure" on page 24
"Typedef" on page 25

Related tasks
"Declaring variables and constants in EGL" on page 50

Related reference
"Arrays" on page 69
"Function invocations" on page 504
"Function part in EGL source format" on page 513
"Options records for MQ records" on page 645
"Primitive types" on page 31
"Use declaration" on page 930

Overview of EGL properties

Most EGL parts have a set of properties that are used to create appropriate output at generation time. The set of valid properties varies by context:

- Each part type defines a set of properties that are for the part type as a whole. Each program part, for example, has a property called **alias**, which identifies the name of the compilable unit.

  If a part is itself a subtype, additional properties are available. A program of type **textUI** has a property called **alias**, as well as a property called **inputForm**. The latter identifies a text form that is presented to the user before the program logic runs.
- Many part types also define a set of properties for use in any of the primitive fields that are components of that part type. A record part of type SQLRecord, for example, includes a set of primitive fields, and each has a **column** property that identifies the SQL table column accessed by the field.

  The properties available in a DataItem part include *all* the primitive field-level properties that are valid in any context. Consider, for example, a DataItem part that represents an ID of nine (and only nine) digits, where in some cases the ID is associated with a relational-database column called *SSN*:

```
DataItem IDPart CHAR(9)
{
  minInput = 9,      // requires 9 input characters
  isDigits = yes,    // requires digits
  columnName = "SSN" // is related to a column
}
```

You can declare a variable of type IDPart as follows:

```
myVariable IDPart;
```

You can declare that variable in a composite part such as a record part or directly in a logic part such as program. In every case, the part type determines whether a given property is used.

In the current example, the property **columnName** is used only if the variable is declared in a record of type SQLRecord. The two validation properties are used only if the variable is declared in a user-interface part such as a pageHandler.

- In some variable declarations, you can override a property that was specified in the related part definition, but only if the property is useful in the context in which the variable is declared:
  - Overriding in context is possible when you declare a variable that is based on a DataItem part. The following statement declares a PageHandler field of type SSN (as defined earlier), but the statement does not require that the user type digits:

    ```
    myVariable IDPart { isDigits = no };
    ```

    In this example, the property **minInput** is unaffected by the override, and the property **columnName** is ignored.
  - In most cases, overrides are not possible for properties of composite parts such as Record parts.
- When you define a fixed structure, you can assign properties to the elementary structure fields and can override those properties when you declare a related variable. You also may assign properties to a structure field that has subordinate structure fields, but in those cases, the assigned properties are ignored unless the property documentation says otherwise.
- When you declare a variable of a primitive type, you can set any of the primitive field-level properties that are useful in the context of the variable declaration.

A property cannot be accessed at run time. When you create variables that are based on an SQL record part, for example, the logic that you write cannot retrieve or change the names assigned to the **tableNames** property, which identifies the SQL tables that are accessed by the record. Even if you override a property value in a variable declaration, the value that you specify at development time cannot be changed by your logic.

The lack of run-time access to a property value means that when you assign the content of a variable or use the variable as an argument, the property value is not transferred along with the content. If you copy data from one SQL record to another, for example, no change is made to the specification of which SQL tables are accessed by the destination record. Similarly, when you pass an SQL record to an EGL function, the parameter receives field content, but retains the SQL-table specifications that were assigned at development time.

Predefined EGL parts such as ConsoleField may include both properties and fields. Unlike properties, the fields *are* available at run time. The logic that you write can read the field value and in many cases can change the field value.

A set-value block is an area of code in which you can set both property and field values. For details, see *Set-value block*.

Related concepts
"References to variables in EGL" on page 55
"Set-value blocks" on page 63

Related reference
"Form part in EGL source format" on page 497
"SQL item properties" on page 63

Field-presentation properties

The EGL field-presentation properties specify characteristics that are meaningful when a field is displayed in an on-screen output, when the destination is a command window, but not a Web browser.

The properties are as follows:

- "color" on page 672
- "highlight" on page 680
- "intensity" on page 681
- "outline" on page 689

In addition, the following properties are meaningful when the field is displayed in a printable output, when the destination is a printer or a print file:

- **highlight** property (but only for *underline* and *noHighLight*, and only for Java output)
- **outline** property, which is appropriate only for devices that support double-byte characters

The field-presentation properties have no effect on data that is returned to the program from a text form; they are solely for output.

Formatting properties

The formatting properties specify characteristics that are meaningful when data is presented on a form or a Web browser:

- "align" on page 670
- "currency" on page 674
- "currencySymbol" on page 674
- "dateFormat" on page 675
- "fillCharacter" on page 679
- "isBoolean" on page 682
- "lineWrap" on page 684
- "lowerCase" on page 685
- "masked" on page 685
- "numericSeparator" on page 689
- "outline" on page 689
- "sign" on page 693
- "timeFormat" on page 695
- "timeStampFormat" on page 696
- "upperCase" on page 697

- "zeroFormat" on page 703

Related concepts
"Overview of EGL properties" on page 60

SQL item properties

The SQL item properties specify characteristics that are meaningful when an item is used in a record of type SQLRecord. You do not need to specify any of the SQL item properties, however, as default values are available.

The properties are as follows:

- "column" on page 672
- "isNullable" on page 683
- "isReadOnly" on page 684
- "maxLen" on page 685
- "persistent" on page 690
- "sqlDataCode" on page 693
- "sqlVariableLen" on page 694

Validation properties

The validation properties restrict what is accepted when the user enters data in a text form.

The properties are as follows:

- "fill" on page 679
- "inputRequired" on page 680
- "inputRequiredMsgKey" on page 681
- "isDecimalDigit" on page 682
- "isHexDigit" on page 682
- "minimumInput" on page 686
- "minimumInputMsgKey" on page 686
- "needsSOSI" on page 687
- "typeChkMsgKey" on page 697
- "validatorDataTable" on page 698
- "validatorDataTableMsgKey" on page 699
- "validatorFunction" on page 699
- "validatorFunctionMsgKey" on page 700
- "validValues" on page 701
- "validValuesMsgKey" on page 702

Set-value blocks

A set-value block is an area of code in which you can set both property and field values. For background, see *Overview of EGL properties*.

A set-value block is available when you take any of the following actions:

- Define a part
- Declare a variable
- Code a special form of an assignment statement

- Code an **openUI** statement, as described in *openUI*

In the latter two cases, you can assign values only to fields.

Note: A restriction applies to fields in fixed structures. You can use set-value blocks to assign the values of the primitive field-level properties, but not to set the values of the fields themselves.

Set-value blocks for elementary situations

Consider the rules that apply in the most elementary cases:

- Each set-value block begins with a left curly brace ({), includes either a list of entries that are separated by commas or a single entry, and ends with a right curly brace (})
- The entries are all in one of two formats:
  - Each entry is composed of an identifier-and-value pair such as **inputRequired = yes**; or
  - Each entry contains values that are assigned positionally, when successive values are assigned to successive elements of an array.

In all cases, the set-value block is in the scope of the part, variable, or field being modified. The variations in syntax are best illustrated by example.

The first example shows a dataItem part, which has two properties (**inputRequired** and **align**) :

```
// the scope of the set-value block is myPart
DataItem myPart INT
  {
  inputRequired = yes,
  align = left
  }
end
```

The next example shows a variable of primitive type.

```
 // the scope is myVariable
 myVariable INT
 {
    inputRequired = yes,
    align = left
 };
```

The next example shows an SQL record part declaration, which includes two record properties (**tableNames** and **keyItems**):

```
// The scope is myRecordPart
Record myRecordPart type SQLRecord
  { tableNames = [["myTable"]],
    keyItems = ["myKey"] }
  myKey CHAR(10);
  myOtherKey CHAR(10);
  myContent01 CHAR(60);
  myContent02 CHAR(60);
end
```

The next example shows a variable declaration that uses the previous part as a type, overrides one of the two record properties, and sets two fields in the record:

```
// The scope is myRecord
myRecord myRecordPart
  {
```

```
    keyItems = ["myOtherKey"],
    myContent01 = "abc",
    myContent02 = "xyz"
  };
```

Additional examples include variable declarations and assignment statements:

```
// the example shows the only case in which a
// record property can be overridden in a
// variable declaration.
// the scope is myRecord
myRecord myRecordPart {keyItems = ["myOtherKey"]};

// the scope is myInteger, which is an array
myInteger INT[5] {1,2,3,4,5};

// these assignment statements
// have no set-value blocks
myRecord02.myContent01 = "abc";
myRecord02.myContent02 = "xyz";

// this abbreviated assignment statement
// is equivalent to the previous two, and
// the scope is myRecord02
myRecord02
  {
    myContent01="abc",
    myContent02="xyz"
  };

// This abbreviated assignment statement
// resets the first four elements of the array
// declared earlier
myInteger{6,7,8,9};
```

The abbreviated assignment statement is not available for fields in a fixed structure.

Set-value blocks for a field of a field

When you are assigning values for a field of a field, you use a syntax in which the set-value block is in a scope such that the entries are modifying only the field of interest.

Consider the following part definitions:

```
record myBasicRecPart03 type basicRecord
  myInt04 INT;
end

record myBasicRecPart02 type basicRecord
  myInt03 INT;
  myRec03 myBasicRecPart03;
end

record myBasicRecPart type basicRecord
  myInt01 INT;
  myInt02 INT;
  myRec02 myBasicRecPart02;
end
```

You can assign a property value for any field as follows:

- Create a set-value block for the record
- Embed a series of field names to narrow the scope

- Create the field-specific set-value block

The syntax for assigning a property value may take any of three forms, as shown in the following examples, which apply to the field myInt04:

```
// dotted syntax, as described in
// References to variables in EGL.
myRecB myBasicRecPart
{
  myRec02.myRec03.myInt04{ align = left }
};

// bracket syntax, as described in
// Bracket syntax for dynamic access.
// You cannot use this syntax to affect
// fields in fixed structures.
myRecC myBasicRecPart
{
  myRec02["myRec03"]["myInt04"]{ align = left }
};

// curly-brace syntax
myRecA myBasicRecPart
{
  myRec02 {myRec03 { myInt04 { align = left }}}
};
```

Even in complex cases, you use a comma to separate one entry in a set-value block from the next; but you need to consider the level at which a given block is nested:

```
// dotted syntax
myRecB myBasicRecPart
{
  myInt01 = 4,
  myInt02 = 5,
  myRec02.myRec03.myInt04{ align = left },
  myRec02.myInt03 = 6
};

// bracket syntax
myRecC myBasicRecPart
{
  myInt01 = 4,
  myInt02 = 5,
  myRec02["myRec03"]["myInt04"]{ align = left },
  myRec02["myInt03"] = 6
};

// curly-brace syntax;
// but this usage is much harder to maintain
myRecA myBasicRecPart
{
  myInt01 = 4,
  myInt02 = 5,
  myRec02
    {
      myRec03
        { myInt04
          { action = label5 }},
      myInt03 = 6
    }
};
```

Use of "this"

In a variable declaration or assignment statement, you can have a container (such as an SQL record) that includes a field (such as *keyItems*) which is named the same

as a record property. To refer to your field rather than to the property, use the keyword **this**, which establishes the correct scope for the set-value block or for an entry in the set-value block.

Consider the following record declaration:

```
Record myRecordPart type SQLRecord
  { tableNames = [["myTable"]],
    keyItems = ["myKey"] }
  myKey CHAR(10);
  myOtherKey CHAR(10);
  keyItems CHAR(60);
end
```

The following record declaration first sets a value for the property keyItems, then sets a value for the field of the same name:

```
myRecord myRecordPart
{
   keyItems = ["myOtherKey"],
   this.keyItems = "abc"
};
```

The next section gives an additional example in an array declaration.

Set-value blocks, arrays, and array elements

When you declare a dynamic array, you can specify the initial number of elements, as in this example:

```
col1 ConsoleField[5];
```

Assignments in a set-value block refer to properties and predefined fields in each of the initial elements of type ConsoleField, though not to any elements that are added later:

```
col1 ConsoleField[5]
{
  position = [1,1],
  color = red
};
```

To assign values to a particular element in a variable declaration, create an embedded set-value block whose scope is that element. As shown in the following example, you specify the scope by using the keyword **this** with a bracketed index:

```
// assign values to the second and fourth element
col1 ConsoleField[5]
{
  this[2] { color = blue },
  this[4] { color = blue }
};
```

For details on another use of the keyword **this**, see *Scoping rules and "this" in EGL*.

You can use positional entries in a set-value block to assigns value to successive elements in an array of any of these types (as is relevant only when processing reports or creating console forms):

- ConsoleField
- Menu
- MenuItem
- Prompt
- Report

- ReportData

The following example could be in an OpenUI statement. The scope of each embedded set-value block is a specific array element:

```
new Menu
{
  labelText = "Universe",
  MenuItems =

  // property value is a dynamic array
  [
    new MenuItem
    { name = "Expand",
       labelText = "Expand" },
    new MenuItem
    { name = "Collapse",
       labelText = "Collapse" }
  ]
}
```

Additional examples

Consider the following parts:

```
Record Point
  x, y INT;
end

Record Rectangle
  topLeft, bottomRight Point;
end
```

The following code is valid:

```
Function test()
  screen Rectangle
  {
    topLeft{x=1, y=1},
    bottomRight{x=80, y=24}
  };

  // change x, y in code, using a statement
  // that is equivalent to the following code:
  //   screen.topLeft.x = 1;
  //   screen.topLeft.y = 2;
  screen.topLeft{x=1, y=2};
end
```

Next, initialize a dynamic array of elements of type Point in the same function:

```
pts Point[2]
{
  this[1]{x=1, y=2},
  this[2]{x=2, y=3}
};
```

Set the value of each element that is now in the array, then set the first element to a different value:

```
pts{ x=1, y=1 };
pts[1]{x=10, y=20};
```

In the previous example, pts[1] is used rather than this[1] because the array name is unambiguous.

Next, consider another dynamic array of type Point:

```
points Point[];
```

The following assignment statement has no effect because no elements exist:

```
points{x=1, y=1};
```

In contrast, the following assignment statement causes an out-of-bounds exception because a particular element is referenced and does not exist:

```
points[1]{x=10, y=20};
```

You can add elements to the array, then use a single statement to set values in all elements:

```
points.resize(2);
points{x=1, y=1};
```

Related concepts
"Bracket syntax for dynamic access" on page 57
"Overview of EGL properties" on page 60
"References to variables in EGL" on page 55
"Scoping rules and ″this″ in EGL" on page 53

Related reference
"Arrays"
"Data initialization" on page 459
"openUI" on page 602

Arrays

EGL supports the following kinds of arrays:

- "Dynamic arrays"
- "Structure-field arrays" on page 73

In either case, the maximum number of supported dimensions is seven.

Dynamic arrays

When you declare an array of records, fixed records, or primitive variables, the array has an identity independent of the elements in the array:

- A set of functions are specific to the array, allowing you to grow or shrink the number of elements at run time.
- The array-specific property **maxSize** indicates how many elements are valid in the array. The default value is unbounded; the number of elements is limited only by the requirements of the target environment.

You do not need to specify a number of elements in your declaration, but if you do, that number indicates the initial number of elements. You also can specify the initial number of elements by listing a series of array constants in the declaration, as is possible only with primitive variables, not with records.

The syntax for declaring a dynamic array is shown in the following examples:

```
// An array of 5 elements or less
myDataItem01 CHAR(30)[] { maxSize=5 };

// An array of 6 elements or less,
// with 4 elements initially
myDataItem02 myDataItemPart[4] { maxSize=6 };
```

```
// An array that has no elements
// but whose maximum size is the largest possible
myRecord myRecordPart[];

// A 3-element array whose elements
// are assigned the values 1, 3, and 5
position int[] = [1,3,5];
```

You can use a literal integer to initialize the number of elements, but neither a variable nor a constant is valid.

When you declare an array of arrays, an initial number of elements is valid in the leftmost-specified dimension and in each subsequent dimension until a dimension lacks an initial number. The following declarations are valid:

```
// Valid, with maxsize giving the maximum
// for the first dimension
myInt01 INT[3][];
myInt02 INT[4][2][] {maxsize = 12};
myInt03 INT[7][3][1];

// In the next example, array constants indicate
// that the outer array initially has 3 elements.
// The first element of the outer array is
// an array of two elements (with values 1 and 2).
// The second element of the outer array is
// an array of three elements (with values 3, 4,5).
// The third element of the outer array is
// an array of two elements (with values 6 and 7).
myInt04 INT[][] = [[1,2],[3,4,5],[6,7]];
```

In the following example, the syntax is not valid because (for instance) the array myInt04 is declared as an array of no elements, but each of those elements is assigned 3 elements:

```
// NOT valid
myInt04 INT[][3];
myInt05 INT[5][][2];
```

An array specified as a program or function parameter cannot specify the number of elements.

When your code references an array or an array element, these rules apply:

- An element subscript can be any numeric expression that resolves to an integer, but the expression cannot include a function invocation.
- If your code refers to a dynamic array but does not specify subscripts, the reference is to the array as a whole.

An out-of-memory situation is treated as a catastrophic error and ends the program.

Dynamic-array functions

A set of functions and read-only variables are available for each dynamic array. In the following example, the array is called *series*:

```
series.reSize(100);
```

The name of the array may include a set of brackets, each containing an integer. An example is as follows:

```
series INT[][];

// resizes the second element
// of series, which is an array of arrays
series[2].reSize(100);
```

In the following sections, substitute the array name for *arrayName* and note that the name may be qualified by a package name, a library name, or both.

appendAll():

```
arrayName.appendAll(appendArray  Array in)
```

This function does as follows:

- Appends to the array that is referenced by *arrayName*, adding a copy of the array that is referenced by *appendArray*
- Increments the array size by the number of added elements
- Assigns an appropriate index value to each of the appended elements

The elements of appendArray must be of the same type as the elements of *arrayName*.

appendElement()

```
arrayName.appendElement(content ArrayElement in)
```

This function places an element to the end of the specified array and increments the size by one. For *content*, you can substitute a variable of the appropriate type; alternatively, you can specify a literal that is assigned to an element created during the operation. The process copies data; if you assign a variable, that variable is still available for comparison or other purposes.

The rules for assigning a literal are as specified in *Assignments*.

getMaxSize()

```
arrayName.getMaxSize ( ) returns (INT)
```

This function returns an integer that indicates the maximum of elements allowed in the array.

getSize()

```
arrayName.getSize ( ) returns (INT)
```

This function returns an integer that indicates the number of elements in the array. It is recommend that you use this function instead of *SysLib.size* when you are working with dynamic arrays.

Another function provides functionality equivalent to that of *arrayName*.getSize:

```
SysLib.size( ) returns (INT)
```

However, it is recommended that you use *arrayName*.**getSize** ( ) when you work with dynamic arrays.

insertElement()

```
arrayName.insertElement (content ArrayElement in, index INT in)
```

This function does as follows:

- Places an element in front of the element that is now at the specified location in the array
- Increments the array size by one
- Increments the index of each element that resides after the inserted element

content is the new content (a constant or variable of the appropriate type for the array), and *index* is an integer literal or a numeric variable that indicates the location of the new element.

If *index* is one greater than the number of elements in the array, the function creates a new element at the end of the array and increments the array size by one.

removeAll()

```
arrayName.removeAll( )
```

This function removes the elements of the array from memory. The array can be used, but its size is zero.

removeElement()

```
arrayName.removeElement(index INT in)
```

This function removes the element at the specified location, decrements the array size by one, and decrements the index of each element that resides after the removed element.

index is an integer literal or a numeric variable that indicates the location of the element to be removed.

resize()

```
arrayName.resize(size INT in)
```

This function adds or shrinks the current size of the array to the size specified in *size*, which is an integer literal, constant, or variable. If the value of *size* is greater than the maximum size allowed for the array, the run unit terminates.

reSizeAll()

```
arrayName.reSizeAll(sizes INT[
] in)
```

This function adds or shrinks every dimension of a multidimensional array. The parameter *sizes* is an array of integers, with each successive element specifying the size of a successive dimension. If the number of dimensions resized is greater than the number of dimensions in *arrayName*, or if a value of an element in *sizes* is greater than the maximum size allowed in the equivalent dimension of *arrayName*, the run unit terminates.

setMaxSize()

```
arrayName.setMaxSize (size INT in)
```

The function sets the maximum of elements allowed in the array. If you set the maximum size to a value less than the current size of the array, the run unit terminates.

setMaxSizes()

```
arrayName.setMaxSizes(sizes INT[
] in)
```

This function sets every dimension of a multidimensional array. The parameter *sizes* is an array of integers, with each successive element specifying the maximum size of a successive dimension. If the number of dimensions specified is greater than the number of dimensions in *arrayName*, or if a value of an element in *sizes* is less than the current number of elements in the equivalent dimension of *arrayName*, the run unit terminates.

Use of dynamic arrays as arguments and parameters

A dynamic array can be passed as an argument to an EGL function. The related parameter must be defined as a dynamic array of the same type as the argument; and for a data item, the type must include the same length and decimal places, if any.

A dynamic array cannot be passed as an argument to another program.

An example of a function that uses a dynamic array as a parameter is as follows:

```
Function getAll (employees Employee[])
 ;
end
```

At run time, the maximum size for a parameter is the maximum size declared for the corresponding argument. The function or called program can change the size of the array, and the change is in effect in the invoking code.

SQL processing and dynamic arrays

EGL lets you use a dynamic array to access rows of a relational database. For details on reading multiple rows, see *get*. For details on adding multiple rows, see *add*.

Structure-field arrays

You declare a structure-field array when you specify that a field in a fixed structure has an occurs value greater than one, as in the following example:

```
Record myFixedRecordPart
  10 mySi CHAR(1)[3];
end
```

If a fixed record called *myRecord* is based on that part, the symbol *myRecord.mySi* refers to a one-dimensional array of three elements, each a character.

Usage of a structure-field array

You can reference an entire array of structure fields (for example, *myRecord.mySi*) in these contexts:

- As the second operand used by an *in* operator. The operator tests whether a given value is contained in the array.
- As the parameter in the function **sysLib.size**. That function returns the occurs value of the structure field.

An array element that is not itself an array is a field like any other, and you can reference that field in various ways; for example, in an assignment statement or as an argument in a function invocation.

An element subscript can be any numeric expression that resolves to an integer, but the expression cannot include a function invocation.

One-dimensional structure-field array

You can refer to an element of a one-dimensional array like *myRecord.mySi* by using the name of the array followed by a bracketed subscript. The subscript is either an integer or a field that resolves to an integer; for example, you can refer to the second element of the example array as *myStruct.mySi[2]*. The subscript can vary from 1 to the occurs value of the structure field, and a run-time error occurs if the subscript is outside of that range.

If you use the name of a structure-field array in a context that requires a field but do not specify a bracketed subscript, EGL assumes that you are referring to the first element of the array, but only if you are in VisualAge Generator compatibility mode. It is recommended that you identify each element explicitly. If you are not in VisualAge Generator compatibility mode, you are required to identify each element explicitly.

The next examples show how to refer to elements in a one-dimensional array. In those examples, valueOne resolves to 1 and valueTwo resolves to 2:

```
// these refer to the first of three elements:
myRecord.mySi[valueOne]

// not recommended; and valid
// only if VisualAge Generator
// compatibility is in effect:
myRecord.mySi

// this refers to the second element:
myRecord.mySi[valueTwo]
```

A one-dimensional array may be substructured, as in this example:

```
record myRecord01Part
  10 name[3];
    20 firstOne CHAR(20);
    20 midOne CHAR(20);
    20 lastOne CHAR(20);
end
```

If a record called *myRecord01* is based on the previous part, the symbol *myRecord01.name* refers to a one-dimensional array of three elements, each of which has 60 characters, and the length of myRecord01 is 180.

You may refer to each element in *myRecord01.name* without reference to the substructure; for example, *myRecord01.name[2]* refers to the second element. You also may refer to a substructure within an element. If uniqueness rules are satisfied, for example, you can reference the last 20 characters of the second element in any of the following ways:

```
myRecord01.name.lastOne[2]
myRecord01.lastOne[2]
lastOne[2]
```

The last two are valid only if the generatable-part property **allowUnqualifiedItemReferences** is set to *yes*.

For details on the different kinds of references, see *References to variables and constants*.

Multidimensional structure-field array

If a structure item with an occurs value greater than one is substructured and if a subordinate structure item also has an occurs value greater than one, the subordinate structure item declares an array with an additional dimension.

Let's consider another record part:

```
record myRecord02Part
  10 siTop[3];
    20 siNext CHAR(20)[2];
end
```

If a record called *myRecord02* is based on that part, each element of the one-dimensional array *myRecord02.siTop* is itself a one-dimensional array. For example, you can refer to the second of the three subordinate one-dimensional arrays as *myRecord02.siTop[2]*. The structure item siNext declares a two-dimensional array, and you can refer to an element of that array by either of these syntaxes:

```
// row 1, column 2.
// the next syntax is strongly recommended
// because it works with dynamic arrays as well
myRecord02.siTop[1].siNext[2]

// the next syntax is supported for compatibility
// with VisualAge Generator
myRecord02.siTop.siNext[1,2]
```

To clarify what area of memory is being referenced, let's consider how data in a multidimensional array is stored. In the current example, myRecord02 constitutes 120 bytes. The referenced area is divided into a one-dimensional array of three elements, each 40 bytes:

```
siTop[1]     siTop[2]     siTop[3]
```

Each element of the one-dimensional array is further subdivided into an array of two elements, each 20 bytes, in the same area of memory:

```
siNext[1,1] siNext[1,2] siNext[2,1] siNext[2,2] siNext[3,1] siNext[3,2]
```

A two-dimensional array is stored in row-major order. One implication is that if you initialize an array in a double while loop, you get better performance by processing the columns in one row before processing the columns in a second:

```
// i, j, myTopOccurs, and myNextOccurs are data items;
// myRecord02 is a record; and
// sysLib.size() returns the occurs value of a structure item.
i = 1;
j = 1;
myTopOccurs = sysLib.size(myRecord02.siTop);
myNextOccurs = sysLib.size(myRecord02.siTop.siNext);
while (i <= myTopOccurs)
  while (j <= myNextOccurs)
    myRecord02.siTop.siNext[i,j] = "abc";
    j = j + 1;
  end
  i = i + 1;
end
```

You must specify a value for each dimension of a multidimensional array. The reference *myRecord02.siTop.siNext[1]*, for example, is not valid for a 2-dimensional array.

An example declaration of a 3-dimensional array is as follows:

```
record myRecord03Part
  10 siTop[3];
    20 siNext[2];
      30 siLast CHAR(20)[5];
end
```

If a record called *myRecord03* is based on that part and if uniqueness rules are satisified, you can reference the last element in the array in any of the following ways:

```
// each level is shown, and a subscript
// is on each level, as is recommended.
myRecord03.siTop[3].siNext[2].siLast[5]

// each level shown, and subscripts are on lower levels
myRecord03.siTop.siNext[3,2].siLast[5]
myRecord03.siTop.siNext[3][2].siLast[5]

// each level is shown, and subscripts are on the lowest level
myRecord03.siTop.siNext.siLast[3,2,5]
myRecord03.siTop.siNext.siLast[3,2][5]
myRecord03.siTop.siNext.siLast[3][2,5]
myRecord03.siTop.siNext.siLast[3][2][5]

// the container and the last level is shown, with subscripts
myRecord03.siLast[3,2,5]
myRecord03.siLast[3,2][5]
myRecord03.siLast[3][2,5]
myRecord03.siLast[3][2][5]

// only the last level is shown, with subscripts
siLast[3,2,5]
siLast[3,2][5]
siLast[3][2,5]
siLast[3][2][5]
```

As indicated by the previous example, you reference an element of a multidimensional array by adding a bracketed set of subscripts, in any of various ways. In all cases, the first subscript refers to the first dimension, the second subscript refers to the second dimension, and so forth. Each subscript can vary from 1 to the occurs value of the related structure item, and a run-time error occurs if a subscript resolves to a number outside of that range.

First, consider the situation when subscripts are not involved:

- You can specify a list that begins with the name of the variable and continues with the names of increasingly subordinate structure items, with each name separated from the next by a period, as in this example:

  ```
  myRecord03.siTop.siNext.siLast
  ```

- You can specify the name of the variable, followed by a period, followed by the name of the lowest-level item of interest, as in this example:

  ```
  myRecord03.siLast
  ```

- If the lowest-level item of interest is unique in a given name space, you can specify only that item, as in this example:

  ```
  siLast
  ```

Next, consider the rules for placing array subscripts:

- You can specify a subscript at each level where one of several elements is valid, as in this example:

  ```
  myRecord03.siTop[3].siNext[2].siLast[5]
  ```

- You can specify a series of subscripts at any level where one of several elements is valid, as in this example:

```
myRecord03.siTop.siNext[3,2].siLast[5]
```

- You can specify a series of subscripts at any level that is at or subordinate to a level where one of several elements is valid, as in this example:

```
myRecord03.siTop.siNext.siLast[3,2,5]
```

- An error occurs if you assign more subscripts than are appropriate at a given level. as in this example:

```
// NOT valid
myRecord03.siTop[3,2,5].siNext.siLast
```

- You can isolate a subscript within a bracket or can display a series of subscripts, each separated from the next by a comma; or you can combine the two usages. The following examples are valid:

```
myRecord03.siTop.siNext.siLast[3,2,5]
myRecord03.siTop.siNext.siLast[3,2][5]
myRecord03.siTop.siNext.siLast[3][2,5]
myRecord03.siTop.siNext.siLast[3][2][5]
```

Related concepts
"Compatibility with VisualAge Generator" on page 428
"References to variables in EGL" on page 55

Related reference
"add" on page 544
"Assignments" on page 352
"EGL system limits" on page 481
"get" on page 567
"in operator" on page 518
"size()" on page 881

Dictionary

A *dictionary part* is a part that is always available; you do not define it. A variable that is based on a dictionary part may include a set of keys and their related values, and you can add and remove key-and-value entries at run time. The entries are treated like fields in a record.

An example of a dictionary declaration is as follows:

```
row Dictionary
  {
    ID        = 5,
    lastName  = "Twain",
    firstName = "Mark",
  };
```

When you include entries in the declaration, each key name is an EGL identifier that must be consistent with the EGL naming conventions. When you add entries at run time, you have greater flexibility; you can specify a string literal, constant, or variable, and in this case the content can be an EGL reserved word or can include characters that would not be valid in an identifier. For details, see *Bracket syntax for dynamic access*.

Example assignments are as follows:

```
row.age = 30;
row["Credit"] = 700;
row["Initial rating"] = 500
```

If you attempt to assign a key that already exists, you override the existing key-and-value entry. The following assignment is valid and replaces ″Twain″ with ″Clemens″:

```
row.lastname = "Clemens";
```

Assignments also can be used for data retrieval:

```
lastname String
age, credit, firstCredit int;

lastname = row.lastname;
age = row.age;
credit = row.credit;
credit = row["Credit"];
firstCredit = row["Initial rating"];
```

The value in a key-and-value entry is of type ANY, which means that you can put different kinds of information into a single dictionary. Each value can be any of these:

- A pre-declared record or other variable
- A constant or literal

Putting a variable into a dictionary assigns a copy of the variable. Consider the following record part:

```
Record myRecordPart
  x int;
end
```

The next code places a variable of type myRecordPart into the dictionary, then changes a value in the original variable:

```
testValue int;

myRecord myRecordPart;

// sets a variable value and places
// a copy of the variable into the dictionary.
myRecord.x = 4;
row Dictionary
{
  theRecord myRecord;
}

// Places a new value in the original record.
myRecord.x = 700;

// Accesses the dictionary's copy of the record,
// assigning 4 to testValue.
testValue = row.theRecord.x;
```

Assigning one dictionary to another replaces the target content with the source content and overrides the values of the target dictionary’s properties, which are described later. The conditional statement in the following code is true, for example:

```
row Dictionary { age = 30 };

newRow Dictionary { };
newRow = row
```

```
// resolves to true
if (newRow.age == 30)
  ;
end
```

A set of properties in the declaration affect how the dictionary is processed. A set of dictionary-specific functions provide data and services to your code.

Dictionary properties

Each property-and-value entry is syntactically equivalent to a key-and-value entry, as shown in this example, and the entries can be in any order:

```
row Dictionary
  {
    // properties
    caseSensitive = no,
    ordering = none,

    // fields
    ID        = 5,
    lastName  = "Twain",
    firstName = "Mark"
    age = 30;
  };
```

Your code can neither add nor retrieve a property or its value. In the unlikely event that you wish to use a property name as a key, use the variable name as a qualifier when you specify or refer to the key, as in this example:

```
row Dictionary
  {
    // properties
    caseSensitive = no,
    ordering = none,

    // fields
    row.caseSensitive = "yes"
    row.ordering = 50,
    age = 30
  };
```

Properties are as follows:

caseSensitive
Indicates whether retrieval of a key or the related value is affected by the case of the key with which that value was stored. Options are as follows:

No (the default)
Key access is unaffected by the case of the key, and the following statements are equivalent:

```
age = row.age;
age = row.AGE;
age = row["aGe"];
```

Yes
The following statements can have different results, even though EGL is primarily a case-insensitive language:

```
age = row.age;
age = row.AGE;
age = row["aGe"];
```

The value of the property **caseSensitive** affects the behavior of several of the functions described in a later section.

ordering
Indicates how key-and-value entries are ordered for purpose of retrieval. The value of this property affects the behavior of the functions **getKeys** and **getValues**, as described in a later section.

Options are as follows:

None (the default)
Your code cannot rely on the order of key-and-value entries.

When the value of the property **ordering** is None, the ordering of keys (when the function **getKeys** is invoked) may not be the same as the ordering of values (when the function **getValues** is invoked).

ByInsertion
The key-and-value pairs are available in the order in which they were inserted. Any entries in the declaration are considered to be inserted first, in left-to-right order.

ByKey
The key-and-value pairs are available in key order.

Dictionary functions

To invoke any of the following functions, qualify the function name with the name of the dictionary, as in this example when the dictionary is called *row*:

```
if (row.containsKey(age))
  ;
end
```

containsKey()

```
dictionaryName.containsKey(key String in) returns (Boolean)
```

This function resolves to true or false, depending on whether the input string (*key*) is a key in the dictionary. If the dictionary property **caseSensitive** is set to no, case is not considered; otherwise, the function seeks an exact match, including by case.

containsKey is used only in a logical expression.

getKeys()

```
dictionaryName.getKeys ( ) returns (String[ ])
```

This function returns an array of strings, each of which is a key in the dictionary.

If the dictionary property **caseSensitive** is set to no, each returned key is in lower case; otherwise, each returned key is in the case in which the key was stored.

If the dictionary property **ordering** is set to no, you cannot rely on the order of the returned keys; otherwise, the order is as specified in the description of that property.

getValues()

```
dictionaryName.getValues ( ) returns (ANY[ ])
```

This function returns an array of values of any type. Each value is associated with a key in the dictionary.

insertAll()

```
dictionaryName.insertAll(sourceDictionary Dictionary in)
```

This function acts as if a series of assignment statements copies the key-and-value entries in the source dictionary (*sourceDictionary*) to the *target*, which is the dictionary whose name qualifies the function name.

If a key is in the source and not in the target, the key-and-value entry is copied to the target. If a key is in both the source and target, the value of the source entry overrides the entry in the target. The determination of whether a key is in the target matches one in the source is affected by the value of property **caseSensitive** in each dictionary.

This function is different from the assignment of one dictionary to another because the function **insertAll** retains these entries:

- The property-and-value entries in the target; and
- The key-and-value entries that are in the target but not the source.

removeElement()

```
dictionaryName.removeElement(key String in)
```

This function removes the entry whose input string (*key*) is a key in the dictionary. If the dictionary property **caseSensitive** is set to no, case is not considered; otherwise, the function seeks an exact match, including by case.

removeAll()

```
dictionaryName.removeAll( )
```

This function removes all key-and-value entries in the dictionary, but has no effect on the dictionary's properties.

size()

```
dictionaryName.size( ) returns (INT)
```

Returns an integer that indicates the number of key-and-value entries in the dictionary.

Related concepts
"Parts" on page 17
"References to variables in EGL" on page 55

Related reference
"Logical expressions" on page 484

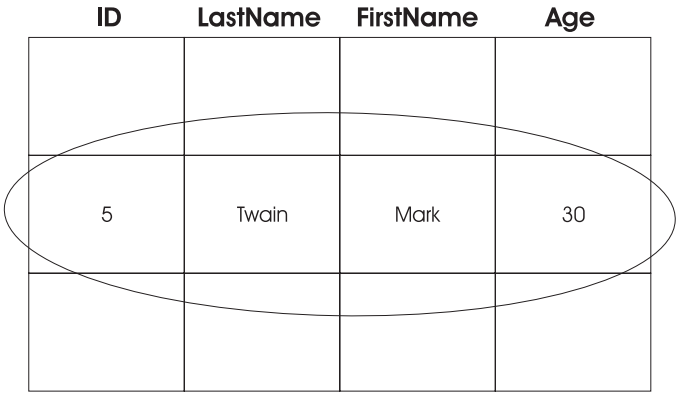
ArrayDictionary

An *arrayDictionary part* is a part that is always available; you do not define it. A variable that is based on an arrayDictionary part lets you access a series of arrays by retrieving the same-numbered element of every array. A set of elements that is retrieved in this way is itself a dictionary, with each of the original array names treated as a key that is paired with the value contained in the array element.

An arrayDictionary is especially useful in relation to the display technology described in *Console user interface*.

The next graphic illustrates an arrayDictionary whose declaration includes arrays that are named *ID*, *lastname*, *firstname*, and *age*. The ellipse encloses a dictionary that includes the following key-and-value entries:

```
ID = 5,
lastName = "Twain",
firstName = "Mark",
age = 30
```



The array of interest is the array of dictionaries, with each dictionary illustrated as being one above the next rather than one alongside the next. The declaration of the arrayDictionary requires an initial list of arrays, however, and those are illustrated as being one alongside the next.

The following code shows the declaration of a list of arrays, followed by the declaration of an arrayDictionary that uses those arrays:

```
ID        INT[4];
lastName  STRING[4];
firstName STRING[4];
age       INT[4];

myRows ArrayDictionary
{
  col1 = ID,
  col2 = lastName,
  col3 = firstName,
  col4 = age
};
```

To retrieve values, your code uses a syntax that isolates a particular dictionary and then a particular field (a key-and-value entry) in that dictionary. You cannot use the arrayDictionary syntax to update a value or to change any characteristic of the arrayDictionary itself.

First, declare a dictionary and assign an arrayDictionary row to that dictionary, as in this example:

```
row Dictionary = myRows[2];
```

Next, declare a variable of the appropriate type and assign an element to that variable, as in either of these examples:

```
cell INT = row["ID"];

cell INT = row.ID;
```

An alternative syntax retrieves the value in one step, as in either of these examples:

```
cell int = myRows[2]["ID"];

cell int = myRows[2].ID;
```

Related concepts
"Console user interface" on page 165
"Dictionary" on page 77
"References to variables in EGL" on page 55

EGL statements

Each EGL function is composed of zero to many EGL statements of the following kinds:

- A *variable declaration* or *constant declaration* provides access to a named area of memory. The value of a variable can be changed at run time; the value of a constant cannot. Either kind of declaration can be anywhere in a function except in a *block*, as described later.
- A *function invocation* directs processing to a function, as in this example:

  ```
  myFunction(myInput);
  ```

  Recursive calls are valid, but only if you are generating for Java.
- An *assignment statement* can copy any of the following values into a variable:
  - Data from a constant or variable
  - A literal
  - A value returned from a function invocation
  - The result of an arithmetic calculation
  - The result of a string concatenation

  Examples of assignment statements are as follows:

  ```
  myItem = 15;
  myItem = readFile(myKeyValue);
  myItem = bigValue - 32;
  record1.message = "Operation " + "successful!";
  ```
- A *keyword statement* provides additional functionality such as file access. Each of these statements is named for the keyword that begins the statement; for example:

  ```
  add record1;     // an add statement
  return (0);      // a return statement
  ```
- A *null statement* is a semicolon that has no effect but may be useful as a placeholder, as in this example:

  ```
  if (myItem == 5)
    ;              // a null statement
  else
    myFunction(myItem);
  end
  ```

Non-null EGL statements have the following characteristics:

- A statement can reference named memory areas, which are of these kinds:
  - Form
  - PageHandler
  - Record
  - DataTable
  - Item (a category that includes data items, as well as structure items in records, forms, and tables)

  - Array (a memory area based on a structure item that has an occurs value greater than 1)
- A statement can include these kinds of expressions--
  - A *datetime expression* resolves to a date, integer, interval, or timestamp
  - A *logical expression* resolves to true or false
  - A *numeric expression* resolves to a number, which may be signed and include a decimal point
  - A *string expression* resolves to a series of characters, which may include single-byte characters, double-byte characters, or a combination of the two
- A statement either ends with a semicolon or with a *block*, which is a series of zero or more subordinate statements that act as a unit. Block-containing statements are terminated with an end delimiter, as in this example:

```
if (record2.status= "Y")
  record1.total = record1.total + 1;
  record1.message = "Operation successful!";
else
  record1.message = "Operation failed!";
end
```

  A semicolon after an end delimiter is not an error but is treated as a null statement.

Names in statements and throughout EGL are case-*in*sensitive; *record1* is identical to *RECORD1*, for example, and both *add* and *ADD* refer to the same keyword.

Note: When you use the source tab in Page Designer, you can manually bind components in a JSP file (specifically, in a JavaServer Faces file) to data areas in a PageHandler. Although EGL is not case sensitive, EGL variable names referenced in the JSP file must have the same case as the EGL variable declaration; and you fail to maintain an exact match, a JavaServer Faces error occurs. It is recommended that you avoid changing the case of an EGL variable after you bind that variable to a JSP field.

System words are a set of words that provide special functionality:

- A *system function* runs code and may return a value; for example:
  - **sysLib.minimum(*arg1*, *arg2*)** returns the minimum of two numbers
  - **strLib.strLen(*arg1*)** returns the length of a character string

  The qualifier (**mathLibstrLib** or **sysLib**) is necessary only if your program has a function of the same name.
- A *system variable* provides a value without invoking a function; for example:
  - **sysVar.errorCode** contains a status code after your program accesses a file and in other situations
  - **sysVar.sqlcode** contains a status code after your program accesses a relational database

  The qualifier **sysVar** is necessary only if your program has a variable of the same name.

A line in a function can have more than one statement. It is recommended that you include no more than one statement per line, however, because you can use the EGL Debugger to set a breakpoint only at the first statement on a line.

See also *Comments*.

Related concepts
“EGL projects, packages, and files” on page 13
“Function part” on page 132
“Parts” on page 17

Related reference
“add” on page 544
“Assignments” on page 352
“call” on page 547
“case” on page 549
“close” on page 551
“Comments” on page 427
“Data initialization” on page 459
“delete” on page 554
“EGL reserved words” on page 474
“execute” on page 557
“Function invocations” on page 504
“get” on page 567
“get next” on page 579
“get previous” on page 584
“if, else” on page 591
“Keywords in alphabetical order”
“Logical expressions” on page 484
“Numeric expressions” on page 491
“open” on page 598
“prepare” on page 611
“replace” on page 613
“set” on page 617
“Text expressions” on page 492
“terminalID” on page 913
“while” on page 629

Keywords in alphabetical order

Keyword	Purpose
“add” on page 544	Places a record in a file, message queue, or database; or places a set of records in a database.
“call” on page 547	Transfers control to another program and optionally passes a series of values. Control returns to the caller when the called program ends. If the called program changes any data that was passed by way of a variable, the storage area available to the caller is changed, too.
“case” on page 549	Marks the start of multiple sets of statements, where at most only one of those sets is run. The **case** statement is equivalent to a C or Java *switch* statement that has a break at the end of each case clause.
“close” on page 551	Disconnects a printer; or closes the file or message queue associated with a given record; or, in the case of an SQL record, closes the cursor that was opened by an EGL **open** or **get** statement.
“continue” on page 553	Presents a text form in a text application.
“converse” on page 554	Presents a text form in a text application.
“delete” on page 554	Removes either a record from a file or a row from a database.

Keyword	Purpose
“display” on page 556	Adds a text form to a run-time buffer but does not present data to the screen.
“execute” on page 557	Lets you write one or more SQL statements; in particular, SQL data-definition statements (of type CREATE TABLE, for example) and data-manipulation statements (of type INSERT or UPDATE, for example).
“exit” on page 560	Leaves the specified block, which by default is the block that immediately contains the **exit** statement.
“for” on page 563	Begins a statement block that runs in a loop for as many times as a test evaluates to true.
“forEach” on page 564	Marks the start of a set of statements that run in a loop. The first iteration occurs only if a specified result set is available and continues (in most cases) until the last row in that result set is processed.
“forward” on page 566	Displays a Web page with variable information. This statement is invoked from a PageHandler.
“freeSQL” on page 567	Frees any resources associated with a dynamically prepared SQL statement, closing any open cursor associated with that SQL statement.
“get” on page 567	Rerieves a single file record or database row and provides an option that lets you replace or delete the stored data later in your code. In addition, this statement allows you to retrieve a set of database rows and place each succeeding row into the next SQL record in a dynamic array. The **get** statement is sometimes identified as **get by key value** and is distinct from the **get by position** statements like **get next**.
“get absolute” on page 573	Reads a numerically specified row in a relational-database result set that was selected by an **open** statement.
“get current” on page 575	Reads the row at which the cursor is already positioned in a database result set that was selected by an **open** statement.
“get first” on page 576	Reads the first row in a database result set that was selected by an **open** statement.
“get last” on page 578	Reads the last row in a database result set that was selected by an **open** statement.
“get next” on page 579	Reads the next record from a file or message queue, or the next row in a database result set.
“get previous” on page 584	Reads the previous record in the file that is associated with a specified EGL indexed record; or reads the previous row in a database result set that was selected by an **open** statement.
“get relative” on page 588	Reads a numerically specified row in a database result set that was selected by an **open** statement. The row is identified in relation to the cursor position in the result set.
“goTo” on page 590	Causes processing to continue at a specified label, which must be in the same function as the statement and outside of a block.
“if, else” on page 591	Marks the start of a set of statements (if any) that run only if a logical expression resolves to true. The optional keyword **else** marks the start of an alternative set of statements (if any) that run only if the logical expression resolves to false. The reserved word **end** marks the close of the **if** statement.
“move” on page 592	Copies data, either byte by byte or by name. The latter operation copies data from the named items in one structure to the same-named items in another.
“open” on page 598	Selects a set of rows from a relational database for later retrieval with **get by position**statements like **get next**. The **open** statement may operate on a cursor or on a called procedure.

Keyword	Purpose
"prepare" on page 611	Specifies an SQL PREPARE statement, which optionally includes details that are known only at run time. You run the prepared SQL statement by running an EGL **execute** statement or (if the SQL statement returns a result set) by running an EGL **open** or **get** statement.
"print" on page 613	Adds a print form to a run-time buffer.
"replace" on page 613	Puts a changed record into a file or database.
"return" on page 616	Exits from a function and optionally returns a value to the invoking function.
"set" on page 617	Has various effects on records, text forms, and items.
"show" on page 626	Presents a text form from a main program along with any other forms buffered using the display statement; ends the current program and optionally forwards the input data from the user and state data from the current program to the program that handles the input from the user.
"transfer" on page 627	Gives control from one main program to another, ends the transferring program, and optionally passes a record whose data is accepted into the receiving program's input record. You cannot use a **transfer** statement in a called program.
"try" on page 628	Indicates that the program continues running if an input/output (I/O) statement, a system-function invocation, or a **call** statement results in an error and is within the **try** statement. If an exception occurs, processing resumes at the first statement in the **onException** block (if any), or at the first statement following the end of the **try** statement. A hard I/O error, however, is handled only if the system variable **VGVar.handleHardIOErrors** is set to 1; otherwise, the program displays a message (if possible) and ends.
"while" on page 629	Marks the start of a set of statements that run in a loop. The first run occurs only if a logical expression resolves to true, and each subsequent iteration depends on the same test. The reserved word **end** marks the close of the **while** statement.

Related reference
"EGL statements" on page 83

Transfer of control across programs

EGL provides several ways to switch control from one program to another:

- The **call** statement gives control to another program and optionally passes a series of values. Control returns to the caller when the called program ends. If the called program changes any data that was passed as a variable, the content of the variable is changed in the caller.

  The call does not commit databases or other recoverable resources, although an automatic server-side commit may occur.

  You may specify characteristics of the call by setting a callLink element of the linkage options part. For details, see *call* and *callLink element*. For details on the automatic server-side commit, see *luwControl in callLink element*.
- The **transfer** statement gives control from one main program to another, ends the transferring program, and optionally passes a record whose data is accepted into the receiving program's *input record*. You cannot use a **transfer** statement in a called program.

Your program can transfer control by a statement of the form *transfer to a transaction* or by a statement of the form *transfer to a program*:

– A transfer to a transaction acts as follows--
  - In a program that runs as a Java main text or main batch program, the behavior depends on the setting of build descriptor option synchOnTrxTransfer--
    • If the value of synchOnTrxTransfer is YES, the transfer statement commits recoverable resources, closes files, closes cursors, and starts a program in the same run unit.
    • If the value of synchOnTrxTransfer is NO (the default), the transfer statement also starts a program in the same run unit, but does not close or commit resources, which are available to the invoked program.
– A *transfer to a program* does not commit or rollback recoverable resources, but closes files, releases locks, and starts a program in the same run unit.

When you are transferring control from EGL-generated Java code, the linkage options part does not affect the characteristics of either kind of transfer. When the transferring program is in COBOL, however, the following statements apply:

– The linkage options part has no effect on transfer to a transaction
– You can set a linkage options part, **transferLink** element to affect the characteristics of transfer to a program

In a PageHandler, a transfer is not valid.

For details, see *transfer*and *transferLink element*.

• The system function **sysLib.startTransaction** starts a run unit asynchronously. The operation does not end the transferring program and does not affect the databases, files, and locks in the transferring program. You have the option to pass data into the *input record*, which is an area in the receiving program.

  If your program invokes **sysLib.startTransaction**, you must generate the program with a linkage options part, **asynchLink** element. For details, see *sysLib.startTransaction* and *asynchLink element*.

• The EGL **show** statement ends the current main program in a text application and shows data to the user by way of a form. After the user submits the form, the **show** statement optionally forwards control to a second main program, which receives data received from the user as well as data that was passed without change from the originating program.

  The **show** statement is affected by the settings in the linkage options part, **transferLink** element.

  For details, see *show*.

• Finally, the **forward** statement is invoked from a PageHandler or from a program that runs in a Java environment. The statement acts as follows:

  1. Commits recoverable resources, closes files, and releases locks
  2. Forwards control
  3. Ends the code

  The target in this case is another program or a Web page. For details, see *forward*.

Related reference

"asynchLink element" on page 355
"call" on page 547
"callLink element" on page 395
"forward" on page 566
"luwControl in callLink element" on page 403

"show" on page 626
"startTransaction()" on page 883
"transfer" on page 627
"transferToProgram element" on page 926

Exception handling

An error may occur when an EGL-generated program acts as follows:

- Accesses a file, queue, or database
- Calls another program
- Invokes a function
- Performs an assignment, comparison, or calculation

try blocks

An EGL *try block* is a series of zero to many EGL statements within the delimiters **try** and **end**. An example is as follows:

```
if (userRequest = "A")
  try
    add record1;
  onException
    myErrorHandler(12);
  end
end
```

In general, a try block allows your program to continue processing even if an error occurs.

The try block may include an *onException clause*, as shown earlier. That clause is invoked if one of the earlier statements in the try block fails; but in the absence of an onException clause, an error in a try block causes invocation of the first statement that immediately follows the try block.

EGL system exceptions

EGL provides a series of system exceptions to indicate the specific nature of a runtime problem. Each of these exceptions is a dictionary from which you can retrieve information, but your retrieval is always by way of the system variable **SysLib.currentException** (also a dictionary), which lets you access the exception thrown most recently in the run unit.

One field in any exception is **code**, which is a string that identifies the exception. You can determine the current exception by testing that field in logic like this:

```
if (userRequest = "A")
  try
    add record1;
  onException
    case (SysLib.currentException.code)
      when (FileIOException)
        myErrorHandler(12);
      otherwise
        myErrorHandler(15);
    end
  end
end
```

In this case, FileIOException is a constant, which is equivalent to the string value ″com.ibm.egl.FileIOException″. The EGL exception constant is always equivalent to the last qualifier in a string that begins ″com.ibm.egl″.

It is strongly recommended that you access the exception fields only in an onException block. The run unit terminates if your code accesses **SysLib.currentException** when no exception has occurred.

The next example accesses the field sqlcode in the exception SQLException:

```
if (userRequest = "A")
  try
    add record01;
  onException
    case (SysLib.currentException.code)
      when ("com.ibm.egl.SQLException")
        if (SysLib.currentException.sqlcode == -270)
          myErrorHandler(16);
        else
          myErrorHandler(20);
        end
      otherwise
        myErrorHandler(15);
    end
  end
end
```

For details on the system exceptions, see *EGL system exceptions*.

Limits of try blocks

The previous details on try blocks must be qualified. First, a try block affects processing only for errors in the following kinds of EGL statements:

- An I/O statement
- A system function
- A call statement

Processing of numeric overflows is not affected by the presence of a try block. For details on those kinds of error, see *VGVar.handleOverflow*.

Second, a try block has no effect on errors inside a user function (or program) that is invoked from within the try block. In the next example, if a statement fails in function myABC, the program ends immediately with an error message unless function myABC itself handles the error:

```
if (userRequest = "B")
  try
    myVariable = myABC();
  onException
    myErrorHandler(12);
  end
end
```

Third, the program ends immediately and with an error message in the following cases:

- An error of a kind that is covered specifically by a try block occurs outside of a try block; or
- One of the following cases applies, even within a try block--
  - A user-written function fails at invocation or on return to the invoker; or

– The system variable **VGVar.handleHardIOErrors** is set to zero when a file or MQSeries I/O statement ends with a hard error (as described later); or

The following cases are also of interest:

- A COBOL run unit ends if a value is divided by zero, although a Java program handles that situation as a numeric overflow
- A COBOL run unit orJava program ends if a non-numeric character is assigned to a numeric variable, although a COBOL program gains some protection if you generate with build descriptor option **spacesZero**

Note: To support the migration of programs written in VisualAge Generator and EGL 5.0, the variable VGVar.handleSysLibraryErrors (previously called ezereply) allows you to process some errors that occur outside of a try block. Avoid use of that variable, which is available only if you are working in VisualAge Generator compatibility mode.

Error-related system variables

EGL provides error-related system variables that are set in a try block either in response to successful events or in response to non-terminating errors. The values in those variables are available in the try block and in code that runs subsequent to the try block, and the values in most cases are restored after a converse, if any.

The EGL run time does not change the value of any error-related variables when statements run outside of a try block. Your program, however, may assign a value to an error-related variable outside of a try block.

The system variable **sysVar.exceptionCode** is given a value in various situations, and in all those situations one or more additional variables are also set, depending on the nature of the program's interaction with the run-time environment:

- The system variables **sysVar.exceptionCode** and **sysVar.errorCode** are both given values after any of the following kinds of statements run in a try block:
  - A **call** statement
  - An I/O statement that operates on an indexed, MQ, relative, or serial file
  - An invocation of almost any system function
- The system variables **sysVar.exceptionCode**, **sysVar.errorCode**, **VGVar.mqConditionCode**, and **sysVar.mqReasonCode** are all given values after an I/O statement in a try block operates on an MQ record
- The system variable **sysVar.exceptionCode** is given a value after a relational database is accessed from a statement in a try block. Values are also assigned to variables in the SQL communication area (SQLCA); for details, see *sysVar.sqlca*.

If a non-terminating error occurs in a try block, the value of **sysVar.exceptionCode** is equivalent to the numeric component of the EGL error message that would be presented to the user if the error occurred outside of the try block. The values of the situation-specific variables like **sysVar.errorCode** and **VGVar.mqConditionCode**, however, are provided by the run-time system. In the absence of an error, the value of **sysVar.exceptionCode** and at least one of the situation-specific variables is the same: a string of eight zeroes.

An error code is assigned to **sysVar.exceptionCode** and **sysVar.errorCode** in the case of a non-terminating numeric overflow, as described in *VGVar.handleOverflow*; but a successful arithmetic calculation does not affect any of the error-related system variables.

Error-related system variables are also not affected by the invocation of a function other than a system function, and **sysVar.errorCode** (the variable affected by most system functions) is not affected by errors in these:

- **sysLib.calculateChkDigitMod10**
- **sysLib.calculateChkDigitMod11**
- **strLib.concatenate**
- **strLib.concatenateWithSeparator**
- **VGLib.connectionService**
- **sysLib.connect**
- **sysLib.convert**
- **sysLib.disconnect**
- **sysLib.disconnectAll**
- **sysLib.purge**
- **sysLib.queryCurrentDatabase**
- **strLib.setBlankTerminator**
- **sysLib.setCurrentDatabase**
- **strLib.strLen**
- **sysLib.verifyChkDigitMod10**
- **sysLib.verifyChkDigitMod11**
- **sysLib.wait**

When an error value is assigned to **sysVar.exceptionCode**, the system variable **sysVar.exceptionMsg** is assigned the text of the related EGL error message, and the system variable **sysVar.exceptionMsgCount** is assigned the number of bytes in the error message, excluding trailing blanks and nulls. When the string of eight zeroes is assigned to **sysVar.exceptionCode**, **sysVar.exceptionMsg** is assigned blanks and **sysVar.exceptionMsgCount** is set to zero.

I/O statements

In relation to I/O statements, an error can be hard or soft:

- A soft error is any of these--
  - No record was found during an I/O operation on an SQL database table
  - One of the following problems occurs in an I/O operation on an indexed, relative, or serial file:
    - Duplicate record (when the external data store allows insertion of a duplicate)
    - No record found
    - End of file
- A hard error is any other problem; for example--
  - Duplicate record (when the external data store prohibits insertion of a duplicate)
  - File not found
  - Communication links are not available during remote access of a data set

If the statement that causes the soft error is in a try block, the following statements apply:

- By default, EGL continues running without passing control to the onException block
- If you wish to pass control to the OnException block, set the property **throwNrfEofExceptions** to *yes* in a program, pageHandler, or library

If a hard I/O error occurs in a try block, the consequence depends on the value of an error-related system variable:

- During access of a file, relational database, or MQSeries message queue, the following rules apply--
  - If **VGVar.handleHardIOErrors** is set to 1, the program continues running
  - If **VGVar.handleHardIOErrors** is set to 0, the program presents an error message, if possible, and ends

  The default setting of that variable is dependent on the value of the property **handleHardIOErrors** , which is available in generatable logic parts like programs, libraries, and pageHandlers. The default value for the property is *yes*, which sets the initial value of the variable **VGVar.handleHardIOErrors** to 1.

If either a hard or soft I/O error occurs outside of a try block, the generated program presents an error message, if possible, and ends.

If you are accessing DB2 directly (not through JDBC), the sqlcode for a hard error is 304, 802, or less than 0.

Error identification

You can determine what kind of error occurred in a try block by including a **case** or **if** statement inside or outside the try block, and in that statement you can test the value of various system variables. If you are responding to an I/O error and if your statement uses an EGL record, however, it is recommended that you use an elementary logical expression. Two formats of the expression are available:

```
recordName is IOerrorValue

recordName not IOerrorValue
```

recordName
Name of the record used in the I/O operation

IOerrorValue
One of several I/O error values that are constant across database management systems

If you don't use the logical expressions with I/O error values and then change database management systems, you may need to modify and regenerate your program. In particular, if you are using JDBC, it is recommended that you use the I/O error values to test for errors rather than the value of **sysVar.sqlcode** or **sysVar.sqlState**or the equivalent values in **sysVar.sqlca**. Those values are dependent on the underlying database implementation when JDBC is in use.

Related concepts
"Compatibility with VisualAge Generator" on page 428
"Dictionary" on page 77

Related reference
"EGL Java runtime error codes" on page 935
"EGL statements" on page 83
"I/O error values" on page 522
"Logical expressions" on page 484
"errorCode" on page 903
"overflowIndicator" on page 906
"sqlca" on page 909
"sqlcode" on page 910
"sqlState" on page 911

"handleSysLibraryErrors" on page 922
"handleHardIOErrors" on page 920
"handleOverflow" on page 921
"mqConditionCode" on page 922

Migrating EGL code to the EGL 6.0 iFix

The EGL V6.0 migration tool converts EGL source from V5.1.2 and V6.0 to comply with the EGL V6.0 iFix. This tool can be used on an entire project, a single file, or a selection of files. Running the tool on a package or folder converts all of the EGL source files in that package or folder. For more information on the code that is changed by the migration tool, see *EGL-to-EGL migration*.

Note: Do not use the migration tool on code that has already been updated to the EGL V6.0 iFix. Doing so can create errors in your code.

To migrate EGL code to the EGL V6.0 iFix, do as follows:

1. In the workbench, click **Window > Preferences**.
2. On the left side of the Preferences window, expand **Workbench** and click **Capabilities**.
3. From the list of capabilities, expand **EGL Developer**.
4. Select the check box for the capability named **EGL V6.0 Migration**.
5. Click **OK**.
6. Again, click **Window > Preferences**.
7. On the left side of the Preferences window, expand **EGL** and click **EGL V6.0 Migration Preferences**.
8. Set the preferences for the EGL V6.0 migration tool. For more information on the preferences in this window, see *Setting EGL-to-EGL migration preferences*.
9. In the Project Explorer view or the Navigator view, select the EGL projects, packages, folders, or files you want to migrate. You can select any number of EGL resources to migrate. To select more than one resource at once, hold CTRL while clicking the resources.
10. Right-click on a selected resource and click **EGL V6.0 Migration > Migrate** from the popup menu.
11. Inspect your code for places that do not comply with the EGL V6.0 iFix.

The migration tool converts the selected EGL source files to comply with the EGL V6.0 iFix. To review the changes that the tool made to the source code, do as follows:

1. In the Project Explorer view or the Navigator view, right-click an EGL source file that has been migrated and click **Compare With > Local History** from the popup menu.
2. Examine the differences between the file in the workspace and the previous version.
3. When you are finished reviewing the changes, click **OK**.

Related concepts
"EGL-to-EGL migration" on page 96
"Setting EGL-to-EGL migration preferences" on page 104

Related tasks
"Enabling EGL capabilities" on page 114

EGL-to-EGL migration

The EGL V6.0 migration tool converts EGL source from V5.1.2 and V6.0 to comply with the EGL V6.0 iFix. This tool can be used on an entire project, a single file, or a selection of files. Running the tool on a package or folder converts all of the EGL source files in that package or folder. For instructions on how to use the migration tool, see *Migrating EGL code to the EGL 6.0 iFix*.

The migration tool can add comments to each file it changes, and it can also add comments to the project's log file. To change these options, see *EGL-to-EGL migration preferences*.

The migration tool changes EGL code in these ways to comply with the EGL V6.0 iFix:

- The migration tool makes changes to the way properties are specified. For information about the changes to properties, see *Changes to properties during EGL-to-EGL migration*.
- The migration tool searches for variables and part names that conflict with reserved words. The migration tool changes those variable and part names by adding a prefix or suffix as defined in the EGL-to-EGL migration preferences. By default, the tool adds the suffix `_EGL` to any name that is now a reserved word. The migration tool does not rename objects of the `CALL` statement, and it does not update references in EGL Build Part files. See *EGL reserved words*. The following is an example of code before and after using the migration tool.

  Before migration:

  ```
  Library Handler
    boolean Bin(4);
  End
  ```

  After migration:

  ```
  Library Handler_EGL
    boolean_EGL Bin(4);
  End
  ```

- The migration tool replaces the single equals sign (=) with the double equals sign (==) when used as a comparison operator. It does not change the single equals sign when used as an assignment operator.

  Before migration:

  ```
  Function test(param int)
    a int;
    If(param = 3)
      a = 0;
    End
  End
  ```

  After migration:

  ```
  Function test(param int)
    a int;
    If(param == 3)
      a = 0;
    End
  End
  ```

- The migration tool adds level numbers to records that do not have level numbers.

  Before migration:

  ```
  Record MyRecord
    item1 int;
    item2 int;
  End
  ```

After migration:

```
Record MyRecord
  10 item1 int;
  10 item2 int;
End
```

- The migration tool changes the declaration syntax of constants.

  Before migration:

  ```
  intConst 3;
  ```

  After migration:

  ```
  const intConst int = 3;
  ```

- The migration tool changes variables and function names that have been moved to different libraries or renamed. This change affects variables and functions from the `SysLib` and `SysVar` libraries.

  Before migration:

  ```
  SysLib.java();
  clearRequestAttr();
  ```

  After migration:

  ```
  JavaLib.invoke();
  J2EELib.clearRequestAttr();
  ```

  Following is a list of changed variables and function names from the SysLib and SysVar libraries:

Table 1. Changed variable and function names from the SysLib and SysVar libraries

Before migration	After migration
SysLib.dateValue	DateTimeLib.dateValue
SysLib.extendTimestampValue	DateTimeLib.extend
SysLib.formatDate	StrLib.formatDate
SysLib.formatTime	StrLib.formatTime
SysLib.formatTimestamp	StrLib.formatTimestamp
SysLib.intervalValue	DateTimeLib.intervalValue
SysLib.timeValue	DateTimeLib.timeValue
SysLib.timeStampValue	DateTimeLib.timestampValue
SysLib.java	JavaLib.invoke
SysLib.javaGetField	JavaLib.getField
SysLib.javaIsNull	JavaLib.isNull
SysLib.javaIsObjID	JavaLib.isObjID
SysLib.javaRemove	JavaLib.remove
SysLib.javaRemoveAll	JavaLib.removeAll
SysLib.javaSetField	JavaLib.setField
SysLib.javaStore	JavaLib.store
SysLib.javaStoreCopy	JavaLib.storeCopy
SysLib.javaStoreField	JavaLib.storeField
SysLib.javaStoreNew	JavaLib.storeNew
SysLib.javaType	JavaLib.qualifiedTypeName
SysLib.clearRequestAttr	J2EELib.clearRequestAttr
SysLib.clearSessionAttr	J2EELib.clearSessionAttr

Table 1. Changed variable and function names from the SysLib and SysVar libraries (continued)

Before migration	After migration
SysLib.getRequestAttr	J2EELib.getRequestAttr
SysLib.getSessionAttr	J2EELib.getSessionAttr
SysLib.setRequestAttr	J2EELib.setRequestAttr
SysLib.setSessionAttr	J2EELib.setSessionAttr
SysLib.displayMsgNum	ConverseLib.displayMsgNum
SysLib.clearScreen	ConverseLib.clearScreen
SysLib.fieldInputLength	ConverseLib.fieldInputLength
SysLib.pageEject	ConverseLib.pageEject
SysLib.validationFailed	ConverseLib.validationFailed
SysLib.getVAGSysType	VGLib.getVAGSysType
SysLib.connectionService	VGLib.connectionService
SysVar.systemGregorianDateFormat	VGVar.systemGregorianDateFormat
SysVar.systemJulianDateFormat	VGVar.systemJulianDateFormat
SysVar.currentDate	VGVar.currentGregorianDate
SysVar.currentFormattedDate	VGVar.currentFormattedGregorianDate
SysVar.currentFormattedJulianDate	VGVar.currentFormattedJulianDate
SysVar.currentFormattedTime	VGVar.currentFormattedTime
SysVar.currentJulianDate	VGVar.currentJulianDate
SysVar.currentShortDate	VGVar.currentShortGregorianDate
SysVar.currentShortJulianDate	VGVar.currentShortJulianDate
SysVar.currentTime	DateTimeLib.currentTime
SysVar.currentTimeStamp	DateTimeLib.currentTimeStamp
SysVar.handleHardIOErrors	VGVar.handleHardIOErrors
SysVar.handleSysLibErrors	VGVar.handleSysLibraryErrors
SysVar.handleOverflow	VGVar.handleOverflow
SysVar.mqConditionCode	VGVar.mqConditionCode
SysVar.sqlerrd	VGVar.sqlerrd
SysVar.sqlerrmc	VGVar.sqlerrmc
SysVar.sqlIsolationLevel	VGVar.sqlIsolationLevel
SysVar.sqlWarn	VGVar.sqlWarn
SysVar.commitOnConverse	ConverseVar.commitOnConverse
SysVar.eventKey	ConverseVar.eventKey
SysVar.printerAssociation	ConverseVar.printerAssociation
SysVar.segmentedMode	ConverseVar.segmentedMode
SysVar.validationMsgNum	ConverseVar.validationMsgNum

- The migration tool changes the way dates, times and timestamps are specified. Following are some examples:

Table 2. Changes to dates, times, and timestamps

Before migration	After migration
dateFormat = ″yy/mm/dd″	dateFormat = ″yy/MM/dd″
dateFormat = ″YYYY/MM/DD″	dateFormat = ″yyyy/MM/dd″
dateFormat = ″YYYY/DDD″	dateFormat = ″yyyy/DDD″
timeFormat = ″hh:mm:ss″	timeFormat = ″HH:mm:ss″

- The migration tool sets the property `HandleHardIOErrors` to `no` for all migrated libraries, programs, and PageHandlers for which that property is not specified.

Related tasks
“Migrating EGL code to the EGL 6.0 iFix” on page 95

Related concepts
“Setting EGL-to-EGL migration preferences” on page 104
“Changes to properties during EGL-to-EGL migration”

Related reference
“EGL reserved words” on page 474

Changes to properties during EGL-to-EGL migration

The migration tool makes significant changes to the way properties are specified. Following is a summary of these changes:

- The migration tool renames properties whose names have changed in the EGL V6.0 iFix. Following is a list of the renamed properties:

Table 3. Renamed properties

Before migration	After migration
action	actionFunction
boolean	isBoolean
getOptions	getOptionsRecord
msgDescriptor	msgDescriptorRecord
onPageLoad	onPageLoadFunction
openOptions	openOptionsRecord
putOptions	putOptionsRecord
queueDescriptor	queueDescriptorRecord
range	validValues
rangeMsgKey	validValuesMsgKey
selectFromList	selectFromListItem
sqlVar	sqlVariableLen
validator	validatorFunction
validatorMsgKey	validatorFunctionMsgKey
validatorTable	validatorDataTable
validatorTableMsgKey	validatorDataTableMsgKey

- The migration tool adds double quotes to property values used as string literals.

  Before migration:

```
{ alias = prog }
```

After migration:

```
{ alias = "prog" }
```

The following properties are affected:

- alias
- column
- currency
- displayName
- fileName
- fillCharacter
- help
- helpKey
- inputRequiredMsgKey
- minimumInputMsgKey
- msgResource
- msgTablePrefix
- pattern
- queueName
- rangeMsgKey
- tableNames
- title
- typeChkMsgKey
- validatorMsgKey
- validatorTableMsgKey
- value
- view

- The migration tool replaces parentheses with square brackets when specifying array literals as values for properties.
  - formSize
  - keyItems
  - outline
  - pageSize
  - position
  - range
  - screenSize
  - screenSizes
  - tableNames
  - tableNameVariables
  - validationBypassFunctions
  - validationBypassKeys
- For properties that take array literals, the migration tool puts single element array literals in brackets to specify that an array with only one element is still an array. The migration tool uses double sets of brackets for properties that take arrays of arrays.

  Before migration:

  ```
  { keyItems = var, screenSizes = (24, 80), range = (1, 9) }
  ```

After migration:

```
{ keyItems = ["var"], screenSizes = [[24, 80]], range = [[1, 9]] }
```

- The migration tool uses the keyword **this** instead of a variable name when overriding properties for a specific element in an array.

  Before migration:

  ```
  Form myForm type TextForm
    fieldArray char(10)[5] { fieldArray[1] {color = red } };
  end
  ```

  After migration:

  ```
  Form myForm type TextForm
    fieldArray char(10)[5] { this[1] {color = red } };
  end
  ```

- The migration tool changes references to parts, functions, and fields, adding quotes and brackets where appropriate.

  Before migration:

  ```
  { keyItems = (item1, item2) }
  ```

  After migration:

  ```
  { keyItems = ["item1", "item2"] }
  ```

  The following properties are affected by the migration tool in this way:

  - action
  - commandValueItem
  - getOptions
  - helpForm
  - inputForm
  - inputPageRecord
  - inputRecord
  - keyItem
  - keyItems
  - lengthItem
  - msgDescriptorRecord
  - msgField
  - numElementsItem
  - onPageLoadFunction
  - openOptionsRecord
  - putOptionsRecord
  - queueDescriptorRecord
  - redefines
  - selectFromListItem
  - tableNameVariables
  - validationBypassFunctions
  - validatorFunction
  - validatorDataTable

- The migration tool assigns a default value of `yes` to any boolean properties that were specified but not assigned a value.

  Before migration:

  ```
  { isReadOnly }
  ```

  After migration:

```
{ isReadOnly = yes}
```

The following properties are affected by the migration tool in this way:

- addSpaceForSOSI
- allowUnqualifiedItemReferences
- boolean
- bypassValidation
- containerContextDependent
- currency
- cursor
- deleteAfterUse
- detectable
- fill
- helpGroup
- includeMsgInTransaction
- includeReferencedFunctions
- initialized
- inputRequired
- isDecimalDigit
- isHexDigit
- isNullable
- isReadOnly
- lowerCase
- masked
- modified
- needsSOSI
- newWindow
- numericSeparator
- openQueueExclusive
- pfKeyEquate
- resident
- runValidatorFromProgram
- segmented
- shared
- sqlVar
- upperCase
- wordWrap
- zeroFormat

• The migration tool splits the **currency** property into two properties: **currency** and **currencySymbol**. The following table gives some examples of how the migration tool changes the **currency** property.

Table 4. Changes to the **currency** *property*

Before migration	After migration
{ currency = yes }	{ currency = yes }
{ currency = no }	{ currency = no }

Table 4. Changes to the **currency** *property (continued)*

Before migration	After migration
{ currency = ″usd″ }	{ currency = yes, currencySymbol = ″usd″ }

- The migration tool changes the values of the **dateFormat** and **timeFormat** properties to be case sensitive. For more information, see *Date, time, and timestamp format specifiers*.
- If the **Add qualifiers to enumeration property values** check box is checked in the preferences menu, the migration tool adds the type of value to the value of the property.

  Before migration:

  ```
  color = red
  outline = box
  ```

  After migration:

  ```
  color = ColorKind.red
  outline = OutlineKind.box
  ```

  This change affects the following properties:

  – align
  – color
  – deviceType
  – displayUse
  – highlight
  – indexOrientation
  – intensity
  – outline
  – protect
  – selectType
  – sign
- The migration tool changes the values of the **tableNames** property to be an array of arrays of strings. Each array of strings must have either one or two elements. The first element is the table name, and the second element, if present, is the table label. The following table gives some examples of how the migration tool changes the **tableNames** property.

Table 5. Changes to the **tableNames** *property*

Before migration	After migration
{ tableNames = (table1, table2) }	{ tableNames = [[″table1″], [″table2″]] }
{ tableNames = (table1 t1, table2) }	{ tableNames = [[″table1″, ″t1″], [″table2″]] }
{ tableNames = (table1 t1, table2 t2) }	{ tableNames = [[″table1″, ″t1″], [″table2″, ″t2″]] }

- The migration tool changes the values of the **tableNameVariables** property in the same way as it changes the values of the **tableNames** property.
- The migration tool changes the values of the **defaultSelectCondition** property to be of type `sqlCondition`.

  Before migration:

```
{ defaultSelectCondition =
  #sql{
    hostVar02 = 4
  }
}
```

After migration:

```
{ defaultSelectCondition =
  #sqlCondition{ // no space between #sqlCondition and the brace
    hostVar02 = 4
  }
}
```

- The migration tool replaces the `NULL` value of the **fillCharacter** to the empty string value `""`.

Related tasks
"Migrating EGL code to the EGL 6.0 iFix" on page 95

Related concepts
"EGL-to-EGL migration" on page 96
"Setting EGL-to-EGL migration preferences"
"Date, time, and timestamp format specifiers" on page 42

Setting EGL-to-EGL migration preferences

You can set preferences that control how the EGL V6.0 migration tool converts EGL source code. For more information about the migration tool, see *EGL-to-EGL migration*. Set the preferences for the migration tool as follows:

1. Click **Window > Preferences**.
2. Expand **EGL**.
3. Click **EGL V6.0 Migration Preferences**.

   Note: If you can not find **EGL V6.0 Migration Preferences**, enable the EGL V6.0 Migration capability. See *Enabling EGL Capabilities*.
4. Choose how to resolve a naming conflict with a new reserved word by clicking a radio button:
   - **Add prefix** sets the migration tool to add a prefix to any words in the source code that are now reserved words. In the text box by this radio button, type the prefix you would like the migration tool to add to the changed word.
   - **Add suffix** sets the migration tool to add a suffix to any words in the source code that are now reserved words. In the text box by this radio button, type the suffix you would like the migration tool to add to the changed word.
5. To add a qualifier to the values of properties that have a finite list of possible values, select the **Add qualifiers to enumeration property values** check box. If this box is checked, the migration tool will add the type of value to the value name.
6. To add comments to the files changed by the migration tool, choose an option under **Logging Level**.
   - **Add comments to all files processed by the migration tool** sets the migration tool to add a comment to each file it processes, even if it makes no changes to that file.
   - **Add comments to all files that are changed by the migration tool** sets the migration tool to add a comment to only the files it changes.
   - **Do not add comments to files** sets the migration tool to not add any comments to the files it processes.

- **Add to beginning of file** sets the migration tool to add comments to the beginning of files.
- **Add to end of file** sets the migration tool to add comments to the end of files.

7. To write a list of the processed files to the file named V60MigrationLog.txt, select the **Append migration results to log file per project** check box.
8. Click **Apply**.
9. Click **OK**.

Related tasks
"Enabling EGL capabilities" on page 114
"Migrating EGL code to the EGL 6.0 iFix" on page 95

Related concepts
"EGL-to-EGL migration" on page 96

Setting up the environment

Setting EGL preferences

Set the basic EGL preferences as follows:

1. Click **Window** > **Preferences**.
2. When a list is displayed, click **EGL** to display the EGL screen.
3. Select or clear the check box for **VisualAge Generator Compatibility**. Your choice affects what options are available at development time, as described in *Compatibility with VisualAge Generator*.
4. In the **Encoding** list box, select the character-encoding set that will be used when you create new EGL build (.eglbld) files. The setting has no effect on existing build files. The default value is UTF-8.
5. In the **User ID** text box, specify the user ID for accessing the remote build machine, if any. The build descriptor option **destUserID** takes precedence, and both that option and the preference value take precedence over the master build descriptor option **destUserID**.
6. In the **Password** text box, specify the password for accessing the remote build machine, if any. The build descriptor option **destPassword** takes precedence, and both that option and the preference value take precedence over the master build descriptor option **destPassword**.
7. Click **Apply**.

To set other EGL preferences, see the list of related tasks at the bottom of this page. When you are finished setting preferences, click **OK**.

Related concepts
"Build" on page 303
"Compatibility with VisualAge Generator" on page 428

Related tasks
"Setting the default build descriptors" on page 109
"Setting preferences for the EGL debugger" on page 108
"Setting preferences for source styles" on page 110
"Setting preferences for SQL database connections" on page 111
"Setting preferences for SQL retrieve" on page 113

Setting preferences for text

To change how text is displayed in the EGL editor, do as follows:

1. Click **Window > Preferences**.
2. When a list of preferences is displayed, expand **Workbench**, then click **Colors and Fonts**. The Colors and Fonts pane is displayed.
3. Expand **EGL** and **Editor**, then click **EGL Editor Text Font**.
4. To choose from a list of fonts and colors, click the **Change** button, then do as follows:
   a. To change the your font preferences, select a font, font style, and size from the scrollable lists.
   b. To change your color preference, select a color from the drop-down list.

c. Select the **Strikeout** check box if you want a line to run through the middle of the text.
d. Select the **Underline** check box if you want a line under the text.
e. You can see a preview of your selections in the Sample box. When you are finished making your selections, click **OK**.

5. To use the default operating system font, click the **Use System Font** button.
6. To use the default workbench font, click the **Reset** button.
7. To set the font for all editors (not just the EGL editor) to the default workbench font, click the **Restore Defaults** button.
8. To save your changes, click **Apply** or (if you are finished setting preferences) click **OK**.

Related tasks
"Setting EGL preferences" on page 107

Setting preferences for the EGL debugger

To set preferences for the EGL debugger, follow these steps:

1. Click **Window** > **Preferences**.
2. When a list is displayed, expand **EGL** and click **Debug**.
3. Clear or select the check box labeled **Prompt for SQL user ID and password when needed**.

   For details on your choice, see *EGL debugger*.
4. Clear or select the check box labeled **Set systemType to DEBUG**.

   For details on your choice, see *EGL debugger*.
5. Set the initial values for sysVar.terminalID, sysVar.sessionID, and sysVar.userID. If you do not specify values, each defaults to your user ID on Windows 2000/NT/XP or Linux™.
6. Set the EGL Debugger Port value. The default is 8345.
7. Select the type of character encoding to use when processing data during a debugging session. The default is the local system's file encoding. For details on your choice, see *Character encoding options for the EGL debugger*.
8. To specify external Java classes for use when the debugger runs, modify the class path. You might need extra classes, for example, to support MQSeries, JDBC drivers, or Java access functions.

   The class path additions are not visible to the WebSphere® Application Server test environment; but you can add to that environment's classpath by working on the Environment tab of the server configuration.

   Use the buttons to the right of the Class Path Order box:

   - To add a project, JAR file, directory, or variable, click the appropriate button: **Add Project**, **Add JARs**, **Add Directory**, or **Add Variable**.
   - To remove an entry, select it and click **Remove**.
   - To move an entry in a list of two or more entries, select the entry and click **Move Up** or **Move Down**.
9. To restore the default settings, click **Restore Defaults**.
10. To save your changes, click **Apply** or (if you are finished setting preferences) click **OK**.

Related concepts
"Compatibility with VisualAge Generator" on page 428

"EGL debugger" on page 261
Character encoding options for the EGL debugger

Related tasks
"Setting preferences for SQL database connections" on page 111

Related reference
"sessionID" on page 909
"terminalID" on page 913
"userID" on page 914

Setting the default build descriptors

For an overview of the default build descriptor and the build descriptor precedence rules, see *Generation in the workbench*.

To specify a preference for build descriptors at the Workbench level, do as follows:

1. Click **Window > Preferences**.
2. When a list is displayed, expand **EGL** and click **Default Build Descriptor**.
3. Select the Debug build descriptor and the Target system build descriptor.
4. Click **Apply**, then click **OK**.

To specify a preference for build descriptors at the file, folder, package, or project level, do as follows:

1. Right-click on the level of interest (for example, on the file or folder name) and, from the context menu, click **Properties**.
2. Select **EGL Default Build Descriptors**.
3. Select the Debug build descriptor and the Target system build descriptor.
4. Click **OK**.

Related concepts
"Generation in the workbench" on page 311

Setting preferences for the EGL editor

To specify the EGL editor preferences, do as follows:

1. Click **Window** > **Preferences**.
2. When a list is displayed, expand **EGL** and click **Editor**.
3. To display line numbers when you review an EGL file, select the **Show line numbers** check box. To clear the line numbers, clear the check box. The file itself is not affected.
4. To show red underlines wherever errors are found in the source code, select the **Annotate errors in text** check box. To clear those underlines, clear the check box. The file itself is not affected.
5. To show a red error indicator in the right margin of the editor (overview ruler) whenever an error is found in the source code, select the **Annotate errors in overview ruler** check box. Clicking on the error indicator will take you to the location of the error in the source code. To clear the error indicator, clear the check box. The file itself is not affected.
6. To specify source styles, follow the process described in *Setting preferences for source styles*.
7. To add, remove, and customize templates for use in content assist, follow the process described in *Setting preferences for templates*.

8. To change how text is displayed, follow the process described in *Setting preferences for text*.

Related tasks
"Setting preferences for source styles"
"Setting preferences for templates"
"Setting preferences for text" on page 107

Related reference
"Content assist in EGL" on page 471

Setting preferences for source styles

You can change how EGL code is displayed in the EGL editor:

1. Click **Window > Preferences**
2. When a list of preferences is displayed, expand **EGL** and **Editor**, then click **Source Styles**.
3. To select the color that you want to appear behind the source type, click the **Custom** radio button in the Background color box. Click the button next to the Custom label. A color palette displays. Select a color, then click **OK**.
4. In the Foreground box, select a type of text, then click the **Color** button. A color palette displays. Select a color, then click **OK**.
5. Select the **Bold** check box if you want to make the type bold.
6. To save your changes, click **Apply** or (if you are finished setting preferences) click **OK**.

Related tasks
"Setting EGL preferences" on page 107

Setting preferences for templates

Do as follows to add, remove, or customize the templates that are displayed when you request content assist in the EGL editor:

1. Click **Window** > **Preferences**.
2. When a list of preferences is displayed, expand **EGL** and **Editor**, then click **Templates**. A list of templates is displayed.

   Note: As in other applications on Windows 2000/NT/XP, you can click an entry to select it; can use **Ctrl-click** to select or unselect an entry without affecting other selections; and can use **Shift-click** to select a set of entries that are contiguous to the entry you last clicked.
3. To make a template available in the EGL editor, select the check box to the left of a template name. To make all the listed templates available, click **Enable All**. Similarly, to make a template unavailable, clear the related check box; and to make all the listed templates unavailable, click **Disable All**.
4. To create a new template, do as follows--
   a. Click **New**
   b. When the New Template dialog is displayed, specify both a name and a description because a template is guaranteed to be displayed in a content-assist list only if the combination of name and description are unique across all templates.

      Note: If the first word used in the template is an EGL keyword (such as Function), the template is available when you request content assist

in the EGL editor, but only when the on-screen cursor is at a place where the word is valid. Similarly, if you type a prefix, then request content assist, all templates beginning with that prefix are available provided the on-screen cursor is in a position where that template is syntactically allowed. For example, type "fun" to request function templates. If you do not type either a prefix or the full first word, you will not see any templates when you request content assist.

c. In the **Pattern** field, type the template itself:
   - Type any text that you wish to display
   - To place a preexisting variable at the on-screen cursor position, click **Insert Variable**, then double-click a variable. When you insert the template in the EGL editor, each of those variables resolves to the appropriate value.
   - To create a custom variable, type a dollar sign ($) followed by a left brace ({), a string, and a right brace (}), as in this example:

     ```
     ${variable}
     ```

     You may find it easier to insert a preexisting variable and change the name for your own use.

     When you insert a custom template in the EGL editor, each variable is underlined to indicate that a value is required.
   - To complete the task, click **OK** and, at the templates screen, click **Apply**.

5. To review an existing template, click on the listed entry and review the Preview box.
6. To edit an existing template, click on the listed entry, then click **Edit**. Interact with the Edit Template dialog as you did with the New Template dialog.
7. To remove an existing template, click on the listed entry, then click **Remove**. To remove multiple templates, use the Windows 2000/NT/XP convention for selecting multiple list entries, then click **Remove**.
8. To import a template from an XML file, click **Import** at the right of the template list and follow the browse mechanism to specify the location of the file.
9. To export a template to an XML file, click **Export** at the right of the template list and follow the browse mechanism to specify the location of the new file. To export multiple templates, use the Windows 2000/NT/XP mechanism for selecting multiple list entries, then click **Export**.
10. To export all the listed templates to an XML file, click **Export All** and follow the browse mechanism to specify the location of the file.
11. To save your changes, click **Apply**. To return to the template list that was in effect at installation time, click **Restore Defaults**.

Related tasks

"Using the EGL templates with content assist" on page 121

Setting preferences for SQL database connections

You use the page for SQL database connections for these reasons:

- You can enable declaration-time and debug-time access to a database that is accessed outside of J2EE.
- Also, you can set a value for the build descriptor option sqlJNDIName, which specifies a name to which the default datasource is bound in the JNDI registry; for example, java:comp/env/jdbc/MyDB. That option is included in the build descriptor that is created for you in the following situation:

– You use the EGL Web Project Wizard, as described in *Creating a project to work with EGL*; and
– When working in that wizard, you request that a build descriptor be created.

Do as follows:

1. Click **Window > Preferences**
2. When a list of preferences is displayed, expand **EGL**, then click **SQL Database Connections**.
3. In the **Connection URL** field, type the URL used to connect to the database through JDBC:
   - For IBM DB2 APP DRIVER for Windows, the URL is `jdbc:db2:`*`dbName`* (where *dbName* is the database name)
   - For the Oracle JDBC thin client-side driver, the URL varies by database location. If the database is local to your machine, the URL is `jdbc:oracle:thin:`*`dbName`* (where *dbName* is the database name). If the database is on a remote server, the URL is `jdbc:oracle:thin:@`*`host`*`:`*`port`*`:`*`dbName`* (where *host* is the host name of the database server, *port* is the port number, and *dbName* is the database name)
   - For the Informix JDBC NET driver, the URL is as follows (with the lines combined into one)--

     ```
     jdbc:informix-sqli://host:port
       /dbName:informixserver=servername;
       user=userName;password=passWord
     ```

     host
     Name of the machine on which the database server resides

     port
     Port number

     dbName
     Database name

     serverName
     Name of the database server

     userName
     Informix user ID

     passWord
     Password associated with the user ID
4. In the **Database** field, type the name of the database.
5. In the **User ID** field, type the user ID for the connection.
6. In the **Password** field, type the password for the user ID.
7. In the **Database vendor type** field, select the database product and version that you are using for your JDBC connection.
8. In the **JDBC driver** field, select the JDBC driver that you are using for your JDBC connection.
9. In the **JDBC driver class** field, type the driver class for the driver you selected. For IBM DB2 APP DRIVER for Windows, the driver class is `COM.ibm.db2.jdbc.app.DB2Driver`; for the Oracle JDBC thin client-side driver, the driver class is `oracle.jdbc.driver.OracleDriver`; and for the Informix JDBC NET driver, the driver class is `com.informix.jdbc.IfxDriver`. For other driver classes, refer to the documentation for the driver.
10. In the **class location** field, type the fully qualified filename of the *.jar or *.zip file that contains the driver class. For IBM DB2 APP DRIVER for Windows,

type the fully qualified filename to the db2java.zip file; for example, `d:\sqllib\java\db2java.zip`. For the Oracle THIN JDBC DRIVER, type the fully qualified pathname to the ojdbc14.jar file; for example, `d:\Ora81\jdbc\lib\ojdbc14.jar` or, if you require Oracle trace, `ojdbc14_g.jar`. For other driver classes, refer to the documentation for the driver.

11. In the **Connection JNDI name** field, specify the database used in J2EE. The value is the name to which the datasource is bound in the JNDI registry; for example, java:comp/env/jdbc/MyDB. As noted earlier, this value is assigned to the option **sqlJNDIName** in the build descriptor that is constructed automatically for a given EGL Web project.
12. If you are accessing DB2 UDB and specify a value in the **Secondary authentication ID** field, the value is used in the SET™ CURRENT SQLID statement used by EGL at validation time. The value is case-sensitive.

You can clear or apply preference settings:

- To restore default values, click **Restore Defaults**.
- To apply preference settings without exiting the preferences dialog, click **Apply**.
- If you are finished setting preferences, click **OK**.

Related tasks
"Creating an EGL Web project" on page 117
"Setting EGL preferences" on page 107

Related reference
"sqlJNDIName" on page 387

Setting preferences for SQL retrieve

At EGL declaration time, you can use the SQL retrieve feature to create an SQL record from the columns of an SQL table. For an overview, see *SQL support*.

To set preferences for the SQL retrieve feature, do as follows:

1. Click **Window > Preferences**, then expand **EGL** and click **SQL Retrieve**
2. Specify rules for creating each structure item that is created by the SQL retrieve feature:
   a. To specify the EGL type to use when creating a structure item from an SQL character data type, click one of the following radio buttons:
      - **Use EGL type string** (the default) maps SQL char data types to EGL string data types
      - **Use EGL type char** maps SQL char data types to EGL char data types
      - **Use EGL type mbChar** maps SQL char data types to EGL mbChar data types
      - **Use EGL type Unicode** maps SQL char data types to EGL Unicode data types
   b. To specify the case of the structure item name, click one of the following radio buttons:
      - **Do not change case** (the default) means that the case of the structure item name is the same as the case of the related table column name
      - **Change to lower case** means that the structure item name is a lower-case version of the table column name
      - **Change to lower case and capitalize first letter after underscore** also means that the structure item name is a lower-case version of the table

column name, except that a letter in the structure item name is rendered in uppercase if, in the table column name, the letter immediately follows an underscore

c. To specify how the underscores in the table column name are reflected in the structure item name, click one of the following radio buttons:
   - **Do not change underscores** (the default) means that underscores in the table column name are included in the structure item name
   - **Remove underscores** means that underscores in the table column name are not included in the structure item name
   - **Change underscores to hyphens** means that underscores in the table column name are rendered as hyphens in the structure item name

3. If you intend to retrieve data from a table that is part of an Informix system schema, clear the check box for **Exclude system schemas**. (In this case, "Informix" is the table owner.) In all other cases, select the check box to improve the performance of the SQL retrieve feature.

   The check box is selected by default.

Related concepts
"SQL support" on page 213

Related tasks
"Retrieving SQL table data" on page 235
"Setting EGL preferences" on page 107
"Setting preferences for SQL database connections" on page 111

Related reference
"Informix and EGL" on page 235

Enabling EGL capabilities

In order to access EGL functionality, the EGL capabilities must be enabled. The following EGL capabilities are available:

EGL Development
Consists of all functionality related to developing and debugging EGL applications.

EGL V6.0 Migration
Consists of all functionality related to converting EGL 5.1.2 and 6.0 source code to comply with the EGL V6.0 iFix.

VisualAge Generator to EGL Migration
Consists of all functionality related to migrating existing VisualAge Generator code to EGL code.

To enable EGL capabilities, do as follows:

1. Click **Window > Preferences**.
2. When a list of preferences is displayed, expand **Workbench**, then click **Capabilities**. The Capabilities pane is displayed.
3. If you would like to receive a prompt when a feature is first used that requires an enabled capability, select the check box for **Prompt when enabling capabilities**.
4. Expand the **EGL Developer** capability folder.

5. Select the check box for the desired EGL capabilities. Alternately, you can select the **EGL Developer** capability folder to enable all of the capabilities that folder contains.
6. To set the list of enabled capabilities back to its state at product install time, click the **Restore Defaults** button.
7. To save your changes, click **Apply**, then click **OK**.

Note: Enabling EGL capabilities will automatically enable any other capabilities that are required to develop and debug EGL applications.

Related tasks
"Setting EGL preferences" on page 107

Beginning code development

Creating a project

Creating an EGL project

For an overview of how to organize your work, see *EGL projects, packages, and files*.

To set up a new EGL project, do as follows:

1. In the Workbench, do either of the following steps:
   - Click **File > New > Project**; or
   - Right-click, then click **New > Project**.

   The New Project wizard opens.
2. Expand **EGL**, then click **EGL Project**. Click **Next**. The **New EGL project** wizard is displayed.

   Note: If EGL Project is not available, check the **Show All Wizards** check box.
3. In the **Project name** field, type a name for the project. By default, the project is placed in your workspace, but you can click **Browse** and choose a different location.
4. In Target Runtime Platform, click the radio button for **Java** or **COBOL**.
5. Select how to specify a build descriptor, which is the part that directs processing at generation time:
   - **Create new project build descriptor(s) automatically** means that EGL provides build descriptors and writes them to a build file (extension .eglbld) that has the same name as the project.

     To specify some of the values in those build descriptors, click **Options**. To change those values later, change the build file that is created for you.

     For further details, see *Specifying database options at project creation*.
   - **Use build descriptor specified in EGL preference** means that EGL points to a build descriptor that you created and identified as an EGL preference.
   - **Select existing build descriptor** allows you to specify a build descriptor from those that are available in your workspace.
6. In most cases, click **Finish**. If you click **Next**, however, you can specify other source folders and projects to reference from the project you are creating. When you have finished selecting other source folders and projects, click **Finish**.

Related concepts
"Build descriptor part" on page 275
"EGL projects, packages, and files" on page 13

Related tasks
"Specifying database options at project creation" on page 118
"Setting preferences for SQL database connections" on page 111

Creating an EGL Web project

For an overview of how to organize your work, see *EGL projects, packages, and files*.

To set up a new EGL Web project, do as follows:

1. In the Workbench, do either of the following steps:
   - Click **File > New > Project**; or
   - Right-click, then click **New > Project**.

   The New Project wizard opens.
2. Expand **EGL**, then click **EGL Web Project**. Click **Next**. The **New EGL Web project** wizard is displayed.
3. In the **Project name** field, type a name for the project. By default, the project is placed in your workspace; but you can click **Browse** and choose a different location.
4. Select how to specify a build descriptor, which is the part that directs processing at generation time:
   - **Create new project build descriptor(s) automatically** means that EGL provides build descriptors and writes them to a build file (extension .eglbld) that has the same name as the project.

     To specify some of the values in those build descriptors, click **Options**. To change those values later, change the build file that is created for you.

     For further details, see *Specifying database options at project creation*.
   - **Use build descriptor specified in EGL preference** means that EGL points to a build descriptor that you created and identified as an EGL preference.
   - **Select existing build descriptor** allows you to specify a build descriptor from those that are available in your workspace.
5. If you requested that a build descriptor be created automatically, you can place a value in the **JNDI Name for SQL Connection** field. The effect is to assign the name to which the default data source is bound in the JNDI registry at debug or generation time. (An example value is java:comp/env/jdbc/MyDB.) Your selection assigns a value to the build descriptor option sqlJNDIName. If the **JNDI Name for SQL Connection** field is already populated, the value was obtained from a Workbench preference, as described in *Setting preferences for SQL database connections*.
6. In most cases, click **Finish**. To do additional customization (as is possible for any Web project), configure the J2EE settings at the bottom of the dialog. Optionally, you can click **Hide Advanced** to conceal the J2EE settings. Click **Next**. Select feature settings, then click **Next**. Select a page template, then click **Finish**.

Related concepts
"Build descriptor part" on page 275
"EGL projects, packages, and files" on page 13

Related tasks
"Specifying database options at project creation"
"Setting preferences for SQL database connections" on page 111

Related reference
"sqlJNDIName" on page 387

Specifying database options at project creation

To assign option values in the build descriptor that is created automatically by EGL, work at the **Project Build Options** dialog. For details on how to display the dialog, see *Creating a project to work with EGL*.

To accept the database-connection information that was specified in preferences, click the check box.

In relation to Java output, the next table shows each on-screen label and the related build descriptor option.

Label	Build descriptor option
Database type	dbms
Database JDBC driver	sqlJDBCDriverClass
Database name	sqlJNDIName (for J2EE output) or sqlDB (for non-J2EE output)

In relation to COBOL output, the next table shows each on-screen label and the related build descriptor option .

Label	Build descriptor option
system	system
output directory	genDirectory
host machine TCP/IP name	destHost
host machine port number	destPort
host userID	destUserID
host password	destPassword
host SQL database	sqlDB
host DB2 userID	sqlID
host DB2 password	sqlPassword

Related concepts
"Build descriptor part" on page 275

Related tasks
"Creating an EGL Web project" on page 117

Related reference
"Build descriptor options" on page 359

Related reference
"Symbolic parameters" on page 392

Creating an EGL source folder

Once you create a project in the workbench, you can create one or more folders within that project to contain your EGL files.

To create a folder for grouping EGL files, do as follows:

1. In the workbench, click **File > New > EGL Source Folder**.
2. Select the project that will contain the EGL folder. In the Folder name field, type the name of the EGL folder, for example myFolder.
3. Click the **Finish** button.

Related concepts
"EGL projects, packages, and files" on page 13
"Introduction to EGL" on page 1

Related tasks
"Creating an EGL Web project" on page 117

Related reference
"Creating an EGL source file"
"Naming conventions" on page 652

Creating an EGL package

An EGL package is a named collection of related source parts. To create an EGL package, do as follows:

1. Identify a project or folder to contain the package. You must create a project or folder if you do not already have one.
2. In the workbench, click **File > New > EGL Package**.
3. Select the project or folder that will contain the EGL package. The Source Folder field may be pre-populated depending on the current selection in the Project Explorer.
4. In the Package Name field, type the name of the EGL package. See *EGL projects, packages, and files* for details on package naming conventions.
5. Click the **Finish** button.

Related concepts
"EGL projects, packages, and files" on page 13
"Introduction to EGL" on page 1

Related tasks
"Creating an EGL source folder" on page 119
"Creating an EGL Web project" on page 117

Related reference
"Creating an EGL source file"

Creating an EGL source file

To create an EGL source file, do as follows:

1. Identify a project or folder to contain the file. You must create a project or folder if you do not already have one.
2. In the workbench, click **File > New > EGL Source File**.
3. Select the project or folder that will contain the EGL file. Select the package that will contain the EGL file. In the EGL Source File Name field, type the name of the EGL file, for example myEGLFile.
4. Click **Finish** to create the file. An extension (.egl) is automatically appended to the end of the file name. The EGL file appears in the Project Explorer view and automatically opens in the default EGL editor.

Related concepts
"EGL projects, packages, and files" on page 13
"Introduction to EGL" on page 1

Related tasks
"Creating an EGL source folder" on page 119
"Creating an EGL Web project" on page 117

Using the EGL templates with content assist

To practice using content assist, do as follows:

1. Open a new EGL file.
2. On an available line, type **P** (for PageHandler or program) and press **Ctrl + Space**.
3. When a pop-up is displayed, click an icon for the part to customize. Do either of these steps:
   - Press **Enter** to select the first icon in the list; or
   - Use the arrow keys to select another icon (for a program) and press **Enter**.

   The editor places a part template in your file.
4. Customize the part.

   When the template is displayed, the editor highlights the first area where you need to type information; in this case, specify the part name. After you type, press **Tab** to highlight the next area where you need to type.

   You can use the **Tab** key repeatedly, and this use of the key is available until you reach the end of the file or until you change your in-file position in any other way.
5. To insert a function into your program or PageHandler, type **F** (for function), then press **Ctrl + Space**. Although you can select a part template again, do as follows
   - Use the arrow keys or your mouse to scroll to the end of the list
   - Press **Enter** or click the word *Function*; note that the absence of an icon means that you are selecting a string rather than a part template

   The ability to select a string is more useful in other contexts, such as when you want to type a variable name quickly.
6. With the cursor at the end of the word *Function*, press **Ctrl + Space** and click an icon from the list.

   The editor places the function template in your file.
7. Customize the part.
8. As you develop your code, periodically press **Ctrl + Space** to understand the range of services that are provided.

Related tasks
"Inserting code snippets into EGL and JSP files" on page 139
"Setting preferences for templates" on page 110

Related reference
"Function part in EGL source format" on page 513
"PageHandler part in EGL source format" on page 659
"Program part in EGL source format" on page 707

Keyboard shortcuts for EGL

The next table shows the keyboard shortcuts that are available in the EGL editor.

Key combination	Function
Ctrl+/	Comment
Ctrl+\	Uncomment
Ctrl+A	Select all
Ctrl+C	Copy
Ctrl+F	Find
Ctrl+H	Search
Ctrl+K	Find next
Ctrl+S	Save
Ctrl+V	Paste
Ctrl+X	Cut
Ctrl+G	Generate
Ctrl+L	Go to a specific line
Ctrl+Y	Redo
Ctrl+Z	Undo
Ctrl+Shift+A	Add an explicit SQL statement to an EGL I/O statement that has an implicit one
Ctrl+Shift+K	Find previous
Ctrl+Shift+N	Access the Open Part dialog
Ctrl+Shift+P	Construct an EGL **prepare** statement and the related **get**, **execute**, or **open** statement
Ctrl+Shift+R	Use the retrieve feature to create or overwrite items in an SQL record part
Ctrl+Shift+S	Show the current file in Project Explorer
Ctrl+Shift+V	View and validate the SQL statement that is associated with an EGL I/O statement and perform related actions
Ctrl+Space	Get content assist
F3	Open the file that contains the part whose name is highlighted
Tab	Indents text to the next tab stop

Developing basic EGL source code

Creating an EGL dataItem part

An EGL dataItem part defines an area of memory that cannot be divided. EGL dataItem parts are contained in EGL files. To create an EGL dataItem part, do as follows:

1. Identify an EGL file to contain the dataItem part and open that file in the EGL editor. You must create an EGL file if you do not already have one.
2. Type the specifics of the dataItem part according to EGL syntax (for details, see *DataItem part in EGL source format*). You can use content assist to place an outline of the dataItem part syntax in the file.
3. Save the EGL file.

Related concepts
"DataItem part"
"EGL projects, packages, and files" on page 13

Related tasks
"Creating an EGL source file" on page 120
"Using the EGL templates with content assist" on page 121

Related reference
"Content assist in EGL" on page 471
"DataItem part in EGL source format" on page 461
"Naming conventions" on page 652

DataItem part

A *dataItem part* defines an area of memory that cannot be subdivided. A dataItem part is a standalone part, unlike a structure field in a fixed structure.

A *primitive variable* is a memory area that is based on a dataItem part or on a primitive declaration such as INT or CHAR(2). You may use a primitive variable in these ways:

- As a parameter that receives data into a function or program
- As a variable in an EGL function; for instance, in an assignment statement or as an argument that passes data to another function or program

Each primitive variable has a series of properties, whether by default or as specified in either the variable or the dataItem part. For details, see *Overview of EGL properties and overrides*.

Related concepts
"Fixed record parts" on page 125
"Fixed structure" on page 24
"Overview of EGL properties" on page 60
"Parts" on page 17
"Record parts" on page 124
"Typedef" on page 25

Related tasks
"Setting preferences for templates" on page 110

Related reference
"DataItem part in EGL source format" on page 461
"EGL source format" on page 478
"Data initialization" on page 459
"Primitive types" on page 31

Creating an EGL record part

A record part defines a structure (a hierarchical layout of fixed-size data elements in storage) and an optional binding, which is a relationship of the record to an external data source (file, database, or message queue). EGL record parts are contained in EGL files. To create an EGL record part, do as follows:

1. Identify an EGL file to contain the record part and open that file in the EGL editor. You must create an EGL file if you do not already have one.
2. Type the specifics of the record part according to EGL syntax (for details, see *Basic record part in EGL source format*, *Indexed record part in EGL source format*, *MQ record part in EGL source format*, *Relative record part in EGL source format*, *Serial record part in EGL source format*, and *SQL record part in EGL source format*). You can use content assist to place an outline of the record part syntax in the file.
3. Save the EGL file.

Related concepts
"EGL projects, packages, and files" on page 13
"Record parts"

Related tasks
"Creating an EGL source file" on page 120
"Using the EGL templates with content assist" on page 121

Related reference
"Basic record part in EGL source format" on page 357
"Content assist in EGL" on page 471
"Indexed record part in EGL source format" on page 520
"MQ record part in EGL source format" on page 642
"Naming conventions" on page 652
"Relative record part in EGL source format" on page 719
"Serial record part in EGL source format" on page 722
"SQL record part in EGL source format" on page 726

Record parts

A *record part* defines a sequence of data whose length is not necessarily known at generation time and whose content is composed of fields. In EGL, a field defines a variable in any record that is based on the record part.

A field can be a dictionary, arrayDictionary, or an array of dictionaries or arrayDictionaries; or can be based on any of the following:

- A primitive type such as STRING
- A DataItem part
- A fixed-record part (as described later)
- Another record part

- An array of any of the preceding kinds

Two types of record parts are available:
- basicRecord, as is used for general processing but not for accessing a data store
- SQLRecord, as is used for accessing a relational database

You may use a *record* in the following contexts:
- In a statement that copies data to or from a relational database
- In an assignment or move statement
- As an argument that passes data to another program or function
- As a parameter that receives data into a program or function

A record part is distinct from a fixed record part, which defines a sequence of data whose length *is* known at generation time. A fixed record part is used primarily for accessing VSAM files, MQSeries messages queues, and other sequential files.

A record part that includes level numbers is a fixed record part, even if the record part is of type basicRecord or SQLRecord. For other details, see *Fixed record parts*.

Related concepts
"DataItem part" on page 123
"Fixed record parts"
"Parts" on page 17
"Record types and properties" on page 126
"Resource associations and file types" on page 286
"Fixed structure" on page 24
"Typedef" on page 25

Related tasks
"Setting the default build descriptors" on page 109
"Setting preferences for the EGL editor" on page 109

Related reference
"EGL source format" on page 478
"Data initialization" on page 459
"Primitive types" on page 31

Fixed record parts

A fixed record part defines a sequence of data whose length is known at generation time. This kind of part is necessarily composed of a series of primitive, fixed-length fields, and each field can be substructured. A field that specifies a telephone number, for example, can be defined as follows:

```
10 phoneNumber    CHAR(10);
   20 areaCode    CHAR(3);
   20 localNumber CHAR(7);
```

Although you can use fixed records (which are variables) for any kind of processing, their best use is for I/O operations on VSAM files, MQSeries messages queues, and other sequential files. Although you can use fixed records for accessing relational databases or for general processing (as was the case with earlier products such as VisualAge Generator), you should avoid using fixed records for those purposes in new development.

A record part of any of the following types is a fixed record part:

- indexedRecord
- mqRecord
- relationalRecord
- serialRecord

In addition, a record part of any of the following types is a fixed record part if each field is preceded by a level number:

- basicRecord
- SQLRecord

You may use a fixed record in the following contexts:

- In a statement that copies data to or from a data source
- In an assignment or move statement
- As an argument that passes data to another program or function
- As a parameter that receives data into a program or function

Any relationship of a fixed record part to an external data source is determined by the type of the fixed record part and by a set of type-specific properties such as fileName. A record based on a part of type indexedRecord, for example, is used for accessing a VSAM Key Sequenced Data Set. The relationship of a record part to a data source determines the operations that are generated when the fixed record is used in an EGL I/O statement such as **add**.

A fixed-record field can be based on another fixed record part; and in assignment statements, that field is treated as a memory area of type CHAR regardless of the types in the fixed record part.

Related concepts
"DataItem part" on page 123
"Record parts" on page 124
"Record types and properties"
"Resource associations and file types" on page 286
"Fixed structure" on page 24
"Typedef" on page 25

Related tasks
"Setting the default build descriptors" on page 109
"Setting preferences for the EGL editor" on page 109

Related reference
"Assignments" on page 352
"EGL source format" on page 478
"Data initialization" on page 459
"Primitive types" on page 31

Record types and properties

Several EGL record types are available:

- basicRecord
- indexedRecord
- mqRecord
- relativeRecord
- serialRecord

- SQLRecord

For details on what target systems support what record types, see *Record and file-type cross-reference*. For details on how record parts are initialized, see *Data initialization*.

basicRecord

A basic record or fixed basic record is used for internal processing and cannot access data storage.

The part is a record part by default; but is a fixed record part if the field definitions are preceded with level numbers.

In a fixed record part of type basicRecord, the property **redefines** is available. If set, that property identifies a declared record, and any record based on the fixed record part will access the run-time memory of the declared record.

In a main program, the program property **inputRecord** identifies a record (or fixed record) that is initialized automatically, as described in *Data initialization*.

indexedRecord

An indexed record is a fixed record that lets you to work with a file that is accessed by a *key value*, which identifies the logical position of a record in the file. You can read the file by invoking a **get**, **get next**, or **get previous** statement. Also, you can write to the file by invoking an **add** or **replace** statement; and you can remove a record from the file by invoking a **delete** statement.

The properties of a part of type indexedRecord include these:

- **fileName** is required. For details on the meaning of your input, see *Resource associations (overview)*. For details on the valid characters, see *Naming conventions*.
- **keyItem** is required and can only be a structure field that is unique in the same record. You must use an unqualified reference to specify the key field; for example, use *myItem* rather than *myRecord.myItem*. (In an EGL statement, however, you can reference the key field as you would reference any field.)

See also *Properties that support variable-length records*.

mqRecord

An MQ record is a fixed record that lets you access an MQSeries message queue. For details, see *MQSeries support*.

relativeRecord

A relative record is a fixed record that lets you work with a data set whose records have these properties:

- Are fixed length
- Can be accessed by an integer that represents the sequential position of the record in the file

The properties of a part of type relativeRecord are as follows:

- **fileName** is required. For details on the meaning of your input, see *Resource associations (overview)*. For details on the valid characters, see *Naming conventions*.
- **keyItem** is required. The key field can be any of these areas of memory:
  - A structure field in the same record

– A structure field in a record that is global to the program or is local to the function that accesses the record
– A primitive variable that is global to the program or is local to the function that accesses the record

You must use an unqualified reference to specify the key field. For example, use *myItem* rather than *myRecord.myItem*. (In an EGL statement, you can reference the key field as you would reference any field.) The key field must be unique in the local scope of the function that accesses the record or must be absent from local scope and unique in global scope.

The key field has these characteristics:

– Has a primitive type of BIN, DECIMAL, INT, or NUM
– Contains no decimal places
– Allows for 9 digits at most

Only the **get** and **add** statements use the key field, but the key field must be available to any function that uses the record for file access.

serialRecord

A serial record is a fixed record that lets you access a sequentially accessed file or data set. You can read from the file by invoking a **get** statement, and a series of **get next** statements reads the file records sequentially, from the first to the last. You can write to the file by invoking an **add** statement, which places a new record at the end of the file.

Serial record properties include **fileName**, which is required. For details on the meaning of your input for that property, see *Resource associations (overview)*. For details on the valid characters, see *Naming conventions*.

See also *Properties that support variable-length records*.

sqlRecord

An SQL record is a record (or a fixed record) that provides special services when you access a relational database.

The part is a record part by default; but is a fixed record part if the field definitions are preceded with level numbers.

Each part has the following properties:

- An entry in **tableNames** identifies an SQL table associated with the part. You may reference multiple tables in a join, but restrictions ensure that you do not write to multiple tables with a single EGL statement. You may associate a given table name with a *label*, which is an optional, short name used to reference the table in an SQL statement.
- **defaultSelectCondition** is optional. The conditions become part of the WHERE clause in the default SQL statements. The WHERE clause is meaningful when an SQL record is used in an EGL **open** or **get** statement or in the statements like **get next** or **get previous**.

  In most cases, the SQL default select condition supplements a second condition, which is based on an association between the key-field values in the SQL record and the key columns of the SQL table.
- **tableNameVariables** is optional. You can specify one or more variables whose content at run time determines what database tables to access, as described in *Dynamic SQL*.

- **keyItems** is optional. Each key field can only be a structure field that is unique in the same record. You must use an unqualified reference to specify each of those fields; for example, use *myItem* rather than *myRecord.myItem*. (In an EGL statement, however, you can reference a key field as you would reference any field.)

For details, see *SQL support*.

Related concepts
"Fixed record parts" on page 125
"Dynamic SQL" on page 224
"MQSeries support" on page 247
"Record parts" on page 124
"Resource associations and file types" on page 286
"SQL support" on page 213

Related reference
"add" on page 544
"close" on page 551

"Data initialization" on page 459
"delete" on page 554
"execute" on page 557
"get" on page 567
"get next" on page 579
"get previous" on page 584
"MQ record properties" on page 644
"Naming conventions" on page 652
"open" on page 598
"prepare" on page 611
"Properties that support variable-length records" on page 716
"Record and file type cross-reference" on page 716
"replace" on page 613
"SQL item properties" on page 63
"terminalID" on page 913

Creating an EGL program part

An EGL program part is the main logical unit used to generate a COBOL program, a Java program, a Java wrapper, or an Enterprise JavaBean session bean. For more information, see *Program part*.

A program part is automatically added to a program file and named appropriately when you create the file in the workbench. Program file specifications allow only one program part per file, and require a program name that matches the file name.

To create a program file with a program part, do as follows:

1. Identify a project or folder to contain the file. You must create a project or folder if you do not already have one.
2. In the workbench, click **File > New > Program**.
3. Select the project or folder that will contain the EGL file, then select a package. Since the program name will be identical to the file name, choose a file name that adheres to EGL part name conventions. In the EGL Source File Name field, type the name of the EGL file, for example myEGLprg. Select an EGL program type (for details, see *Basic program in EGL source format* or *TextUI program in EGL*

source format). If the program part is a main program, click to remove the check mark from Create as called program.

4. Click the **Finish** button.

Related concepts
"EGL projects, packages, and files" on page 13
"Introduction to EGL" on page 1
"Program part"

Related tasks
"Creating an EGL source folder" on page 119

Related reference

"Basic program in EGL source format" on page 708
"Creating an EGL source file" on page 120
"Naming conventions" on page 652
"Text UI program in EGL source format" on page 710

Program part

A *program part* defines the central logical unit in a run-time COBOL or Java program. For an overview of main and called programs and of the program types (basic and textUI), see *Parts*.

Any kind of program part includes a function called *main*, which represents the logic that runs at program start up. A program can include other functions and can access functions that are outside of the program. The function *main* can invoke those other functions, and any function can give control to other programs.

The most important program properties are as follows:

- Each *parameter* references an area of memory that contains data received from a caller. Parameters are global to the program and are valid only in called programs.
- Each *variable* references an area of memory that is allocated in and global to the program.
- A *form group* is a collection of forms that present data to the user:
  - A basic program can present data to a printer by way of *print forms*
  - A text program can present data interactively (by way of *text forms*) or to a printer

  For details, see *FormGroup part*.
- An *input record* is an area of global memory that receives data when control is transferred asynchronously from another program. An input record is available only in a main program.
- In main text programs, the *segmented* property determines what actions are taken automatically before the program issues a **converse** statement to present a text form. For details, see *Segmentation*.
- Also in text programs, an *input form* has one of two purposes at program start up:
  - The form is presented to a user who invokes the program from a monitor or terminal
  - Alternatively, data that was entered by a user is received into the input form, which is a memory area in the program itself. This situation applies only in

the case of a *deferred program switch*, which is a two-step transfer of control that is caused by a variant of the **show** statement--

1. A program submits a text form to the user, then terminates
2. The user submits the form, and by virtue of information in the form, the submission automatically invokes a second program, which contains the input form

For a complete list of program properties, see *Program part properties*.

Related concepts
"FormGroup part" on page 143
"Function part" on page 132
"Parts" on page 17
"References to variables in EGL" on page 55
"Segmentation in text applications" on page 149

Related tasks
"Creating an EGL program part" on page 129

Related reference
"Content assist in EGL" on page 471
"Data initialization" on page 459
"EGL source format" on page 478
"EGL statements" on page 83
"Program part in EGL source format" on page 707
"Program part properties" on page 713

Creating an EGL function part

A function part is a logical unit that either contains the first code in a program or is invoked from another function. EGL function parts are contained in EGL files. To create an EGL function part, do as follows:

1. Identify an EGL file to contain the function part and open that file in the EGL editor. You must create an EGL file if you do not already have one.
2. Type the specifics of the function part according to EGL syntax (for details, see *Function part in EGL source format*). You can use content assist to place an outline of the function part syntax in the file.
3. Save the EGL file.

Related concepts
"EGL projects, packages, and files" on page 13
"Function part" on page 132

Related tasks
"Creating an EGL source file" on page 120
"Using the EGL templates with content assist" on page 121

Related reference
"Content assist in EGL" on page 471
"Function invocations" on page 504
"Function part in EGL source format" on page 513
"Naming conventions" on page 652

Function part

A *function part* is a logical unit that either contains the first code in the program or is invoked from another function. The function that contains the first code in the program is called *main*.

The function part can include the following properties:

- A *return value*, which describes the data that the function part returns to the caller
- A set of *parameters*, each of which references memory that is allocated and passed by another logic part
- A set of other *variables*, each of which allocates other memory that is local to the function
- EGL statements
- A specification as to whether the function requires the program context, as described in *containerContextDependent*

The function *main* is unusual in that it cannot return a value or include parameters, and it must be declared inside a program part.

Related concepts
"Parts" on page 17
"Program part" on page 130
"References to variables in EGL" on page 55
"SQL support" on page 213

Related reference
"containerContextDependent" on page 453
"Data initialization" on page 459
"EGL source format" on page 478
"Function invocations" on page 504
"Function part in EGL source format" on page 513
"EGL statements" on page 83

Creating an EGL library part

An EGL library part contains a set of functions, variables, and constants that can be used by programs, PageHandlers, or other libraries. To create an EGL library part, do as follows:

1. Identify a project or folder to contain the file. You must create a project or folder if you do not already have one.
2. In the workbench, click **File > New > Library**.
3. Select the project or folder that will contain the EGL file, then select a package. Since the library name will be identical to the file name, choose a file name that adheres to EGL part name conventions. In the EGL Source File Name field, type the name of the EGL file, for example myLibrary.
4. Select the type of library by clicking one of the following radio buttons:
   - **Basic - Create a basic library**
   - **Native - Create a native library**
5. Click the **Finish** button.

Related concepts
"EGL projects, packages, and files" on page 13

"Introduction to EGL" on page 1
"Library part of type basicLibrary"
"Library part of type basicLibrary"

Related tasks
"Creating an EGL source folder" on page 119
"Creating an EGL Web project" on page 117

Related reference
"Creating an EGL source file" on page 120
"Library part in EGL source format" on page 630
"Naming conventions" on page 652

Library part of type basicLibrary

A *library part* of type basicLibrary contains a set of functions, variables, and constants that can be used by programs, PageHandlers, or other libraries. It is recommended that you use libraries to maximize your reuse of common code and values.

The type specification *basicLibrary* indicates that the part is generated into a compilable unit and includes EGL values and code for local execution. This type is the default when the keyword **type** is not specified. For details on creating a library to access a native DLL from an EGL-generated Java program, see *Library part of type nativeLibrary*.

Rules for a library of type *basicLibrary* are as follows:

- You can reference a library's functions, variables, and constants without specifying the library name, but only if you include the library in a program-specific Use declaration.
- Library functions can access any system variables that are associated with the invoking program or PageHandler. The following rules apply:
  - When a function in a library receives a record as an argument, the record cannot be used for input or output (I/O) or for testing an I/O state such as endOfFile. The code that invokes the library, however, can use the record in either way.
  - When you declare a record in a library, the library-based functions can use the record for input or output (I/O) and for testing the I/O state (for end of file, for example). The code that invokes the library, however, cannot use the record in either way.
- Library functions can use any statements except these:
  - converse
  - forward
  - show
  - transfer
- A library cannot access a text form.
- A library that accesses a print form must include a use declaration for the related form group.
- You can use the modifier **private** on a function, variable, or constant declaration to keep the element from being used outside the library.
- Library functions that are declared as public (as is the default) are available outside the library and cannot have parameters that are of a loose type, which is

a special kind of primitive type that is available only if you wish the parameter to accept a range of argument lengths. For details on the loose type, see *Function part in EGL source format*.

The library is generated separately from the parts that use it. EGL run time accesses the library part by using the setting of the library property **alias**, which defaults to the EGL library name.

At run time, the library is loaded when first used and is unloaded when the program or PageHandler that accessed the library leaves memory, as occurs when the run unit ends..

A PageHandler gets a new copy of the library whenever the PageHandler is loaded. Also, a library that is invoked by another library remains in memory as long as the invoking library does.

In EGL-generated Java code, a library that is used only for its constants is not loaded at run time because constants are generated as literals in the programs and PageHandlers that reference them.

Related concepts
"forward" on page 566
"Function part in EGL source format" on page 513
"Library part in EGL source format" on page 630
"Library part of type basicLibrary" on page 133
"Run unit" on page 721
"Segmentation in text applications" on page 149
"show" on page 626
"transfer" on page 627
"Use declaration" on page 930

Related reference
"converse" on page 554
"forward" on page 566
"Function part in EGL source format" on page 513
"Library part in EGL source format" on page 630
"Run unit" on page 721
"Segmentation in text applications" on page 149
"show" on page 626
"transfer" on page 627
"Use declaration" on page 930

Library part of type nativeLibrary

A library of type nativeLibrary enables your EGL-generated Java code to invoke a single, locally running DLL. The code for that DLL is not written in the EGL language. For information on developing a basic library, which contains shared functions and values that *are* written in the EGL language, see *Library part of type basicLibrary*.

In a library of type nativeLibrary, the purpose of each function is to provide an interface to a DLL function. You cannot define statements in the EGL function, and you cannot declare variables or constants anywhere in the library.

EGL run time accesses a DLL-based function by using the setting of the EGL function property **alias**, which defaults to the EGL function name. Set that property

explicitly if the name of the DLL-based function does not conform to the conventions described in *Naming conventions*.

The library property **callingConvention** specifies how the EGL run time passes data between the two kinds of code:

- The EGL code that invokes the library function; and
- The function in the DLL.

The only value now available for **callingConvention** is *I4GL*:

- Data is passed in accordance with the Informix stack format. Each input parameter is placed on an input stack, and each output parameter is placed on an output stack.
- You cannot pass arguments such as records or dictionaries. Only these are valid:
  - Primitive variables, including variables of type ANY
  - Fields that are in dataTables, print forms, text forms, and fixed records, but only if the field lacks a substructure

  The parameters in the library functions must be primitive variables and may be of type ANY, but cannot be of a loose type and cannot include the **field** modifier.

The library property **dllName** specifies the DLL name, which is final; it cannot be overridden at deployment time. If you do not specify a value for the library property **dllName**, you must specify the DLL name in the Java runtime property vgj.defaultI4GLNativeLibrary. Only one such Java runtime property is available for a run unit, so only one DLL can be specified, aside from DLLs that are identified in the EGL libraries.

Whether you specify the DLL name at development time (in **dllName**) or at deployment time (in vgj.defaultI4GLNativeLibrary), the DLL must reside in the directory path identified in a runtime variable; that variable is either PATH (on Windows 2000/NT/XP) or LIBPATH (on UNIX platforms).

Library functions are automatically declared as public to ensure that they are available outside the library. In your other EGL code, you can reference a function by its function-alias name alone, without specifying the library name, but only if you include the library in a program-specific Use declaration.

The EGL library is generated as a Java class that is separate from the code that accesses the library and from the DLL. EGL run time accesses that class by using the setting of the library property **alias**, which defaults to the EGL library name. Set that property explicitly if the name of the library part does not conform to Java conventions.

At run time, a DLL is loaded when first used and is unloaded when the accessing program or PageHandler leaves memory, as occurs when the run unit ends.

A PageHandler gets a new copy of the DLL whenever the PageHandler is loaded. Also, a DLL that is invoked by an EGL library of type basicLibrary remains in memory as long as the invoking library does.

The following native library provides access to a DLL written in C:

```
Library myLibrary type nativeLibrary
{callingConvention="I4GL", dllname="mydll"}

  Function entryPoint1( p1 int nullable in,
```

```
                        p2 date in, p3 time in,
                        p4 interval in, p5 any out)
    end

    Function entryPoint2( p1 float in,
                        p2 String in,
                        p3 smallint out)
    end

    Function entryPoint3( p1 any in,
                        p2 any in,
                        p3 any out,
                        p4 CLOB inout)
    end
end
```

Related concepts
"Java runtime properties" on page 327
"Library part of type basicLibrary" on page 133

Related reference
"Function part in EGL source format" on page 513
"Java runtime properties (details)" on page 525
"Library part in EGL source format" on page 630
"Naming conventions" on page 652
"Run unit" on page 721
"Use declaration" on page 930

Creating an EGL dataTable part

An EGL dataTable part associates a data structure with an array of initial values for the structure. To create an EGL dataTable part, do as follows:

1. Identify a project or folder to contain the file. You must create a project or folder if you do not already have one.
2. In the workbench, click **File > New > Data Table**.
3. Select the project or folder that will contain the EGL file, then select a package. Since the dataTable name will be identical to the file name, choose a file name that adheres to EGL part name conventions. In the EGL Source File Name field, type the name of the EGL file, for example myDataTable. Select a dataTable sub-type (for details, see *Data Table part in EGL source format*).
4. Click the **Finish** button.

Related concepts
"DataTable" on page 137
"EGL projects, packages, and files" on page 13
"Introduction to EGL" on page 1

Related tasks
"Creating an EGL source folder" on page 119
"Creating an EGL Web project" on page 117

Related reference
"Creating an EGL source file" on page 120
"DataTable part in EGL source format" on page 462
"Naming conventions" on page 652

DataTable

An EGL *dataTable* is primarily composed of these components:

- A structure, with each top-level item defining a column.
- An array of values that are consistent with those columns. Each element of that array defines a row.

A dataTable of error messages, for example, might include these components:

- The declaration of a numeric field and a character field
- A list of paired values like these—

```
001   Error 1
002   Error 2
003   Error 3
```

You do not declare a dataTable as if you were declaring a record or data item. Instead, any code that can access a dataTable can treat that part as a variable. For details on part access, see *References to parts*.

Any code that can access a dataTable has the option of referencing the part name in a Use declaration.

Types of dataTables

Some types of dataTables are for run-time validation; specifically, to hold data for comparison against form input. (You relate the dataTable to the input field when you declare the form part.) Three types of validation dataTables are available:

matchValidTable
The user's input must match a value in the first dataTable column.

matchInvalidTable
The user's input must be different from any value in the first dataTable column.

rangeChkTable
The user's input must match a value that is between the values in the first and second column of at least one dataTable row. (The range is inclusive; the user's input is valid if it matches a value in the first or second column of any row.)

The other types of dataTables are as follows:

msgTable
Contains run-time messages.

basicTable
Contains other information that is used in the program logic; for example, a list of countries and related codes.

DataTable generation

The output generated for a dataTable part varies by the output language:

- If you are generating output in Java, each dataTable is generated as a pair of files, each named for the dataTable. One file has the extension .java, the other has the extension .tab. The .tab file is not processed by the Java compiler, but is included in the root of the directory structure that contains the package. If the package is *my.product.package*, for example, the directory structure is *my/product/package*, and the .tab file is in the directory that contains the subdirectory *my*.

- If you are generating output for COBOL, a dataTable is generated as a separate program, with the file extension .cbl. EGL also produces a binary file that has the file extension .tab. You do not compile that file; it is read as is at runtime.

You do not need to generate the dataTables if you are generating into a directory or Java package to which you had previously generated the same dataTables.

To save generation time when you do not need to generate dataTables, assign NO to the build descriptor option **genTables**.

Properties of the dataTable

You can set the following properties:

- An **alias** is incorporated into the names of generated output. If you do not specify an alias, the part name (or a truncated version)is used instead.
- The **shared** property indicates whether multiple users can access the dataTable. The default is *no*.
- The **resident** property indicates whether the dataTable remains in memory even when no program is using the dataTable. (The program goes into memory when first accessed.) The default is *no*. You can specify *yes* only if the shared specification is also *yes*.

Related concepts

"References to parts" on page 20

Related reference

"DataTable part in EGL source format" on page 462
"Use declaration" on page 930

Inserting code snippets into EGL and JSP files

The Snippets view lets you insert reusable programming objects into your code. The Snippets view contains several pieces of EGL code, as well as code for many other technologies. You can use the snippets provided or add your own to the Snippets view. For more information about using the Snippets view, see *Snippets view*.

To insert an EGL code snippet into your code, do as follows:

1. Open the file to which you want to add a snippet.
2. Open the Snippets view.
   a. Click **Window > Show View > Other**.
   b. Expand **Basic** and click **Snippets**.
   c. Click **OK**.
3. In the Snippets view, expand the **EGL** drawer. This drawer contains the available EGL code snippets.
4. Use one of these methods to insert a snippet into the file:
   - Click and drag a snippet into the source code.
   - Double-click a snippet to insert that snippet at the current cursor position. You may see a window describing the variables and values in the snippet. If so, click **Insert**.

   Note: If the cursor turns into a circle with a strike through it, indicating that the snippet can not be inserted at that point, you may be trying to insert the snippet into the wrong place. Check the snippet's details to find out where it should be inserted in the code.
5. Change the pre-defined names of functions, variables, and data parts in the snippet as appropriate to your code. Most snippets include comments that explain what names need to be changed.

Following are the snippets available in EGL:

Table 6. Snippets available in EGL

Snippet name	Description
`setCursorFocus`	A JavaScript™ function that sets the cursor focus to a specified form field on a Web page.
`autoRedirect`	A JavaScript function that tests for the presence of a session variable. If the session variable is not present, it forwards the browser to a different page.
`getClickedRowValue`	An EGL function that retrieves the hyperlinked value of a clicked row in a data table.
`databaseUpdate`	An EGL function that updates a single row of a relational table when passed a record from a PageHandler.

Related concepts
Snippets view

Related tasks
"Using the EGL templates with content assist" on page 121
"Setting the focus to a form field"
"Testing browsers for a session variable"
"Retrieving the value of a clicked row in a data table" on page 141
"Updating a row in a relational table" on page 141

Related reference

Setting the focus to a form field

The `setCursorFocus` snippet in the JSP drawer of the Snippets view is a JavaScript function that sets the cursor focus to a specified form field on a Web page. It must be placed within a `<script>` tag in a JSP page. To insert and configure this snippet, follow these directions:

1. Insert the snippet's code into the source code of the page. For more information, see *Inserting EGL code snippets*.
2. Replace `[n]` with the number of the form field which will receive focus. For example, use `[3]` to set focus to the fourth field on the page.
3. Set the form name to `form1`.
4. Change the `<body>` tag of the JSP page to `<body onload="setfocus();">`.

The code inserted by this snippet is as follows:

```
function setFocus() {
 document.getElementById('form1').elements[n].select();
 document.getElementById('form1').elements[n].focus();
}
```

Related tasks
"Inserting code snippets into EGL and JSP files" on page 139

Testing browsers for a session variable

The `autoRedirect` snippet in the JSP drawer of the Snippets view tests for the presence of a session variable. If the session variable is not present, it forwards the browser to a different page. This snippet must be placed within the `<head>` tag of a JSP page after the `<pageEncoding>` tag. To insert and configure this snippet, follow these directions:

1. Insert the snippet's code into the `<head>` tag of the page after the `<pageEncoding>` tag. For more information, see *Inserting EGL code snippets*.
2. Replace `{SessionAttribute}` with the name of the session variable that is being tested.
3. Replace `{ApplicationName}` with the name of your project or application.
4. Replace `{PageName}` with the name of the page that the browser will be redirected to if the session variable is absent.

The code inserted by this snippet is as follows:

```
<%
if ((session.getAttribute("userID") == null ))
  {
   String redirectURL =
```

```
   "http://localhost:9080/EGLWeb/faces/Login.jsp";
   response.sendRedirect(redirectURL);
  }
%>
```

Related tasks
"Inserting code snippets into EGL and JSP files" on page 139

Retrieving the value of a clicked row in a data table

The `getClickedRowValue` snippet in the EGL drawer of the Snippets view is a function that retrieves the hyperlinked value of a clicked row in a data table. This snippet must be placed in an EGL PageHandler. This snippet has the following prerequisites:

1. The JSP page has a data table.
2. The names of the JSP identifiers have not been changed from the default.
3. The page is defined as request in scope in faces-config.xml, not session.

To insert and configure this snippet, follow these directions:

1. Insert the snippet's code into the PageHandler. For more information, see *Inserting EGL code snippets*.
2. Define a char or string variable to receive the clicked value.
3. Add a command hyperlink (from the Faces Components drawer in the Palette view) to a field in the data table.
4. For the target of the command hyperlink, specify the name of the JSP page. The hyperlink links to its own page.
5. Add a parameter to the hyperlink and give that parameter the same name as the variable in the PageHandler that receives the clicked value.
6. Set the `action` property (located on the All tab of the Properties view) to the `getVal()` function.

The code inserted by this snippet is as follows:

```
function getVal()
 javaLib.store((objId)"context",
   "javax.faces.context.FacesContext",
   "getCurrentInstance");
 javaLib.store((objId)"root",
   (objId)"context", "getViewRoot");
 javaLib.store((objId)"parm",
   (objId)"root",
   "findComponent",
   "form1:table1:param1");
 recVar = javaLib.invoke((objId)"parm",
                         "getValue");
end
```

Related tasks
"Inserting code snippets into EGL and JSP files" on page 139

Updating a row in a relational table

The `databaseUpdate` snippet in the EGL drawer of the Snippets view is a function that updates a single row of a relational table when passed a record from a PageHandler. This snippet is intended to be placed in an EGL library. To insert and configure this snippet, follow these directions:

1. Insert the snippet's code into the PageHandler. For more information, see *Inserting EGL code snippets*.
2. Replace {tableName} and {keyColumn} with the name of the table and its primary key column.

The code inserted by this snippet is as follows:

```
Function updateRec(${TableName}New  ${TableName})

  // Function name - call this function
  // passing the ${TableName} Record as a parameter
  ${TableName}Old ${TableName};

  // A copy of the Record, used
  // to lock the table row and to obtain
  // the existing row values prior to update try
  ${TableName}Old.${KeyColumn} =
      ${TableName}New.${KeyColumn};
  get ${TableName}Old forUpdate;

  // Get the existing row.
  // Note that if you had custom processing to do,
  // you would insert your code after this call
  move ${TableName}New to ${TableName}Old byName;

  //Move the updated values to the copy-row
  replace ${TableName}Old;

  //And replace the row in the database.
  sysLib.commit();

  //Commit your changes to the Database
  onException
    //If the update fails...
    sysLib.rollback();

    // cancel all database updates
    // (assuming this is permitted
    // by your database) and call
    // a custom error handling routine
  end
end
```

Related tasks

"Inserting code snippets into EGL and JSP files" on page 139

Working with text and print forms

Creating an EGL formGroup part

An EGL formGroup part defines a collection of text and print forms. To create an EGL formGroup part, do as follows:

1. Identify a project or folder to contain the file. You must create a project or folder if you do not already have one.
2. In the workbench, click **File > New > Form Group**.
3. Select the project or folder that will contain the EGL file, then select a package. Since the formGroup name will be identical to the file name, choose a file name that adheres to EGL part name conventions. In the EGL Source File Name field, type the name of the EGL file, for example myFormGroup.
4. Click the **Finish** button.

Related concepts
"EGL projects, packages, and files" on page 13
"EGL form editor overview" on page 153
"Editing form groups with the EGL form editor" on page 153
"FormGroup part"
"Introduction to EGL" on page 1

Related tasks
"Creating an EGL source folder" on page 119
"Creating an EGL Web project" on page 117

Related reference
"Creating an EGL source file" on page 120
"Form part in EGL source format" on page 497
"Naming conventions" on page 652

FormGroup part

An EGL FormGroup part serves two purposes:

- Defines a collection of text and print forms. (Forms that are unique to the part are defined within the part or are included by way of a Use declaration. Forms that are common to several FormGroup parts are included by way of a Use declaration.)
- Defines zero to many *floating areas*, as described in *Form part*

You do not declare a form group as if you were declaring a record or data item. Instead, your program accesses a FormGroup part (and the related forms) only if the following statements apply:

- The location of the FormGroup part is accessible to the program, as described in *References to parts*
- A Use declaration in the program references the FormGroup part

A program can include no more than two formGroup parts; and if two are specified, one must be a *help group*. A help group contains one or more *help forms*, which are read-only forms that give information in response to a user keystroke.

Forms are available at run time only if you generate the FormGroup. The generated output for Javais a class for the FormGroup part and a class for each Form part. The generated output for a COBOL program is as follows:

- Text forms are generated into an object module
- Print forms are generated into a printing-services program

At preparation time, each of those entities is processed into a separate run-time load module. The EGL run time handles the interaction of your generated program and the form-specific code.

Form parts cannot be generated separately.

Related concepts
"Form part"
"References to parts" on page 20
"Editing form groups with the EGL form editor" on page 153

Related reference
"Use declaration" on page 930

Form part

A *form part* is a unit of presentation. It describes the layout and characteristics of a set of fields that are shown to the user at one time.

You do not declare a form as if you were declaring a record or data item. To access a form part, your program must include a use declaration that refers to the related form group.

A form part is of one of two types, *text* or *print*:

- A form of type *text* defines a layout that is displayed in a 3270 screen orin a command window. With one exception, any text form can have both constant fields and variable fields, including variable fields that accept user input. The exception is a *help form*, which is solely for presenting constant information.
- A form of type *print* defines a layout that is sent to a printer. Any print form can have both constant and variable fields.

Form properties determine the size and position of the output on a screen or page and specify formatting characteristics of that output.

A given form can be displayed on one or more *devices*, each of which is an output peripheral or is the operational equivalent of an output peripheral:

- A *screen device* is a terminal, monitor, or terminal emulator. The output surface is a screen.
- A *print device* is a file that can be sent to a printer or is the printer itself. The output surface is a page.

Whether of type *text* or *print*, a form is further categorized as follows:

- A *fixed form* has a specific starting row and column in relation to the output surface of the device. You could assign a fixed print form, for example, to start at line 10, column 1 on a page.
- A *floating form* has no specific starting row or column; instead, the placement of a floating form is at the next unoccupied line in an output surface sub-area that you declare. The declared sub-area is called a *floating area*.

You might declare a floating area to be a rectangle that starts at line 10, extends through line 20, and is the maximum width of the output device. If you have a one-line floating form of the same width, you can construct a loop that acts as follows for each of 20 times:

1. Places data in the floating map
2. Writes the floating map to the next line in the floating area

One or more floating areas are declared in the FormGroup part, but only one can accept floating forms for a particular device. If you try to present a floating form in the absence of a floating area, the entire output surface is treated as a floating area.

- A *partial form* is smaller than the standard size of the output surface for a particular device. You can declare and position partial forms so that multiple forms are displayed at different horizontal positions. Although you can specify the starting and ending columns for a partial form, you cannot display forms that are next to one another.

Additional details are specific to the form type:

- Print forms
- Text forms

Related concepts
"Print forms" on page 146
"Text forms" on page 148
"Editing form groups with the EGL form editor" on page 153
"Form templates in the EGL form editor" on page 159

Related tasks
"Creating a form in the EGL form editor" on page 155

Related reference
"FormGroup part in EGL source format" on page 494
"Form part in EGL source format" on page 497

Creating an EGL print form

A print form is an EGL form part that defines a layout to send to a printer. To create an EGL print form, do as follows:

1. Identify an EGL file to contain the print form and open that file in the EGL editor. You must create an EGL file if you do not already have one.
2. Type the specifics of the print form according to EGL syntax (for details, see *Form part in EGL source format*). You can use content assist to place an outline of the form part syntax in the file.
3. Save the EGL file.

Related concepts
"EGL projects, packages, and files" on page 13
"Form part" on page 144
"Print forms" on page 146
"Editing form groups with the EGL form editor" on page 153

Related tasks
"Creating an EGL source file" on page 120
"Using the EGL templates with content assist" on page 121
"Creating a form in the EGL form editor" on page 155

Related reference
"Content assist in EGL" on page 471
"Form part in EGL source format" on page 497
"Naming conventions" on page 652

Print forms

Forms and their types are introduced in *Form part*. The current page outlines how to present print forms.

Print process: Printing is a two-step process:

- First, you code **print** statements, each of which adds a form to a run-time buffer
- Next, the EGL run time adds the symbols needed to start a new page, sends all the buffered forms to a print device, and erases the contents of the buffer. Those services are provided in response to any of the following circumstances:
  - The program runs a **close** statement on a print form that is destined for the same print device; or
  - The program is in segmented mode (as described in *Segmentation*) and runs a **converse** statement; or
  - The program was called by a non-EGL (and non-VisualAge Generator) program, and the called program ends; or
  - The main program in the run unit ends; or
  - The system variable **ConverseVar.printerAssociation** (which assigns an output destination for print forms) is set in a COBOL program that is running on iSeries.

In the case of multiform output, the **print** statements must be invoked in the order in which you want to present the forms. Consider the following example:

- At the top of the output, a fixed form identifies a purchasing company and an order number
- In a subsequent floating area, a series of identically formatted floating forms identify each item of the company's order
- At the bottom of the output, a fixed form indicates the number of screens or pages needed to scroll through the list of items

You can achieve that output by submitting a series of **print** statements that each operate on a print form. Those statements reference the forms *in the following order*:

1. Top form
2. Floating form, as presented by a **print** statement that is invoked repeatedly in a loop
3. Bottom form

The symbols needed to start a new page are inserted in various circumstances, but you can cause the insertion by invoking the system function ConverseLib.pageEject before issuing a **print** statement.

Considerations for fixed forms: The following statements apply to fixed forms:

- If you issue a print statement for a fixed form that has a starting line greater than the current line, EGL inserts the symbols needed to advance the print device to the specified line. Similarly, if you issue a print statement for a fixed form that has a starting line less than the current line, EGL inserts the symbols needed to start a new page.

- If a fixed form overlays *some but not all* lines in another fixed form, EGL automatically inserts the symbols needed to start a new page and places the second fixed form on the new page.
- If a fixed form overlays *all* lines in another fixed form, EGL replaces the existing form without clearing the rest of the output from the buffer. To keep the existing output and place the new form on the next page, invoke the system function ConverseLib.pageEject before issuing the **print** statement for the new form.

Considerations for floating forms: The following mistakes can occur if you are using floating forms:

- You issue a **print** statement to place a floating form beyond the end of the floating area; or
- You issue a **print** statement that at least partially overlays a floating area with a fixed form, then issue a **print** statement to add a floating form to the floating area.

The result in either case is that EGL inserts the symbols needed to start a new page, and the floating form is placed on the first line of the floating area on the new page. If the page is similar to the order-and-item output described earlier, for example, the new page does not include the topmost fixed form.

Print destination: When EGL processes a **close** statement to present a print file, the output is sent to a printer or data set. You can specify the destination at any of three times:

- At test time (as described in *EGL debugger*)
- At generation time (as described in *Resource associations and file types*)
- At run time (as described in relation to the system variable ConverseVar.printerAssociation)

Related concepts
"EGL debugger" on page 261
"FormGroup part in EGL source format" on page 494
"Form part in EGL source format" on page 497
"Form part" on page 144
"Resource associations and file types" on page 286
"Segmentation in text applications" on page 149

Related reference
"pageEject()" on page 767
"printerAssociation" on page 896

Creating an EGL text form

A text form is an EGL form part that defines a layout to display in a 3270 screen orin a command window. To create an EGL text form, do as follows:

1. Identify an EGL file to contain the text form and open that file in the EGL editor. You must create an EGL file if you do not already have one.
2. Type the specifics of the text form according to EGL syntax (for details, see *Form part in EGL source format*). You can use content assist to place an outline of the form part syntax in the file.
3. Save the EGL file.

Related concepts
"EGL projects, packages, and files" on page 13

"Form part" on page 144
"Text forms"
"Editing form groups with the EGL form editor" on page 153

Related tasks
"Creating an EGL source file" on page 120
"Creating a form in the EGL form editor" on page 155
"Using the EGL templates with content assist" on page 121

Related reference
"Content assist in EGL" on page 471
"Form part in EGL source format" on page 497
"Naming conventions" on page 652

Text forms

Forms and their types are introduced in *Form part*. The current page outlines how to present text forms.

The **converse** statement is sufficient for giving the user access to a single, fixed text form. The logical flow of your program continues only after the user responds to the displayed form. You can also construct output from multiple forms, as in the following case:

- At the top of the output, a fixed form identifies a purchasing company and an order number
- In a subsequent floating area, a series of identically formatted floating forms identify each item of the company's order
- At the bottom of the output, a fixed form indicates the number of screens needed to scroll through the list of items

Two steps are necessary:

1. First, you construct the order-and-item output by coding a series of **display** statements, each of which adds a form to a run-time buffer but does not present data to the screen. Each **display** statement operates on one of the following forms:
   - Top form
   - Floating form, as presented by a **display** statement that is invoked repeatedly in a loop
   - Bottom form
2. Next, the EGL run time presents all the buffered text forms to the output device in response to either of these situations:
   - The program runs a **converse** statement; or
   - The program ends.

In most cases, you present the last form of your screen output by coding a **converse** statement rather than a **display** statement.

The fixed forms each have an on-screen position, so the order in which you specify them, in relation to each other and in relation to the repeated display of floating forms, does not matter. The contents of the buffer are erased when output is sent to the screen.

If you overlay one text form with another, no error occurs, but the following statements apply:

- If a partial form overlays any lines in another fixed form, EGL replaces the existing form without clearing the rest of the output from the buffer. If you want to erase the existing output before displaying the new form, invoke the system function ConverseLib.clearScreen before issuing the **display** or **converse** statement for the new form.
- If you use a **display** or **converse** statement to place a floating map beyond the bottom of the floating area, all the floating forms in that floating area are erased, and the added form is placed on the first line of the same floating area.
- If a floating form overlays a fixed form, these statements apply- -
  - Only the fixed-form lines that are in the floating area are overwritten by the floating form
  - The result is unpredictable if a fixed-form line is overwritten by a floating-form line that includes a variable field

Whether you are presenting one form or many, the output destination is the screen device at which the user began the run unit.

Related concepts
"Form part" on page 144

Related reference
"Form part in EGL source format" on page 497
"FormGroup part in EGL source format" on page 494
"clearScreen()" on page 766

Segmentation in text applications

Segmentation concerns how a program interacts with its environment before issuing a **converse** statement.

By default a program that presents text forms is *non-segmented*, which means that the program behaves as if it were always in memory and providing a service to only one user. The following rules are in effect before a non-segmented program issues a **converse** statement:

- Databases and other recoverable resources are not committed
- Locks are not released
- File and database positions are retained
- Single-user EGL tables are not refreshed; their values are the same before and after the converse
- Similarly, system variables are not refreshed

A called program is always non-segmented.

A non-segmented program can be easier to code. For example, you do not need to reacquire a lock on an SQL row after a **converse**. Disadvantages include the fact that SQL rows are held during user think time, a behavior that leads to performance problems for other users who need to access the same SQL row.

Two techniques are available for releasing or refreshing resources before a converse in a non-segmented program:

- You can set the system variable **ConverseVar.commitOnConverse** to 1. Results are as follows before a converse:
  - Databases and other recoverable resources are committed
  - Locks are released

  - File and database positions are not retained, except when the database open statement includes the withHold option, as is available only for COBOL programs

  The setting of **ConverseVar.commitOnConverse** never affects system variables or EGL tables.
- A second technique for handling converse is to set the *segmented* property of the text program to *yes*, either by changing a program property at development time or by setting the system variable **ConverseVar.segmentedMode** to 1 at run time. Segmentation causes the following results before a converse:
  - Databases and other recoverable resources are committed
  - Locks are released
  - File and database positions are not retained, even when the database open statement includes the withHold option
  - Single-user EGL tables are refreshed; their values become the same as when the program began
  - System variables are refreshed; their values become the same as when the program began, except for a subset of variables whose values are saved *across segments*

  The behavior of a segmented program is unaffected by the value of the system variable **ConverseVar.commitOnConverse**.

Related concepts

"Program part" on page 130

Modified data tag and modified property

Each item on a text form has a *modified data tag*, which is a status value that indicates whether the user is considered to have changed the form item when the form was last presented.

As described later, an item's modified data tag is distinct from the item's **modified** property, which is set in the program and which pre-sets the value of the modified data tag.

Interacting with the user: In most cases, the modified data tag is pre-set to *no* when the program presents the form to the user; then, if the user changes the data in the form item, the modified data tag is set to *yes*, and your program logic can do as follows:

- Use a data table or function to validate the modified data (as occurs automatically when the modified data tag for the item is *yes*)
- Detect that the user modified the item (for example, by using a conditional statement of the type *if item modified*)

The user sets the modified data tag by typing a character in the item or by deleting a character. The modified data tag stays set, even if the user, before submitting the form, returns the field content to the value that was presented.

When a form is re-displayed due to an error, the form is still processing the same converse statement. As a result, any fields that were modified on the converse have the modified data tag set to *yes* when the form is re-displayed. For example, if data is entered into a field that has a validator function, the function can invoke the ConverseLib.validationFailed function to set an error message and cause the form to re-display. In this case, when an action key is pressed, the validator function will execute again because the field's modified data tag is still set to *yes*.

Setting the modified property: You may want your program to do a task regardless of whether the user modified a particular field; for example:

- You may want to force the validation of a password field even if the user did not enter data into that field
- You may specify a validation function for a critical field (even for a protected field) so that the program always does a particular *cross-field validation*, which means that your program logic validates a group of fields and considers how one field's value affects the validity of another.

To handle the previous cases, you can set the **modified** property for a particular item either in your program logic or in the form declaration:

- In the logic that precedes the form presentation, include a statement of the type *set item modified*. The result is that when the form is presented, the modified data tag for the item is pre-set to *yes*.
- In the form declaration, set the ***modified*** property of the item to *yes*. In this case, the following rules apply:
  - When the form is presented for the first time, the modified data tag for the item is pre-set to *yes*.
  - If any of the following situations occurs before the form is presented, the modified data tag is pre-set to *yes* when the form is presented:
    - The code runs a statement of the type *set item initial*, which reassigns the original content and property values for the item; or
    - The code runs a statement of the type *set item initialAttributes*, which reassigns the original property values (but not content) for each item on the form; or
    - The code runs a statement of the type *set form initial*, which reassigns the original content and property values for each item on the form; or
    - The code runs a statement of the type *set form initialAttributes*, which reassigns the original property values (but not content) for each item on the form

The *set* statements affect the value of the **modified** property, not the current setting of the modified data tag. A test of the type *if item modified* is based on the modified data tag value that was in effect when the form data was last returned to your program. If you try to test the modified data tag for an item before your logic presents the form *for the first time*, an error occurs at run time.

If you need to detect whether the user (rather than the program) modified an item, make sure that the value of the modified data tag for the item is pre-set to *no*:

- If the **modified** property of the item is set to *no* in the form declaration, do not use a statement of the type *set item modified*. In the absence of that statement, the **modified** property is automatically set to *no* prior to each form presentation.
- If the **modified** property of the item is set to *yes* in the form declaration, use a statement of the type *set item normal* in the logic that precedes form presentation. That statement sets the **modified** property to *no* and (as a secondary result) presents the item as unprotected, with normal intensity.

Testing whether the form is modified: The form as a whole is considered to be modified if the modified data tag is set to *yes* for any of the variable form items. If you test the modified status of a form that was not yet presented to the user, the test result is FALSE.

Examples: Assume the following settings in the form *form01*:

- The **modified** property for the field *item01* is set to *no*
- The **modified** property for the field *item02* is set to *yes*

The following logic shows the result of various tests:

```
// tests false because a converse statement
// was not run for the form
if (form01 is modified)
  ;
end

// causes a run-time error because a converse
// statement was not run for the form
if (item01 is modified)
  ;
end

// assume that the user modifies both items
converse form01;

// tests true
if (item01 is modified)
  ;
end

// tests true
if (item02 is modified)
  ;
end

// sets the modified property to no
// at the next converse statement for the form
set item01 initialAttributes;

// sets the modified property to yes
// at the next converse statement for the form
set item02 initialAttributes;

// tests true
// (the previous set statement takes effect only
// at the next converse statement for the form
if (item01 is modified)
  ;
end

// assume that the user does not modify either item
converse form01;

// tests false because the program set the modified
// data tag to no, and the user entered no data
if (item01 is modified)
  ;
end

// tests true because the program set the modified
// data tag to yes
if (item02 is modified)
  ;
end

// assume that the user does not modify either item
converse form01;

// tests false
if (item01 is modified)
  ;
end
```

```
// tests false because the presentation was not
// the first, and the program did not reset the
// item properties to their initial values
if (item02 is modified)
  ;
end
```

EGL form editor overview

The EGL form editor lets you edit a formGroup part graphically. The form editor works with formGroup parts, their form parts, and the fields in those form parts in much the same way as other graphical editors work with files like Web pages and Web diagrams.

The form editor has these parts:

- The editor itself, which displays the graphical representation of the form group and that form group's source code. You can switch between the graphical representation and the source code by clicking the **Design** and **Source** tabs at the bottom of the editor. Changes to the Source view or Design view are reflected immediately in the other view.
- The Properties view, which displays the EGL properties of the form or field currently selected in the editor.
- The Palette view, which displays the types of forms and fields that can be created in the editor.
- The Outline view, which displays a hierarchical view of the form group open in the editor.

For more information on using the form editor, see *Editing form groups with the EGL form editor*.

Related concepts
"FormGroup part" on page 143
"Form part" on page 144
"Display options for the EGL form editor" on page 163
"Form filters in the EGL form editor" on page 164
"Editing form groups with the EGL form editor"

Related tasks
"Creating a form in the EGL form editor" on page 155
"Setting preferences for the EGL form editor" on page 163
"Setting preferences for the EGL form editor palette entries" on page 158

Editing form groups with the EGL form editor

The EGL form editor lets you edit a formGroup part graphically. The form editor works with formGroup parts, their form parts, and the fields in those form parts in much the same way as other graphical editors work with files like Web pages and Web diagrams. At any time, you can click the **Source** tab at the bottom of the editor and see the EGL source code that the editor is generating. The form editor has the following features:

- The form editor can edit the size and properties of a form group. To edit the properties of a form group, open the form group in the form editor and change

its properties in the Properties view. To resize a form group, open it in the form editor and choose a size in characters from the list at the top of the editor.

- The form editor can create, edit, and delete forms in a form group. To create a form, click the appropriate type of form on the Palette view and draw a rectangle representing the size and location of the form in the editor. To edit a form, click it to select it, and then use the Properties view to edit its properties. You can also drag a form to move it, or resize it using the resize handles that appear on the border of a selected form. Many of the same options are available when you right-click a form to open its popup menu. See *Creating a form in the EGL form editor*.
- The form editor uses templates to create commonly used types of forms, such as popup forms and popup menus. These forms have pre-made borders, sections, and fields. See *Form templates in the EGL form editor*.
- The form editor can create, edit, and delete fields in a form. To create a field, click the appropriate type of field on the Palette view and draw a rectangle representing the size and location of the field in the editor. You can add a field only within an existing form. To edit a field, click it to select it, and then use the Properties view to edit its properties. You can also drag a field to move it, or resize it using the resize handles that appear on the border of a selected form. Many of the same options are available when you right-click a field to open its popup menu. See *Creating a constant field* or *Creating a variable field*.
- Filters can prevent forms from being shown in the form editor, allowing you to mimic the appearance of the form group at run time. To switch filters, create a filters, or edit filters, use the **Filters** button at the top of the editor. See *Form filters in the EGL form editor* or *Creating a filter*.
- You can customize the appearance of the form editor by using the display options at the top of the editor and by setting the editor's preferences in the Preferences window. For example, these options can display a grid over the form group, increase or decrease the zoom level, and show or hide sample values in fields. See *Display options for the EGL form editor* or *Setting preferences for the EGL form editor*.

Related concepts
"EGL form editor overview" on page 153
"FormGroup part" on page 143
"Form part" on page 144
"Display options for the EGL form editor" on page 163
"Form filters in the EGL form editor" on page 164
"Form templates in the EGL form editor" on page 159

Related tasks
"Creating a filter" on page 155
"Creating a popup form" on page 159
"Creating a popup menu" on page 160
"Displaying a record in a text or print form" on page 161
"Setting preferences for the EGL form editor" on page 163
"Setting preferences for the EGL form editor palette entries" on page 158
"Creating a form in the EGL form editor" on page 155
"Creating a constant field" on page 156
"Creating a variable field in a print or text form" on page 157

Related reference
"FormGroup part in EGL source format" on page 494
"Form part in EGL source format" on page 497

Creating a filter

To create a new filter in the EGL form editor, follow these steps:

1. Open a form group in the form editor.
2. In the form editor, click the **Filters** button. The Filters window opens.
3. In the Filters view, click the **New** button. The New Filter dialog opens.
4. In the New Filter dialog, type a name for the filter and click **OK**.
5. Select the forms to be displayed while the filter is active by doing one or more of the following steps:
   - Clear the check boxes next to the forms you want hidden by the filter.
   - Select the check boxes next to the forms you want shown by the filter.
   - Click the **Select All** button to show every form.
   - Click the **Deselect All** button to hide every form.
6. Click **OK**.

   The new filter is now active. You can switch filters by using the list next to the **Filters** button.

Related concepts
"EGL form editor overview" on page 153
"Editing form groups with the EGL form editor" on page 153
"Display options for the EGL form editor" on page 163
"Form filters in the EGL form editor" on page 164

Related tasks
"Creating a form in the EGL form editor"

Creating a form in the EGL form editor

To create a form in the EGL form editor, follow these steps:

1. Open a form group in the form editor.
2. On the Palette view, click either **Text Form** or **Print Form**.
3. On the form group in the editor, click and drag a rectangle that indicates the size and shape of the form. The Create Form Part window opens.
4. In the Create Form Part Window, type a name for the form in the **Enter part name** field. This name will be the name of the form part in the EGL source code.
5. Click **OK**.
6. Click the form and edit its properties in the Properties view.
7. Add fields to the form as appropriate. See *Creating a constant field* and *Creating a variable field*.

You can also create forms based on the templates in the Palette view. These templates create forms with predefined appearances and fields. See *Creating a popup form* or *Creating a popup menu*.

Related concepts
"EGL form editor overview" on page 153
"Editing form groups with the EGL form editor" on page 153
"FormGroup part" on page 143
"Form part" on page 144
"Display options for the EGL form editor" on page 163

"Form filters in the EGL form editor" on page 164
"Form templates in the EGL form editor" on page 159

Related tasks
"Creating a filter" on page 155
"Creating a popup form" on page 159
"Creating a popup menu" on page 160
"Displaying a record in a text or print form" on page 161
"Creating a constant field"
"Creating a variable field in a print or text form" on page 157

Related reference
"FormGroup part in EGL source format" on page 494
"Form part in EGL source format" on page 497

Creating a constant field

Constant fields display a string of text that does not change in a form. Unlike variable fields, constant fields can not be accessed by EGL code. To insert a constant field into a form, follow these steps:

1. Open a form group in the EGL form editor.
2. If the form group has no forms, add a form to the form group. See *Creating a form*.
3. On the Palette view, click a type of constant field to add. The following types of constant fields are available by default:

Table 7. Constant fields available in the Palette view

Field name	Default color	Default intensity	Default highlighting	Default Protection
Title	Blue	Bold	None	Skip
Column Heading	Blue	Bold	None	Skip
Label	Cyan	Normal	None	Skip
Instructions	Cyan	Normal	None	Skip
Help	White	Normal	None	Skip

These fields are samples of commonly used constant text fields in a text-based interface. You can customize the individual fields after placing them on a form. You can also customize the default color, intensity, and highlighting of the fields available in the Palette view. See *Setting preferences for the EGL form editor palette entries*.

4. Within a form in the editor, click and hold the mouse to draw a rectangle that represents the size and location of the field. A preview box next to the mouse cursor shows you the size of the field and its location relative to the form.

   Note: You can add a field only within an existing form.
5. When the field is the correct size, release the mouse. The new field is created.
6. Type the text you want to display in the field.
7. In the Properties view, set the properties for the new field.

Related concepts
"EGL form editor overview" on page 153
"Editing form groups with the EGL form editor" on page 153
"Form part" on page 144

Related tasks
"Setting preferences for the EGL form editor palette entries" on page 158
"Creating a variable field in a print or text form"

Related reference
"Form part in EGL source format" on page 497

Creating a variable field in a print or text form

Variable fields can serve as input or output text in a form. Each variable field is based on an EGL primitive or a DataItem part. Unlike constant fields, variable fields can be accessed by EGL code. To insert a variable field into a form, follow these steps:

1. Open a form group in the EGL form editor.
2. If the form group has no forms, add a form to the form group. See *Creating a form*.
3. On the Palette view, click a type of variable field to add. The following types of variable fields are available by default:

Table 8. Variable fields available in the Palette view

Field name	Default color	Default intensity	Default highlighting	Default Protection
Input	Green	Normal	Underlined	No
Output	Green	Normal	None	Skip
Message	Red	Bold	None	Skip
Password	Green	Invisible	None	No

These fields are samples of commonly used variable text fields in a text-based interface. You can customize the individual fields after placing them on a form. You can also customize the default color, intensity, and highlighting of the fields available in the Palette view. See *Setting preferences for the EGL form editor palette entries*.

4. Within a form in the editor, click and hold the mouse to draw a rectangle that represents the size and location of the field. A preview box next to the mouse cursor shows you the size of the field and its location relative to the form.

   Note: You can add a field only within an existing form.
5. When the field is the correct size, release the mouse. The New EGL Field window opens.
6. In the New EGL Field window, enter the name of the new field in the **Name** field.
7. Do one of the following to select the type of field:
   - To use a primitive type, click a primitive type from the **Type** list.
   - To use a DataItem part, follow these steps:
     a. Click **dataItem** from the **Type** list. The Select a DataItem Part window opens.
     b. In the Select a DataItem Part window, click a DataItem part from the list or type the name of one.
     c. Click **OK**.
8. As necessary, type values in the **Dimensions** field or fields to set the dimensions of the new variable field.
9. If you want to make the field an array, select the **Array** check box.

10. If the **Array** check box is selected, click **Next** and continue following these steps. Otherwise, click **Finish** and stop following these steps. The new field is created and you do not need to follow the rest of these steps, because they are applicable only if you are creating an array.
11. On the Array Properties page of the New EGL Field window, type the size of the array in the **Array Size** field.
12. Choose an orientation of **Down** or **Across** from the **Index Orientation** buttons.
13. Under **Layout**, enter the number of vertical and horizontal fields in the **Fields Down** and **Fields Across** fields.
14. Under **Spaces**, enter the amount of space between the array's rows and columns in the **Lines between rows** and **Spaces between columns** fields.
15. Click **Finish**. The new field is created in the form group.

Once you have created the new field, click the field to select it and set the properties for the field in the Properties view.

Since variable fields have no default value, they can be invisible if they are not highlighted. To mark each variable field with appropriate sample text, click the **Toggle Sample Values** button at the top of the editor.

Once you have created a variable field, you can double-click it in the editor to open the Edit Type Properties window. From this window you can edit the field in the following ways:

- Change the field's name by typing a new name in the **Field Name** field.
- Select a new type of field from the **Type** list.
- Change the precision of the field by entering a new number in the **Precision** field.

When you are finished editing the field's properties in the Edit Type Properties window, click **OK**.

Related concepts
"EGL form editor overview" on page 153
"Editing form groups with the EGL form editor" on page 153
"Form part" on page 144

Related tasks
"Setting preferences for the EGL form editor palette entries"
"Creating a constant field" on page 156

Related reference
"Form part in EGL source format" on page 497

Setting preferences for the EGL form editor palette entries

The preferences for the EGL form editor palette control the default color, intensity, and highlighting for the types of constant and variable fields in the palette. To change these preferences, follow these steps:

1. From the menu bar, click **Window** > **Preferences**. The Preferences window opens.
2. In the left pane of the Preferences window, expand **EGL** > **EGL Form Editor**and click **EGL Palette Entries**.

3. In the right pane of the Preferences window, for each type of constant and variable field in the **Palette Entries** list, select the following options:
   - From the **Color** list, click a default color for that type of field.
   - From the **Intensity** list, click a default intensity for that type of field.
   - From the **Highlight** radio buttons, click a default highlighting style for that type of field.
   - From the **Protect** radio buttons, choose whether the field is protected from user update by default. For more information about protecting fields, see *ConsoleField properties and fields*.

   Note: You can restore all of the palette entries to their default settings by clicking **Restore Defaults**.
4. When you are finished setting the preferences for the palette entries, click **OK**

Related concepts
"EGL form editor overview" on page 153
"Editing form groups with the EGL form editor" on page 153

Related tasks
"Setting preferences for the EGL form editor" on page 163
"Creating a constant field" on page 156
"Creating a variable field in a print or text form" on page 157

Related reference
"ConsoleField properties and fields" on page 429

Form templates in the EGL form editor

The EGL form editor uses templates to create commonly used types of forms and fields. These templates are listed in the **Templates** drawer of the Palette view.

The form editor can create forms from templates. These forms have pre-made borders, sections, and fields. To create a form from a template, see *Creating a popup form* or *Creating a popup menu*.

The form editor can create a group of fields using an EGL record as a template. To create fields representing data from an EGL record, see *Displaying a record in a form*.

Related concepts
"EGL form editor overview" on page 153
"Editing form groups with the EGL form editor" on page 153

Related tasks
"Creating a popup form"
"Creating a popup menu" on page 160
"Displaying a record in a text or print form" on page 161
"Setting preferences for the EGL form editor" on page 163
"Setting preferences for the EGL form editor palette entries" on page 158
"Creating a form in the EGL form editor" on page 155

Creating a popup form

A popup form is a special kind of form that can be added to a form group. Fundamentally, a popup form is the same as an ordinary text form, but popup forms are created with pre-made features like borders and sections. To create a popup form in the EGL form editor, follow these steps:

1. Open a form group in the form editor.
2. On the Palette view, click **Popup Form**.
3. On the form group in the editor, click and drag a rectangle that indicates the size and shape of the popup form. The Create Form Part window opens.
4. In the Create Form Part Window, type a name for the form in the **Enter part name** field. This name will be the name of the form part in the EGL source code.
5. Click **OK**. The New Popup Form Template window opens.
6. In the New Popup Form Template window, enter the characters to use for the form's borders in the **Vertical Character** and **Horizontal Character** fields.
7. Click a color for the border from the **Color** list.
8. Click an intensity for the border from the **Intensity** list.
9. Click a highlight value from the **Highlight** radio buttons.
10. Repeat the following steps for each section you want to add to the form. You must add at least one section to the form.
    a. Under **Popup Sections**, click the **Add** button. The Create Popup Form Section window opens.
    b. In the Create Popup Form Section window, type a name for the section in the **Section Name** field.
    c. In the **Number of rows** field, enter the number of rows in the section. Do not enter a number greater than the number of remaining effective rows, which is displayed at the bottom of the New Popup Form Template window.
    d. Click **OK**.
    e. Use the **Up** and **Down** buttons to set the order of the fields.

    Note: The total number of rows in the popup field's sections cannot exceed the total number of rows in the popup field. When adding sections, pay attention to the **Remaining Effective Rows** field and remember that dividers between the sections require an additional row for each new field.
11. When you are finished adding sections to the popup field, click **Finish**. The new popup form is created in the editor.
12. Add fields to the form as appropriate. See *Creating a constant field* and *Creating a variable field*.

Related concepts
"EGL form editor overview" on page 153
"Editing form groups with the EGL form editor" on page 153
"Form templates in the EGL form editor" on page 159

Related tasks
"Setting preferences for the EGL form editor" on page 163
"Creating a form in the EGL form editor" on page 155
"Creating a constant field" on page 156
"Creating a variable field in a print or text form" on page 157
"Creating a popup menu"

Creating a popup menu

A popup menu is a special kind of form that can be added to a form group. Fundamentally, a popup menu is the same as an ordinary text form, but popup

menus are created with pre-made features like a title, help text, and a specified number of menu options. To create a popup menu in the EGL form editor, follow these steps:

1. Open a form group in the form editor.
2. On the Palette view, click **Popup Menu**.
3. On the form group in the editor, click and drag a rectangle that indicates the size and shape of the popup menu. The Create Form Part window opens.
4. In the Create Form Part Window, type a name for the form in the **Enter part name** field. This name will be the name of the form part in the EGL source code.
5. Click **OK**. The New Popup Menu Template window opens.
6. In the New Popup Menu Template, enter the size of the popup menu in the **Width** and **Height** fields. By default, these fields are populated with the size of the form you created in the editor.
7. In the **Menu Title** field, type the title of the menu.
8. In the **Number of Menu Options** field, type the number of menu options that the popup menu will have.
9. In the **Menu Help Text** field, type any additional help text for the menu.
10. Click **Finish**. The new popup menu is created in the editor.
11. Add fields to the new popup menu and edit the existing fields as appropriate. See *Creating a constant field* and *Creating a variable field*.

Related concepts
"EGL form editor overview" on page 153
"Editing form groups with the EGL form editor" on page 153
"Form templates in the EGL form editor" on page 159

Related tasks
"Setting preferences for the EGL form editor" on page 163
"Creating a form in the EGL form editor" on page 155
"Creating a constant field" on page 156
"Creating a variable field in a print or text form" on page 157
"Creating a popup form" on page 159

Displaying a record in a text or print form

The **Record** template, found in the **Templates** drawer of the Palette view, creates a group of form fields that are equivalent to the fields in an EGL record part. To create the form fields, follow these steps:

1. Open a form group in the EGL form editor.
2. Create a form. See *Creating a form in the EGL form editor*.
3. On the Palette view, click **Record**.
4. Within a form in the editor, click and hold the mouse to draw a rectangle that represents the size and location of the fields. A preview box next to the mouse cursor shows you the size of the record fields and their location relative to the field.

   Note: You can add a record only within an existing form.
5. When the record is the correct size, release the mouse. The EGL Record Placement window opens.
6. In the EGL Record Placement, click **Browse**. The Select a Record Part dialog opens.

7. In the Select a Record Part dialog, click the name of the record part you want to use or type the name of a record part.
8. Click **OK**. Now the Create a Record window is populated with a list of the fields in that record.
9. Using one or more of the following methods, select and organize the record part fields you want to display as fields in the form:
   - To remove a field, click its name and then click **Remove** .
   - To add a field, follow these steps:
     a. Click the **Add** button. The Edit Table Entry window opens.
     b. In the Edit Table Entry window, type a name for the field in the **Field Name** box.
     c. In the **Type** list, select a type for the field.
     d. If necessary for the type you have selected, enter the precision for the field in the **Precision** field.
     e. Enter a width for the field in the **Field Width** field.
     f. If you want the field to be an input field, select the **Make this field an input field** check box. Otherwise, clear the check box.
     g. Click **OK**.
   - To edit a field, follow these steps:
     a. Click the field's name.
     b. Click the **Edit** button. The Edit Table Entry window opens.
     c. In the Edit Table Entry window, type a name for the field in the **Field Name** box.
     d. In the **Type** list, select a type for the field.
     e. If necessary for the type you have selected, enter the precision for the field in the **Precision** field.
     f. Enter a width for the field in the **Field Width** field.
     g. If you want the field to be an input field, select the **Make this field an input field** check box. Otherwise, clear the check box.
     h. Click **OK**.
   - To move fields up or down in the list, use the **Up** and **Down** buttons.
10. Using the **Orientation** radio buttons, choose a vertical or horizontal orientation for the fields.
11. In the **Number of Rows** field, enter the number of rows you want the group of fields to have.
12. If you want the group of fields to have a header row, select the **Create header row** check box.
13. Click **Finish**.

Related concepts
"EGL form editor overview" on page 153
"Editing form groups with the EGL form editor" on page 153
"Form templates in the EGL form editor" on page 159

Related tasks
"Creating a form in the EGL form editor" on page 155
"Creating a constant field" on page 156
"Creating a variable field in a print or text form" on page 157

Display options for the EGL form editor

The EGL form editor has display options that allow you to control how form groups appear in the editor at design time. These options do not change the appearance of the form group at run time. From left to right at the top of the editor, these are the buttons that control the display options:

Toggle Gridlines
This option displays a grid over the form group to help in sizing and arranging forms. To change the color of the grid, see *Setting preferences for the EGL form editor*.

Toggle Sample Values
This option inserts sample values into variable fields, which would otherwise be invisible.

Toggle Black and White Mode
This option switches the editor's background from black to white.

Zoom Level
Sets the magnification level of the editor.

There are other buttons at the top of the editor that control the size of the form group and the editor's filters. See *Editing form groups with the EGL form editor* or *Form filters in the EGL form editor*.

Related concepts
"EGL form editor overview" on page 153
"Form filters in the EGL form editor" on page 164

Related tasks
"Editing form groups with the EGL form editor" on page 153
"Creating a filter" on page 155
"Setting preferences for the EGL form editor"

Setting preferences for the EGL form editor

The preferences for the EGL form editor can change the appearance of the form editor, such as the background color and grid color. To change the preferences for the form editor, follow these steps:

1. From the menu bar, click **Window** > **Preferences**. The Preferences window opens.
2. In the left pane of the Preferences window, expand **EGL** and click **EGL Form Editor**.
3. In the right pane of the Preferences window, select the preferences for the form editor:
   - In the **Background Color** field, select a background color for the form editor.
   - In the **Grid Color** field, select a grid color for the form editor.
   - If you want to show a border around fields, select the **Highlight fields** check box and select a color.
   - If you want to show rulers at the top and left side of the form editor, select the **Show rulers** check box.
   - In the **Font** list, click a font for the fields and click a size from the adjacent list.

Note: Choose a monospaced font to ensure that your fields display at the correct size in the form editor. A monospaced font is a font whose characters all have the same width, such as Courier New.

- If you want blinking fields to be displayed in italic type in the editor, select the **Visually demonstrate blinking fields** check box. This option does not change the appearance of the fields at run time; it only changes their appearance at design time.

Note: You can restore the EGL form editor preferences window to its default settings by clicking **Restore Defaults**.

4. When you are finished setting the preferences for the EGL form editor, click **OK**.

Related concepts
"EGL form editor overview" on page 153
"Editing form groups with the EGL form editor" on page 153
"Form filters in the EGL form editor"

Related tasks
"Setting preferences for the EGL form editor palette entries" on page 158

Form filters in the EGL form editor

Filters limit which forms are shown in the EGL form editor. You can define any number of filters, but only one filter can be active at a time. Filters affect only the presentation of the form group at design time; they do not affect the EGL code in any way. See *Creating a filter*.

You can switch between active filters with the list next to the **Filters** button at the top of the editor. To create, edit, or delete filters, click the **Filters** button.

From the Filters window, you can manage your filters by using the following functions:

- Select filters from the list.
- Add a new filter by clicking **New**.
- Delete a filter by selecting it from the list and clicking **Remove**.
- Select which forms are displayed when the filter is active.

Related concepts
"EGL form editor overview" on page 153
"Editing form groups with the EGL form editor" on page 153

Related tasks
"Creating a filter" on page 155
"Creating a form in the EGL form editor" on page 155

Creating a Console User Interface

Console user interface

Console user interface (ConsoleUI) is a technology for displaying data in a text-based format on a Windows or UNIX screen. This technology is available only in EGL-generated Java programs, not in PageHandlers or COBOL programs.

The interface that you create with ConsoleUI can be displayed in Windows 2000/NT/XP or UNIX X-windows, either locally or by way of a remote terminal session.

ConsoleUI is distinct from Text user interface (TextUI), and the two cannot operate in the same program:

- When TextUI is in effect, the style of interace is like that used in a mainframe program interacting with 3270 terminals. The program presents a text form but does not process user input as the user moves from one field to the next. When the user submits the form (by pressing the **Enter** key, in most cases), all the data in the form returns to the program, and only then does the program validate your data; if validation succeeds, the program runs your next coded statement.
- When ConsoleUI is in effect, the style of interface is like that used in a UNIX-based program interacting with character-based terminals. The program presents a console form and can respond immediately to a user event, as when the user presses the **Tab** key to move an on-screen cursor to the next field. Validation is on a field-by-field basis, and you can restrict the cursor to the current field until the user has typed valid data there.

When you use consoleUI, you typically code a program as follows:

1. Declare a set of variables that are based on the ConsoleUI parts, which are always available; you do not define the parts that are specific to ConsoleUI.
2. Open a visual entity such as a form by including a consoleUI variable as an argument when you invoke the appropriate EGL function. Alternatively, you can open a visual entity by invoking an EGL function like **displayFormByName**, which accepts a name that is known at run time.
3. Reference the visual entity in an EGL **openUI** statement, which allows for user interaction by tying particular events (such as user keystrokes) to particular logic.

The user of a consoleUI application can press keys to interact with the on-screen display, but mouse clicks have no effect.

ConsoleUI can accept user input into a field, but only if you have specified a *binding*, which is a correspondence between the input field and a variable of primitive type. EGL run time acts as follows:

- Uses the variable value as the initial content of a displayed field; and
- Moves the user's input to that variable as soon as the user leaves the field.

ConsoleUI also allows you to interact with users in *line mode*, which is a mode of processing in which your code reads or writes only one line at a time. The implications of line mode are as follows:

- In the Eclipse workbench, the user interacts with the Console view
- In a program that was invoked with a command prompt, the user interacts with the command window
- In a program that runs under Curses in UNIX, the user interacts with the window in which the UI is displayed; and the usual, window-based interaction is suspended

ConsoleUI is equivalent to the user-interface technology in the Informix 4GL product.

Related tasks
"Creating an interface with consoleUI"

Related reference
"ConsoleUI parts and related variables" on page 167
"ConsoleUI screen options for UNIX" on page 171
"EGL library ConsoleLib" on page 735
"openUI" on page 602
"Use of new in ConsoleUI" on page 170

Creating an interface with consoleUI

Console user interface (ConsoleUI) is a technology for displaying data in a text-based format on a Windows or UNIX screen.

The steps for creating an interface with consoleUI are as follows:

1. Create an EGL source file
2. Write a program that includes the language elements described in *ConsoleUI parts and related variables*
3. Generate Java code from the EGL source file
4. Run the generated Java file as an application

Each of these tasks are detailed below.

Creating an EGL source file

1. In the workbench, from the EGL Perspective, select **File>New>EGL Source File**. Or, from any perspective, select **File>New>Other>EGL Source File.**.
2. In the wizard screen, enter the following information:
   - **Source Folder**: the directory location that will contain the EGL source file.
   - **Package**: the package location that will contain the EGL source file. This field is optional.
   - **EGL Source File Name**: the filename of the Console UI source file, such as **myConsoleUI**.
3. Select **Finish** to create the file. An extension (**.egl**) is automatically appended to the end of the file name. The EGL file appears in the Project Explorer view and automatically opens in the default EGL editor.

Writing the ConsoleUI program

To populate the source file and create the ConsoleUI, you will need to use the ConsoleUI language elements, which are introduced in the **egl.ui.console** overview help topic, and defined thoroughly in the individual ConsoleUI **Library**, **OpenUI Statement**, **Record types**, and **Enumerations** help topics.

A ConsoleUI application must include, at a minimum the following elements:

1. PROGRAM...END
2. Function main ()
3. OpenUI Statement

Note: Although the **OpenUI statement** is fundamental to the ConsoleUI, you can write a successful ConsoleUI program without an **OpenUI statement**.

Generating Java code from EGL source

To generate a Java file:

1. In the EGL Editor, right-click on the ConsoleUI file. A context menu displays.
2. Select **Generate**.

Note: A ConsoleUI **.egl** source file cannot be generated to COBOL.

Run the generated Java file as an application

To run the generated Java file:

1. From the Project Explorer, right-click on the generated Java (**.java**) file. A context menu displays.
2. Select **Run>Run As>Java Application**.
3. Or, with the Java file open in the editor, select **Run>Run As>Java Application** from the main menu.
4. Your ConsoleUI will display to a window.

A ConsoleUI application can display in either a curses-based terminal session or a Swing-based graphical window. UNIX users have a more flexible display choice, which is described in the *ConsoleUI screen options for UNIX* help topic.

Note: IBM does not support usage of both ConsoleUI and TextUI in the same program.

Related concepts
"Console user interface" on page 165

Related reference
"EGL library ConsoleLib" on page 735
"ConsoleUI parts and related variables"
"ConsoleUI screen options for UNIX" on page 171
"openUI" on page 602

ConsoleUI parts and related variables

When you work with consoleUI, you create the following kinds of variables, which are based on the related consoleUI parts:

- Window
- Prompt
- ConsoleField
- ConsoleForm
- Menu
- MenuItem

The library **ConsoleLib** also includes system variables of type PresentationAttributes. The system variables control visual aspects of displayed output; and to change aspects of your display, you can change those variables by setting the PresentationAttributes fields **color**, **highlight**, and **intensity**. For details on those fields, see *PresentationAttributes fields in EGL consoleUI*.

Window

A window is a rectangular area in which you can place other visual entities that are represented as variables.

When you display a window and no other windows are in effect, the new window is inside the *screen window*, which is a rectangle that has the basic characteristics of any window in the operating system. A difference is in effect for UNIX, when the Curses library is in use; in that case, the display of a consoleUI window puts the existing terminal window into windowing mode.

Any additional window that you display appears in the content portion of the screen window, usually on top of the window that you already opened. A side-by-side presentation of windows is also possible.

When you declare a window, you can set various properties. **Position**, for example, is the location relative to the upper left corner of the display; and **size**is the window's height and width in number of characters.

An example of a window declaration is as follows:

```
myWindow WINDOW
{name="myWindow", position = [2,2],
 size = [18,75], color = red, hasborder=yes};
```

You display a window by using an EGL function whose name begins with *ConsoleLib.openWindow*. If you have not displayed a window when presenting other data, EGL provides a window for you.

Prompt

A prompt is a one-line statement that elicits user input. A declaration of a prompt is as follows:

```
  myPrompt Prompt { message = "Type your ID: "};
```

You display a prompt by including the variable in an **openUI** statement, which binds the prompt to a variable of type String, but only for input. You can configure the prompt to accept a single character or a string.

ConsoleField

A consoleField is an on-screen field that is declared in the context of a console form (as described later). The next example declares a consoleField whose content can vary at run time:

```
  myField ConsoleField (
    name="myFieldName",
    position=[1,31],
    fieldLen=20,
    binding = "myVariable" );
```

To specify constant text, use an asterisk (*) in place of the variable name, as in the following example:

```
*    ConsoleField
     { position=[2,5], value="Title:  " };
```

It is highly recommended that when you declare a named consoleField, you use the same name for the consoleField and for the value of the name attribute within the consoleField. However, different names are valid for those two uses. You would reference the consoleField name (like *myField*) when access to the consoleField is resolved at generation time. You would reference the name-attribute value (like *myFieldName*) when access is resolved at run time, as when the consoleField is used to define an event in the **openUI** statement.

ConsoleForm

A consoleForm is primarily a set of consoleFields. To make a consoleForm active, you invoke the system function **ConsoleLib.displayForm**. To display a read-only consoleForm, for example, you can do as follows:

1. Invoke **ConsoleLib.displayForm**
2. Invoke the system function **ConsoleLib.getKey** to wait for a user keystroke

To allow the user to write to a consoleField, do this instead:

1. Invoke **ConsoleLib.displayForm**
2. Issue an **openUI** statement that references either the displayed consoleForm or specific consoleFields in the consoleForm.

The consoleForm is a record of subtype ConsoleForm and can include not only consoleFields, but any of the fields that are valid in any EGL record.

To allow for user interaction with an on-screen table of consoleFields, do this:

1. In the consoleForm, declare an arrayDictionary that in turn references consoleField arrays that are also declared in the consoleForm
2. Use that arrayDictionary in an **openUI** statement

To allow for user interaction with only a subset of consoleFields in the consoleForm, you can list the consoleFields in the **openUI** statement, either in explicitly or by referencing a dictionary. Like the arrayDictionary, the dictionary is declared in the consoleForm and references consoleFields that are also declared in the consoleForm.

EGL does not display any primitive variable that you declare in the consoleForm. You can use such a variable to bind a consoleField, as you can use a variable declared outside of the consoleForm.

In general, you create consoleForm bindings in either of two ways:

- By setting a default binding when you declare the consoleForm.
- By setting a binding when you code the **openUI** statement.

Any binding specified in the **openUI** statement overrides the default binding in total; none of the consoleForm-declaration bindings remain.

If you use the **openUI** statement to bind variables, one option is to use the statement property **isConstruct**, which acts as follows:

- Formats user input into a string appropriate to an SQL WHERE clause

- Places that string into a single variable so you can easily code an SQL SELECT statement that retrieves user-requested data from a relational database, as when you code an EGL **prepare** statement

For details on the property **isConstruct**, see *OpenUI statement*.

Tab order is the order in which the user tabs from one consoleField to another. By default, the tab order is the order of the consoleFields in the consoleForm declaration. If you provide a list of consoleFields in an **openUI** statement, the tab order is the order of consoleFields in that statement; similarly, if you provide a dictionary or arrayDictionary in an **openUI** statement, the tab order is the order of consoleFields in the declaration of the dictionary or arrayDictionary.

By default, the user exits a consoleForm-related **openUI** statement by pressing the **Esc** key.

Menu

A menu is a set of labels displayed horizontally. One label is for the menu as a whole and one for each menuItem in the menu. To ensure that a response occurs when the user selects a particular menuItem, you reference the menu as a whole in the **openUI** statement and reference the menuItem in an OnEvent clause of that statement.

MenuItem

A menuItem displays a label and is used as described in the previous section.

Related concepts
"ArrayDictionary" on page 81
"Console user interface" on page 165
"Dictionary" on page 77

Related reference
"ConsoleField properties and fields" on page 429
"ConsoleForm properties in EGL consoleUI" on page 442
"EGL library ConsoleLib" on page 735
"ConsoleUI screen options for UNIX" on page 171
"Menu fields in EGL consoleUI" on page 443
"MenuItem fields in EGL consoleUI" on page 444
"openUI" on page 602
"PresentationAttributes fields in EGL consoleUI" on page 446
"Prompt fields in EGL consoleUI" on page 447
"Window fields in EGL consoleUI" on page 449

Related tasks
"Creating an interface with consoleUI" on page 166

Use of new in ConsoleUI

When you create an EGL program that uses consoleUI, every variable of type Menu, MenuItem, Prompt, and Window is a *reference variable*, which contains a memory address that refers to a value stored outside the variable.

You can declare a reference variable as you declare any other, as in this example:

```
myPrompt Prompt { message = "Type your ID: "};
```

Alternatively, you can declare a reference variable and initialize it with the reserved word **new**, as in this example:

```
myPrompt Prompt = new Prompt { message = "Type your ID: "};
```

When you are declaring variables, the difference between the two formats has little practical effect; but when you code the openUI statement, the word **new** provides a coding convenience, as shown in *openUI*.

The general syntax for **new** is as follows:

```
new partName
```

partName
One of the following words, which refers to a particular kind of part:

- Menu
- MenuItem
- Prompt
- Window

For details on other implications of reference variables, see *Reference compatibility in EGL*.

Related concepts
"Console user interface" on page 165

Related reference
"ConsoleUI parts and related variables" on page 167
"openUI" on page 602
"Reference compatibility in EGL" on page 718

Related tasks
"Creating an interface with consoleUI" on page 166

ConsoleUI screen options for UNIX

EGL users on the supported UNIX platforms have the ability to run their ConsoleUI application using either a graphical display mode or a UNIX curses mode.

Graphical display mode

To run a ConsoleUI application in graphical display mode, you must make sure that the EGL curses library is not located in the Library Path environment variable of your running shell. This is the default mode.

UNIX curses mode

To run a ConsoleUI application in UNIX curses mode, you must have the appropriate platform-specific EGL curses library in the Library Path environment variable of your running shell. The EGL curses libraries must be downloaded from the EGL Support website.

To download the EGL curses library:

1. Locate the appropriate EGL Support website.
   - The URL for Rational Application Developer is:

```
http://www3.software.ibm.com/ibmdl/pub/software/rationalsdp/rad/60/redist
```

- The URL for Rational Web Developer is:

```
http://www3.software.ibm.com/ibmdl/pub/software/rationalsdp/rwd/60/redist
```

.

2. Download the **EGLRuntimesV60IFix001.zip** file to your preferred directory.
3. Unzip **EGLRuntimesV60IFix001.zip** to identify the following files:
   - AIX: **EGLRuntimes/Aix/bin/libCursesCanvas6.so**
   - Linux: **EGLRuntimes/Linux/bin/libCursesCanvas6.so**
4. Insert the appropriate EGL curses library in the Library Path environment variable.

   AIX: Set the '**LIBPATH**' Library Path environment variable using the following bourne-shell:

   ```
   "LIBPATH=$INSTDIR/aix; export LIBPATH"
   ```

   Linux: Set the '**LD_LIBRARY_PATH**' Library Path environment variable using the following bourne-shell:

   ```
   "LD_LIBRARY_PATH=$INSTDIR/aix; export LD_LIBRARY_PATH"
   ```

Related concepts
"Console user interface" on page 165

Related reference
"EGL library ConsoleLib" on page 735
"ConsoleUI parts and related variables" on page 167
"openUI" on page 602

Related tasks
"Creating an interface with consoleUI" on page 166

Creating an EGL Web application

Web support

EGL provides support for Web-based applications in the following ways:

- You can develop a *PageHandler*, which is a logic part whose functions are each invoked by a specific user action at a Web page. Generation of this logic part also can provide a JavaServer Faces JSP for customization.
- You can provide functionality that is common to several Web pages, as when each page in an application displays a button that the user clicks to log out. The user's click in this case can invoke an EGL program, which acts as a common subroutine.
- Finally, when you work in the WebSphere Page Designer, you can customize JavaServer Faces JSPs and can affect PageHandlers, as described in *Page Designer Support for EGL*.

Related concepts

"PageHandler" on page 180

"WebSphere Application Server and EGL" on page 321

Related tasks
"Starting a Web application on the local machine" on page 320

Related reference
"Page Designer support for EGL" on page 178

Creating a single-table EGL Web application

EGL Data Parts and Pages wizard

The EGL Data Parts and Pages wizard gives you a convenient way to create a Web-based utility that lets you maintain a specific table in a relational database.

The wizard creates these entities:

- A set of PageHandlers that you later generate into a set of parts that run under Java Server Faces
- An SQL record part, as well as the related data-item parts and library-based function parts
- A set of JSP files that provide the following Web pages:
  - A *selection condition page*, which accepts selection criteria from the user
  - A *list page*, which displays multiple rows, based on the user's criteria
  - A *create detail page*, which lets the user display or insert one row
  - A *detail page*, which lets the user display, update, or delete one row

  The user first encounters the selection criteria page; but if you fail to specify the information needed for that page, the user first encounters the list page, which provides access (in this situation) to every row in the table.

When working in the wizard, you can do as follows:

- Customize the Web pages described earlier, as by varying the fields displayed or including links from one page to another.
- Specify the SQL-record key fields that are used to create, read, update, or delete a row from a given database table or view.
- Customize explicit SQL statements for creating, reading, or updating a row. (The SQL statement for deleting a row cannot be customized.)
- Specify the SQL-record key fields that are used to select a set of rows from a given database or view.
- Customize an explicit SQL statement for selecting a set of rows.
- Validate and run each SQL statement

The output includes these files:

- An HTML file (index.html) that invokes the Web application.
- A set of JSP files that provide the Web pages described earlier.
- An EGL source file that contains all the data-item parts referenced by the structure items in the SQL record parts.
- For each SQL record part, the wizard also produces two files: one for the record part itself, one for the related, library-based functions. You can reduce the number of files if you select the **Record and library in the same file** check box.

You can customize the Web-based utility after the wizard creates it.

Related concepts
"Java program, PageHandler, and library" on page 306
"SQL support" on page 213

Related tasks
"Creating a single-table EGL Web application"
"Creating, editing, or deleting a database connection for the EGL wizards" on page 239
"Customizing SQL statements in the EGL wizards" on page 240
"Defining Web pages in the EGL Data Parts and Pages wizard" on page 176

Creating a single-table EGL Web application

To create an EGL Web application from a single relational database table, do as follows:

1. Select **File** > **New** >**Other...**. A dialog is displayed for selecting a wizard.
2. Expand **EGL** and double-click **EGL Data Parts and Pages**. The EGL Data Parts and Pages dialog is displayed.
3. Enter an EGL Web project name, or select an existing project from the drop-down list. The EGL parts will be generated into this project.
4. Select an existing database connection from the drop-down list or establish a new database connection--
   - To establish a new database connection, click **Add** and follow the directions in the help topic *Database connection page*, which you can access by pressing F1
   - For details on editing or deleting a database connection, see *Creating, editing, or deleting a database connection for the EGL wizards*

   When a connection is made to the database, a list of database tables is displayed.

5. If you do not want to accept the default EGL file name for data items, type a new file name.
6. In the **Select your data** field, click on the name of the database table of interest.
7. In the **Record name** field, either specify the name of the EGL record to be created or accept the default name.
8. If you want to include the library part and SQL record parts in the same file, select the check box.
9. To set additional fields to non-default values, click **Next**; otherwise, click **Finish**. The remaining steps assume that you clicked **Next**.
10. Select the key field to use when reading, updating, and deleting individual rows, then click the right arrow. To select multiple key fields, hold down the **Ctrl** key while clicking on different field names. To remove a key field from the list on the right, highlight the field name and click the left arrow.
11. Choose the selection condition field to use when selecting a set of rows, then click the right arrow. To select multiple fields, hold down the **Ctrl** key while clicking on different field names. To remove a field from the list on the right, highlight the field name and click the left arrow.
12. To customize the implicit SQL statements, see *Customizing SQL statements in the EGL wizards*. This option is not available for the EGL **delete** statement.
13. Click **Next**.
14. If you want to apply a template to the new Web pages, follow these steps:
    a. Select the **Select page template** check box.
    b. Select a page template type by clicking either **Sample page template** or **User-defined page template**.
    c. Click on the page template you want to use. You can either select a thumbnail or browse to the location of the template by clicking the **Browse** button.
15. Click **Next**.
16. To customize the Web pages, see *Defining Web pages in the EGL Data Parts and Pages wizard*.
17. Click **Next**.
18. The Generate the Web application screen is displayed, including (at the bottom) a list of the files and Web pages that will be produced:
    a. To change the name of the EGL Web project that will receive the EGL parts, type a project name in the **EGL Web project name** field or select a project from the related drop-down list.
    b. To specify the EGL and Java packages for a specific type of part (PageHandler, data, or library), type a package name in the related field or select a name from the related drop-down list.
    c. To change the name of the JSP and EGL files that are produced for a given Web page, click on the appropriate entry under **Web pages** and type the new name. Each file name includes any letters or numbers that you type, but excludes spaces and other characters.

    Type or select packages for the PageHandler parts, data parts, and library parts.
19. Click **Finish**.

Related concepts
"SQL support" on page 213

Related tasks
"Creating, editing, or deleting a database connection for the EGL wizards" on page 239
"Creating EGL data parts from relational database tables" on page 238
"Customizing SQL statements in the EGL wizards" on page 240
"Defining Web pages in the EGL Data Parts and Pages wizard"

Defining Web pages in the EGL Data Parts and Pages wizard

The EGL Data Parts and Pages wizard creates a Web application from a relational-database table. When you are working in this wizard, you can specify the following aspects of each type of Web page that is produced:

- Page title
- Style sheet
- Fields to display, including their order and properties
- Links to other pages

When you are working in the wizard dialog called *Define the Web pages of the application*, you can click the tabs to navigate between pages. Do as follows for each page (where possible):

1. Set the page title in the **Page title** field
2. Select a style sheet from a drop-down list in the **Style sheet** field
3. To see the effect of accepting the current page definition, click **Preview**.
4. If you wish to specify the number of rows to display on a page, select **Pagination** and assign the number of rows (a positive integer) to **Page size**.
5. Select the fields to be included on the page:
   a. To include a field, select the related check box. To select every field, click **All**.
   b. To exclude a field, clear the related check box. To exclude every field, click **None**.
   c. To change the display location of a field, click on the field, then use the Up and Down arrows to move the field to another location.
   d. To set properties for a field, click on the field then double-click the **Value** field in the Properties pane. Enter the value or, in some cases, select from a drop-down list.
6. Select the actions that the user can perform on the page:
   a. To include an action, select the related check box. To select every action, click **Select All**.

      The individual actions are **create**, **delete**, **extract**, **list**, and **read**:

      - **Create** links to the create detail page, where the user can display or insert one row. The option is present on the create detail page only to indicate that the user can create a row from that page.
      - **Delete** is available only on the detail page. This option deletes the record that has the key specified by the user.
      - **Extract** links to the selection condition page, which accepts selection criteria from the user. The option is present on the selection condition page only to indicate that the user can cause the return of a result set from that page.
      - **List** links to the list page, which displays multiple rows in accordance with the user's criteria.
      - **Read** is available only on the create detail page. This option displays the record that has the key specified by the user.

- **Update** is available only on the detail page. This option updates the record that was modified by the user.

b. To exclude an action, clear the related check box. To exclude every action, click **None**.

   If a given action is always available, you cannot clear the check box.

c. To set the Web-page label for an action, click on the action name, then double-click the **Value** field in the Properties pane and enter a value.

Related concepts
"SQL support" on page 213
"EGL Data Parts and Pages wizard" on page 173

Related tasks
"Creating a single-table EGL Web application" on page 174
"Creating EGL data parts from relational database tables" on page 238
"Creating, editing, or deleting a database connection for the EGL wizards" on page 239
"Setting EGL preferences" on page 107
"Starting a Web application on the local machine" on page 320

Creating an EGL pageHandler part

A pageHandler part controls a user's run-time interaction with a Web page by providing data and services to a Java Server Faces JSP. When you create a JSP, a pageHandler part is automatically created for you in a package called *pagehandlers* within the *EGLSource* folder. The name of the pageHandler part is the same as the corresponding JSP, but with a .egl file extension.

Optionally, you can create a pageHandler part and let the system automatically add the JSP to your project, provided a JSP file with the same name does not already exist within the EGL Web project. To create an EGL pageHandler part, do as follows:

1. If your EGL Web project does not contain a package named *pagehandlers*, you must create one. Page Designer requires that all pageHandler parts reside in a package named *pagehandlers*. For details on creating packages, see *Creating an EGL package*.
2. Identify an EGL file within the *pagehandlers* package to contain the pageHandler part. Open the file in the EGL editor. You must create an EGL file if you do not already have one.
3. Type the specifics of the pageHandler part according to EGL syntax (for details, see *PageHandler part in EGL source format*). You can use content assist to place an outline of the pageHandler part syntax in the file.
4. Save the EGL file.

Related concepts
"EGL projects, packages, and files" on page 13
"PageHandler" on page 180

Related tasks
"Creating an EGL package" on page 120
"Creating an EGL source file" on page 120
"Using the EGL templates with content assist" on page 121
"Using the Quick Edit view for PageHandler code" on page 187

Related reference
"Content assist in EGL" on page 471
"Naming conventions" on page 652
"PageHandler part in EGL source format" on page 659

Page Designer support for EGL

When you create a JSP file in an EGL Web project, EGL automatically creates a PageHandler, and that PageHandler includes skeletal EGL code for you to customize. Then, in Page Designer, do as follows:

1. Drag components from the palette to a JSP
2. Use the Attributes view to set component-specific characteristics such as color and to set up *bindings*, which are relationships between components and either data or logic

You can do work that is specific to EGL:

- Create EGL variables and place them in an existing PageHandler.
- Bind PageHandler items to JSP user-interface components.
- Bind PageHandler functions to buttons and hyperlink controls. The functions act as event handlers.

When you use the source tab in Page Designer, you can manually bind components in a JSP file (specifically, in a JavaServer Faces file) to data areas and functions in a PageHandler. Although EGL is not case sensitive, EGL names referenced in the JSP file must have the same case as the EGL variable or function declaration; and if you fail to maintain an exact match, a JavaServer Faces error occurs. It is recommended that you avoid changing the case of an EGL variable or function after you bind that variable or function to a JSP field.

For further details on naming issues, see *Changes to EGL identifiers in JSP files and generated Java beans*.

Binding components to data areas in the PageHandler

Most components on the JSP have a one-to-one correspondence with data. A text box, for example, shows the content of the EGL item to which the text box is bound. An input text box also updates the EGL item if the user changes the data.

A more complex situation occurs when you are specifying a check box group, list box, radio button group, or combo box. In those cases, you need two different kinds of bindings:

- One is to bind the component to the text that you want to display to the user. An example is the text of an item in a list box.
- One is to bind the component to a PageHandler data area that receives a value to indicate the user's choice. You might create a data item, for example, to receive the numeric index of a user-selected list-box item.

In the Properties view, you can follow either of two procedures to bind the component to the text that the user sees:

- You can use **Add Choice** to indicate that the component is associated with a single character string, which may be specified explicitly or by identifying a PageHandler item

- You can use **Add Set of Choices** to indicate that the component is associated with a list of character strings, which may be specified explicitly or by identifying a PageHandler area such as a data table or an array of character items

Alternatively, you can bind a single-select component (combo box, single-select list box, or radio button group) to an array of character items by dragging the array from the Page Data view to the component.

To bind a component to a data area that will receive a value indicating the user's choice, you can work in either the Page Data view or the Properties view. The procedure is the same as when you are binding any component, even a simple text box.

If the value can be only one of two alternatives, you can bind the component to an EGL item for which the item property **boolean** is set to *yes*. The component populates the item with one of two values:

- For a character item, the value is **Y** (for yes) or **N** (for no)
- For a numeric item, the value is **1** (for yes) or **0** (for no)

When a check box is displayed, the status (whether checked or not) is dependent on the value in the bound item.

For details on the properties that can be applied to data items in the PageHandler, see *Page item properties*.

Binding components to functions

After you drag a command button or a command hyperlink to the page surface, you can bind that component to an existing EGL function or to an event handler that the Page Designer creates:

- You can bind the component to an existing event handler in any of these ways--
  - By dragging the EGL function from the Actions node in the Page Data view to the component, as is recommended
  - By opening the component in the Quick Edit view
  - By right-clicking on the component and selecting **Edit Faces Command Event**
- You can cause Page Designer to create a new event handler either when you open the component in the Quick Edit view or when you right-click on the component and select **Edit Faces Command Event**

If the Page Designer creates an event handler in the PageHandler and gives you access to that PageHandler function, the name of the function is the tool-assigned button ID plus the string "Action". If the name is not unique to the PageHandler, the Page Designer appends a number to the function name.

Related concepts
"PageHandler" on page 180

Related tasks
"Creating an EGL field and associating it with a Faces JSP" on page 184
"Associating an EGL record with a Faces JSP" on page 185

"Using the Quick Edit view for PageHandler code" on page 187

Related reference
"PageHandler part in EGL source format" on page 659
"PageHandler field properties" on page 665

PageHandler

An EGL *PageHandler* is an example of *page code*; it controls a user's run-time interaction with a Web page and can do any of these tasks:

- Assign data values for submission to a JSP file. Those values are ultimately displayed on a Web page.
- Change the data returned from the user or from a called program.
- Forward control to another JSP file.

You can work most easily by customizing a JSP file and creating the PageHandler in Page Designer; for details, see *Page Designer support for EGL*.

The PageHandler itself includes variables and the following kinds of logic--

- An OnPageLoad function, which is invoked the first time that the JSP renders the Web page
- A set of event handler functions, each of which is invoked in response to a specific user action (specifically, by the user clicking a button or hypertext link)
- Optionally, validation functions that are used to validate Web-page input fields
- Private functions that can be invoked only by other PageHandler functions

The variables in the PageHandler are accessed in two ways:

- The run-time environment accesses the data automatically. If a field in the JSP is *bound* to an item in the PageHandler, the result is as follows--
  - After the OnPageLoad function runs and before the Web page is displayed, each PageHandler item value is written to the JSP field to which the data is bound.
  - When the user submits a form in which bound JSP fields reside, the value in each field of the submitted form is copied to the associated PageHandler item. Only then is control passed to an event handler. (However, this description does not include the validation steps, which are covered later in this topic.)
- The event handlers and the OnPageLoad function also can interact with the data, as well as with data stores (such as SQL databases) and with called programs.

The pageHandler part should be simple. Although the part might include light-weight data validations such as range checks, you are advised to invoke other programs to perform complex business logic. Database access, for example, should be reserved to a called program.

Output associated with a PageHandler

When you save a PageHandler, EGL places a JSP file in the project folder WebContent\WEB-INF, but only in this case:

- You assigned a value to the PageHandler **view** property, which specifies a JSP file name
- The folder WebContent\WEB-INF does not contain a JSP file of the specified name

EGL never overwrites a JSP file.

If a Workbench preference is set to automatic build on save, PageHandler generation occurs whenever you save the PageHandler. In any case, when you generate a PageHandler, the output is composed of the following objects:

- The *page bean* is a Java class that contains data and that provides initialization, data validation, and event-handling services for the Web page.
- A <managed-bean> element is placed in the JSF configuration file in your project, to identify the page bean at run time.
- A <navigation-rule> element is created in the JSF application configuration file to associate a JSF outcome (the name of the PageHandler) with the JSP file to be invoked.
- A JSP file, in the same situation as when you save the PageHandler.

All data tables and records that are used by the part handler are also generated.

Validation

If the JSP-based JSF tags perform data conversion, validation, or event handling, the JSF run time does the necessary processing as soon as the user submits the Web page. If errors are found, the JSF run time may re-display the page without passing control to the PageHandler. If the PageHandler receives control, however, it may conduct a set of EGL-based validations.

The EGL-based validations occur if you specify the following details when you declare the PageHandler:

- The element edits (such as minimum input length) for individual input fields.
- The type-based edits (character, numeric) for individual fields.
- The DataTable edits (range, match valid, and match invalid) for individual input fields, as explained in *DataTable part*.
- The edit functions for individual input fields.
- The edit function for the PageHandler as a whole.

The PageHandler oversees the edits in the following order, but only for items whose values were changed by the user:

1. All the elementary and type-based edits, even if some fail
2. (If the prior edits were successful) all the table edits, even if some fail
3. (If the prior edits were successful) all the field-edit functions, even if some fail
4. (If all prior edits were successful) the pageHandler edit function

The page-item property **validationOrder** defines the order in which both the individual input fields are edited and the field validator functions are invoked.

If no validationOrder properties are specified, the default is the order of items defined in the PageHandler, from top to bottom. If validationOrder is defined for some but not all of the items in a PageHandler, validation of all items with the validationOrder property occurs first, in the specified order. Then, validation of items without the validationOrder property occurs in the order of items in the PageHandler, from top to bottom.

Run-time scenario

This section gives a technical overview of the run-time interaction of user and Web application server.

When the user invokes a JSP that is supported by a PageHandler, the following steps occur:

1. The Web application server initializes the environment--
   a. Constructs a session object to retain data that the user-accessed applications need across multiple interactions
   b. Constructs a request object to retain data on the user's current interaction
   c. Invokes the JSP
2. The JSP processes a series of JSF tags to construct a Web page--
   a. Creates an instance of the PageHandler, causes the onPageLoad function (if any) to be invoked with user-specified arguments, and places the PageHandler into the request object
   b. Accesses data stored in the request and session objects, for inclusion in the Web page

   Note: The pageHandler part has a property called onPageLoadFunction, which identifies the PageHandler function that is invoked at JSP startup. The function automatically retrieves any user-supplied arguments that were passed to it; can call other code; and can place additional data in the request or session object of the Web application server; but the function can neither forward control to another page nor cause an error message to be displayed when the page is first presented to the user.
3. The JSP submits the Web page to the user, and the Web application server destroys the response object, leaving the session object and the JSP.

If the user supplies data in the on-screen fields associated with an HTML <FORM> tag and submits the form, the following steps occur:

1. The Web application server re-initializes the environment--
   a. Constructs a request object
   b. Places the received data for the submitted form into the page bean, for validation
   c. Re-invokes the JSP
2. The JSP processes a series of JSF tags to store the received data in the page bean.
3. The run-time PageHandler validates data:
   a. Does relatively elementary edits (such as minimum input length), as specified in the pageHandler data declarations
   b. Invokes any item-specific validation functions, as specified in the pageHandler data declarations
   c. Invokes a pageHandler validator function, as is needed if you wish to validate one field at least partially on the basis of the content of another field

   (For details on validation, see the previous section.)
4. If an error occurs, the EGL run time places errors on a JSF queue, and the JSP re-displays the Web page with embedded messages. If no error occurs, however, the result is as follows:
   a. Data stored in the page bean is written to the record bean
   b. Subsequent processing is determined by an event handler, which is identified in the JSF tag that is associated with the user-clicked button or hyperlink.

The event handler may forward processing to a JSF label, which identifies a mapping in a run-time JSF-based configuration file. The mapping in turn identifies the object to invoke, whether a JSP (usually one associated with an EGL PageHandler) or a servlet.

Related concepts
"References to parts" on page 20
"Web support" on page 173

Related reference
"Page Designer support for EGL" on page 178
"PageHandler part in EGL source format" on page 659
"PageHandler field properties" on page 665

JavaServer Faces controls and EGL

JavaServer Faces (JSF) is a server-side user interface component framework. In simple terms, JSF is a set of tools and components that allow you to create interfaces for Web pages. JSF components can display data on a Web page and accept input from the user.

This topic explains the relationship between JSF components and EGL. For more details on JSF, see Creating Faces applications - overview. To see a related tutorial, click **Help > Tutorials Gallery**; expand **Do and Learn**; and select *Display dynamic information on Web pages with JavaServer Faces*.

Two methods are available for displaying EGL data on a Web page using JSF controls:

- You can create JSF controls automatically from data items in the Page Data view or from data items you create in the Page Designer view. To use this method, select the check box named **Add controls to display the EGL element on the Web page** as you follow the directions in *Associating an EGL record with a Faces JSP* or *Creating an EGL data item and associating it with a Faces JSP*.
- You can add JSF controls manually and bind them to data in the Page Data view. This method allows you to customize the layout of the JSF controls on the page, rather than using the default layout. To use this method, see one of the following topics:
  - *Binding a JavaServer Faces input or output component to an EGL PageHandler*
  - *Binding a JavaServer Faces check box component to an EGL PageHandler*
  - *Binding a JavaServer Faces single-selection component to an EGL PageHandler*
  - *Binding a JavaServer Faces multiple-selection component to an EGL PageHandler*

You can also bind EGL functions in PageHandlers to JSF controls. See *Binding a JavaServer Faces command component to an EGL PageHandler*.

Related tasks
"Creating an EGL field and associating it with a Faces JSP" on page 184
"Associating an EGL record with a Faces JSP" on page 185
"Binding a JavaServer Faces command component to an EGL PageHandler" on page 186
"Binding a JavaServer Faces input or output component to an EGL PageHandler" on page 187
"Binding a JavaServer Faces check box component to an EGL PageHandler" on page 188
"Binding a JavaServer Faces single-selection component to an EGL PageHandler" on page 189
"Binding a JavaServer Faces multiple-selection component to an EGL PageHandler" on page 190

Related reference
"Page Designer support for EGL" on page 178

Creating an EGL field and associating it with a Faces JSP

To create an EGL primitive field and associate it with a Faces JSP, do as follows:

1. Open a Faces JSP file in the Page Designer. To open a JSP file, double-click on the JSP file in the Project Explorer. The JSP opens in the Page Designer. Click the **Design** tab to access the Design view.

   Note: You can access the related PageHandler by right clicking in the Design view (or Source view) and clicking **Edit Page Code**.

   .
2. From the **Window** menu, select **Show View** > **Other** >**Basic** > **Palette**.
3. In the Palette view, click the **EGL** drawer to display the EGL data object types.
4. Drag **New Field** from the palette to the JSP. The Create a New EGL Data Field dialog is displayed.
5. Type a field name in the **Name** field.
6. Select the field type from the **Type** drop-down list and, if you need to specify the field's primitive characteristics (length and possibly decimals), type the information in the **Dimensions** text box. Default masks are used if you declare items of the following types:
   - Date (mask *yyyymmdd*)
   - Time (mask *hhmmss*)
   - Timestamp (mask *yyyymmddhhmmss*)

   If you wish to specify a DataItem part as the type, select **DataItem**, which is the last value in the list. In this case, the Select a DataItem part dialog is displayed, and you either select a DataItem part from the list or type the name, then click **OK**.
7. If you are creating an array of data items, select the **Array** check box and type an integer in the **Size** text box.
8. If you do not want to include the field on the page, clear the check box named **Add controls to display the EGL element on the Web page** and click **OK**. The field is now available in the Page Data view. You can add it to the JSP file later by dragging it from the Page Data view to the JSP.
9. If you want to include the fields in the JSP file, follow these additional steps:
10. Select the check box named **Add controls to display the EGL element on the Web page** and click **OK**. The Insert Control window opens.
11. In the Insert Control window, select the radio button that indicates your intended use of the field:
    - For output (**Displaying an existing record**)
    - For input or output (**Updating an existing record**)
    - For input (**Creating a new record**)

    Your choice affects the types of controls that are available.
12. To change the field label, select the label that is displayed next to the field name, then type the new content.
13. To select a control type different from the one identified, select a type from the **Control Type** list.

14. If you click **Options**, the Options dialog is displayed, and the specific options that are available depend on whether you are using the field for input, for output, or for both. One option in any case is to include or exclude the JSF tag <h:outputLabel> around field labels.

    When you complete your work in the Options dialog, click **OK**.
15. Click **Finish**.

Related reference
"Page Designer support for EGL" on page 178
"Primitive types" on page 31

Associating an EGL record with a Faces JSP

To associate an EGL record with a Faces JSP, do as follows:

1. Open a Faces JSP file in the Page Designer. If you do not have a JSP file open, double-click on the JSP file in the Project Explorer. The JSP file opens in the Page Designer. Click the **Design** tab to access the Design view.

   Note: You can access the related PageHandler by right clicking in the Design view (or Source view) and clicking **Edit Page Code**.
2. From the **Window** menu, select **Show View** > **Other** >**Basic** > **Palette**.
3. In the Palette view, click the **EGL** drawer to display the EGL data object types.
4. Drag **Record** from the palette to the JSP page. The Select a Record Part dialog is displayed.
5. Select a record from the list.
6. Specify the field name or accept the default value, which is the record-part name.
7. If you are declaring an array of records, select the **Array** check box and type an integer in the **Size** text box.
8. If you do not want to include the record on the page, clear the check box named **Add controls to display the EGL element on the Web page** and click **OK**. The record is now available in the Page Data view. You can add it to the JSP file later by dragging it from the Page Data view to the JSP.
9. If you want to include the fields in the JSP file, follow these additional steps:
10. Select the check box for **Add controls to display the EGL element on the Web page** and click **OK**. The Insert Control window opens.
11. In the Insert Control window, select the radio button that indicates your intended use of the field:
    - For output (**Displaying an existing record**)
    - For input or output (**Updating an existing record**)
    - For input (**Creating a new record**)

    Your choice affects the types of controls that are available.
12. To change the order of fields, use the up and down arrows.
13. If you wish to select only a subset of the listed fields, click **None** and select the fields of interest. To select all fields instead, click **All**.
14. Do as follows for each field:
    a. To exclude a field, clear the related check box. To include the field, ensure that the check box is selected.
    b. To change the field label, select the label that is displayed next to the field name, then type the new content.

c. To select a control type different from the one identified (if possible), select from a list of types.

15. If you click **Options**, the Options dialog is displayed, and the specific options that are available depend on whether you are using fields for input, for output, or for both. One option in any case is to include or exclude the JSF tag <h:outputLabel> around field labels.

    When you complete your work in the Options dialog, click **OK**.

16. Click **Finish**.

Related concepts
"Record parts" on page 124

Related reference
"Page Designer support for EGL" on page 178

Binding a JavaServer Faces command component to an EGL PageHandler

To bind a JavaServer Faces command component (button or hypertext link) to an EGL PageHandler function, do as follows:

1. Open a Faces JSP file in the Page Designer. If you do not have a JSP file open, double-click on the JSP file in the Project Explorer. The JSP opens in the Page Designer. Click the **Design** tab to access the Design view.
2. From the **Window** menu, select **Show View** > **Other** >**Basic** > **Palette**.
3. In the Palette view, click the **Faces Components** drawer to display the Faces Components object types.
4. Drag a command component from the palette to the JSP. Command components have the word **Command** in the label. The component object is placed on the JSP.
5. Bind the an event handler to the command component using one of these methods:
   - To bind the component to an existing event handler, drag the event handler from the Actions node in the Page Data view to the component object on the JSP.
   - To a create a new event handler that is bound to the component:
     a. Right-click on the component and click **Edit Events** from the popup menu.
     b. Using the Quick Edit view, enter the EGL code for the event handler. For details on using the Quick Edit view, see *Using Quick Edit view for PageHandler code*.

     The event handler is visible in the Page Data view and a corresponding function is added to the PageHandler. For details, see *PageHandler part in EGL source format*.

Related concepts
"PageHandler" on page 180

Related tasks
"Creating an EGL pageHandler part" on page 177
"Using the Quick Edit view for PageHandler code" on page 187

Related reference
"Page Designer support for EGL" on page 178
"PageHandler part in EGL source format" on page 659

Using the Quick Edit view for PageHandler code

The Quick Edit view allows you to maintain EGL PageHandler code for JSP server events without opening the PageHandler file. To use the Quick Edit view, do as follows:

1. Open a JSP file in the Page Designer. If you do not have a file open, double-click on the JSP file in the Project Explorer. The JSP opens in the Page Designer. Click the **Design** tab to access the Design view.
2. Right-click in the Page Designer, then select **Edit Events**. The Quick Edit view opens.
3. Follow these steps to maintain PageHandler functions for command components:
   a. Select a command component in the JSP.
   b. If the command component already has a PageHandler function associated with it, the function displays in the script editor (right pane) of the Quick Edit view. Any changes you make to the code are reflected in the PageHandler.
   c. To create a PageHandler function for the selected command component, click **Command** in the event pane (left pane) of the Quick Edit view, then click in the script editor (right pane) of the Quick Edit view. The function displays. Type the PageHandler code for the function.
4. Follow these steps to maintain the onPageLoad function:
   a. Click inside the JSP.
   b. Click **onPageLoad** in the event pane (left pane) of the Quick Edit view.
   c. The onPageLoad function displays in the script editor (right pane) of the Quick Edit view. Any changes you make to the code are reflected in the PageHandler.

Related concepts
"PageHandler" on page 180

Related reference
"PageHandler part in EGL source format" on page 659

Binding a JavaServer Faces input or output component to an EGL PageHandler

To bind a JavaServer Faces input or output component to an existing EGL PageHandler data area, do as follows:

1. Open a Faces JSP file in the Page Designer. If you do not have a JSP file open, double-click on the JSP file in the Project Explorer. The JSP opens in the Page Designer. Click the **Design** tab to access the Design view.
2. From the **Window** menu, select **Show View** > **Other** >**Basic** > **Palette**.
3. In the Palette view, click the **Faces Components** drawer to display the Faces Components object types.
4. Drag an input or output component from the palette to the JSP. Input and output components have the words **Input** and **Output** in the labels. The component object is placed on the JSP.

5. To bind the component to an existing PageHandler data area, do one of the following:
   - Drag the data area from the Page Data view to the component object on the JSP.
   - Select the component object on the JSP, then right-click the data area in the Page Data view and select **Bind to 'component name'**.
   - Select the component object on the JSP. Click the button next to the **Value** field of the Properties view, then select a data area from the Select Page Data Object list and click **OK**.

Related concepts
"PageHandler" on page 180

Related tasks
"Creating an EGL pageHandler part" on page 177

Related reference
"Page Designer support for EGL" on page 178
"PageHandler part in EGL source format" on page 659

Binding a JavaServer Faces check box component to an EGL PageHandler

A JavaServer Faces check box component is unique in that the data area to which it is bound must have the item property **isBoolean** (formerly the **boolean** property) set to *yes*. Examples of boolean data area declarations are as follows:

```
DataItem CharacterBooleanItem char(1)
{
   value = "N",
   isBoolean = yes
}
end

DataItem NumericBooleanItem smallInt
{
   value = "0",
   isBoolean = yes
}
end
```

To bind a JavaServer Faces check box component to an existing EGL PageHandler data area, do as follows:

1. Open a Faces JSP file in the Page Designer. If you do not have a JSP file open, double-click on the JSP file in the Project Explorer. The JSP opens in the Page Designer. Click the **Design** tab to access the Design view.
2. From the **Window** menu, select **Show View** > **Other** >**Basic** > **Palette**.
3. In the Palette view, click the **Faces Components** drawer to display the Faces Components object types.
4. Drag a check box component from the palette to the JSP. The component object is placed on the JSP.
5. To bind the component to an existing PageHandler data area, do one of the following:
   - Drag the data area from the Page Data view to the component object on the JSP.
   - Select the component object on the JSP, then right-click the data area in the Page Data view and select **Bind to 'component name'**.

- Select the component object on the JSP. Click the button next to the **Value** field in the Properties view, then select a data area from the Select Page Data Object list and click **OK**.

Related concepts
"PageHandler" on page 180

Related tasks
"Creating an EGL pageHandler part" on page 177

Related reference
"Page Designer support for EGL" on page 178
"PageHandler part in EGL source format" on page 659

Binding a JavaServer Faces single-selection component to an EGL PageHandler

A single-selection component allows a user to make one selection from a list of values. The user's selection is stored in a PageHandler data area. Radio buttons, single-select list boxes, and combo boxes are single-selection JavaServer Faces components.

A binding is a relationship between the component and a data area. The data area must be declared in the PageHandler before you can bind a component to it. A single-selection component needs two different kinds of bindings:

- A binding to one or more data areas that contain the values from which the user can make a selection
- A binding to a data area that will receive the user's selection

To bind a JavaServer Faces single-selection component to existing EGL PageHandler data areas, do as follows:

1. Open a Faces JSP file in the Page Designer. If you do not have a JSP file open, double-click on the JSP file in the Project Explorer. The JSP opens in the Page Designer. Click the **Design** tab to access the Design view.
2. From the **Window** menu, select **Show View** > **Other** >**Basic** > **Palette**.
3. In the Palette view, click the **Faces Components** drawer to display the Faces Components object types.
4. Drag a single-selection component from the palette to the JSP. The component object is placed on the JSP.
5. To bind the component to one or more PageHandler data areas that contain the values you want to display to the user, do one of the following procedures:
   - You can bind the component to individual PageHandler data areas, each of which contains one list item. Perform the following procedure for each data area:
     a. Select the object component on the JSP.
     b. In the Properties view, click **Add Choice**. The Name and Value fields are populated with default values.
     c. Click the **Name** field, then type the text that you want to display to the user.
     d. Click the **Value** field, then click the button next to the **Value** field. Select an individual data area from the Select Page Data Object list and click **OK**. This data area holds the value that will be moved to the receiving data area.

- You can bind the component to a PageHandler array data area that contains the values you want to display to the user. Perform the following procedure to bind the component to an array data area:
  a. Select the object component on the JSP.
  b. In the Properties view, click **Add Set of Choices**. The Name and Value fields are populated with default values.
  c. Click the **Value** field, then click the button next to the **Value** field. Select an array data area from the Page Data Object list and click **OK**. The values in the array data area are the values that will be displayed to the user. The properties described later determine whether the values in the array data area or their equivalent index values will be moved to the receiving data area.

6. If you are using an array data area to supply the values displayed to the user, you must define the receiving data area with two properties: **selectFromListItem** and **selectType**. The selectFromListItem property points to the array that holds the list items. The selectType property indicates whether the receiving data area is to be populated with a text value or an index value. Examples of receiving data areas are as follows:

```
colorSelected char(10)
{selectFromListItem = "colorsArray",
 selectType = value};

colorSelectIdx smallInt
{selectFromListItem = "colorsArray",
selectType = index};
```

7. To bind the component to a PageHandler data area that will receive the user's selection, do one of the following:
   - Drag the data area from the Page Data view to the component object on the JSP.
   - Select the component object on the JSP, then right-click the data area in the Page Data view and select **Bind to 'component name'**.
   - Select the component object on the JSP. Click the button next to the **Value** field in the Properties view, then select a data area from the Select Page Data Object list and click **OK**.

Related concepts
"PageHandler" on page 180

Related tasks
"Creating an EGL pageHandler part" on page 177

Related reference
"Page Designer support for EGL" on page 178
"PageHandler part in EGL source format" on page 659

Binding a JavaServer Faces multiple-selection component to an EGL PageHandler

A multiple-selection component allows a user to make one or more selections from a list of values. The user's selections are stored in a PageHandler array data area. Check box groups and multiple-select list boxes are multiple-selection JavaServer Faces components.

A binding is a relationship between the component and a data area. The data area must be declared in the PageHandler before you can bind a component to it. A multiple-selection component needs two different kinds of bindings:

- A binding to one or more data areas that contain the values from which the user can make a selection
- A binding to an array data area that will receive the user's selections

To bind a JavaServer Faces multiple-selection component to existing EGL PageHandler data areas, do as follows:

1. Open a Faces JSP file in the Page Designer. If you do not have a JSP file open, double-click on the JSP file in the Project Explorer. The JSP opens in the Page Designer. Click the **Design** tab to access the Design view.
2. From the **Window** menu, select **Show View** > **Other** >**Basic** > **Palette**.
3. In the Palette view, click the **Faces Components** drawer to display the Faces Components object types.
4. Drag a multiple-selection component from the palette to the JSP. The component object is placed on the JSP.
5. To bind the component to one or more PageHandler data areas that contain the values you want to display to the user, do one of the following procedures:
   - You can bind the component to individual PageHandler data areas, each of which contains one list item. Perform the following procedure for each data area:
     a. Select the object component on the JSP.
     b. In the Properties view, click **Add Choice**. The Name and Value fields are populated with default values.
     c. Click the **Name** field, then type the text that you want to display to the user.
     d. Click the **Value** field, then click the button next to the **Value** field. Select an individual data area from the Select Page Data Object list and click **OK**. This data area holds the value that will be moved to the receiving data area.
   - You can bind the component to a PageHandler array data area that contains the values you want to display to the user. Perform the following procedure to bind the component to an array data area:
     a. Select the object component on the JSP.
     b. In the Properties view, click **Add Set of Choices**. The Name and Value fields are populated with default values.
     c. Click the **Value** field, then click the button next to the **Value** field. Select an array data area from the Select Page Data Object list and click **OK**. The values in the array data area are the values that will be displayed to the user. The properties described later determine whether the values in the array data area or their equivalent index values will be moved to the receiving data area.
6. If you are using an array data area to supply the values displayed to the user, you must define the receiving data area with two properties: **selectFromListItem** and **selectType**. The selectFromListItem property points to the array that holds the list items. The selectType property indicates whether the receiving data area is to be populated with a text value or an index value. Examples of receiving data areas are as follows:

```
colorSelected char(10)
{selectFromListItem = "colorsArray",
 selectType = value};
```

```
colorSelectIdx smallInt
{selectFromListItem = "colorsArray",
selectType = index};
```

7. To bind the component to a PageHandler array data area that will receive the user's selections, do as follows:
   a. Select the component object on the JSP
   b. Click the button next to the **Value** field of the Properties view
   c. Select a data area from the Select Page Data Object list
   d. Click **OK**

Related concepts
"PageHandler" on page 180

Related tasks
"Creating an EGL pageHandler part" on page 177

Related reference
"Page Designer support for EGL" on page 178
"PageHandler part in EGL source format" on page 659

Creating EGL Reports

EGL reports overview

EGL can produce reports that are based on the functionality of JasperReports, which is an open-source, Java-based reporting library. For details on that library, see the following Web site:

http://jasperreports.sourceforge.net

EGL does not provide a mechanism for report layout. You must act as follows:

- Import a JasperReports output file (file extension *jasper*); or
- Use a text editor or specialized tool to create a design file that is transformed into a JasperReports output file when you click **Project > Build All** in the Workbench.

  Following are two specialized tools for creating a design file:

  – JasperAssistant, as described on this Web site:

    http://www.jasperassistant.com

  – iReport, as described on this Web site:

    http://ireport.sourceforge.net

In the EGL file that you write to drive report production, you submit data to the JasperReports output file (or accept the data source specified in that file), then export the report into one or more output files, each potentially in a different format such as HTML or Adobe Acrobat PDF.

If you also code an EGL handler of type JasperReport, you can respond to user events that occur as the report is filled with data; for example, you can add run-specific details to the report when report production is almost complete. To ensure that event-handling works, however, you must ensure that the output generated from the report handler is referenced in the JasperReports output file.

The **EGL report handler wizard** lets you create an EGL report handler easily.

When you write EGL code that interacts with a report, you use functions in the system library **ReportLib**; and in the code that produces a report, you create variables of type Report and ReportData.

The EGL parts mentioned here (Handler, Report, and Report Data) are all defined for you.

Related concepts
EGL report creation process overview

Related reference
EGL report library
Data sources
EGL report handler

EGL report creation process overview

This topic provides an overview of the general processes for creating and generating a report for an EGL project. Additional details about these processes are contained in EGL reports task help topics.

To create a report, you complete the three processes described below. Two of these processes, creating an XML design and writing code to drive a report, are required. A third process, creating a report handler, is optional. You do not need to complete these processes in the order described. For example, if you want a report handler, you can create it before you create an XML design document or you can work on creating a design document and a report handler simultaneously, with the exceptions described in the ″Code interrelationships between the report handler and the XML design document″ paragraphs in step 2 below.

You cannot generate a report if you do not have an XML design document and the code for driving the report.

The three processes you complete to create a report are:

1. Create an XML design document to specify layout information for the report. You can create this document in either of the following ways:
   - By using a third-party JasperReport design tool (like Jasper Assistant or iReports).
   - By using a text editor to write Jasper XML design information into a new text file.

   The XML design document must have a .jrxml extension. If the file you created does not have this extension, rename the file as a .jrxml file. In addition, be sure to save the XML design document in the same EGL package that will contain the EGL report handler and report-invocation code files.

   The .jrxml file that you create will be compiled into a .jasper file. If you do not create a new .jrxml file, you must import a .jasper file that was previously compiled.
2. If you want to use a report handler, which provides the logic for handling events during the filling of the report, you can create a report handler in either of the following ways:
   - By using the EGL report handler wizard to specify information for the report handler.
   - By creating a new EGL source file and either inserting a handler using the report handler template or manually entering the handler code.

   Code interrelationships between the report handler and the XML design document. In the .jrxml file, you can specify the scriptletClass that references the report handler file that is generated from the EGL report handler. Be aware that:
   - If the .jrxml file uses Java code that is produced by a report handler, you must generate the report handler before you create the .jrxml file.
   - If you change a report handler, you must recompile the .jrxml file.
   - If you need to resolve any compilation errors in the .jrxml file or want to recompile the .jasper file after making changes to a report handler, you must modify the .jrxml file and save it.
3. Use EGL ReportLib functions to write report-invocation code in your EGL project. You can use the EGL Program Part wizard when creating report-invocation code.

Important: You must give the report handler and the report-invocation code files names that are different from the XML design document. If you do not do this, the compilation of the design file causes the overwriting of Java code. To avoid problems, name your report handlers as *reportName*_handler.egl and name your XML design documents as *reportName*_XML.jrxml. For example, you can name the report abc_handler.egl and the design documents abc_XML.jrxml. You must also be sure that the XML design file has a unique name so that it does not clash with any EGL program files.

To build and generate a report after you create an XML design document, a report handler if you want to use one, and report-invocation code, you complete these processes:

1. Build the EGL project by selecting **Project > Build All**.

   EGL automatically generates Java code from the EGL report handler and compiles the XML design document (the .jrxml file) into a .jasper file.
2. Run the EGL program that has the report invocation code.

After the EGL program runs, the JasperReports program used by EGL automatically saves the generated report in the location specified by *reportDestinationFileName* in the report-invocation code.

The JasperReports program that generates the report also generates and stores a .jprint file, which is an intermediate file format that is exported into the final report format (.pdf, .html, .xml, .txt, or .csv).

The program can reuse one .jprint for multiple exports.

The exportReport() function in the report-invocation code causes EGL to export the report in the specified format. For example, the following code causes EGL to export a report in .pdf format:

```
reportLib.exportReport(myReport, ExportFormat.pdf);
```

EGL does not automatically refresh exported reports. If you change the report design or if data changes, you must refill and reexport the report.

Related concepts
EGL report overview

Related tasks
Adding a design document to a package
Using report templates
Creating an EGL report handler
Creating an EGL report handler manually
Writing code to drive a report
Running a report
Exporting reports
Using content assist in EGL

Related Reference
EGL report library
Data sources
EGL report handler

Data sources

The EGL report library includes references to the primary data source that contains data or to information on how to get the data.

You can use the following statements to specify data-source information:

- *DataSource.databaseConnection*
- *DataSource.sqlStatement*
- *DataSource.reportData*

For example, if you specify *fillReport (eglReport, DataSource.databaseConnection)* when using the *ReportLib.fillReport* function, EGL retrieves the database connection and passes it to the JasperReports engine.

See Sample code for EGL report-driver functions for examples of how a report is generated using a database connection, report data, and an SQL statement as a data source.

Related concepts
"EGL reports overview" on page 193
"EGL report creation process overview" on page 194

Related reference
"EGL library ReportLib" on page 834
"fillReport()" on page 837
"Sample code for EGL report-driver functions" on page 201

Data records in the library

The EGL Library contains a *ReportData* record and a *Report* record.

The *ReportData* record contains information for a particular set of data to be used in a report. The record contains these fields:

Field	Explanation	Data type
connectionName	Name of the connection established in the EGL program.	String
sqlStatement	The SQL statement that EGL should execute. Report data comes from the results of the execution of the statement.	String
Data	A dynamic array of records.	Any (This is the EGL *Any* type.)

The *Report* record contains information for a particular report. The record contains these fields:

Field	Explanation	Data Type
reportDesignFile	Name of the report design file, which is an XML file with a .jrxml extension.	String

Field	Explanation	Data Type
reportDestinationFile	Location of the .jrprint file.	String
reportExportFile	Location of the final, saved .xml, .pdf, .html, .txt., or .csv file.	String
reportData	The report data that is used as the primary data source for the report.	

Related concepts
"EGL reports overview" on page 193
"EGL report creation process overview" on page 194

Related reference
"EGL library ReportLib" on page 834

EGL report handler

The EGL report handler provides added functionality for handling events that occur when a report is being filled with data. You can use the New EGL Report Handler wizard to specify information for a report handler, or you can create a report handler manually.

When a report handler file is generated, EGL creates these files:

- *handlerName*.java
- *handlerName*_lib.java file.

handlerName
Alias of the EGL report handler

When EGL generates .java files, the class names are lower case. Be sure that any class name entered in an XML design document is in lower case.

See Creating an EGL report handler manually for sample syntax for and examples of report handler code.

Technical Details: The EGL report handler is an EGL handler part of type JasperReport. The report handler maps to the JasperReports scriptlet class. The report handler Java generation extends the JRDefaultScriptlet class and defines a Java class that contains the generated Java functions representing the scriptlet functions. The definition section of the XML design document contains the name of the scriptlet class. The JasperReports engine loads the scriptlet class and calls the different methods as defined in the report definition. (For more information on JasperReports scriplets and scriplet class, see JasperReports documentation.)

The Report Handler maintains an internal list of *ReportData* records that are returned when requested.

Related concepts
"EGL reports overview" on page 193
"EGL report creation process overview" on page 194

Related tasks

"Migrating EGL code to the EGL 6.0 iFix" on page 95
"Creating an EGL report handler manually" on page 205
"Writing code to drive a report" on page 209

Related reference
"Additional EGL report handler functions" on page 199
"EGL library ReportLib" on page 834
"Predefined report handler functions"

Predefined report handler functions

The Report Handler provides the following predefined functions that you can use as function templates:

Function	Where the function operates
beforeReportInit();	Before report initialization
afterReportInit();	After report initialization
beforePageInit();	Entering a page
afterPageInit();	Leaving a page
beforeColumnInit();	Before column initialization
afterColumnInit();	After column initialization
beforeGroupInit (*groupName String*);	Before group initialization. *groupName* is the name of the group in the report.
afterGroupInit(*groupName String*);	After group initialization.
beforeDetailEval();	Before every field. If this function is set, each row, before printing, calls to this function.
afterDetailEval();	After every field. If this function is set, each row, before printing, calls to this function.

Within one of these functions, you can make calls to other functions. For example, you can make a call to setReportVariable(), as follows:

```
function afterGroupInit(groupName String)
 if (groupName == "cat")
  setReportVariableValue ("NewGroupName", "dog");
 else
  setReportVariableValue ("NewGroupName", groupName);
 end
end
```

You can also create your own functions. See JasperReports documentation for information about creating custom functions.

For examples using predefined report handler functions, see *Creating an EGL report handler manually*.

Related concepts
"EGL reports overview" on page 193
"EGL report creation process overview" on page 194

Related tasks
"Migrating EGL code to the EGL 6.0 iFix" on page 95

"Creating an EGL report handler manually" on page 205

Related reference
"Additional EGL report handler functions"
"Data records in the library" on page 196
"EGL library ReportLib" on page 834
"EGL report handler" on page 197

Additional EGL report handler functions

You can invoke any of the following ReportLib functions from within the predefined report handler functions:

Function for getting report parameters

Function	Purpose
getReportParameter (*parameter* **String** in)	Returns the value of the specified parameter from the report that is being filled.

Functions for setting and getting report variables

These variables can be used for many reasons, for example, for storing an expression that is frequently used or for performing a complex calculation on the expression defined on the row that is being processed.

Function	Purpose
getReportVariableValue (*variable* **String** in)	Returns the value of the specified variable from the report that is being filled. The returned value is of type ANY.
setReportVariableValue (*variable String* in, *value* **Any** in);	Sets the value of the specified variable to the provided value.

Function for getting field values

Function	Purpose
getFieldValue (*fieldName* **String** in)	Returns the value of the specified field value for the row currently being processed. The returned value is of type ANY.

Functions for adding or getting data for subreports

A subreport is a report that is called from within another report. Sometimes data is exchanged between the main report and the subreport. A subreport can also be the main report of another subreport.

Function	Purpose
addReportData (*rd* **ReportData** in, *dataSetName* **String** in);	Adds the report data object with the specified name to the current Report Handler.
getReportData (*dataSetName* **String** in)	Retrieves the report data record with the specified name. The returned value is of type ReportData.

For examples using the functions described in this topic, see *Creating an EGL report handler manually*.

Related concepts
"EGL report creation process overview" on page 194
"EGL reports overview" on page 193

Related tasks
"Migrating EGL code to the EGL 6.0 iFix" on page 95
"Creating an EGL report handler manually" on page 205

Related reference
"Data records in the library" on page 196
"EGL report handler" on page 197
"EGL library ReportLib" on page 834
"Predefined report handler functions" on page 198

Data types in XML design documents

In XML report design documents, data types are described as Java data types. If you create EGL scriptlet code to use in the design document, use the Java data type that corresponds to the applicable EGL primitive type. Data returned as a result of a call to EGL scriptlet code must be declared using the Java data type.

The following table shows how EGL primitive types map to Java data types. JavaReports documentation contains information on the Java data types that you can use.

EGL primitive type	Java data type
bigint	java.lang.Long
bin	java.math.BigDecimal
blob	
char	java.lang.String
clob	
date	java.util.Date
dbchar	java.lang.String
decimal	java.math.BigDecimal
decimalfloat	java.lang.Double
float	java.lang.Float
hex	java.lang.byte
int	java.lang.Integer
interval	java.lang.String
mbchar	java.lang.String
money	java.math.BigDecimal
numc	java.math.BigDecimal
pacf	java.math.BigDecimal
smallfloat	java.lang.Float

EGL primitive type	Java data type
smallint	java.lang.Short
string	java.lang.String
time	java.sql.Time
timestamp	java.sql.Timestamp
unicode	java.lang.String

Related concepts
"EGL reports overview" on page 193
"EGL report creation process overview" on page 194

Related tasks
"Adding a design document to a package" on page 203

Related reference
"Data records in the library" on page 196
"EGL library ReportLib" on page 834
"EGL report handler" on page 197

Sample code for EGL report-driver functions

This topic contains code snippets that show how a report is generated using three different data sources:

- A database connection
- A data record
- An SQL statement

The following code snippet shows how a report is generated using a database connection as the data source:

```
//Variable declaration
myReport     Report;
myReportData ReportData;

//Function containing report invocation code
 function makeReport()
       //Initialize Report file locations
      myReport.reportDesignFile = "reportDesignFileName.jasper";
      myReport.reportDestinationFile =
 "reportDestinationFileName.jrprint";

       //Set the report data via a connection using the SQL statement
       //embedded in the report design
      sysLib.defineDatabaseAlias("alias", "databaseName");
       sysLib.connect("alias", "userid", "password");
      myReportData.connectionName="connectionName";
       myReport.reportData = myReportData;


       //Fill the report with data
       reportLib.fillReport(myReport, DataSource.databaseConnection);
       //Export the report in PDF format
      myReport.reportExportFile = "reportDesignFileName.pdf";
      reportLib.exportReport(myReport, ExportFormat.pdf);
  end
```

The following code snippet shows how a report is generated using a report data flexible record as the data source:

```
//Variable declaration
myReport     Report;
myReportData ReportData;

//Function containing the report driving code
function makeReport()
    //Initialize myReport file locations
    myReport.reportDesignFile = "reportDesignFileName.jasper";
     myReport.reportDestinationFile =
 "reportDestinationFileName.jrprint";

         //Set the report data
        populateReportData();
        myReport.reportData = myReportData;

        //Fill the report with data
       reportLib.fillReport(myReport, DataSource.reportData);

        //Export the report in HTML format
       myReport.reportExportFile = "reportDesignFileName.html";
       reportLib.exportReport(myReport, ExportFormat.html);
end

function populateReportData()
   //Insert EGL code here which populates myReportData
    ...
end
```

The following code snippet shows how a report is generated using an SQL statement as the data source:

```
//Variable declaration
myReport     Report;
myReportData ReportData;

//Function containing report driving code
function makeReport()
     //Initialize Report file locations
   myReport.reportDesignFile = "reportDesignFileName.jasper";
   myReport.reportDestinationFile = "reportDestinationFileName.jrprint";

    //Set the report data via a SQL statement
   myReportData.sqlStatement = "SELECT * FROM dataBaseTable";
   myReport.reportData = myReportData;

    //Fill the report with data
   reportLib.fillReport(myReport, DataSource.sqlStatement);

    //Export the report in text format
   myReport.reportExportFile = "reportOutputFileName.txt";
   reportLib.exportReport(myReport, ExportFormat.text);
end
```

Related concepts
"EGL report creation process overview" on page 194
"EGL reports overview" on page 193
Related tasks
"Writing code to drive a report" on page 209

Related reference
"Data records in the library" on page 196

"Data sources" on page 196
"EGL report handler" on page 197
"EGL library ReportLib" on page 834

Adding a design document to a package

You must have an XML design document with a .jrxml extension that specifies layout and other design information for the report.

To add a design document to a package, follow these steps:

1. Create a design document in either of the following ways:
   - Use a third-party JasperReports design tool (like JasperAssistant or iReports). If the file you create does not have a .jrxml extension, rename the file so it has a .jrxml extension.
   - Use a text editor to write Jasper XML design information into a new text file and save the file as a .jrxml file.
2. Place the XML design document in same EGL package that will contain EGL report handler and report driver files.

If you do not create a new XML design document, you must import a .jasper file that was previously compiled.

The .jrxml file is automatically compiled into a .jasper file when you select **Project > Build All** to build all EGL project components.

Note: See EGL report creation process overview for guidelines to follow if you are creating an XML design document and a report handler simultaneously. See Creating an EGL report handler part manually for an example showing how an XML design document gets a report data record from the report handler.

Related conceptsEGL reports overview
EGL report creation process overview

Related tasks
Creating an EGL report handler
Creating an EGL report handler manually
Writing code to drive a report

Related reference
EGL report library
Data types in XML design documents

Using report templates

You can select and modify any of the following EGL report templates:

- Database connection template
- Report data template
- SQL statement template
- Report handler template

To use a report template, follow these steps:

1. Select **Window > Preferences**.
2. When a list of preferences is displayed, expand **EGL**.

3. Expand **Editor**and select **Templates**.
4. Scroll through the list of templates and select a template. For example, select **handler** to display the report handler template.
5. Click **Edit**.
6. Change the template to meet your needs.

   Type **handler** followed by **Ctrl+space** to edit the report handler template. For more information, including code examples, see Creating an EGL report handler manually.

   Type **jas** followed by **Ctrl+space** to edit a data source template.
7. Click **Apply** and then **OK** to save your changes.

Related concepts
EGL reports overview
EGL report creation process overview

Related tasks
Creating an EGL report handler manually
Writing code to drive a report
Using content assist in EGL
Setting preferences for templates

Related Reference
EGL report handler
EGL report library
Sample code for EGL report-driver functions

Creating an EGL report handler

An EGL report handler provides the logic for handling events that occur when a report is being filled. To create an EGL report handler, do as follows:

1. Identify a project or folder to contain the file. You must create a project or folder if you do not already have one.
2. In the workbench, click **File > New > Other**.
3. In the New window, expand **EGL**.
4. Click **Report Handler**.
5. Click **Next**.
6. Select the project or folder that will contain the EGL file, then select a package.
7. Since the report handler name will be identical to the file name, choose a file name that adheres to EGL part name conventions. In the EGL Source File Name field, type the name of the EGL file, for example myReportHandler.
8. Click **Finish**.

Related concepts
"Development process" on page 8
"EGL projects, packages, and files" on page 13
"Generated output" on page 515
"Parts" on page 17
"Run-time configurations" on page 9

Related tasks
"Creating an EGL Web project" on page 117

Related reference
"EGL editor" on page 471
"EGL source format" on page 478

Creating an EGL report handler manually

If you do not want to use the New EGL Report Handler Wizard to create a report handler, you can create the report handler manually.

To create the report handler manually, follow these steps:

1. Create a new EGL source file.
2. Either:
   - Manually enter the handler code.
   - Insert a handler using the report handler template, as follows:
     a. Navigate to the report handler template and select the template you want.
     b. Click **Edit**.
     c. Type **handler** followed by **Ctrl+space**.
     d. Change the template code as needed.

The remainder of this topic contains code examples that show the following:

- The syntax for creating a report handler manually
- How to get report parameters in a report-handler
- How to set and get report variables
- How to get field values
- How to add a flexible record
- How an XML design document gets a report data record from the report handler

You can copy this code and modify it for your application.

Sample code showing syntax for creating a report handler manually

The following code shows the general syntax for creating an EGL report handler manually:

```
handler handlerName type jasperReport

  // Use Declarations (optional)
  use usePartReference;

   // Constant Declarations (optional)
  const constantName constantType = literal;

  // Data Declarations (optional)
  identifierName declarationType;

  // Pre-defined Jasper callback functions (optional)
  function beforeReportInit()
  ...
  end

  function afterReportInit()
...
  end

  function beforePageInit()
  ...
```

```
  end

  function afterPageInit()
  ...
  end

  function beforeColumnInit()
  ...
  end

  function afterColumnInit()
  ...
  end

  function beforeGroupInit(stringVariable string)
  ...
  end

  function afterGroupInit(stringVariable string)
  ...
  end

  function beforeDetailEval()
  ...
  end

  function afterDetailEval()
  ...
  end

  // User-defined functions (optional)
  function myFirstFunction()
  ...
  end

  function mySecondFunction()
  ...
  end
end
```

Example showing how to get report parameters

The following code snippet shows how to get report parameters in a report handler:

```
handler myReportHandler type jasperReport

  // Data Declarations
  report_title String;

  // Jasper callback function
  function beforeReportInit()
...

     report_title = getReportTitle();

...
  end

 ...

  // User-defined function
  function getReportTitle() Returns (String)
     return (getReportParameter("ReportTitle"));
   end

end
```

Example showing how to set and get report variables

The code snippet below shows how to set and get report variables in a report handler:

```
handler myReportHandler type jasperReport

   // Data Declarations
  employee_serial_number int;

  // Jasper callback function
  function afterPageInit()
  ...
    employee_serial_number = getSerialNumberVar();
  ...
  end

 ...

  // User-defined function
  function getSerialNumberVar() Returns (int)
     employeeName String;
     employeeName = "Ficus, Joe";
     setReportVariableValue("employeeNameVar", employeeName);
     return (getReportVariableValue("employeeSerialNumVar"));
   end
end
```

Example showing how to get report field values in a report handler

The example code snippet below shows how to get report field values in a report handler

```
handler myReportHandler type jasperReport

  // Data Declarations
  employee_first_name String;

  // Jasper callback function
  function beforeColumnInit()
    ...
    employee_first_name = getFirstNameField();
    ...
  end

   ...

  // User-defined function
  function getFirstNameField() Returns (String)
     fldName String;
     fldName = "fname";
     return (getFieldValue(fldName));
   end

end
```

Example showing how to add a report data flexible record in a report handler

The example code below shows how to add a report data flexible record in a report handler:

```
handler myReportHandler type jasperReport

 // Data Declarations
 customer_array  customerRecordType[];
 c    customerRecordType;
```

```
 // Jasper callback function
  function beforeReportInit()
    customer ReportData;
    datasetName String;

     //create the ReportData object for the Customer subreport
    c.customer_num  = getFieldValue("c_customer_num");
    c.fname         = getFieldValue("c_fname");
    c.lname         = getFieldValue("c_lname");
    c.company       = getFieldValue("c_company");
    c.address1      = getFieldValue("c_address1");
    c.address2      = getFieldValue("c_address2");
    c.city          = getFieldValue("c_city");
    c.state         = getFieldValue("c_state");
    c.zipcode       = getFieldValue("c_zipcode");
    c.phone         = getFieldValue("c_phone");
    customer_array.appendElement(c);
    customer.data = customer_array;
     datasetName = "customer";
     addReportData(customer, datasetName);
  end
end
```

Example showing how an XML design document gets a report data record from the report handler

The code snippet below shows how an XML design document gets a report data flexible record from the report handler:

```
<jasperReport name="MasterReport"
   scriptletClass="subreports.SubReportHandler">
...

<subreport>
  <dataSourceExpression>
    <![CDATA[(JRDataSource)(((subreports.SubReportHandler)
    $P{REPORT_SCRIPTLET}).getReportData(
       new String("customer")))]]&gt;
  </dataSourceExpression>
  <subreportExpression class="java.lang.String">
    <![CDATA["C:/RAD/workspaces/Customer.jasper"]]&gt;
  </subreportExpression>
</subreport>

...
</jasperReport>
```

Related concepts

"EGL reports overview" on page 193
"EGL report creation process overview" on page 194

Related tasks

"Creating an EGL source file" on page 120
"Migrating EGL code to the EGL 6.0 iFix" on page 95
"Using report templates" on page 203

Related Reference

"Additional EGL report handler functions" on page 199
"EGL library ReportLib" on page 834
"EGL report handler" on page 197
"Predefined report handler functions" on page 198

Writing code to drive a report

Use the New EGL Program Part wizard to create a new EGL basic program that uses the report library to run reports.

To create this report driver, follow these steps:

1. Choose **File > New > Program** and then select the folder that will contain the EGL file.
2. Select a package.
3. Specify a file name for the source file, select the *BasicProgram* type and click **Finish**.
4. Find the program line.
5. In the main() function, just after the program line, type **jas** followed by **Ctrl+space** to insert code for the report driver.
6. In the window containing data-source connection types and code, select one of the data-source connection types.
7. You can modify existing code or add your own code. If you modify the code, insert specific values for the variables used by the report driver. These variables include *reportDesignFileName*, *reportDestinationFileName*, *exportReportFile*, *alias*, *databaseName*, *userid*, *password*, and *connectionName*.

Code showing report-invocation information

The following code shows report-invocation information:

```
  myReport     Report;
  myReportData ReportData;

  myReport.reportDesignFile = "myReport_XML.jasper";
  myReport.reportDestinationFile = "myReport.jrprint";
  myReport.reportExportFile = "myReport.pdf";

  myReportData.sqlStatement = "Select * From myTable";

  myReport.reportData = myReportData;

ReportLib.fillReport(myReport, DataSource.sqlStatement);

ReportLib.exportReport(myReport, ExportFormat.pdf);
```

Code	Explanation
`myReport Report;`	This is a report library record declaration.
`myReportData ReportData;`	This is a report library data record declaration.
`myReport.reportDesignFile = "myReport_XML.jasper";`	This statement defines the report design to use to create a report.
`myReport.reportDestinationFile ="myReport.jrprint";`	This statement specifies the file name for generated report output.
`myReport.reportExportFile = "myReport.pdf`	This statement specifies the file name for exported output.
`myReport.sqlStatement = "Select * From myTable";`	This provides information on the SQL Select statement used in the report.
`myReport.reportData = myReportData;`	This provides information on report data.
`ReportLib.fillReport(myReport, DataSoucre.sqlStatement );`	This statement specifies source information for the report.

Code	Explanation
`ReportLib.exportReport(myReport, ExportFormat.pdf);`	This statement specifies the report output format.

Example code fragment:

```
//location where the .jprint file is stored.
abcReport.reportDestinationFile="C:\\temp\\MasterReport.jrprint";

//location for the exported report.
abcReport.reportExportFile="C:\\temp\\MasterReport.pdf";

//perform the export.
ReportLib.exportReport(abcReport, ExportFormat.pdf);
```

Related concepts
EGL report overview
EGL report creation process overview

Related tasks
Creating an EGL report handler
Creating an EGL report handler manually
Using report templates

Related reference
EGL report handler
EGL report library
Sample code for EGL report-driver functions

Generating files for and running a report

You must have an XML design document with a .jrxml extension that specifies layout and other design information for the report.

To build and generate a report for an EGL project, follow these steps:

1. Build the EGL project by selecting **Project > Build All**.

   EGL automatically generates Java code from the EGL report handler and compiles the XML design document (the .jrxml file) into a .jasper file.
2. Run the EGL program that has the report invocation code. One way to do this in the Package Explorer is to navigate to and right-click the .egl file that contains the code. Then, select **Generate** from the popup menu.

In addition to generating the report, the JasperReports program used by EGL automatically saves the generated report in the location specified by *reportDestinationFileName* in the report-invocation code.

The JasperReports program that generates the report also generates and stores a .jprint file, which is an intermediate file format that is exported into the final report format (.pdf, .html, .xml, .txt, or .csv). The program can reuse one .jprint file for multiple exports.

The *exportReport()* function in the report-invocation code causes EGL to export the report in the specified format.

Related conceptsEGL reports overview
EGL report creation process overview

Related tasks
Creating an EGL report handler
Creating an EGL report handler manually
Writing code to drive a report
Exporting reports

Related reference
EGL report library
EGL report handler

Exporting Reports

You can export filled reports as PDF, HTML, XML, CSV (comma-separated values), and plain-text output. The *exportReport()* function in EGL report driver code causes EGL to export the report in the specified format.

The *exportReportFile* value in the report-driver code specifies the location of the exported file.

To specify the format of exported reports, use one of the following parameters in the call to the *exportReport()* function:

- ExportFormat.html
- ExportFormat.pdf
- ExportFormat.text
- ExportFormat.xml
- ExportFormat.csv

For example, the following code causes EGL to export a report in .pdf format:

```
reportLib.exportReport(myReport, ExportFormat.pdf);
```

Important: EGL does not automatically refresh exported reports. If you change the report design or if data changes, you must refill and re-export the report.

Related conceptsEGL reports overview
EGL report creation process overview

Related tasks
Creating an EGL report handler
Creating an EGL report handler manually
Writing code to drive a report
Running a report

Related reference
EGL report library
EGL report handler

Working with files and databases

SQL support

As shown in the next table, EGL-generated code can access a relational database on any of the target systems.

Target System	Support for access of relational databases
AIX, iSeries, Linux, Windows 2000/NT/XP, UNIX System Services	JDBC provides access to DB2 UDB, Oracle, or Informix

As you work on a program, you can code SQL statements as you would when coding programs in most other languages. To ease your way, EGL provides SQL statement templates for you to fill.

Alternatively, you can use an SQL record as the I/O object when you code an EGL statement. Using the record in this way means that you access a database either by customizing an SQL statement provided to you or by relying on a default that removes the need to code SQL.

In either case, be aware of these aspects of EGL support for SQL:

- If you want to test for a null in a particular table column, you must receive the column value into an SQL record, into a record item that is declared as nullable. For details, see *Testing for and setting NULL* described later.
- In the overview sections that follow (and in keeping with SQL terminology), each item that is referenced in an SQL statement is called a *host variable*. The word *host* refers to the language that embeds the SQL statement; in this case, to the EGL procedural language. A host variable in an SQL statement is preceded by a colon, as in this example:

```
select empnum, empname
from employee
where empnum >= :myRecord.empnum
for update of empname
```

EGL statements and SQL

The next table lists the EGL keywords that you can use to access a relational database. Included in this table is an outline of the SQL statements that correspond to each keyword. When you code an EGL **add** statement, for example, you generate an SQL INSERT statement.

In many business applications, you use the EGL **open** statement and various kinds of **get by position** statements. The code helps you to declare, open, and process a *cursor*, which is a run-time entity that acts as follows:

- Returns a *result set*, which a list of rows that fulfill your search criteria
- Points to a specific row in the result set

When your output is Java code, you can use the EGL **open** statement to call a stored procedure. That procedure is composed of logic that is written outside of

EGL, is stored in the database management system, and also returns a result set. (Regardless of your output language, you can use the EGL **execute** statement to call a stored procedure.)

Later sections give details on processing a result set.

If you intend to code SQL statements explicitly, you use the EGL **execute** statement and possibly the EGL **prepare** statement.

Keyword/Purpose	Outline of SQL statements	Can you modify the SQL?
add Places a row in a database; or (if you use a dynamic array of SQL records), places a set of rows based on the content of successive elements of the array.	INSERT row (as occurs repeatedly, if you specify a dynamic array).	Yes
close Releases unprocessed rows.	CLOSE cursor.	No
delete Deletes a row from a database.	DELETE row. The row was selected in either of two ways: • When you invoked a **get** statement with the forUpdate option (as appropriate when you wish to select the first of several rows that have the same key value) • When you invoked an **open** statement with the forUpdate option and then a **get next** statement (as appropriate when you wish to select a set of rows and to process the retrieved data in a loop)	No
forEach Marks the start of a set of statements that run in a loop. The first iteration occurs only if a specified result set is available and continues (in most cases) until the last row in that result set is processed.	EGL converts a **forEach** statement into an SQL FETCH statement that runs inside a loop.	No
freeSQL Frees any resources associated with a dynamically prepared SQL statement, closing any open cursor associated with that SQL statement.		No

Keyword/Purpose	Outline of SQL statements	Can you modify the SQL?
get (also called **get by key value**) Reads a single row from a database; or (if you use a dynamic array of SQL records), reads successive rows into successive elements in the array.	SELECT row, but only if you set the option singleRow. Otherwise, the following rules apply: • EGL converts a **get** statement to this: – DECLARE cursor with SELECT or (if you set the forUpdate option) with SELECT FOR UPDATE. – OPEN cursor. – FETCH row. • If you did not specify the option forUpdate, EGL also closes the cursor. • The singleRow and forUpdate options are not supported with dynamic arrays; in that case, EGL run time declares and opens a cursor, fetches a series of rows, and closes the cursor.	Yes
get absolute Reads a numerically specified row in a result set that was selected by an **open** statement.	EGL converts a **get absolute** statement to an SQL FETCH statement.	No
get current Reads the row at which the cursor is already positioned in a result set that was selected by an **open** statement.	EGL converts a **get current** statement to an SQL FETCH statement.	No
get first Reads the first row in a result set that was selected by an **open** statement.	EGL converts a **get first** statement to an SQL FETCH statement.	No
get last Reads the last row in a result set that was selected by an **open** statement.	EGL converts a **get last** statement to an SQL FETCH statement.	No
get next Reads the next row in a result set that was selected by an **open** statement.	EGL converts a **get next** statement to an SQL FETCH statement.	No
get previous Reads the previous row in a result set that was selected by an **open** statement.	EGL converts a **get previous** statement to an SQL FETCH statement.	No

Keyword/Purpose	Outline of SQL statements	Can you modify the SQL?
get relative Reads a numerically specified row in a result set that was selected by an **open** statement. The row is identified in relation to the cursor position in the result set.	EGL converts a **get relative** statement to an SQL FETCH statement.	No

execute
 	
Lets you run an SQL data-definition statement (of type CREATE TABLE, for example); or a data-manipulation statement (of type INSERT or UPDATE, for example); or a prepared SQL statement that does not begin with a SELECT clause.
	The SQL statement you write is made available to the database management system.
	
	The primary use of **execute** is to code a single SQL statement that is fully formatted at generation time, as in this example--
	`try`
	`  execute`
	`  #sql{    // no space after "#sql"`
	`    delete`
	`    from EMPLOYEE`
	`    where department =`
	`      :myRecord.department`
	`  };`
	`onException`
	`  myErrorHandler(10);`
	`end`
	
	A fully formatted SQL statement may include host variables in the WHERE clause.
		Yes

open Selects a set of rows from a relational database for later retrieval with **get next** statements.	EGL converts an **open** statement to a CALL statement (for accessing a stored procedure) or to these statements: • DECLARE cursor with SELECT or with SELECT FOR UPDATE. • OPEN cursor.	Yes

Keyword/Purpose	Outline of SQL statements	Can you modify the SQL?

prepare
 	
Specifies an SQL PREPARE statement, which optionally includes details that are known only at run time; you run the prepared SQL statement by running an EGL execute statement or (if the SQL statement begins with SELECT) by running an EGL open or get statement.
	EGL converts a **prepare** statement to an SQL PREPARE statement, which is always constructed at run time. In the following example of an EGL **prepare** statement, each parameter marker (?) is resolved by the USING clause in the subsequent **execute** statement:
	`myString =`
	`  "insert into myTable " +`
	`  "(empnum, empname) " +`
	`  "value ?, ?";`
	
	`try`
	`  prepare myStatement`
	`  from myString;`
	`onException`
	`  // exit the program`
	`  myErrorHandler(12);`
	`end`
	
	`try`
	`  execute myStatement`
	`  using :myRecord.empnum,`
	`  :myRecord.empname;`
	`onException`
	`  myErrorHandler(15);`
	`end`
		Yes

replace Puts a changed row back into a database.	UPDATE row. The row was selected in either of two ways: • When you invoked a **get** statement with the forUpdate option (as appropriate when you wish to select the first of several rows that have the same key value); or • When you invoked an **open** statement with the forUpdate option and then a **get next** statement (as appropriate when you wish to select a set of rows and to process the retrieved data in a loop).	Yes

Note: Under no circumstances can you update multiple database tables by coding a single EGL statement.

Result-set processing

A common way to update a series of rows is as follows:

1. Declare and open a cursor by running an EGL **open** statement with the option forUpdate; that option causes the selected rows to be locked for subsequent update or deletion
2. Fetch a row by running an EGL **get next** statement
3. Do the following in a loop:
   a. Change the data in the host variables into which you retrieved data
   b. Update the row by running an EGL **replace** statement
   c. Fetch another row by running an EGL **get next** statement
4. Commit changes by running the EGL function **commit**.

The statements that open the cursor and that act on the rows of that cursor are related to each other by a result-set identifier, which must be unique across all result-set identifiers, program variables, and program parameters within the program. You specify that identifier in the **open** statement that opens the cursor, and you reference the same identifier in the **get next**, **delete**, and **replace** statements that affect an individual row, as well as on the **close** statement that closes the cursor. For additional details, see *resultSetID*.

The following code shows how to update a series of rows when you are coding the SQL yourself:

```
VGVar.handleHardIOErrors  = 1;

try
  open selectEmp forUpdate with
  #sql{            // no space after "#sql"
    select empname
    from EMPLOYEE
    where empnum >= :myRecord.empnum
    for update of empname
  };

onException
  myErrorHandler(8);    // exits program
end

try
  get next from selectEmp into :myRecord.empname;
onException
  if (sysVar.sqlcode != 100)
    myErrorHandler(8);  // exit the program
  end
end

while (sysVar.sqlcode != 100)
  myRecord.empname = myRecord.empname + " " + "III";

  try
    execute
    #sql{
      update EMPLOYEE
      set empname = :empname
      where current of selectEmp
    };
  onException
    myErrorHandler(10);   // exits program
  end

  try
    get next from selectEmp into :myRecord.empname;
  onException
    if (sysVar.sqlcode != 100)
      myErrorHandler(8);  // exits program
    end
  end
end  // end while; cursor is closed automatically
     // when the last row in the result set is read

sysLib.commit;
```

If you wish to avoid some of the complexity in the previous example, consider SQL records. Their use allows you to streamline your code and to use I/O error values that do not vary across database management systems. The next example is equivalent to the previous one but uses an SQL record called emp:

```
VGVar.handleHardIOErrors  = 1;

try
  open selectEmp forUpdate for emp;
onException
  myErrorHandler(8);    // exits program
end

try
  get next emp;
onException
  if (sysVar.sqlcode not noRecordFound)
    myErrorHandler(8);  // exit the program
  end
end

while (sysVar.sqlcode not noRecordFound)
  myRecord.empname = myRecord.empname + " " + "III";

  try
    replace emp;
  onException
    myErrorHandler(10);   // exits program
  end

  try
    get next emp;
  on exception
    if (sysVar.sqlcode not noRecordFound)
      myErrorHandler(8);  // exits program
    end
  end
end  // end while; cursor is closed automatically
     // when the last row in the result set is read

sysLib.commit;
```

Later sections describe SQL records.

SQL records and their uses

An SQL record is a variable that is based on an SQL record part. This type of record allows you to interact with a relational database as though you were accessing a file. If the variable EMP is based on an SQL record part that references the database table EMPLOYEE, for example, you can use EMP in an EGL **add** statement:

```
add EMP;
```

In this case, EGL inserts the data from EMP into EMPLOYEE. The SQL record also includes state information so that after the EGL statement runs, you can test the SQL record to perform tasks conditionally, in accordance with the I/O error value that resulted from database access:

```
VGVar.handleHardIOErrors  = 1;

try
  add EMP;
onException
  if (EMP is unique)     // if a table row
                         // had the same key
    myErrorHandler(8);
  end
end
```

An SQL record like EMP allows you to interact with a relational database as follows:

- Declare an SQL record part and the related SQL record
- Define a set of EGL statements that each use the SQL record as an I/O object
- Accept the default behavior of the EGL statements or make the SQL changes that are appropriate for your business logic

Declaring an SQL record part and the related record

You declare an SQL record part and associate each of the record items with a column in a relational table or view. You can let EGL make this association automatically by way of the EGL editor's retrieve feature, as described later in *Database access at declaration time*.

If the SQL record part is not a fixed record part, you can include primitive fields as well as other variables. You are especially likely to include the following kinds of variables:

- Other SQL records. The presence of each represents a one-to-one relationship between the parent and child tables.
- Arrays of SQL records. The presence of each represents a one-to-many relationship between the parent and child tables.

Only fields of a primitive type can represent a database column.

If level numbers precede the fields, the SQL record part is a fixed record part. The following rules apply:

- The structure in each SQL record part must be *flat* (without hierarchy)
- All of the fields must be primitive fields, but not of type BLOB, CLOB, or STRING
- None of the record fields can be a structure-field array

After you declare an SQL record part, you declare an SQL record that is based on that part.

Defining the SQL-related EGL statements

You can define a set of EGL statements that each use the SQL record as the I/O object in the statement. For each statement, EGL provides an *implicit SQL statement*, which is not in the source but is implied by the combination of SQL record and EGL statement. In the case of an EGL **add** statement, for example, an implicit SQL INSERT statement places the value of a given record item into the associated table column. If your SQL record includes a record item for which no table column was assigned, EGL forms the implicit SQL statement on the assumption that the name of the record item is identical to the name of the column.

Using implicit SELECT statements: When you define an EGL statement that uses an SQL record and that generates either an SQL SELECT statement or a cursor declaration, EGL provides an implicit SQL SELECT statement. (That statement is embedded in the cursor declaration, if any.) For example, you might declare a variable that is named EMP and is based on the following record part:

```
Record Employee type sqlRecord
  { tableNames = [["EMPLOYEE"]],
    keyItems = ["empnum"] }
  empnum decimal(6,0);
  empname char(40);
end
```

Then, you might code a **get** statement:

```
get EMP;
```

The implicit SQL SELECT statement is as follows:

```
SELECT empnum, empname
FROM   EMPLOYEE
WHERE  empnum = :empnum
```

EGL also places an INTO clause into the standalone SELECT statement (if no cursor declaration is involved) or into the FETCH statement associated with the cursor. The INTO clause lists the host variables that receive values from the columns listed in the first clause of the SELECT statement:

```
INTO :empnum, :empname
```

The implicit SELECT statement reads each column value into the corresponding host variable; references the tables specified in the SQL record; and has a search criterion (a WHERE clause) that depends on *a combination of two factors*:

- The value you specified for the record property **defaultSelectCondition**; and
- A relationship (such as an equality) between two sets of values:
  - Names of the columns that constitute the table key
  - Values of the host variables that constitute the record key

A special situation is in effect if you read data into a dynamic array of SQL records, as is possible with the **get** statement:

- A cursor is open, successive rows from the database are read into successive elements of the array, the result set is freed, and the cursor is closed.
- If you do not specify an SQL statement, the search criterion depends on the record property **defaultSelectCondition**, but also depends on a relationship (specifically, a greater-than-or-equal-to relationship) between the following sets of values:
  - Names of columns, as specified indirectly when you specify items in the EGL statement
  - Values of those items

  Any host variables specified in the property **defaultSelectCondition** must be outside the SQL record that is the basis of the dynamic array.

For details on the implicit SELECT statement, which vary by keyword, see *get* and *open*.

Using SQL records with cursors: When you are using SQL records, you can relate cursor-processing statements by using the same SQL record in several EGL statements, as you can by using a result-set identifier. However, any cross-statement relationship that is indicated by a result-set identifier takes precedence over a relationship indicated by the SQL record; and in some cases you must specify a resultSetID.

In addition, only one cursor can be open for a particular SQL record. If an EGL statement opens a cursor when another cursor is open for the same SQL record, the generated code automatically closes the first cursor.

Customizing the SQL statements

Given an EGL statement that uses an SQL record as the I/O object, you can progress in either of two ways:

- You can accept the implicit SQL statement. In this case, changes made to the SQL record part affect the SQL statements used at run time. If you later indicate that a different record item is to be used as the key of the SQL record, for example, EGL changes the implicit SELECT statement used in any cursor declaration that is based on that SQL record part.
- You can choose instead to make the SQL statement explicit. In this case, the details of that SQL statement are isolated from the SQL record part, and any subsequent changes made to the SQL record part have no effect on the SQL statement that is used at run time.

  If you remove an explicit SQL statement from the source, the implicit SQL statement (if any) is again available at generation time.

Example of using a record in a record

To allow a program to retrieve data for a series of employees in a department, you can create two record parts and a function, as follows:

```
DataItem DeptNo { column = deptNo } end

Record Dept type SQLRecord
  deptNo DeptNo;
  managerID CHAR(6);
  employees Employee[];
end

Record Employee type SQLRecord
  employeeID CHAR(6);
  empDeptNo DeptNo;
end

Function getDeptEmployees(dept Dept)
  get dept.employees usingKeys dept.deptNo;
end
```

Testing for and setting NULL

In some cases, EGL internally maintains a null indicator for a subset of variables in your code. If you accept the default behavior, EGL internally maintains a null indicator only for each variable having these characteristics:

- Is in an SQL record
- Is declared with the property **isNullable** set to *yes*

Do not code host variables for null indicators in your SQL statements, as you might in some languages. To test for null in a nullable host variable, use an EGL **if** statement. You also can test for retrieval of a truncated value, but only when a null indicator is available.

You can null an SQL table column in either of two ways:

- Use an EGL **set** statement to null a nullable host variable, then write the related SQL record to the database; or
- Use the appropriate SQL syntax, either by writing an SQL statement from scratch or by customizing an SQL statement that is associated with the EGL **add** or **replace** statement

For additional details on null processing, see *itemsNullable* and *SQL item properties*.

Database access at declaration time

You receive the following benefits from accessing (at declaration time) a database that has similar characteristics to the database that your code will access at run time:

- If you access a database table or view that is equivalent to a table or view associated with an SQL record, you can use the retrieve feature of the EGL part editor to create or overwrite the record items. The retrieve feature accesses information stored in the database management system so that the number and data characteristics of the created items reflect the number and characteristics of the table columns. After you invoke the retrieve feature, you can rename record items, delete record items, and make other changes to the SQL record.
- Your access of an appropriately structured database at declaration time helps to ensure that your SQL statements will be valid in relation to an equivalent database at run time.

The retrieve feature creates record items that each have the same name (or almost the same name) as the related table column.

You cannot retrieve a view that is defined with the DB2 condition WITH CHECK OPTIONS.

For further details on using the retrieve feature, see *Retrieving SQL table data*. For details on naming, see *Setting preferences for SQL retrieve*.

To access a database at declaration time, specify connection information in a preferences page, as described in *Setting preferences for SQL database connections*.

Related concepts

"Dynamic SQL" on page 224
"Logical unit of work" on page 288
"resultSetID" on page 722

Related tasks

"Retrieving SQL table data" on page 235
"Setting preferences for SQL database connections" on page 111
"Setting preferences for SQL retrieve" on page 113

Related reference

"add" on page 544
"close" on page 551
"Database authorization and table names" on page 453
"Default database" on page 234
"delete" on page 554
"execute" on page 557
"get" on page 567
"get next" on page 579
"Informix and EGL" on page 235
"itemsNullable" on page 377
"open" on page 598
"prepare" on page 611
"replace" on page 613
"SQL data codes and EGL host variables" on page 723
"SQL examples" on page 224

"SQL item properties" on page 63
"SQL record internals" on page 726
"SQL record part in EGL source format" on page 726
"Testing for and setting NULL" on page 222

Dynamic SQL

The SQL statement associated with an EGL statement can be specified statically, with every detail in place at generation time. When dynamic SQL is in effect, however, the SQL statement is built at run time, each time that the EGL statement is invoked.

Use of dynamic SQL decreases the speed of run-time processing, but lets you vary a database operation in response to a run-time value:

- For a database query, you may want to vary the selection criteria, how data is aggregated, or the order in which rows are returned; those details are controlled by the WHERE, HAVING, GROUP BY, and ORDER BY clauses. In this case, you can use the prepare statement.
- For many kinds of operations, you may want a run-time value to determine which table to access. You can accomplish dynamic specification of a table in either of two ways:
  - Use the prepare statement; or
  - Use an SQL record and specify a value for the property **tableNameVariables**, as described in *SQL record part in EGL source format*.

Related concepts
"SQL support" on page 213

Related reference
"Database authorization and table names" on page 453
"prepare" on page 611
"SQL record part in EGL source format" on page 726

SQL examples

You can access an SQL data base in any of these ways:

- By hand-coding an SQL statement whose format is known at generation time.
- By using an SQL record as the I/O object of an EGL statement, when the format of the SQL statement is known at generation time--
  - If you place an explicit SQL statement in the EGL source, that SQL statement is used at run time;
  - Otherwise, an implicit SQL statement is used at run time.
- By coding an EGL **prepare** statement, which generates an SQL PREPARE statement that in turn creates an SQL statement at run time.

In every case, you can use an SQL record as a memory area and to provide a simple way to test for successful operation. The examples in this section assume that a record part is declared in an EGL file and that a record based on the part was declared in a program in that file:

- The SQL record part is as follows--

```
Record Employee type sqlRecord
  {
    tableNames = [["employee"]],
    keyItems = ["empnum"],
    defaultSelectCondition =
```

```
        #sqlCondition{   // no space
                         // between #sqlCondition
                         // and the brace
          aTableColumn = 4   -- start each SQL comment
                             -- with a double hyphen
        }
    }

    empnum  decimal(6,0) {isReadonly=yes};
    empname char(40);
  end
```

- The SQL record is as follows--

```
emp Employee;
```

For further details on SQL records and implicit statements, see SQL support.

Coding SQL statements

To prepare to code SQL statements, declare variables:

```
empnum decimal(6,0);
empname char(40);
```

Adding a row to an SQL table: To prepare to add a row, assign values to variables:

```
empnum = 1;
empname = "John";
```

To add the row, associate an EGL **execute** statement with an SQL INSERT statement as follows:

```
try
  execute
    #sql{
      insert into employee (empnum, empname)
      values (:empnum, :empname)
    };
onException
  myErrorHandler(8);
end
```

Reading a set of rows from an SQL table: To prepare to read a set of rows from an SQL table, identify a record key:

```
empnum = 1;
```

To get the data, code a series of EGL statements:

- To select a result set, run an EGL open statement--

```
open selectEmp
  with #sql{
    select empnum, empname
    from employee
    where empnum >= :empnum
    for update of empname
  }
  into empnum, empname;
```

- To access the next row of the result set, run an EGL get next statement--

```
get next from selectEmp;
```

If you did not specify the into clause in the open statement, you need to specify the into clause in the get next statement; and if you specified the into clause in both places, the clause in the get next statement takes precedence:

```
get next from selectEmp
  into empnum, empname;
```

The cursor is closed automatically when the last record is read from the result set.

A more complete example is as following code, which updates a set of rows:

```
VGVar.handleHardIOErrors  = 1;

try
  open selectEmp
    with #sql{
      select empnum, empname
      from employee
      where empnum >= :empnum
      for update of empname
    }
    into empnum, empname;
onException
  myErrorHandler(6);    // exits program
end

try
  get next from selectEmp;
onException
  if (sqlcode != 100)
    myErrorHandler(8);  // exits program
  end
end

while (sqlcode != 100)
  empname = empname + " " + "III";

  try
    execute
      #sql{
        update employee
        set empname = :empname
        where current of selectEmp
      };
  onException
    myErrorHandler(10); // exits program
  end

  try
    get next from selectEmp;
  onException
    if (sqlcode != 100)
      myErrorHandler(8);  // exits program
    end
  end
end  // end while; cursor is closed automatically
     // when the last row in the result set is read

sysLib.commit();
```

Instead of coding the get next and while statements, you can use the forEach statement, which executes a block of statements for each row in a result set:

```
VGVar.handleHardIOErrors  = 1;

try
  open selectEmp
    with #sql{
      select empnum, empname
      from employee
      where empnum >= :empnum
      for update of empname
    }
```

```
    into empnum, empname;
onException
  myErrorHandler(6);    // exits program
end

try
  forEach (from selectEmp)
    empname = empname + " " + "III";

    try
      execute
        #sql{
          update employee
          set empname = :empname
        where current of selectEmp
        };
    onException
      myErrorHandler(10); // exits program
    end
  end  // end forEach; cursor is closed automatically
       // when the last row in the result set is read

onException
  // the exception block related to forEach is not run if the condition
  // is "sqlcode = 100", so avoid the test "if (sqlcode != 100)"
  myErrorHandler(8);  // exits program
end

sysLib.commit();
```

Using SQL records with implicit SQL statements

To begin using EGL SQL records, declare an SQL record part:

```
Record Employee type sqlRecord
  {
    tableNames = [["employee"]],
    keyItems = ["empnum"],
    defaultSelectCondition =
      #sqlCondition{
        aTableColumn = 4 -- start each SQL comment
                         -- with a double hyphen
      }
  }

  empnum  decimal(6,0) {isReadonly=yes};
  empname char(40);
end
```

Declare a record that is based on the record part:

```
emp Employee;
```

Adding a row to an SQL table: To prepare to add a row to an SQL table, place values in the EGL record:

```
emp.empnum = 1;
emp.empname = "John";
```

Add an employee to the table by specifying the EGL add statement:

```
try
  add emp;
onException
  myErrorHandler(8);
end
```

Reading rows from an SQL table: To prepare to read rows from an SQL table, identify a record key:

```
emp.empnum = 1;
```

Get a single row in either of these ways:

- Specify the EGL get statement in a way that generates a series of statements (DECLARE cursor, OPEN cursor, FETCH row, and in the absence of forUpdate, CLOSE cursor):

```
try
  get emp;
onException
  myErrorHandler(8);
end
```

- Specify the EGL get statement in a way that generates a single SELECT statement:

```
try
  get emp singleRow;
onException
  myErrorHandler(8);
end
```

Process multiple rows in either of these ways:

- Use the EGL open, get next, and while statements--

```
VGVar.handleHardIOErrors  = 1;

try
  open selectEmp forUpdate for emp;
onException
  myErrorHandler(6);    // exits program
end

try
  get next emp;
onException
  if (emp not noRecordFound)
    myErrorHandler(8);  // exit the program
  end
end

while (emp not noRecordFound)
  myRecord.empname = myRecord.empname + " " + "III";

  try
    replace emp;
  onException
    myErrorHandler(10);   // exits program
  end

  try
    get next emp;
  onException
    if (emp not noRecordFound)
      myErrorHandler(8);  // exits program
    end
  end
end  // end while; cursor is closed automatically
     // when the last row in the result set is read

sysLib.commit();
```

- Use the EGL open and forEach statements:

```
VGVar.handleHardIOErrors  = 1;

try
  open selectEmp forUpdate for emp;
onException
  myErrorHandler(6);    // exits program
end

try
  forEach (from selectEmp)
    myRecord.empname = myRecord.empname + " " + "III";

    try
      replace emp;
    onException
      myErrorHandler(10);   // exits program
    end
  end  // end forEach; cursor is closed automatically
       // when the last row in the result set is read

onException

  // the exception block related to forEach is not run if the condition
  // is noRecordFound, so avoid the test "if (not noRecordFound)"
  myErrorHandler(8);  // exit the program
end

sysLib.commit();
```

Using SQL records with explicit SQL statements

Before using SQL records with explicit SQL statements, you declare an SQL record part. This part is different from the previous one, in the syntax for SQL item properties and in the use of a calculated value:

```
Record Employee type sqlRecord
  {
    tableNameVariables = [["empTable"]],
                 // use of a table-name variable
                 // means that the table is specified
                 // at run time
    keyItems = ["empnum"]
  }
  empnum decimal(6,0) { isReadonly = yes };
  empname char(40);

  // specify properties of a calculated column
  aValue decimal(6,0)
    { isReadonly = yes,
      column = "(empnum + 1) as NEWNUM" };
end
```

Declare variables:

```
emp Employee;
empTable char(40);
```

Adding a row to an SQL table: To prepare to add a row to an SQL table, place values in the EGL record and in a table name variable:

```
emp.empnum = 1;
emp.empname = "John";
empTable = "Employee";
```

Add an employee to the table by specifying the EGL add statement and modifying the SQL statement:

```
// a colon does not precede a table name variable
try
  add emp
    with #sql{
      insert into empTable (empnum, empname)
      values (:empnum, :empname || ' ' || 'Smith')
    }

onException
  myErrorHandler(8);
end
```

Reading rows from an SQL table: To prepare to read rows from an SQL table, identify a record key:

```
emp.empnum = 1;
```

Get a single row in any of these ways:

- Specify the EGL get statement in a way that generates a series of statements (DECLARE cursor, OPEN cursor, FETCH row, CLOSE cursor):

  ```
  try
    get emp into empname   // The into clause is optional. (It
                           // cannot be in the SELECT statement.)
    with #sql{
      select empname
      from empTable
      where empum = :empnum + 1
    }
  onException
    myErrorHandler(8);
  end
  ```

- Specify the EGL get statement in a way that generates a single SELECT statement:

  ```
  try
    get emp singleRow // The into clause is derived
                      // from the SQL record and is based
                      // on the columns in the select clause
    with #sql{
      select empname
      from empTable
      where empnum = :empnum + 1
    }
  onException
    myErrorHandler(8);
  end
  ```

Process multiple rows in either of these ways:

- Use the EGL open, get next, and while statements:

  ```
  try

  // The into clause is derived
  // from the SQL record and is based
  // on the columns in the select clause
    open selectEmp forUpdate
      with #sql{
        select empnum, empname
        from empTable
        where empnum >= :empnum
        order by NEWNUM       -- uses the calculated value
        for update of empname
      } for emp;
  onException
    myErrorHandler(8);        // exits the program
  ```

```
end

try
  get next emp;
onException
  myErrorHandler(9);    // exits the program
end

while (emp not noRecordFound)
  try
    replace emp
    with #sql{
      update :empTable
      set empname = :empname || ' ' || 'III'
    } from selectEmp;

  onException
    myErrorHandler(10); // exits the program
  end

  try
    get next emp;
  onException
    myErrorHandler(9);  // exits the program
  end
end  // end while

// no need to say "close emp;" because emp
// is closed automatically when the last
// record is read from the result set or
// (in case of an exception) when the program ends

sysLib.commit();
```

- Use the EGL open and forEach statements:

```
try

  // The into clause is derived
  // from the SQL record and is based
  // on the columns in the select clause
  open selectEmp forUpdate
    with #sql{
      select empnum, empname
      from empTable
      where empnum >= :empnum
      order by NEWNUM       -- uses the calculated value
      for update of empname
    } for emp;

onException
  myErrorHandler(8);        // exits the program
end

try
  forEach (from selectEmp)

    try
      replace emp
      with #sql{
        update :empTable
        set empname = :empname || ' ' || 'III'
      } from selectEmp;

    onException
      myErrorHandler(9);  // exits program
    end
```

```
    end   // end forEach statement, and there is
          // no need to say "close emp;" because emp
          // is closed automatically when the last
          // record is read from the result set or
          // (in case of an exception) when the program ends

  onException
    // the exception block related to forEach is not run if the condition
    // is noRecordFound, so avoid the test "if (not noRecordFound)"
    myErrorHandler(9);  // exits program
  end

  sysLib.commit();
```

Using EGL prepare statements

You have the option to use an SQL record part when coding the EGL prepare statement. Declare the following part:

```
Record Employee type sqlRecord
  {
    tableNames = [["employee"]],
    keyItems = ["empnum"],
    defaultSelectCondition =
      #sqlCondition{
        aTableColumn = 4 -- start each SQL comment
                         -- with a double hyphen
      }
  }

  empnum  decimal(6,0) {isReadonly=yes};
  empname char(40);
end
```

Declare variables:

```
emp Employee;
empnum02 decimal(6,0);
empname02 char(40);
myString char(120);
```

Adding a row to an SQL table: Before adding a row, assign values to variables:

```
emp.empnum = 1;
emp.empname = "John";
empnum02 = 2;
empname02 = "Jane";
```

Develop the SQL statement:

- Code the EGL prepare statement and reference an SQL record, which provides an SQL statement that you can customize:

  ```
  prepare myPrep
    from "insert into employee (empnum, empname) " +
    "values (?, ?)" for emp;

  // you can use the SQL record
  // to test the result of the operation
  if (emp is error)
    myErrorHandler(8);
  end
  ```

- Alternatively, code the EGL prepare statement without reference to an SQL record:

  ```
  myString = "insert into employee (empnum, empname) " +
    "values (?, ?)";

  try
  ```

```
  prepare addEmployee from myString;
onException
  myErrorHandler(8);
end
```

In each of the previous cases, the EGL prepare statement includes placeholders for data that will be provided by an EGL execute statement. Two examples of the execute statement are as follows:

- You can provide values from a record (SQL or otherwise):

  ```
  execute addEmployee using emp.empnum, emp.empname;
  ```

- You can provide values from individual items:

  ```
  execute addEmployee using empnum02, empname02;
  ```

Reading rows from an SQL table: To prepare to read rows from an SQL table, identify a record key:

```
empnum02 = 2;
```

You can replace multiple rows in either of these ways:

- Use the EGL open, while, and get next statements--

```
myString = "select empnum, empname from employee " +
           "where empnum >= ? for update of empname";

try
  prepare selectEmployee from myString for emp;
onException
  myErrorHandler(8);    // exits the program
end

try
  open selectEmp with selectEmployee
    using empnum02
    into emp.empnum, emp.empname;
onException
  myErrorHandler(9);    // exits the program
end

try
  get next from selectEmp;
onException
  myErrorHandler(10);   // exits the program
end

while (emp not noRecordFound)

  emp.empname = emp.empname + " " + "III";

  try
    replace emp
      with #sql{
        update employee
        set empname = :empname
      }
    from selectEmp;
  onException
    myErrorHandler(11); // exits the program
  end

  try
    get next from selectEmp;
  onException
    myErrorHandler(12); // exits the program
```

```
  end
end  // end while; close is automatic when last row is read

sysLib.commit();
```

- Use the EGL open and forEach statements--

```
myString = "select empnum, empname from employee " +
           "where empnum >= ? for update of empname";

try
  prepare selectEmployee from myString for emp;
onException
  myErrorHandler(8);    // exits the program
end

try
  open selectEmp with selectEmployee
    using empnum02
    into emp.empnum, emp.empname;
onException
  myErrorHandler(9);    // exits the program
end

try
  forEach (from selectEmp)
    emp.empname = emp.empname + " " + "III";

    try
      replace emp
        with #sql{
          update employee
          set empname = :empname
        }
      from selectEmp;
    onException
      myErrorHandler(11); // exits the program
    end
  end // end forEach; close is automatic when last row is read
onException

  // the exception block related to forEach is not run if the condition
  // is noRecordFound, so avoid the test "if (not noRecordFound)"
  myErrorHandler(12); // exits the program
end

sysLib.commit();
```

Default database

The default database is a relational database that is accessed when an SQL-related I/O statement runs in EGL-generated code and when no other database connection is current. The default database is available from the beginning of a run unit; however, you can dynamically connect to a different database for subsequent access in the same run unit, as described in *VGLib.connectionService*.

In relation to a Java run unit, the default database is specified in the optional run-time property **vgj.jdbc.default.database**, which receives a generated value from build descriptor option **sqlDB** if option **genProperties** is set to GLOBAL or PROGRAM at generation time.

In relation to an iSeries COBOL program, the idea of a database connection is not meaningful. The build descriptor option **destLibrary** identifies the library used at run time, and if you wish to reference another library, you must use the name of

that other library as a qualifier. For instance, if you need to access the SQL table SAMPLE in library LIBRARY02, you would refer to the table as LIBRARY02.SAMPLE.

Related concepts
"Java runtime properties" on page 327
"Run unit" on page 721
"SQL support" on page 213

Related tasks
"Setting up a J2EE JDBC connection" on page 341
"Setting up the J2EE run-time environment for EGL-generated code" on page 333
"Understanding how a standard JDBC connection is made" on page 245

Related reference
"destLibrary" on page 369
"Java runtime properties (details)" on page 525
"sqlDB" on page 384
"connectionService()" on page 888

Informix and EGL

The following rules are specific to Informix databases and EGL:

- An Informix database that is accessed by EGL or by an EGL-generated program must have transactions enabled.
- If you are coding an SQL statement and use a colon (:) when identifying an Informix table, use quote marks to separate the Informix identifier from the rest of the statement, as in these examples:

```
INSERT INTO "myDB:myTable"
  (myColumn) values (:myField)

INSERT INTO "myDB@myServer:myTable"
  (myColumn) values (:myField)
```

- If you are using the SQL retrieve feature of EGL to access data from a non-ANSI Informix database, make sure that any database column of type DECIMAL includes a scale value. Instead of defining a column as DECIMAL (4), for example, define the column as DECIMAL (4,0).
- If you intend to use the SQL retrieve feature to retrieve data from a table that is part of an Informix system schema, you must set a special preference, as described in *Setting preferences for SQL retrieve*.

Related concepts
"SQL support" on page 213

Related tasks
"Retrieving SQL table data"
"Setting preferences for SQL retrieve" on page 113

SQL-specific tasks

Retrieving SQL table data

EGL provides a way to create SQL record items from the definition of an SQL table, view, or join; for an overview, see *SQL support*.

Do as follows:

1. Ensure that you have set SQL preferences as appropriate. For details, see *Setting preferences for SQL retrieve*.
2. Decide where to do the task--
   - In an EGL source file, as you develop each SQL record; or
   - In the Outline view, as may be easier when you already have SQL records.
3. If you are working in the EGL source file, proceed in this way--
   a. If you do not have the SQL record, create it:
      1) Type **R**, press Ctrl-Space, and in the content-assist list, select one of the SQL table entries (usually **SQL record with table names**).
      2) Type the name of the SQL record; press Tab; and type a table name, or a comma-delimited list of tables, or the alias of a view.

      You also can create an SQL record by typing the minimal content, as appropriate if the name of the record is the same as the name of the table, as in this example:

      ```
      Record myTable type sqlRecord
      end
      ```
   b. Right-click anywhere in the record.
   c. In the context menu, click **SQL record > Retrieve SQL**.
4. If you are working in the Outline view, right click on the entry for the SQL record and, in the context menu, click **Retrieve SQL**.

Note: You cannot retrieve an SQL view that is defined with the DB2 condition WITH CHECK OPTIONS.

After you create record items, you may want to gain a productivity benefit by creating the equivalent dataItem parts; see *Overview on creating dataItem parts from an SQL record part*.

Related concepts
"Creating dataItem parts from an SQL record part (overview)"
"SQL support" on page 213

Related tasks
"Creating dataItem parts from an SQL record part" on page 237
"Setting preferences for SQL database connections" on page 111
"Setting preferences for SQL retrieve" on page 113

Related reference
"SQL item properties" on page 63

Creating dataItem parts from an SQL record part (overview)

After you declare structure items in an SQL record part, you can use a special mechanism in the EGL editor to create data item parts that are equivalent to the structure items. The benefit is that you can more easily create a non-SQL record (usually a basic record) for transferring data to and from the related SQL record at run time.

Consider the following structure items:

```
10 myHostVar01 CHAR(3);
10 myHostVar02 BIN(9,2);
```

You can request that dataItem parts be created:

```
DataItem myHostVar01 CHAR(3) end

DataItem myHostVar02 BIN(9,2) end
```

Another effect is that the structure item declarations are rewritten:

```
10 myHostVar01 myHostVar01;
10 myHostVar02 myHostVar02;
```

As shown in this example, each dataItem part is given the same name as the related structure item and acts as a typedef for the structure item. Each data item part is also available as a typedef for other structure items.

Before you can use a structure item as the basis of a dataItem part, the structure item must have a name, must have valid primitive characteristics, and must not point to a typedef.

Related concepts
"SQL support" on page 213

Related tasks
"Creating dataItem parts from an SQL record part"

Related reference
"DataItem part in EGL source format" on page 461
"SQL record part in EGL source format" on page 726

Creating dataItem parts from an SQL record part

After you declare structure items in an SQL record part, you can use a special mechanism in the EGL editor to create dataItem parts that are equivalent to the structure items. For general information, see *Overview on creating dataItem parts from an SQL record part*.

If the Outline view is not displayed, open that view by selecting **Show View** > **Outline** from the Window menu.

Do as follows in the Outline view:

1. For a given SQL record part, hold down **Ctrl** while clicking on each of the structure items of interest. To select all the structure items in a given record, click the topmost structure item, then hold down **Shift** while clicking on the bottommost structure item.
2. Right-click on the selected structure items.
3. In the context menu, click **Create DataItem part**.

The data-item parts are written at the bottom of the EGL source file, and each structure item is changed to refer to the equivalent part.

Related concepts
"Creating dataItem parts from an SQL record part (overview)" on page 236
"SQL support" on page 213

Related tasks
"Retrieving SQL table data" on page 235

Related reference
"SQL record part in EGL source format" on page 726

Creating EGL data parts from relational database tables

EGL Data Parts wizard

The EGL Data Parts wizard lets you create SQL record parts, as well as related data-item parts and library-based function parts, from one or more relational database tables or pre-existing views.

After connecting to the database, you can do as follows:

- Specify the SQL-record key fields that are used to create, read, update, or delete a row from a given database table or view.
- Customize explicit SQL statements for creating, reading, or updating a row. (The SQL statement for deleting a row cannot be customized.)
- Specify the SQL-record key fields that are used to select a set of rows from a given database or view.
- Customize an explicit SQL statement for selecting a set of rows.
- Validate and run each SQL statement

The output includes these files:

- An EGL source file that defines each record part
- An EGL library for each record part
- An EGL source file that contains all the data-item parts referenced by the structure items in the SQL record parts

You can reduce the number of files if you select the **Record and library in the same file** check box.

Related concepts
"SQL support" on page 213

Related tasks
"Creating, editing, or deleting a database connection for the EGL wizards" on page 239
"Creating EGL data parts from relational database tables"
"Customizing SQL statements in the EGL wizards" on page 240

Creating EGL data parts from relational database tables

To create EGL data parts from relational database tables without creating a separate Web application, do as follows:

1. Select **File** > **New** >**Other...**. A dialog is displayed for selecting a wizard.
2. Expand **EGL** and double-click **EGL Data Parts**. The EGL Data Parts dialog is displayed.
3. Enter an EGL or EGL Web project name, or select an existing project from the drop-down list. The parts will be generated into this project.
4. Select an existing database connection from the drop-down list or establish a new database connection--
   - To establish a new database connection, click **Add** and interact with the *New Database Connection Wizard*. For details on the kind of input required in a particular field, right click into the field and press F1.
   - For details on editing or deleting a database connection, see *Creating, editing, or deleting a database connection for the EGL wizards*.

   When a connection is made to the database, a list of database tables is displayed.

5. If you do not want to accept the default EGL file name for data items, type a new file name.
6. In the **Select your data** field, click on the name of the table whose columns will help you to declare data parts. To select multiple tables, hold down the **Ctrl** key while clicking on different table names. To transfer the highlighted name or names to the list of selected tables, click the right arrow.
7. For each of the selected tables (on the right), either specify the name of the EGL record to be created or accept the default name. To remove one or more tables from that list, highlight the entries of interest and click the left arrow.
8. If you want to include the library part and SQL record parts in the same file, select the check box.
9. Click **Next**.
10. A tab is available for each table. In each tab, select the key field to use when reading, updating, and deleting individual rows, then click the right arrow. To select multiple key fields, hold down the **Ctrl** key while clicking on different field names. To remove a key field from the list on the right, highlight the field name and click the left arrow.
11. Choose the selection condition field to use when selecting a set of rows, then click the right arrow. To select multiple fields, hold down the **Ctrl** key while clicking on different field names. To remove a field from the list on the right, highlight the field name and click the left arrow.
12. To customize an implicit SQL statement, see *Customizing SQL statements in the EGL wizards*. This option is not available for the EGL delete statement.
13. Click **Next**.
14. The Generate EGL Data Parts screen is displayed, including (at the bottom) a list of the files that will be produced:
    a. To change the name of the EGL project that will receive the EGL parts, type a project name in the Destination project field or select a project from the related drop-down list.
    b. To specify the EGL packages for a specific type of part (data or library), type a package name in the related field or select a name from the related drop-down list.
15. Click **Finish**.

Related concepts
"EGL Data Parts wizard" on page 238
"EGL Data Parts and Pages wizard" on page 173
"SQL support" on page 213

Related tasks
"Creating a single-table EGL Web application" on page 174
"Creating, editing, or deleting a database connection for the EGL wizards"
"Customizing SQL statements in the EGL wizards" on page 240

Creating, editing, or deleting a database connection for the EGL wizards: When you are at the first screen in an EGL wizard for creating data parts from a relational database table or for creating a Web application from a relational database table, you specify a database connection in either of two ways:

- Select an existing connection from a drop-down list; or
- Interact with the *New Database Connection Wizard*.

To use that wizard to create a connection, click **Add** and add information as required. For details on the kind of input required in a particular field, left click into the field and press F1.

To edit an existing database connection, do as follows:

1. Select **Window** > **Open Perspective** > **Other**. At the Select Perspective dialog, select the **Show all** check box and double-click **Data**.
2. In the Database Explorer view, right-click on the database connection, then select **Edit Connection**. Step through the pages of the database connection wizard and change information as appropriate. For help, press F1.
3. To complete the edit, click **Finish**.

To delete an existing database connection, do as follows:

1. Select **Window** > **Open Perspective** > **Other**. At the Select Perspective dialog, select the **Show all** check box and double-click **Data**.
2. In the Database Explorer view, right-click on the database connection, then select **Delete**.

Related concepts
"EGL Data Parts wizard" on page 238
"EGL Data Parts and Pages wizard" on page 173
"SQL support" on page 213

Related tasks
"Creating a single-table EGL Web application" on page 174
"Creating EGL data parts from relational database tables" on page 238
"Setting EGL preferences" on page 107

Customizing SQL statements in the EGL wizards: When you are using an EGL wizard to create data parts from a relational table or create a Web application from a relational table, you can change the SQL statement that is associated with an action like read or update:

1. Select an action from the Edit actions list, then click **Edit SQL**.
2. Edit the SQL statement (as is possible for all actions except delete), then click **Validate**. Validation ensures that the statement has the correct syntax and adheres to the rules for host variable names. If the statement contains errors, a message is displayed. Correct the errors and validate again.

   Revert to Last changes the statement into its last valid modified version. Previous versions become unavailable after you close the dialog.
3. Click **Execute**, then click **Execute** again.
4. If the SQL statement requires values for host variables, the Specify Variable Values dialog is displayed. Double-click the **Value** field to enter the value of a host variable, then press the **Enter** key. When you have entered values for all host variables, click **Finish**.

   Note: For host variables defined as type *character*, you must enclose the value in single quotes.
5. When you are finished executing the SQL statement, click **Close**.
6. When you are finished editing the SQL statements, click **OK**.

Related concepts
"EGL Data Parts wizard" on page 238
"EGL Data Parts and Pages wizard" on page 173
"SQL support" on page 213

Related tasks
"Creating a single-table EGL Web application" on page 174
"Creating EGL data parts from relational database tables" on page 238
"Creating, editing, or deleting a database connection for the EGL wizards" on page 239
"Setting EGL preferences" on page 107

Viewing the SQL SELECT statement for an SQL record

EGL provides an implicit SQL SELECT statement for a given SQL record part. To view the implicit SQL SELECT statement, do as follows:

1. Open the EGL file that contains the SQL record part. If you do not have the file open, right-click on the EGL file in the Project Explorer, then select **Open With** > **EGL Editor**.
2. Click inside the SQL record part, then right-click. A context menu displays.
3. Select **SQL Record** > **View Default Select**.
4. To validate the SQL SELECT statement against a database, click **Validate**.

Note: Before using the validate function, DB2 UDB users must set the DEFERREDPREPARE option. You can set this option interactively in the CLP (DB2 command line processor) using the **db2 update cli cfg for section COMMON using DEFERREDPREPARE 0** command. This command will put the keyword under the COMMON section. Execute the command **db2 get cli cfg for section common** to verify that the keyword is being picked up.

Related concepts
"SQL support" on page 213

Related tasks
"Validating the SQL SELECT statement for an SQL record"

Related reference
"SQL record part in EGL source format" on page 726

Validating the SQL SELECT statement for an SQL record

EGL provides an implicit SQL SELECT statement for a given SQL record part. To validate the implicit SQL SELECT statement against a database, do as follows:

1. Open the EGL file that contains the SQL record part. If you do not have the file open, right-click on the EGL file in the Project Explorer, then select **Open With** > **EGL Editor**.
2. Click inside the SQL record part, then right-click. A context menu displays.
3. Select **SQL Record** > **Validate Default Select**.

Note: Before using the validate function, DB2 UDB users must set the DEFERREDPREPARE option. You can set this option interactively in the CLP (DB2 command line processor) using the **db2 update cli cfg for section COMMON using DEFERREDPREPARE 0** command. This command will

put the keyword under the COMMON section. Execute the command **db2 get cli cfg for section common** to verify that the keyword is being picked up.

Related concepts
"SQL support" on page 213

Related tasks
"Viewing the SQL SELECT statement for an SQL record" on page 241

Related reference
"SQL record part in EGL source format" on page 726

Constructing an EGL prepare statement

Within a function, you can construct the following kinds of EGL statements that are based on an SQL record part:

- An EGL **prepare** statement; and
- The related EGL **execute**, **open**, or **get** statement.

Do as follows:

1. Open an EGL file with the EGL editor. The file must contain a function and a coded SQL statement. If you do not have a file open, right-click on the EGL file in the workbench Project Explorer, then select **Open With** > **EGL Editor**.
2. Click inside the function at the location where the EGL **prepare** statement will reside, then right-click. A context menu displays.
3. Select **Add SQL Prepare Statement**.
4. Type a name to identify the EGL **prepare** statement. For rules, see *Naming conventions*.
5. If you have an SQL record variable defined, select it from the drop-down list. The corresponding SQL record part name displays. If you do not have an SQL record variable defined, you can type a name in the SQL record variable name field, then select an SQL record part name using the **Browse** button. You must eventually define an SQL record variable with that name in the EGL source code.
6. Select an execution statement type from the drop-down list.
7. If the execution statement is of type open, enter a result-set identifier.
8. Click **OK**. EGL statements are constructed inside the function.

Related concepts
"SQL support" on page 213

Related tasks
"Validating the SQL SELECT statement for an SQL record" on page 241
"Viewing the SQL SELECT statement for an SQL record" on page 241

Related reference
"Naming conventions" on page 652
"SQL record part in EGL source format" on page 726

Constructing an explicit SQL statement from an implicit one

EGL provides an implicit SQL statement for each SQL-related EGL input/output (I/O) statement. To construct an explicit SQL statement from an implicit one, do as follows:

1. Open the EGL file that contains the EGL I/O statement. If you do not have the file open, right-click on the EGL file in the Project Explorer, then select **Open With** > **EGL Editor**.
2. Click on the EGL I/O statement, then right-click. A context menu displays.
3. To construct an explicit SQL statement without an INTO clause, select **SQL Statement** > **Add**. To construct an explicit SQL statement with an INTO clause, select **SQL Statement** > **Add with Into**. The implicit SQL statement is appended to the EGL I/O statement making it an explicit SQL statement.

   Note: The INTO clause is only valid with **open**, **get**, and **get next** statements.

Related concepts
"SQL support" on page 213

Related tasks
"Removing an SQL statement from an SQL-related EGL statement" on page 244
"Resetting an explicit SQL statement" on page 244
"Validating an implicit or explicit SQL statement"
"Viewing the implicit SQL for an SQL-related EGL statement"

Viewing the implicit SQL for an SQL-related EGL statement

EGL provides an implicit SQL statement for each SQL-related EGL input/output (I/O) statement. To view the implicit SQL for an EGL I/O statement, do as follows:

1. Open the EGL file that contains the EGL I/O statement. If you do not have the file open, right-click on the EGL file in the Project Explorer, then select **Open With** > **EGL Editor**.
2. Click on the EGL I/O statement, then right-click. A context menu displays.
3. Select **SQL Statement** > **View**.

Related concepts
"SQL support" on page 213

Related tasks
"Constructing an explicit SQL statement from an implicit one"
"Removing an SQL statement from an SQL-related EGL statement" on page 244
"Resetting an explicit SQL statement" on page 244
"Validating an implicit or explicit SQL statement"

Validating an implicit or explicit SQL statement

To validate an implicit or explicit SQL statement against a database, do as follows:

1. Open the EGL file that contains the SQL-related EGL statement or explicit SQL statement. If you do not have the file open, right-click on the EGL file in the Project Explorer, then select **Open With** > **EGL Editor**.
2. Click the EGL statement or SQL statement, then right-click. A context menu displays.
3. Select **SQL Statement** > **Validate**.

Note: Before using the validate function, DB2 UDB users must set the DEFERREDPREPARE option. You can set this option interactively in the

CLP (DB2 command line processor) using the **db2 update cli cfg for section COMMON using DEFERREDPREPARE 0** command. This command will put the keyword under the COMMON section. Execute the command **db2 get cli cfg for section common** to verify that the keyword is being picked up.

Related concepts
"SQL support" on page 213

Related tasks
"Constructing an explicit SQL statement from an implicit one" on page 243
"Removing an SQL statement from an SQL-related EGL statement"
"Resetting an explicit SQL statement"
"Viewing the implicit SQL for an SQL-related EGL statement" on page 243

Resetting an explicit SQL statement

EGL provides an implicit SQL statement for each SQL-related EGL input/output (I/O) statement. An implicit SQL statement can be appended to an EGL I/O statement making it an explicit SQL statement. If you change the explicit SQL statement, do as follows to return to an SQL statement based on the implicit SQL:

1. Open the EGL file that contains the explicit SQL statement. If you do not have the file open, right-click on the EGL file in the Project Explorer, then select **Open With** > **EGL Editor**.
2. Click on the explicit SQL statement, then right-click. A context menu displays.
3. Select **SQL Statement** > **Reset**.

Related concepts
"SQL support" on page 213

Related tasks
"Constructing an explicit SQL statement from an implicit one" on page 243
"Removing an SQL statement from an SQL-related EGL statement"
"Validating an implicit or explicit SQL statement" on page 243
"Viewing the implicit SQL for an SQL-related EGL statement" on page 243

Removing an SQL statement from an SQL-related EGL statement

EGL provides an implicit SQL statement for each SQL-related EGL input/output (I/O) statement. An implicit SQL statement can be appended to an EGL I/O statement making it an explicit SQL statement (see *Constructing an explicit SQL statement from an implicit one)*. To remove the appended SQL statement, do as follows:

1. Open the EGL file that contains the explicit SQL statement. If you do not have the file open, right-click on the EGL file in the Project Explorer, then select **Open With** > **EGL Editor**.
2. Click on the explicit SQL statement, then right-click. A context menu displays.
3. Select **SQL Statement** > **Remove**. The EGL I/O statement remains.

Related concepts
"SQL support" on page 213

Related tasks
"Constructing an explicit SQL statement from an implicit one" on page 243

"Resetting an explicit SQL statement" on page 244
"Validating an implicit or explicit SQL statement" on page 243
"Viewing the implicit SQL for an SQL-related EGL statement" on page 243

Resolving a reference to display an implicit SQL statement

Consider what happens when you specify the following EGL statement:

```
open myRecord;
```

When the EGL editor tries to create a default SQL statement, the editor attempts to find a variable named myRecord and to identify the SQL record part on which that variable is based. If the variable is unavailable at development time or if the variable is undeclared, the editor attempts to use an SQL record part named myRecord as the basis for the default SQL statement. The editor assumes that you intend to create a variable whose name is the name of the SQL record part.

If you wish to store an SQL-related function in a file that does not include the variable myRecord, you can do as follows:

1. In the program part, declare the global variable
2. Create the function as a nested function in the program part
3. Create the default SQL statement and modify it as appropriate; then, save the file
4. Move the function to the other file

After the function is moved from the program part, the record name cannot be resolved at development time, and the editor cannot display any default SQL statements that are based on that record.

Related concepts
"SQL support" on page 213

Understanding how a standard JDBC connection is made

A standard JDBC connection is created for you at run time if you are debugging a generated Java program and if the program properties file includes the necessary values. For details on the meaning of the program properties, including details on how the values are derived, see *Java run-time properties (details)*.

The JDBC connection is based on the following kinds of information:

Connection URL

If your code tries to access a database before invoking the system function sysLib.connect or VGLib.connectionService, the connection URL is the value of property **vgj.jdbc.default.database**.

If your code tries to access a database in response to an invocation of the system function sysLib.connect or VGLib.connectionService, the connection URL is the value of property **vgj.jdbc.database***SN*.

For details on the format of a connection URL, see *sqlValidationConnectionURL*.

User ID

If your code tries to access a database before invoking the system function sysLib.connect or VGLib.connectionService, the user ID is the value of property **vgj.jdbc.default.userid**.

If your code tries to access a database in response to an invocation of one of those system functions, the user ID is a value specified in the invocation.

Password
If your code tries to access a database before invoking the system function sysLib.connect or VGLib.connectionService, the password is the value of property **vgj.jdbc.default.password**.

If your code tries to access a database in response to an invocation of one of those system functions, the password is a value specified in the invocation. You can use a system function to avoid exposing the password in the program properties file.

JDBC driver class
The JDBC driver class is the value of property **vgj.jdbc.drivers**.

Related concepts
"Program properties file" on page 329

Related tasks
"Setting up a J2EE JDBC connection" on page 341
"Setting up the J2EE run-time environment for EGL-generated code" on page 333

Related reference
"connect()" on page 867
"connectionService()" on page 888
"genProperties" on page 375
"Java runtime properties (details)" on page 525
"JDBC driver requirements in EGL" on page 543
"sqlDB" on page 384
"sqlID" on page 385
"sqlPassword" on page 387
"sqlValidationConnectionURL" on page 387
"sqlJDBCDriverClass" on page 386

VSAM support

EGL-generated COBOL code can access local or remote VSAM files. VSAM support for EGL-generated Java code is as follows:

- AIX-based code can access local VSAM files
- The following code can access remote VSAM files on z/OS:
  - EGL-generated Java code that runs on Windows 2000/NT/XP
  - The EGL debugger, which runs on Windows 2000/NT/XP

Access prerequisites

Access requires that you first define the VSAM file on the system where you want the file to reside. Remote access from Windows 2000/NT/XP (whether for the EGL debugger or at run time) also requires that you install Distributed File Manager (DFM) on the workstation as follows:

1. Locate the following file in your EGL installation directory:
   ```
   workbench\bin\VSAMWIN.zip
   ```
2. Unzip the file into a new directory and follow the directions in the INSTALL.README file.

System name

To access a local VSAM file, specify the system name in the resource associations part and use the naming convention that is appropriate to the operating system. To

access a remote VSAM file from the EGL debugger or from EGL-generated Java code, specify the system name in the following way:

```
\\machineName\qualifier.fileName
```

machineName
The SNA LU alias name as specified in the SNA configuration

qualifier.fileName
The VSAM data set name, including a qualifier

The naming convention is similar to the Universal Naming Convention (UNC) format. For details on UNC format, refer to the *Distributed FileManager User's Guide*, which is in the following file in your EGL installation directory:

```
workbench\bin\VSAMWIN.zip
```

MQSeries support

EGL supports access of MQSeries message queues on any of the target platforms. You can provide such access in either of the following ways:

- Use MQSeries-related EGL keywords like **add** and **get next** on an MQ record; in this case, EGL hides details of MQSeries so you can focus on the business problem your code is addressing
- Invoke EGL functions that call MQSeries commands directly, in which case some commands are available that are not supported by the EGL keywords

You can mix the two approaches in a given program. For most purposes, however, you use one or the other approach exclusively.

Regardless of your approach, you can control various run-time conditions by customizing *options records*, which are global basic records that EGL run-time services passes on calls to MQSeries. When you declare an options record as a program variable, you can use an EGL-installed options record part as a typedef; or you can copy the installed part into your own EGL file, customize the part, and use the customized part as a typedef.

Your approach determines how EGL run-time services makes the options records available to MQSeries:

- If you are working with the EGL **add** and **get next** statements, you identify the options records when you specify properties of an MQ record. If you do not identify a particular options record, EGL uses a default.
- If you are invoking the EGL functions that call MQSeries directly, you use options records as arguments when you invoke the functions. Defaults are not available in this case.

For details on options records and on the values that are passed to MQSeries by default, see *Options records for MQ records*. For details on MQSeries itself, refer to these documents:

- *An Introduction to Messaging and Queueing* (GC33-0805–01)
- *MQSeries MQI Technical Reference* (SC33-0850)
- *MQSeries Application Programming Guide* (SC33-0807-10)
- *MQSeries Application Programming Reference* (SC33-1673-06)

Connections

You connect to a queue manager (called the *connecting queue manager*) the first time you invoke a statement from the following list:

- An EGL **add** or **get next** statement that accesses a message queue
- An invocation of the EGL function MQCONN or MQCONNX

You can access only one connecting queue manager at a time; however, you can access multiple queues that are under the control of the connecting queue manager. If you wish to connect directly to a queue manager other than the current connecting queue manager, you must disconnect from the first by invoking MQDISC, then connect to the second queue manager by invoking **add**, **get next**, MQCONN, or MQCONNX.

You can also access queues that are under the control of a *remote queue manager*, which is a queue manager with which the connecting queue manager can interact. Access between the two queue managers is possible only if MQSeries itself is configured to allow for that access.

Access to the connecting queue manager is terminated when you invoke MQDISC or when your code ends.

Include message in transaction

You can embed queue-access statements in a unit of work so that all your changes to the queues are committed or rolled back at a single processing point. If a statement is in a unit of work, the following is true:

- An EGL **get next** statement (or an EGL MQGET invocation) removes a message only when a commit occurs
- The message placed on a queue by an EGL **add** statement (or an EGL MQPUT invocation) is visible outside the unit of work only when a commit occurs

When queue-access statements are not in a unit of work, each change to a message queue is committed immediately.

An MQSeries-related EGL **add** or **get next** statement is embedded in a unit of work if the property **includeMsgInTransaction** is in effect for the MQ record. The generated code includes these options:

- For MQGET, MQGMO_SYNCPOINT
- For MQPUT, MQPMO_SYNCPOINT

If you do not specify the property **includeMsgInTransaction** for an MQ record, the queue-access statements run outside of a unit of work. The generated code includes these options:

- For MQGET, MQGMO_NO_SYNCPOINT
- For MQPUT, MQPMO_NO_SYNCPOINT

When your code ends a unit of work, EGL commits or rolls back *all* recoverable resources being accessed by your program, including databases, message queues, and recoverable files. This outcome occurs whether you use the system functions (**sysLib.commit**, **sysLib.rollback**) or the EGL calls to MQSeries (MQCMIT, MQBACK); the appropriate EGL system function is invoked in either case.

A rollback occurs if an EGL program terminates early because of an error detected by EGL run-time services.

Customization

If you wish to customize your interaction with MQSeries rather than relying on the default processing of **add** and **get next** statements, you need to review the information in this section.

EGL dataTable part

A set of EGL dataTable parts is available to help you interact with MQSeries. Each part allows EGL-supplied functions to retrieve values from memory-based lists at run time. The next section includes details on how data tables are deployed.

Making customization possible

To make customization possible, you must bring a variety of installed EGL files into your project *without changing them in any way*. The files are as follows:

records.egl
Contains basic record parts that can be used as typedefs for the options records that are used in your program; also includes structure parts that are used by those records and that give you the flexibility to develop record parts of your own

functions.egl
Contains two sets of functions:

- MQSeries command functions, which access MQSeries directly
- Initialization functions, which let you place initial values in the options records that are used in your program

mqrcode.egl, mqrc.egl, mqvalue.egl
Contains a set of EGL dataTable parts that are used by the command and initialization functions

Your tasks are as follows:

1. Using the process for importing files into the workbench, bring those files into an EGL project. The files reside in the following directory:

   ```
   installationDir\egl\eclipse\plugins\
   com.ibm.etools.egl.generators_version\MqReusableParts
   ```

   installationDir
   The product installation directory, such as C:\Program Files\IBM\RSPD\6.0. If you installed and kept a Rational Developer product before installing the product that you are using now, you may need to specify the directory that was used in the earlier install.

   version
   The latest version of the plugin; for example, 6.0.0

2. To make the parts more easily available to your program, write one or more *EGL* import statements in the file that contains your program. If the files to be imported reside in a project other than the one in which you are developing code, make sure that your project references the other project.

   For details, see *Import*.

3. In your program, declare global variables:
   - Declare MQRC, MQRCODE, and MQVALUE, each of which must use as a typedef the dataTable part that has the same name as the variable.
   - For each options record that you wish to pass to MQSeries, declare a basic record that uses an options record part as a typedef. For details on each part, see *Options records for MQ records*.

4. In your function, initialize the options records that you intend to pass to MQSeries. You can do this easily by invoking the imported EGL initialization function for a given options record. The name of each function is the name of the part that is used as a typedef for the record, followed by _INIT. An example is MQGMO_INIT.
5. Set values in the options records. In many cases you set a value by assigning an EGL symbol that represents a constant, each of which is based on a symbol described in the MQSeries documentation. You can specify multiple EGL symbols by summing individual ones, as in this example:

```
MQGMO.GETOPTIONS = MQGMO_LOCK
                 + MQGMO_ACCEPT_TRUNCATED_MSG
                 + MQGMO_BROWSE_FIRST
```

MQSeries-related EGL keywords

When you work with the MQSeries-related EGL keywords like *add* and *scan*, you define an MQ record for each message queue you wish to access. The record layout is the format of the message.

The next table lists the keywords.

Keyword	Purpose
add	Places the content of an MQ record at the end of the specified queue. The EGL add statement invokes as many as three MQSeries commands: • MQCONN connects the generated code to a queue manager and is invoked when no connection is active. • MQOPEN establishes a connection to a queue and is invoked when a connection is active but the queue is not open. • MQPUT puts the record in the queue and is always invoked unless an error occurred in an earlier MQSeries call. After adding an MQ record, you must close a message queue before reading an MQ record from the same queue.
close	Relinquishes access to the message queue that is associated with an MQ record. The EGL close statement invokes the MQSeries MQCLOSE command, which also is invoked automatically when your program ends. You should close the message queue after an add or scan if another program requires access to the queue. The close is particularly appropriate if your program runs for a long time and no longer needs access.
scan	Reads the first message in a queue into a message queue record and (by default) removes the message from the queue. The EGL scan statement invokes as many as three MQSeries commands: • MQCONN connects the generated code to a queue manager and is invoked when no connection is active. • MQOPEN establishes a connection to a queue and is invoked when a connection is active but the queue is not open. • MQGET removes the record from the queue and is always invoked unless an error occurred in an earlier MQSeries call. After reading an MQ record, you must close the queue before adding an MQ record to the same queue.

Manager and queue specification

When you work with the MQSeries-related EGL keywords, you identify a queue in the following situations:

- At declaration time, you specify a logical queue name, and you do so by setting the **queueName** property of the MQ record part. That logical queue name acts as a default for the queue name accessed at run time; but in most cases the name is meaningful only as way of associating the MQ record with a physical queue. The logical queue name can be no more than 8 characters.
- At generation time, you control the generation process with a buildDescriptor part that in turn can reference a resource associations part. The resource associations part associates the queue name with the name of a physical queue.
- At run time, your code can change the value in the record-specific variable ***record*.resourceAssociation** to override any queue name you specified at declaration or generation time.

The name of the physical queue has the following format:

```
queueManagerName:physicalQueueName
```

queueManagerName
Name of the queue manager; if this name is omitted, the colon is omitted, too

physicalQueueName
Name of the physical queue, as known to the specified queue manager

The first time that you issue an add or scan statement against a message queue record, a connecting queue manager must be specified, whether by default or otherwise. In the simplest case, you do not specify a connecting queue manager at all, but rely on a default value in the MQSeries configuration.

The record-specific variable ***record*.resourceAssociation** always contains at least the name of the message queue for a given MQ record.

Remote message queues

If you want to access a queue that is controlled by a remote queue manager, you must do the following:

- Issue the EGL close statement to relinquish access to the queue now in use
- Set the record-specific variable ***record*.resourceAssociation** to ensure later access of the remote queue

You set ***record*.resourceAssociation** in one of two ways, depending on how the queue-manager relationships are established in MQSeries:

- If the connecting queue manager has a local definition of the remote queue, set ***record*.resourceAssociation** as follows:
  - Accept the same value for the connecting queue manager (either by specifying the name of the connecting queue manager or by specifying no name; in the latter case, omit the colon).
  - Specify the name of the local definition of the remote queue.

  Your next use of the *add* or *scan* statement issues an MQOPEN to establish access to the remote queue.
- Alternatively, set ***record*.resourceAssociation** with the name of the remote queue manager, along with the name of the remote queue. The connecting queue manager does not change in this case. Your next use of the *add* or *scan* statement issues MQOPEN and uses the connection already in place.

Related concepts
"Direct MQSeries calls"
"MQSeries support" on page 247

Related reference
"MQ record properties" on page 644
"Options records for MQ records" on page 645

Direct MQSeries calls

You can use a set of installed EGL functions that mediate between your code and MQSeries, as described in *MQSeries support*.

The next table lists the available functions and identifies the required arguments. MQBACK (MQSTATE), for example, indicates that when you invoke MQBACK you pass an argument that is based on the MQSTATE record part. The record parts are described later.

MQSeries-related EGL function invocation	Effect
MQBACK (*MQSTATE*)	Invokes the system function sysLib.rollback to rollback a logical unit of work. The rollback affects *all* recoverable resources being accessed by your program, including databases, message queues, and recoverable files.
MQBEGIN (*MQSTATE*, *MQBO*)	Begins a logical unit of work.
MQCHECK_COMPLETION (*MQSTATE*)	Sets the mqdescription field of the record that is based on MQSTATE. The setting is based on the last-returned reason code. The function MQCHECK_COMPLETION is called automatically from the EGL functions MQBEGIN, MQCLOSE, MQCONN, MQCONNX, MQDISC, MQGET, MQINQ, MQOPEN, MQPUT, MQPUT1, and MQSET.
MQCLOSE (*MQSTATE*)	Closes the message queue to which MQSTATE.hobj refers.
MQCMIT (*MQSTATE*)	Invokes the system function sysLib.commit to commit a logical unit of work. The commit affects *all* recoverable resources being accessed by your program, including databases, message queues, and recoverable files.
MQCONN (*MQSTATE*, *qManagerName*)	Connects to a queue manager, which is identified by *qManagerName*, a string of up to 48 characters. MQSeries sets the connection handle (MQSTATE.hconn) for use in subsequent calls. **Note:** Your code can be connected to one queue manager at a time.
MQCONNX(*MQSTATE*, *qManagerName*, *MQCNO*)	Connects to a queue manager with options that control the way that the call works. The queue manager is identified by *qManagerName*, a string of up to 48 characters. MQSeries sets the connection handle (MQSTATE.hconn) for use in subsequent calls.
MQDISC (*MQSTATE*)	Disconnects from a queue manager.

MQSeries-related EGL function invocation	Effect
MQGET(*MQSTATE*, *MQMD*, *MQGMO*, *BUFFER*)	Reads and removes a message from the queue. The buffer cannot be more that 32767 bytes, but that restriction does not apply if you are using the EGL get next statement.
MQINQ(*MQSTATE*, *MQATTRIBUTES*)	Requests attributes of a queue.
MQNOOP()	Used only by EGL.
MQOPEN(*MQSTATE*, *MQOD*)	Opens a message queue. MQSeries sets the queue handle (MQSTATE.hobj) for use in subsequent calls.
MQPUT(*MQSTATE*, *MQMD*, *MQPMO*, *BUFFER*)	Adds a message to the queue. The buffer cannot be more that 32767 bytes, but that restriction does not apply if you are using the EGL **add** statement.
MQPUT1(*MQSTATE*, *MQOD*, *MQMD*, *MQPMO*, *BUFFER*)	Opens a queue, writes a single message, and closes the queue.
MQSET(*MQSTATE*, *MQATTRIBUTES*)	Sets attributes of a queue.

The next table lists the options records that are used as arguments when you invoke the MQSeries-related EGL functions. Also listed is the initialization function that should invoked for a given argument.

Your first step is to initialize the argument that is based on the MQSTATE record part. In the following example (as in the table that follows), the argument name is assumed to be the same as the name of the record part:

```
MQSTATE_INIT(MQSTATE);
```

Argument (the record part name)	Initialization function	Description	For Java or COBOL output?
MQATTRIBUTES	none	Arrays of attributes and attribute selectors, plus other information used in the command MQINQ or MQSET	Either
MQBO	MQBO_INIT (MQBO)	Begin options	Either
MQCNO	MQCNO_INIT (MQCNO)	Connect options	Either
MQDH	MQDH_INIT (MQDH)	Distribution header	COBOL only
MQDLH	MQDLH_INIT (MQDLH)	Dead-letter header	COBOL only
MQGMO	MQGMO_INIT (MQGMO)	Get-message options	Either
MQIIH	MQIIH_INIT (MQIIH)	IMS™ information header; describes information that is required at the start of an MQSeries message sent to IMS	Either; however, MQSeries documentation indicates that use of this header is not supported on Windows 2000/NT/XP

Argument (the record part name)	Initialization function	Description	For Java or COBOL output?
MQINTATTRS	none	Arrays of integer attributes for use in the command MQINQ or MQSET	Either
MQMD	MQMD_INIT (MQMD, MQSTATE)	Message descriptor (MQSeries version 2)	Either
MQMD1	MQMD1_INIT (MQMD1, MQSTATE)	Message descriptor (MQSeries version 1)	COBOL only
MQMDE	MQMDE_INIT (MQMDE, MQSTATE)	Message descriptor extension	Supported for COBOL; but for Java, use only the fields that are in MQSeries version 2
MQOD	MQOD_INIT (MQOD)	Object descriptor	Either
MQOO	MQOO_INIT (MQOO)	Open options	Either
MQOR	MQOR_INIT (MQOR)	Object record	COBOL only
MQPMO	MQPMO_INIT (MQPMO)	Put-message options	Either
MQRMH	MQRMH_INIT (MQRMH, MQSTATE)	Message reference header	COBOL only
MQRR	MQRR (MQRR)	Response record	COBOL only
MQSELECTORS	none	An array of attribute selectors, used only if you wish to access MQSeries without use of EGL functions	Either
MQSTATE	MQSTATE_INIT (MQSTATE)	A collection of arguments that are each used in one or more calls to MQSeries; for example, when you connect with the EGL function MQCONN or MQCONNX, MQSeries sets the connection handle (MQSTATE.hconn) for use in subsequent calls	Either
MQTM	MQTM_INIT (MQTM)	Trigger message	COBOL only
MQTMC2	MQTMC2_INIT (MQTMC2)	Trigger message 2 (character format)	COBOL only

Argument (the record part name)	Initialization function	Description	For Java or COBOL output?
MQXQH	MQXQH_INIT (MQXQH, MQSTATE)	Transmission queue header	Either

As shown, the supported arguments are more numerous when you generate in COBOL than when you generate in Java.

Note: The record parts each contain only one structure item, and the structure item uses a structure part as a typeDef. This setup gives you maximum flexibility. You can create your own record parts that are each composed of a series of structure parts.

The name of each structure part is the name of the record part followed by _S; the record part MQGMO, for example, uses a structure part named MQGMO_S.

Related concepts
"MQSeries-related EGL keywords" on page 250
"MQSeries support" on page 247
"Record parts" on page 124
"Typedef" on page 25

Related reference
"get next" on page 579
"commit()" on page 866
"rollback()" on page 878

Maintaining EGL code

Line commenting EGL source code

To comment one line of code, do as follows:

1. Click on the line, then right-click. A context menu is displayed.
2. Select **Comment**. Comment indicators (//) are placed at the beginning of the line.

To comment multiple consecutive lines of code, do as follows:

1. Click on the starting line. Holding down the left mouse button, drag the cursor to the ending line. Release the mouse button, and the range of lines is highlighted.
2. Right-click, then select **Comment** from the context menu. Comment indicators (//) are placed at the beginning of each of the lines in the selected range.

Use the same procedures to uncomment lines, but select **Uncomment** from the context menu.

Related tasks
"Creating an EGL source file" on page 120
"Opening a part in an .egl file" on page 259

Related reference
"EGL editor" on page 471

Searching for parts

If you have a file open in the EGL editor, you can search for parts after setting the search criteria:

1. Open an EGL file. You cannot use the search facility unless the EGL editor is active; however, your search is not limited to the file that is open in the editor.
2. On the Workbench menu, click **Search** > **EGL**. The Search dialog is displayed.
3. If the EGL Search tab is not already displayed, click **EGL Search**. Notice that the conditions specified throughout the Search tab can affect the results.
4. Type the name of a part you want to locate; or to display a list of parts with names that match a specific pattern of characters, embed wildcard symbols in the name:
   - A question mark (?) represents any one character
   - An asterisk (*) represents a series of any characters

   For example, type *myForm?Group* to locate parts named myForm1Group and myForm2Group, but not myForm10Group. Type *myForm*Group* to locate parts named myForm1Group, myForm2Group, and myForm10Group.
5. To make the search case-sensitive (so that myFormGroup is different from MYFORMGROUP), click the check box.
6. In the Search For box, select a type of part, or select **Any element** to expand your search to all part types.
7. In the Limit To box, select the option to limit your search to part declarations, part references, or both.

8. In the Scope box, select **Workspace** to search your workspace, **Enclosing Projects** to search the project that is currently highlighted in Project Explorer, or **Working Set** to search a defined set of projects. If you choose the Working Set scope, click the **Choose** button to select an existing working set or to define a new working set.
9. Click the **Search** button. The results of the search are displayed in the Search view.
10. If you double-click a file in the Search view, the file opens in the EGL editor, and the matching part is highlighted. If there is more than one match in the file, the first match is highlighted.

    Arrows in the left margin of the editor indicate the locations of each matching part.

Related concepts
"Parts" on page 17

Related tasks
"Opening a part in an .egl file" on page 259

Related reference
"EGL editor" on page 471

Viewing part references

You can display a hierarchical view of the EGL parts that are referenced in a program, library, PageHandler, or report handler part; and you can access those parts:

1. Open the Parts Reference view in one of two ways:
   - In the Project Explorer, right-click on an EGL file that contains a program, library, PageHandler, or report handler part. Select **Open in Parts Reference**.
   - Alternatively, open an EGL file in the EGL editor:
     a. If the Outline view is not displayed, open that view by selecting **Show View** > **Outline** from the Window menu.
     b. In the Outline view, right-click on a file, then click **Open in Parts Reference**.
2. The program, library, PageHandler, or report handler part is at the top level of the hierarchy; each referenced part is a sub-item in that hierarchy; and for each part, the view displays parameters, data declarations, use declarations, and functions, as appropriate.
3. Double-click on a part. The related source file opens in the EGL editor, and the part name is highlighted.

Related concepts
"EGL projects, packages, and files" on page 13
"Parts" on page 17

Related tasks
"Locating an EGL source file in the Project Explorer" on page 259
"Opening a part in an .egl file" on page 259

Related reference
"EGL editor" on page 471

Opening a part in an .egl file

With a few keystrokes you can access an EGL part other than a build part, anywhere in your workspace:

1. In the workbench, click **Navigate > Open Part** or click the **Open Part** button on the toolbar. The Open Part dialog is displayed.
2. Type the name of the part you want to locate; or to display a list of parts with names that match a specific pattern of characters, embed wildcard symbols in the name:
   - A question mark (?) represents any one character
   - An asterisk (*) represents a series of any characters

   For example, type *myForm?Group* to locate parts named myForm1Group and myForm2Group, but not myForm10Group. Type *myForm*Group* to locate parts named myForm1Group, myForm2Group, and myForm10Group.

   As you type the name, qualifying parts are displayed in the Open Part dialog, in the Matching parts section.
3. From the list of parts, select the part you want to open. The dialog's Qualifier section displays the path containing the folder, project, package, and source file that holds the selected part. In the event that multiple parts have the same name, select a part by clicking on the path of the file you want to open.
4. Click **OK**. The source file containing the part you selected opens in the EGL editor, with the part name highlighted.

Related concepts
"EGL projects, packages, and files" on page 13
"Parts" on page 17

Related tasks
"Creating an EGL source file" on page 120
"Locating an EGL source file in the Project Explorer"

Related reference
"EGL editor" on page 471

Locating an EGL source file in the Project Explorer

If you are editing an EGL source file, you can quickly locate the file in the Project Explorer view. The Show in Project Explorer context menu option does the following:

- Opens the Project Explorer view, if it is not already open
- Expands the Project Explorer tree nodes needed to locate the source file
- Highlights the source file

To locate an EGL source file in the Project Explorer, do as follows:

1. Right-click within the editor area of an open EGL source file. A context menu displays.
2. Select **Show in Project Explorer** from the context menu.

Related tasks
"Creating an EGL source file" on page 120
"Opening a part in an .egl file"

Related reference
"EGL editor" on page 471

Deleting an EGL file in the Project Explorer

To delete an EGL file in the Project Explorer, do this:

1. Click the EGL file and press the Delete key. Alternatively, right-click the EGL file and when the context menu is displayed, select **Delete**.
2. You will be asked to confirm that you want to delete the file. Click **Yes** to delete the file or **No** to cancel the deletion.

Related tasks
"Creating an EGL source file" on page 120
"Locating an EGL source file in the Project Explorer" on page 259

Debugging EGL code

EGL debugger

When you are in the Workbench, the EGL debugger lets you debug EGL code without requiring that you first generate output. These categories are in effect:

- To debug PageHandlers, as well as programs used in a J2EE context, you can use the local WebSphere Application Server test environment in debug mode--
  - You must use that environment for all code that runs under J2EE in a Web application.
  - You may use that environment for programs that run in a batch application under J2EE.
- To debug other code (batch applications that do not run under J2EE; or text applications), use a launch configuration that is outside of the WebSphere Test Environment. In this case, you can start the debug session with a few keystrokes.

If you are working on a batch program that you intend to deploy in a J2EE context, you can use the launch configuration to debug the program in a non-J2EE context. Although your setup is simpler, you need to adjust some values:

- You need to set the value of the build descriptor option J2EE to NO when you use the launch configuration.
- Also, you need to adjust Java property values to conform to differences in accessing a relational database--
  - For J2EE you specify a string like *jdbc/MyDB*, which is the name to which a data source is bound in the JNDI registry. You specify that string in these ways:
    - By setting the build descriptor option sqlJNDIName; or
    - By placing a value in the EGL SQL Database Connections preference page, in the Connection JNDI Name field; for details, see *Setting preferences for SQL database connections*.
  - For non-J2EE you specify a connection URL like *jdbc:db2:MyDB*. You specify that string in these ways:
    - By setting the build descriptor option sqlDB; or
    - By placing a value in EGL SQL Database Connections preference page, in the field Connection URL; for details, see *Setting preferences for SQL database connections*.

A later section describes the interaction of build descriptors and EGL preferences.

Debugger mode

The debugger has two modes: Java and COBOL, as determined by the build descriptor option **system**. If no build descriptor is in use or if you set the system type to DEBUG as a debug preference, the mode is Java.

The mode controls how the debugger acts in situations where the EGL run-time behavior differs for Java and COBOL output.

Debugger commands

You use the following commands to interact with the EGL debugger:

Add breakpoint
Identifies a line at which processing pauses. When code execution pauses, you can examine variable values as well as the status of files and screens.

Breakpoints are remembered from one debugging session to the next, unless you remove the breakpoint.

You cannot set a breakpoint at a blank line or at a comment line.

Disable breakpoint
Inactivates a breakpoint but does not remove it.

Enable breakpoint
Activates a breakpoint that was previously disabled.

Remove breakpoint
Clears the breakpoint so that processing no longer automatically pauses at the line.

Remove all breakpoints
Clears every breakpoint.

Run
Runs the code until the next breakpoint or until the run unit ends. (In any case the debugger stops at the first statement in the main function.)

Run to line
Runs all statements up to (but not including) the statement on a specified line.

Step into
Runs the next EGL statement and pauses.

The following list indicates what happens if you issue the command **step into** for a particular statement type:

call
Stops at the first statement of a called program if the called program runs in the EGL debugger. Stops at the next statement in the current program if the called program runs outside of the EGL debugger.

The EGL debugger searches for the receiving program in every project in the workbench.

converse
Waits for user input. That input causes processing to stop at the next running statement, which may be in a validator function.

forward
If the code forwards to a PageHandler, the debugger waits for user input and stops at the next running statement, which may be in a validator function.

If the code forwards to a program, the debugger stops at the first statement in that program.

function invocation
Stops at the first statement in the function.

JavaLib.invoke and related functions
Stops at the next Java statement, so you can debug the Java code that is made available by the Java access functions.

show, transfer
Stops at the first statement of the program that receives control. The target program is EGL source that runs in the EGL debugger and is not EGL-generated code.

After either a **show** statement or a **transfer** statement of the form *transfer to a transaction*, the behavior of the EGL debugger depends on the debugger mode:

- In Java mode, the EGL debugger switches to the build descriptor for the new program or (if no such build descriptor is in use) prompts the user for a new build descriptor. The new program can have a different set of properties from the program that ran previously.
- In COBOL mode, the build descriptor for the previous program remains in use, and the new program cannot have a different set of properties.

The EGL debugger searches for the receiving program in every project in the workbench.

Step over
Runs the next EGL statement and pauses, but does not stop within functions that are invoked from the current function.

The following list indicates what happens if you issue the command **step over** for a particular statement type:

converse
Waits for user input, then skips any validation function (unless a breakpoint is in effect). Stops at the statement that follows the **converse** statement.

forward
If the code forwards to a PageHandler, the debugger waits for user input and stops at the next running statement, but not in a validator function, unless a breakpoint is in effect.

If the code forwards to a program, the debugger stops at the first statement in that program.

show, transfer
Stops at the first statement of the program that receives control. The target program is EGL source that runs in the EGL debugger and is not EGL-generated code.

After either a **show** statement or a **transfer** statement of the form *transfer to a transaction*, the behavior of the EGL debugger depends on the debugger mode:

- In Java mode, the EGL debugger switches to the build descriptor for the new program or (if no such build descriptor is in use) prompts the user for a new build descriptor. The new program can have a different set of properties from the program that ran previously.
- In COBOL mode, the build descriptor for the previous program remains in use, and the new program cannot have a different set of properties.

The EGL debugger searches for the receiving program in every project in the workbench.

Step return
Runs the statements needed to return to an invoking program or function; then, pauses at the statement that receives control in that program or function.

An exception is in effect if you issue the command **step return** in a validator function. In that case, the behavior is identical to that of a **step into** command, which primarily means that the EGL debugger runs the next statement and pauses.

The EGL debugger treats the following EGL statements as if they were null operators:

- **sysLib.audit**
- **sysLib.purge**
- **sysLib.startTransaction**

You can add a breakpoint at these statements, for example, but a **step into** command merely continues to the subsequent statement, with no other effect.

Finally, if you issue the command **step into** or **step over** for a statement that is the last one running in the function (and if that statement is not **return**, **exit program**, or **exit stack**), processing pauses in the function itself so that you can review variables that are local to the function. To continue the debug session in this case, issue another command.

Use of build descriptors

A build descriptor helps to determine aspects of the debugging environment. The EGL debugger selects the build descriptor in accordance with the following rules:

- If you specified a debug build descriptor for your program or PageHandler, the EGL debugger uses that build descriptor. For details on how to establish the debug build descriptor, see *Setting the default build descriptors*.
- If you did not specify a debug build descriptor, the EGL debugger prompts you to select from a list of your build descriptors or to accept the value **None**. If you accept the value **None**, the EGL debugger constructs a build descriptor for use during the debugging session; and a preference determines whether VisualAge Generator compatibility is in effect.
- If you specified either **None** or a build descriptor that lacks some of the required database-connection information, the EGL debugger gets the connection information by reviewing your preferences. For details on how to set those preferences, see *Setting preferences for SQL database connections*.

If you are debugging a program that is intended for use in a text or batch application in a Java environment, and if that program issues a **transfer** statement that switches control to a program that is also intended for use in a different run unit in a Java environment, the EGL debugger uses a build descriptor that is assigned to the receiving program. The choice of build descriptor is based on the rules described earlier.

If you are debugging a program that is called by another program, the EGL debugger uses the build descriptor that is assigned to the called program. The choice of build descriptor is based on the rules described above, except that if you do not specify a build descriptor, the debugger does not prompt you for a build descriptor when the called program is invoked; instead, the build descriptor for the calling program remains in use.

Note: You must use a different build descriptor for the caller and the called program if one of those programs (but not both) takes advantage of

VisualAge Generator compatibility. The generation-time status of VisualAge compatibility is determined by the value of build descriptor option **VAGCompatibility**.

A build descriptor or resource association part that you use for debugging code may be different from the one that you use for generating code. For example, if you intend to access a VSAM file from a program that is written for a COBOL environment, you are likely to reference a resource association part in the build descriptor. The resource association part must refer to the run-time target system and must refer to a file type that is appropriate for the target system. The difference between the two situations is as follows:

- At generation time, the resource association part indicates the file's system name that is used in the target environment
- At debug time, the system name must reflect another naming convention, as appropriate when you access a remote VSAM file from an EGL-generated Java program on Windows 2000/NT/XP; for details on that naming convention, see *VSAM support*

SQL-database access

To determine the user ID and password to use for accessing an SQL database, the EGL debugger considers the following sources in order until the information is found or every source is considered:

1. The build descriptor used at debug time; specifically, the build descriptor options sqlID and sqlPassword.
2. The SQL preferences page, as described in *Setting preferences for SQL database connections*; at that page, you also specify other connection information.
3. An interactive dialog that is displayed at connection time. Such a dialog is displayed only if you select the checkbox **Prompt for SQL user ID and password when needed**.

call statement

As noted earlier, the EGL debugger responds to a **transfer** or **show** statement by interpreting EGL source code. The EGL debugger responds to a **call** statement, however, by reviewing the linkage options part specified in the build descriptor, if any. If the referenced linkage options part includes a **callLink** element for the call, the result is as follows:

- If the **callLink** property **remoteComType** is set to DEBUG, the EGL debugger interprets EGL source code. The debugger finds the source by referencing the **callLink** properties **package** and **location**.
- If the **callLink** property **remoteComType** is not set to DEBUG, the debugger invokes EGL-generated code and uses the information in the linkage options part as if the debugger were running an EGL-generated Java program, even if the debugger is running in COBOL mode.

In the absence of linkage information, the EGL debugger responds to a **call** statement by interpreting EGL source code. Linkage information is unavailable in these cases:

- No build descriptor is used; or
- A build descriptor is used, but no linkage options part is specified in that build descriptor; or
- A linkage options part is specified in the build descriptor, but the referenced part does not have a **callLink** element that references the called program.

If the debugger runs EGL source code, you can run statements in that program by issuing the **step into** command from the caller. If the debugger calls generated code, however, the debugger runs the entire program; the **step into** command works like the **step over** command.

System type used at debug time

A value for system type is available in sysVar.systemType. Also, a second value is available in VGLib.getVAGSysType if you requested development-time compatibility with VisualAge Generator).

The value in **sysLib.systemType** is the same as the value of the build descriptor option **system**, except that the value is DEBUG in either of two cases:

- You select the preference **Set systemType to DEBUG**, as mentioned in *Setting preferences for the EGL debugger*; or
- You specified NONE as the build descriptor to use during the debugging session, regardless of the value of that preference.

The system function **VGLib.getVAGSysType** returns the VisualAge Generator equivalent of the value in **sysLib.systemType**; for details, see the table in *VGLib.getVAGSysType*.

EGL debugger port

The EGL debugger uses a port to establish communication with the Eclipse workbench. The default port number is 8345. If another application is using that port or if that port is blocked by a firewall, set a different value as described in *Setting preferences for the EGL debugger*.

If a value other than 8345 is specified as the EGL debugger port and if an EGL program will be debugged on the J2EE server, you must edit the server configuration:

1. Go to the Environment tab, System Properties section
2. Click Add
3. For Name, type com.ibm.debug.egl.port
4. For Value, type the port number

Invoking the EGL debugger from generated code

You can invoke the EGL debugger from an EGL-generated Java program or wrapper so you can use the EGL debugger when you work on a partly deployed application. The program needs a call statement that you associate with a linkage options part, callLink element. Similarly, you must associate the wrapper with a callLink element. In either case, the element must specify property removeComType as DEBUG.

Different rules apply, depending on whether or not the program to be debugged runs in J2EE:

- When the called program does not run in J2EE, its caller may be running anywhere, including a remote system.

  Before the call occurs, you must start a listener program that runs in Eclipse. A listener is started using an EGL Listener launch configuration that has only one configurable setting, a port number. The default port number is 8346.

  To specify a different port number, do as follows:

  1. At the Run menu, click Debug

2. When the Debug dialog is displayed, select EGL Listener
3. Click New

You must specify a port if multiple EGL Listeners run at the same time, because each EGL Listener requires its own port. You also must specify a port if another application is using port 8346 or if a firewall prevents use of port 8346.

The listener port is not the same as the EGL debugger port, which is specified as an EGL preference.

- When the program to be debugged is to be run in J2EE, it must run in the same J2EE server as its caller. The EGL Debugger jars must have been added to the server, and the server must be running in debug mode.

Recommendations

As you prepare to work with the EGL debugger, consider these recommendations (most of which assume that **sysVar.systemType** is set to DEBUG when you are debugging the code):

- If you are retrieving a date from a database but expect the runtime code to retrieve that date in a format other than the ISO format, write a function to convert the date, but invoke the function only when the system type is DEBUG. The ISO format is yyyy-mm-dd, which is the only one available to the debugger.
- To specify external Java classes for use when the debugger runs, modify the class path, as described in *Setting preferences for the EGL debugger*. You might need extra classes, for example, to support MQSeries, JDBC drivers, or Java access functions.
- When you are debugging a PageHandler function that was invoked by JSF (rather than by another EGL function), use Run to leave the function rather than Step Over, Step Into, or Step Return. Using any of the three Step commands takes you to the generated Java code of the PageHandler, which is not useful when you are debugging EGL. If you use one of the Step commands, use Run to leave the generated Java code and display the Web page in a browser.
- If you are using the SQL option WITH HOLD (or the EGL equivalent), you need to know that the option WITH HOLD is unavailable for EGL-generated Java or in the EGL debugger. You may be able to work around the limitation, in part by placing commit statements inside a conditional statement that is invoked only at run time, as in the following example:

```
if (systemType not debug)
  sysLib.commit();
end
```

If EGL programs are debugged on the J2EE server or by an EGL Listener, the server or EGL Listener must be configured to indicate the number for the EGL debugger port:

- To configure a J2EE server, edit the server configuration--
  1. Go to the Environment tab, System Properties section
  2. Click **Add**
  3. For **Name**, type *com.ibm.debug.egl.port*
  4. For **Value**, type the new port number
- To configure an EGL Listener, edit the EGL Listener launch configuration--
  1. Go to the Arguments tab
  2. In the **VM Arguments** field, type this:

     `-Dcom.ibm.debug.egl.port=`*`portNumber`*

portNumber
The new port number

Related concepts
"Compatibility with VisualAge Generator" on page 428
Character encoding options for the EGL debugger
"VSAM support" on page 246

Related tasks
"Setting preferences for SQL database connections" on page 111
"Setting preferences for the EGL debugger" on page 108
"Setting the default build descriptors" on page 109

Related reference
"remoteComType in callLink element" on page 408
"sqlDB" on page 384
"sqlID" on page 385
"sqlJNDIName" on page 387
"sqlPassword" on page 387
"getVAGSysType()" on page 892
"systemType" on page 911

Debugging applications other than J2EE

Starting a non-J2EE application in the EGL debugger

To start debugging an EGL text program or non-J2EE basic program in an EGL debugging session, a launch configuration is required. A launch configuration defines a program's file location and specifies how the program should be launched. You can let the EGL application create the launch configuration (implicit creation), or you can create one yourself (see *Creating a launch configuration in the EGL debugger*).

To launch a program using an implicitly created launch configuration, do as follows:

1. In the Project Explorer view, right-click the EGL source file you want to launch. Alternatively, if the EGL source file is open in the EGL editor, you can right-click on the program in the Outline view.
2. A context menu displays.
3. Click **Debug EGL Program**. A launch configuration is created, and the program is launched in the EGL debugger.

To view the implicitly created launch configuration, do as follows:

1. Click the arrow next to the Debug button on the toolbar. A context menu displays.
2. Click **Debug**. The Debug dialog displays. The name of the launch configuration is displayed in the Name field. Implicitly created launch configurations are named according to the project and source file names.

Note: You can also display the Debug dialog by clicking **Debug** from the Run menu.

Related concepts
"EGL debugger" on page 261

Related tasks
"Creating a launch configuration in the EGL debugger"
"Stepping through an application in the EGL debugger" on page 273
"Using breakpoints in the EGL debugger" on page 272
"Viewing variables in the EGL debugger" on page 273

Creating a launch configuration in the EGL debugger

To start debugging an EGL text program or non-J2EE basic program in an EGL debugging session, a launch configuration is required. A launch configuration defines how a program should be launched. You can create a launch configuration (explicit creation), or you can let the EGL application create one for you (see *Starting a non-J2EE program in the EGL debugger*).

To start a program using an explicitly created launch configuration, do as follows:

1. Click the arrow next to the Debug button on the toolbar, then click **Debug**, or select **Debug** from the Run menu.
2. The Debug dialog is displayed.
3. Click **EGL Program** in the Configurations list, then click **New**.
4. If you did not have an EGL source file highlighted in the Project Explorer view, the launch configuration is named *New_configuration*. If you had an EGL source file highlighted in the Project Explorer view, the launch configuration has the same name as the EGL source file. If you want to change the name of the launch configuration, type the new name in the Name field.
5. If the name in the Project field of the Load tab is not correct, click **Browse**. A list of projects displays. Click a project, then click **OK**.
6. If the name in the EGL program source file field is not correct or the field is empty, click **Search**. A list of EGL source files displays. Click a source file, then click **OK**.
7. If you made changes to any of the fields on the Debug dialog, click **Apply** to save the launch configuration settings.
8. Click **Debug** to launch the program in the EGL debugger.

Note: If you have not yet used **Apply** to save the launch configuration settings, clicking **Revert** will remove all changes that you have made.

Related concepts
"EGL debugger" on page 261

Related tasks
"Starting a non-J2EE application in the EGL debugger" on page 268
"Stepping through an application in the EGL debugger" on page 273
"Using breakpoints in the EGL debugger" on page 272
"Viewing variables in the EGL debugger" on page 273

Creating an EGL Listener launch configuration

To debug a non-J2EE EGL application called from an EGL-generated Java application or wrapper, an EGL Listener launch configuration is required. To create an EGL Listener launch configuration, do as follows:

1. Click the arrow next to the Debug button on the toolbar, then click **Debug**, or select **Debug** from the Run menu.
2. The Debug dialog is displayed.
3. Click **EGL Listener** in the Configurations list, then click **New**.

4. The Listener launch configuration is named *New_configuration*. If you want to change the name of the launch configuration, type the new name in the Name field.
5. If you do not enter a port number, the port defaults to 8346; otherwise, enter a port number. Each EGL Listener requires its own port.
6. Click **Apply** to save the Listener launch configuration.
7. Click **Debug** to launch the EGL Listener.

Related concepts
"EGL debugger" on page 261

Related tasks
"Creating a launch configuration in the EGL debugger" on page 269
"Starting a non-J2EE application in the EGL debugger" on page 268
"Stepping through an application in the EGL debugger" on page 273
"Using breakpoints in the EGL debugger" on page 272
"Viewing variables in the EGL debugger" on page 273

Debugging J2EE applications

Preparing a server for EGL Web debugging

To debug EGL Web programs that run in the WebSphere Application Server, you must prepare the server for debugging. The preparation step must be done once per server and does not need to be done again, even if the Workbench is shut down.

To prepare a server for debugging, do as follows:

1. If you are working with WebSphere v5.1 Test Environment, make sure that the server is stopped. If you are working with WebSphere Application Server v6.0, make sure that the server is running. The explanation for this difference is that the v6.0 code is a functioning server.
2. In the Server view, right-click on the server. A context menu displays.
3. Select **Enable/Disable EGL Debugging**. A message indicates that you have enabled EGL debugging.
4. If you want to debug the generated Java instead of EGL, right-click on the server again and select **Enable/Disable EGL Debugging**. A message indicates that you have disabled EGL debugging.

Related concepts
"EGL debugger" on page 261
"WebSphere Application Server and EGL" on page 321"Web support" on page 173

Related tasks
"Starting an EGL Web debugging session" on page 271
"Starting a server for EGL Web debugging"
"Stepping through an application in the EGL debugger" on page 273
"Using breakpoints in the EGL debugger" on page 272
"Viewing variables in the EGL debugger" on page 273

Starting a server for EGL Web debugging

If you are working with an EGL-based Web application that accesses a JNDI data source, you cannot follow the instructions in the current topic unless you

previously configured a Web application server. For background information that is specific to WebSphere, see *WebSphere Application Server and EGL*.

Also, if you wish to debug an EGL Web program, you must prepare the server for that purpose as described in *Preparing a server for EGL Web debugging*.

To start the server for debugging, do as follows:

1. In the Server view, right-click on the server
2. Select **Debug > Debug on Server**

Related concepts
"EGL debugger" on page 261
"WebSphere Application Server and EGL" on page 321"Web support" on page 173

Related tasks
"Preparing a server for EGL Web debugging" on page 270
"Starting an EGL Web debugging session"
"Stepping through an application in the EGL debugger" on page 273
"Using breakpoints in the EGL debugger" on page 272
"Viewing variables in the EGL debugger" on page 273

Starting an EGL Web debugging session

If you are working with an EGL-based Web application that accesses a JNDI data source, you cannot follow the instructions in the current topic unless you previously configured a Web application server. For background information that is specific to WebSphere, see *WebSphere Application Server and EGL*.

Also, if you wish to debug an EGL Web program, you must prepare the server for that purpose as described in *Preparing a server for EGL Web debugging*. You will save time on the current procedure if you already started the server for debugging, as described in *Starting a server for EGL Web debugging*.

To start an EGL Web debugging session, do as follows:

1. In the Project Explorer, expand the **WebContent** and **WEB-INF** folders. Right-click the JSP file you want to run, then select **Debug > Debug on Server**. The Server Selection dialog is displayed.
2. If you have already configured a server for this Web project, select **Choose an existing server**, then select a server from the list. Click **Finish** to start the server (if necessary), to deploy the application to the server, and to start the application.
3. If you have not configured a server for this Web project, you can proceed as follows, but only if your application does not access a JNDI data source--
   a. Select **Manually define a server**.
   b. Specify the host name, which (for the local machine) is **localhost**.
   c. Select a server type that is similar to the Web application server on which you intend to deploy your application at run time. Choices include **WebSphere v5.1 Test Environment** and **WebSphere v6.0 Server**.
   d. If you do not intend to change your choices as you work on the current project, select the check box for **Set server as project default**.
   e. In most cases, you can avoid this step; but if you wish to specify settings that are different from the defaults, click **Next** and make your selections.
   f. Click **Finish** to start the server, to deploy the application to the server, and to start the application.

Related concepts
"EGL debugger" on page 261
"WebSphere Application Server and EGL" on page 321"Web support" on page 173

Related tasks
"Preparing a server for EGL Web debugging" on page 270
"Starting a server for EGL Web debugging" on page 270
"Stepping through an application in the EGL debugger" on page 273
"Using breakpoints in the EGL debugger"
"Viewing variables in the EGL debugger" on page 273

Using breakpoints in the EGL debugger

Breakpoints are used to pause execution of a program. You can manage breakpoints inside or outside of an EGL debugging session. Keep the following in mind when working with breakpoints:

- A blue marker in the left margin of the Source view indicates that a breakpoint is set and enabled.
- A white marker in the left margin of the Source view indicates that a breakpoint is set but disabled.
- The absence of a marker in the left margin indicates that a breakpoint is not set.

Add or remove a breakpoint

Add or remove a single breakpoint in an EGL source file by doing one of the following:

- Position the cursor at the breakpoint line in the left margin of the Source view and double-click.
- Position the cursor at the breakpoint line in the left margin of the Source view and right-click. A context menu displays. Click the appropriate menu item.

Disable or enable a breakpoint

Disable or enable a single breakpoint in an EGL source file by doing the following:

1. In the Breakpoint view, right-click on the breakpoint. A context menu displays.
2. Click the appropriate menu item.

Remove all breakpoints

Remove all breakpoints from an EGL source file by doing the following:

1. Right-click on any of the breakpoints displayed in the Breakpoints view. A context menu displays.
2. Click **Remove All**.

Related concepts
"EGL debugger" on page 261

Related tasks
"Creating a launch configuration in the EGL debugger" on page 269
"Starting a non-J2EE application in the EGL debugger" on page 268
"Stepping through an application in the EGL debugger" on page 273
"Viewing variables in the EGL debugger" on page 273

Stepping through an application in the EGL debugger

As explained in *EGL debugger*, the EGL debugger provides the following commands to control execution of a program during a debugging session:

Resume
Runs the code until the next breakpoint or until the end of the program.

Run to Line
Allows you to select an executable line in the Source view and run the code to that line.

Step Into
Runs the next EGL statement and pauses. The program stops at the first statement of a called function.

Step Over
Runs the next EGL statement and pauses, but does not stop within functions that are invoked from the current function.

Step Return
Returns to an invoking program or function.

With the exception of Run to Line, each of the commands can be accessed in the following ways:

- Click the appropriate button on the toolbar of the Debug view; or
- Click the appropriate menu item on the Run menu; or
- Right-click a highlighted thread in the Debug view, then click the appropriate menu item.

To use Run to Line, do as follows when the program is paused:

1. Position the cursor in the left margin of the Source view at an executable line, then right-click. A context menu displays.
2. Click **Run to Line**.

When using Run to Line, keep in mind the following:

- The operation is not available from the Debug view or the Run menu
- Run to Line stops at enabled breakpoints

Related concepts
"EGL debugger" on page 261

Related tasks
"Creating a launch configuration in the EGL debugger" on page 269
"Starting a non-J2EE application in the EGL debugger" on page 268
"Using breakpoints in the EGL debugger" on page 272
"Viewing variables in the EGL debugger"

Viewing variables in the EGL debugger

Whenever a program is paused, you can view the current values of the program's variables.

To view a program's variables, do as follows:

1. In the Variables view, expand the parts in the navigator to see their variables.

2. To display the variables' types, click the **Show Type Names** button on the toolbar.
3. To display the details of a variable in a separate pane, click on the variable, then click the **Show Detail** button on the toolbar.

Related concepts
"EGL debugger" on page 261

Related tasks
"Creating a launch configuration in the EGL debugger" on page 269
"Starting a non-J2EE application in the EGL debugger" on page 268
"Stepping through an application in the EGL debugger" on page 273
"Using breakpoints in the EGL debugger" on page 272

Working with EGL build parts

Creating a build file

To create a build file, do as follows:

1. Identify a project or folder to contain the file. You must create a project or folder if you do not already have one. The project should be an EGL or EGL Web project.
2. In the workbench, click **File > New > EGL Build File**.
3. Select the project or folder that will contain the EGL build file. In the File name field, type the name of the EGL build file, for example MyEGLbuildParts. The extension .eglbld is required for the file name. An extension is automatically appended to the end of the file name if no extension or an invalid extension is specified.
4. Click **Finish** to create the build file with no EGL build part declaration. The build file appears in the Project Explorer view and automatically opens in the EGL build parts editor.
5. To add an EGL build part before creating the build file, click **Next**. Select the type of build part to add, then click **Next**. Type a name and a description for the build part, then click **Finish**. The build file appears in the Project Explorer view and automatically opens in the EGL build parts editor.

Related concepts
"EGL projects, packages, and files" on page 13
"Introduction to EGL" on page 1

Related tasks

"Adding a build descriptor part to an EGL build file" on page 279
"Adding a linkage options part to an EGL build file" on page 294

"Adding an import statement to an EGL build file" on page 299
"Adding a resource associations part to an EGL build file" on page 289
"Creating an EGL Web project" on page 117

Setting up general build options

Build descriptor part

A build descriptor part controls the generation process. The part contains several kinds of information:

- *Build descriptor options* specify how to generate and prepare EGL output, and a subset of the build descriptor options can cause other build parts to be included in the generation process. For details on specific options, see *Build descriptor options*.
- *Java run-time properties* assign values to the following properties:
  - vgj.datemask.gregorian.long.*locale*, which contains the date mask used in either of two cases:
    - The Java code generated for the system variable VGVar.currentFormattedGregorianDate is invoked; or

- EGL validates a page item or text-form field that has a length of 10 or more, if the item property **dateFormat** is set to *systemGregorianDateFormat*.

The meaning of *locale* is described at the end of this section.

– vgj.datemask.gregorian.short.*locale*, which contains the date mask used when EGL validates a page item or text-form field that has a length of less than 10, if the item property **dateFormat** is set to *systemGregorianDateFormat*.

  The meaning of *locale* is described at the end of this section.

– vgj.datemask.julian.long.*locale*, which contains the date mask used in either of two cases:

  - The Java code generated for the system variable VGVar.currentFormattedJulianDate is invoked; or
  - EGL validates a page item or text-form field that has a length of 8 or more, if the item property **dateFormat** is set to *systemJulianDateFormat*.

  The meaning of *locale* is described at the end of this section.

– vgj.datemask.julian.short.*locale*, which contains the date mask used when EGL validates a page item or text-form field that has a length of less than 10, if the item property **dateFormat** is set to *systemJulianDateFormat*.

  The meaning of *locale* is described at the end of this section.

– vgj.jdbc.database.*SN*, which identifies a database that is made available to your Java code.

  You must customize the name of the property itself when you specify a substitution value for *SN*, at deployment time. The substitution value in turn must match either the server name that is included in the invocation of VGLib.connectionService or the database name that is included in the invocation of sysLib.connect.

You also must customize the name of the date-mask properties:

– In a given run unit, each property that is initially in effect has a name whose last qualifier (the *locale*) matches the value in the program property **vgj.nls.code**

– In a Web application, a different set of properties is in effect if a program sets the system variable sysLib.setLocale

The Java run-time properties have no effect when you generate COBOL.

Master build descriptors: Your system administrator may require that you use a *master build descriptor* to specify information that cannot be overridden and that is in effect for every generation that occurs in your installation of EGL. By a mechanism described in Master build descriptor, the system administrator identifies that part by name, along with the EGL build file that contains the part.

If the information in the master build descriptor is not sufficient for a particular generation process or if no master build descriptor is identified, you can specify a build descriptor at generation time, along with the EGL build file that contains the generation-specific part. The generation-specific build descriptor (like the master build descriptor) must be at the top level of an EGL build file.

You can create a chain of build descriptors from the generation-specific build descriptor, so that the first in the chain is processed before the second, and the second before the third. When you define a given build descriptor, you begin a chain (or continue one) by assigning a value to the build descriptor option **nextBuildDescriptor**. Your system administrator can use the same technique to create a chain from the master build descriptor. The implication of chaining information is described later.

Any build part referenced by a build descriptor must be visible to the referencing build descriptor, in accordance with the rules described in References to parts. The build part can be a linkage options part or a resource associations part, for example, or the next build descriptor.

Precedence of options: For a given build descriptor option (or Java run-time property), the value that is initially processed at generation time stays in effect, and the overall order of precedence is as follows:

1. The master build descriptor
2. The generation-specific build descriptor, followed by the chain that extends from it
3. The chain that extends from the master build descriptor

The benefit of this scheme is convenience:

- The system administrator can specify unchanging values by setting up a master build descriptor.
- You can use a generation-specific build descriptor to assign values that are specific to a generation.
- A project manager can specify a set of defaults by customizing one or more build descriptors. In most situations of this kind, the generation-specific build descriptor points to the first build descriptor in a chain that was developed by the project manager.

  Default options can be useful when your organization develops a set of programs that must be generated or prepared similarly.
- The system administrator can create a set of general defaults by establishing a chain that extends from the master build descriptor, although use of this feature is unusual.

If a given build descriptor is used more than once, only the first access of that build descriptor has an effect. Also, only the first specification of a particular option has an effect.

Example: Let's assume that the master build descriptor contains these (unrealistic) option-and-value pairs:

```
OptionX               02
OptionY               05
```

In this example, the generation-specific build descriptor (called myGen) contains these option-and-value pairs.

```
OptionA               20
OptionB               30
OptionC               40
OptionX               50
```

As identified in myGen, the next build descriptor is myNext01, which contains these:

```
OptionA               120
OptionD               150
```

As identified in myNext01, the next build descriptor is myNext02, which contains these:

```
OptionB               220
OptionD               260
OptionE               270
```

As identified in the master build descriptor, the next build descriptor is myNext99, which contains this:

```
OptionZ               99
```

EGL accepts option values in the following order:

1. Values for options in the master build descriptor:

   ```
   OptionX              02
   OptionY              05
   ```

   Those options override all others.
2. Values in the generation-specific build descriptor myGen:

   ```
   OptionA             20
   OptionB             30
   OptionC             40
   ```

   The value for optionX in myGen was ignored.
3. Values for other options in myNext01and myNext02:

   ```
   OptionD             150
   OptionE             270
   ```

   The value for optionA in myNext01 was ignored, as was the value for optionD in myNext02.
4. Values for other options in myNext99:

   ```
   OptionZ              99
   ```

Related concepts
"Build" on page 303
"Java runtime properties" on page 327
"References to parts" on page 20
"Master build descriptor"
"Parts" on page 17

Related tasks
"Adding a build descriptor part to an EGL build file" on page 279
"Adding a resource associations part to an EGL build file" on page 289
"Editing general options in a build descriptor" on page 280
"Editing Java run-time properties in a build descriptor" on page 284
"Editing a resource associations part in an EGL build file" on page 290

Related reference
"Build descriptor options" on page 359
"Java runtime properties (details)" on page 525
"Symbolic parameters" on page 392
"connect()" on page 867
"connectionService()" on page 888
"setLocale()" on page 880
"currentFormattedGregorianDate" on page 916
"currentFormattedJulianDate" on page 917

Master build descriptor

An installation can provide its own set of default values for build options and control whether those default values can be overridden.

To set up the master build descriptor, create two build descriptor parts in the same build file, the first referencing the second by use of the build descriptor option

nextBuildDescriptor. The options in the first part specify default values for options that may not be overridden. The options in the second part specify default values for options that can be overridden.

To install the master build descriptor in the workbench, add a plugin xml file like following to the workbench plugins directory:

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
 id="egl.master.build.descriptor.plugin"
 name="EGL Master Build Descriptor Plug-in"
 version="5.0"
 vendor-name="IBM">
 <requires />
 <runtime />
 <extension point =
 "com.ibm.etools.egl.generation.base.framework.masterBuildDescriptor">
 <masterBuildDescriptor
 file = "filePath.buildFileName"
 name = "masterBuildPartName" />
 </extension>
 </plugin>
```

The file path (*filePath*) is in relation to the workspace directory.

If you are using the EGL SDK, you declare the name and file path name of the master build descriptor in a file named eglmaster.properties. This file must be in a directory that is listed in the CLASSPATH environment variable. The format of the properties file is as follows:

```
masterBuildDescriptorName=masterBuildPartName
masterBuildDescriptorFile=fullyQualifiedPathforEGLBuildFile
```

Related concepts
"Build" on page 303
"Build descriptor part" on page 275
"Build plan" on page 305
"EGL projects, packages, and files" on page 13

Related tasks
"Adding a build descriptor part to an EGL build file"

Related reference
"Build descriptor options" on page 359
"Format of eglmaster.properties file" on page 478
"Format of master build descriptor plugin.xml file" on page 493

Adding a build descriptor part to an EGL build file

A build descriptor part controls the generation process. It contains option names and their related values, and those option-and-value pairs specify how to generate and prepare EGL output. Some options specify other control parts, such as a resource association part, that are in the generation process. You can add a build descriptor part to an EGL build file. See *Build descriptor part* for more information. To add a build descriptor part, do as follows:

1. To open an EGL build file with the EGL build parts editor, do as follows in the Project Explorer:
   a. Right-click on the EGL build file
   b. Select **Open With** > **EGL Build Parts Editor**.

2. If the Outline view is not displayed, open that view by selecting **Show View** > **Outline** from the Window menu.
3. In the Outline view, right-click on the build file, then click **Add Part**.
4. Click the **Build Descriptor** radio button, then click **Next**.
5. Choose a name for your build descriptor that adheres to EGL part name conventions. In the Name field, type the name of your build descriptor.
6. In the Description field, type a description of your build part.
7. Click **Finish**. The build descriptor is declared in the EGL build file and the build descriptor general options are displayed in the EGL build parts editor.
8. You can optionally create a chain of build descriptors, so that the first in the chain is processed before the second, and the second before the third. If you want to begin or continue a chain of build descriptors, specify the next build descriptor in the **nextBuildDescriptor** option field of the Options list. To populate the **nextBuildDescriptor** option field, do as follows:
   a. Using the scroll bar on the Options list, scroll down until the **nextBuildDescriptor** option is in view.
   b. If the nextBuildDescriptor row is not highlighted, click once to select the row.
   c. Click the Value field once to put the field into edit mode.
   d. You can type the name of the next build descriptor in the Value field or select an existing build descriptor from the drop-down list.

Related concepts
"Build descriptor part" on page 275

Related tasks
"Editing general options in a build descriptor"
"Editing Java run-time properties in a build descriptor" on page 284
"Removing a build descriptor part from an EGL build file" on page 285

Related reference
"EGL build-file format" on page 358
"Naming conventions" on page 652

Editing general options in a build descriptor

A build descriptor part controls the generation process. To edit the general build descriptor options and symbolic parameters, do as follows:

1. To open an EGL build file with the EGL build parts editor, do as follows in the Project Explorer:
   a. Right-click on the EGL build file
   b. Select **Open With** > **EGL Build Parts Editor**
2. If the Outline view is not displayed, open that view by selecting **Show View** > **Outline** from the Window menu.
3. In the Outline view, right-click on a build descriptor and select **Open**. There are two buttons in the upper right corner of the editor view. Make sure that the Show General Build Descriptor Options button (the first of the two buttons) is pressed. The EGL build parts editor displays the general build descriptor options for the current part definition.
4. You can optionally create a chain of build descriptors, so that the first in the chain is processed before the second, and the second before the third. If you want to begin or continue a chain of build descriptors, specify the next build descriptor in the **nextBuildDescriptor** field. If the nextBuildDescriptor row is

not highlighted, click once to select the row, then click the Value field once to put the field into edit mode. You can type the name of the next build descriptor in the Value field or select an existing build descriptor from the drop-down list.

5. To specify the generation and preparation of EGL output, select a grouping of option-and-value pairs from the **Build option filter** drop-down list. If the option you want to define is not highlighted, click once to select the row, then click the Value field once to put the field into edit mode. You can type the option value, or if a drop-down list is available, select an existing value. If you want to limit your view of option-and-value pairs to the ones you have defined, click the Show only options specified check box.

Related concepts
"Build descriptor part" on page 275

Related tasks
"Adding a build descriptor part to an EGL build file" on page 279
"Editing Java run-time properties in a build descriptor" on page 284
"Removing a build descriptor part from an EGL build file" on page 285

Related reference
"EGL build-file format" on page 358
"Symbolic parameters" on page 392

Choosing options for Java generation

Build descriptor options are set in build descriptor parts. To choose build descriptor options for Java generation, start the EGL editor and edit the build descriptor part.

When you begin editing a build descriptor part from the GUI, the EGL editor contains a pane listing all EGL build descriptor options. To limit the display to options that are applicable to a program generated in Java, select a category from the Build option filter drop-down menu.

Select each option you want, and set its value. The value can be literal, symbolic, or a combination of literal and symbolic You can define symbolic parameters in the EGL part editor; for details, see *Editing general build descriptor options*.

Two build descriptor options—**genDirectory** and **destDirectory**—let you use a symbolic parameter for the value or a portion of the value. For example, for the value of **genDirectory** you can specify `C:\genout\%EZEENV%`. Then if you generate for a Windows environment, the actual generation directory is C:\genout\WIN.

You do not need to specify all the options listed. If you do not specify a value for a build descriptor option, the default for the option is used when the option is applicable in the generation context.

If you have specified a master build descriptor, the option values in that build descriptor override the values in all other build descriptors. When you generate, the master and generation build descriptors can chain to other build descriptors as described in *Build descriptor part*.

Related concepts
"Build descriptor part" on page 275

Related tasks
"Editing general options in a build descriptor" on page 280
"Generating Java wrappers"

Related reference
"Build descriptor options" on page 359

Generating Java wrappers

You can generate Java wrapper classes when you generate the related program. For details on how to set up the build descriptor, see *Java wrapper*.

Related concepts
"Generation" on page 301
"Generation of Java code into a project" on page 301"Java wrapper"

Related tasks
"Building EGL output" on page 305
"Processing Java code that is generated into a directory" on page 315

Related reference
"Build descriptor options" on page 359
"Java wrapper classes" on page 535
"Output of Java wrapper generation" on page 656

Java wrapper: A Java wrapper is a set of classes that act as an interface between the following executables:

- A servlet or a hand-written Java program, on the one hand
- A generated program or EJB session bean, on the other

You generate the Java wrapper classes if you use a build descriptor that has these characteristics:

- The build descriptor option **enableJavaWrapperGen** is set to **yes** or **only**; and
- The build descriptor option **linkage** references a linkage options part that includes a **callLink** element to guide the call from wrapper to program; and
- One of two statements apply:
  - The call from wrapper to program is by way of an EJB session bean (in which case the **callLink** element, **linkType** property is set to **ejbCall**); or
  - The call from wrapper to program is remote (in which case the **callLink** element, **type** property is set to **remoteCall**); also, the **callLink** element, **javaWrapper** property is set to **yes**.

If an EJB session bean mediates between the Java wrapper classes and an EGL-generated program, you generate the EJB session if you use a build descriptor that has these characteristics:

- The build descriptor option **enableJavaWrapperGen** is set to **yes** or **only**; and
- The build descriptor option **linkage** references a linkage options part that includes a **callLink** element to guide the call from wrapper to EJB session bean (in which case the **type** property of the **callLink** element is set to **ejbCall**).

For further details on using the classes, see *Java wrapper classes*. For details on the class names, see *Generated output (reference)*.

Related concepts
"COBOL program" on page 306

"Generated output" on page 515
"Java program, PageHandler, and library" on page 306
"Run-time configurations" on page 9

Related tasks
"Generating Java wrappers" on page 282

Related reference
"Generated output (reference)" on page 516
"Java wrapper classes" on page 535
"Output of Java wrapper generation" on page 656

Choosing options for COBOL generation

Build descriptor options are set in build descriptor parts. To choose build descriptor options for COBOL generation, start the EGL editor and edit the build descriptor part.

When you begin editing a build descriptor part from the GUI, the EGL editor contains a pane listing all EGL build descriptor options. To limit the display to options that are applicable to a particular kind of generated output, select a category from the Build option filter drop-down menu. The categories with the word *Basic* contain build descriptor options that are used most frequently, while the categories with the word *All* contain every option possible for the specified output. The word *iseriesc* refers to EGL-generated COBOL programs that run on iSeries.

Select each option you want and set its value. The value can be literal, symbolic, or a combination of literal and symbolic. You can define symbolic parameters in the EGL part editor, as described in *Editing general build descriptor options*.

Two build descriptor options—**genDirectory** and **destDirectory**—let you use a symbolic parameter for the value or a portion of the value. For example, for the value of **genDirectory** you can specify `C:\genout\%EZEENV%`. Then if you generate for a Windows environment, the actual generation directory is C:\genout\WIN.

You do not need to specify all the options listed. If you do not specify a value for a build descriptor option, the default for the option is used when the option is applicable in the generation context.

If you have specified a master build descriptor, the option values in that build descriptor override the values in all other build descriptors. When you generate, the master and generation build descriptors can chain to other build descriptors as described in *Build descriptor part*.

Related concepts
"Build descriptor part" on page 275

Related tasks
"Choosing options for Java generation" on page 281
"Editing general options in a build descriptor" on page 280

Related reference
"Build descriptor options" on page 359

Editing Java run-time properties in a build descriptor

When you are editing a build descriptor part, you can assign values to the following Java run-time properties, which are detailed in *Java run-time properties (details)*:

- vgj.jdbc.database.*SN*
- vgj.datemask.gregorian.long.*locale*
- vgj.datemask.gregorian.short.*locale*
- vgj.datemask.julian.long.*locale*
- vgj.datemask.julian.short.*locale*

To edit the properties, do as follows:

1. To open an EGL build file with the EGL build parts editor, do as follows in the Project Explorer:
   a. Right-click on the EGL build file
   b. Select **Open With** > **EGL Build Parts Editor**
2. If the Outline view is not displayed, open that view by selecting **Show View** > **Outline** from the Window menu.
3. In the Outline view, right-click on a build descriptor and select **Open**. The EGL part editor displays the general build descriptor options for the current part definition.
4. Click the **Show Java Run-time Properties** button on the editor toolbar.
5. To add the Java run-time property **vgj.jdbc.database.*SN***, do this:
   a. In the screen area that is titled "Database mappings for connect", click the**Add** button
   b. Type a "Server name" that you use when coding the system word VGLib.connectionService; this value is substituted for *SN* in the name of the generated property
   c. If the row in the Database mappings for connect list is not highlighted, click once to select the row, then click the JNDI name or URL field once to put the field into edit mode. Type a value whose meaning is different for J2EE connections as compared with non-J2EE connections:
      - In relation to J2EE connections (as is needed in a production environment), the value is the name to which the datasource is bound in the JNDI registry; for example, jdbc/MyDB
      - In relation to a standard JDBC connection (as may be used for debugging), the value is the connection URL; for example, jdbc:db2:MyDB
6. To assign the date masks used when you code either VGVar.currentFormattedGregorianDate (for a Gregorian date) or VGVar.currentFormattedJulianDate (for a Julian date); or EGL validates a page item or a text-form field that has a length of 10 or more and a **dateFormat** property of *systemGregorianDateFormat* or *systemJulianDateFormat*, do this:
   a. In the screen area that is titled "Date Masks", click the **Add** button
   b. In the Locale column, select one of the codes in the listbox; the selected value is substituted for ***locale*** in the date-mask properties listed earlier. Only one of your entries is used at run time: the entry for which the value of ***locale*** matches the value of the Java run-time property **vgj.nls.code**
   c. If the row in the Date Masks list is not highlighted, click once to select the row, then click the Long Gregorian Mask field once to put the field into edit mode. Either select a mask from the listbox or type a mask; characters other than D, Y, or digits can be used as separators, and the default value is specific to the locale

d. If the row in the Date Masks list is not highlighted, click once to select the row, then click the Long Julian Mask field once to put the field into edit mode. Either select a mask from the listbox or type a mask; characters other than D, Y, or digits can be used as separators, and the default value is specific to the locale

7. To assign the date masks used when EGL validates a page item or a text-form field that has a length less than 10 and a **dateFormat** property of *systemGregorianDateFormat* or *systemJulianDateFormat*, do this:
   a. In the screen area that is titled "Date Masks", click the **Add** button
   b. In the Locale column, select one of the codes in the listbox; the selected value is substituted for ***locale*** in the date-mask properties listed earlier. Only one of your entries is used at run time: the entry for which the value of ***locale*** matches the value of the Java run-time property **vgj.nls.code**
   c. If the row in the Date Masks list is not highlighted, click once to select the row, then click the Short Gregorian Mask field once to put the field into edit mode. Either select a mask from the listbox or type a mask; characters other than D, Y, or digits can be used as separators, and the default value is specific to the locale
   d. If the row in the Date Masks list is not highlighted, click once to select the row, then click the Short Julian Mask field once to put the field into edit mode. Either select a mask from the listbox or type a mask; characters other than D, Y, or digits can be used as separators, and the default value is specific to the locale
8. To remove an assignment, click on it, then click the **Remove** button.

Related concepts
"Build descriptor part" on page 275
"Java runtime properties" on page 327

Related tasks
"Adding a build descriptor part to an EGL build file" on page 279
"Editing general options in a build descriptor" on page 280
"Removing a build descriptor part from an EGL build file"

Related reference
"EGL build-file format" on page 358
"Java runtime properties (details)" on page 525
"connectionService()" on page 888
"currentFormattedGregorianDate" on page 916
"currentFormattedJulianDate" on page 917

Removing a build descriptor part from an EGL build file

To remove a build descriptor part from an EGL build file, do as follows:

1. To open an EGL build file with the EGL build parts editor, do as follows in the Project Explorer:
   a. Right-click on the EGL build file
   b. Select **Open With** > **EGL Build Parts Editor**
2. If the Outline view is not displayed, open that view by selecting **Show View** > **Outline** from the Window menu
3. In the Outline view, right-click on the build descriptor part, then click **Remove**

Related concepts
"Build descriptor part" on page 275

Related tasks
"Adding a build descriptor part to an EGL build file" on page 279
"Editing general options in a build descriptor" on page 280
"Editing Java run-time properties in a build descriptor" on page 284

Setting up external file, printer, and queue associations

Resource associations and file types

An EGL fixed record that accesses an external file, printer, or queue has a logical file or queue name. (In the case of a printer, the logical file name is *printer* for most run-time systems.) The name can be no more than 8 characters and is meaningful only as a way of relating the record to a *system name*, which the target system uses to access a physical file, printer, or queue.

In relation to files or queues, the file or queue name is a default for the system name. In relation to printers, a default exists only for COBOL output.

Instead of accepting a default, you can take one or both of these actions:

- At generation time, you control the generation process with a build descriptor that in turn references a specific resource associations part. The resource associations part relates the file name with a system name on the target platform where you intend to deploy the generated code.
- At run time (in most cases) you can change the value in the record-specific variable resourceAssociation (for files or queues) or in the system variable ConverseVar.printerAssociation (for print output). Your purpose is to override the system name that you specified either by default or by specifying a resource associations part.

The resource associations part does not apply to these record types:
- basicRecord, because basic records do not interact with data stores
- SQLRecord, because SQL records interact with relational databases

Resource associations part: The resource associations part is a set of association elements, each of which has these characteristics:

- Is specific to a logical file or queue name
- Has a set of entries, each specific to a target system; each entry identifies the file type on the target platform, along with the system name and in some cases additional information

You can think of an association element as a set of properties and values in a hierarchical relationship, as in the following example:

```
// an association element
property:  fileName
value:     myFile01

  // an entry, with multiple properties
  property:  system
  value:     aix
  property:  fileType
  value:     spool
  property:  systemName
  value:     employee

  // a second entry
  property:  system
  value:     win
```

```
property:  fileType
value:     seqws
property:  systemName
value:     c:\myProduct\myFile.txt
```

In this example, the file name myFile01 is related to these files:

- *employee* on AIX
- *myFile.txt* on Windows 2000/NT/XP

The file name must be a valid name, an asterisk, or the beginning of a valid name followed by an asterisk. The asterisk is the wild-card equivalent of one or more characters and provides a way to identify a set of names. An association element that includes the following value for a file name, for example, pertains to any file name that begins with the letters *myFile*:

```
myFile*
```

If multiple elements are valid for a file name used in your program, EGL uses the first element that applies. A series of association elements, for example, might be characterized by the following values for file name, in order:

```
myFile
myFile*
*
```

Consider the element associated with the last value, where the value of myFile is only an asterisk. Such an element could apply to any file; but in relation to a particular file, the last element applies only if the previous elements do not. If your program references myFile01, for instance, the linkage specified in the second element supersedes the third element to define how the reference is handled.

At generation time, EGL selects a particular association element, along with the first entry that is appropriate. An entry is appropriate in either of two cases:

- A match exists between the target system for which you are generating, on the one hand, and the **system** property, on the other; or
- The **system** property has the following value:

  ```
  any
  ```

If you are generating for AIX, for example, EGL uses the first entry that refers to **aix** or to **any**.

File types: A file type determines what properties are necessary for a given entry in an association element. The next table describes the EGL file types.

Record types and VSAM: Each of three types of fixed records is appropriate for accessing a VSAM data set, but only if the file type in the association element for the record is ibmcobol, vsam, or vsamrs:

- If the fixed record is of type indexedRecord, the VSAM data set is a Key Sequenced Data Set with a primary or alternate index
- If the fixed record is of type relativeRecord, the VSAM data set is a Relative Record Data Set
- If the fixed record is of type serialRecord, the VSAM data set is an Entry Sequenced Data Set

For further details: For further details on resource associations, see these topics:

- *Record and file type cross-reference*

- *Association elements*

Related concepts
"Fixed record parts" on page 125
"MQSeries support" on page 247
"Parts" on page 17
"Record types and properties" on page 126
"Record parts" on page 124
"VSAM support" on page 246

Related task
"Adding a resource associations part to an EGL build file" on page 289
"Editing a resource associations part in an EGL build file" on page 290
"Removing a resource associations part from an EGL build file" on page 291

Related reference
"Association elements" on page 352
"Record and file type cross-reference" on page 716
"recordName.resourceAssociation" on page 832
"resourceAssociations" on page 381
"system" on page 389
"printerAssociation" on page 896

Logical unit of work

When you change resources that are categorized as *non-recoverable* (such as serial files on Windows 2000), your work is relatively permanent; neither your code nor EGL run-time services can simply rescind the changes. When you change resources that are categorized as *recoverable* (such as relational databases), your code or EGL run-time services either can commit the changes to make the work permanent or can rollback the changes to return to content that was in effect when changes were last committed.

Recoverable resources are as follows:

- Relational databases
- CICS queues and files that are configured to be recoverable
- MQSeries message queues, unless your MQSeries record specifies otherwise, as described in *MQSeries support*

A *logical unit of work* identifies input operations that are either committed or rolled back as a group. A unit of work begins when your code changes a recoverable resource; and ends when the first of these events occurs:

- Your code invokes the system function **sysLib.commit** or **sysLib.rollback** to commit or roll back the changes
- EGL run-time services performs a rollback in response to a hard error that is not handled in your code; in this case, all the programs in the run unit are removed from memory
- An implicit commit occurs, as happens in the following cases--
  - A program issues a **show** statement.
  - The top-level program in a run unit ends successfully, as described in *Run unit*.
  - A Web page is displayed, as when a PageHandler issues a **forward** statement.
  - A program issues a **converse** statement and any of the following applies:

- You are not in VisualAge Generator compatibility mode, and the program is a segmented program
- **ConverseVar.commitOnConverse** is set to 1
- You are in VisualAge Generator compatibility mode, and **ConverseVar.segmentedMode** is set to 1

Unit of work for Java: In a Java run unit, the details are as follows:

- When any of the Java programs ends with a hard error, the effect is equivalent to performing rollbacks, closing cursors, and releasing locks.
- When the run unit ends successfully, EGL performs a commit, closes cursors, and releases locks.
- You can use multiple connections to read from multiple databases, but you should update only one database in a unit of work because only a one-phase commit is available. For related information, see *VGLib.connectionService*.
- When an EGL-generated program is accessed by way of an EGL-generated EJB session bean, transaction control may be affected by a transaction attribute (also called the container transaction type), which is in the deployment descriptor of the EJB session bean. The transaction attribute affects transaction control only when the linkage options part, callLink element, property **remoteComType** for the call is direct, as described in *remoteComType in callLink element*.

  The EJB session bean is generated with transaction attribute REQUIRED, but you can change the value at deployment time. For details on the implications of the transaction attribute, see your Java documentation.

Related concepts
"MQSeries support" on page 247
"Run unit" on page 721
"SQL support" on page 213

Related tasks
"Setting up a J2EE JDBC connection" on page 341
"Understanding how a standard JDBC connection is made" on page 245

Related reference
"Default database" on page 234
"commit()" on page 866
"connectionService()" on page 888
"rollback()" on page 878
"Java wrapper classes" on page 535
"luwControl in callLink element" on page 403
"remoteComType in callLink element" on page 408
"sqlDB" on page 384

Adding a resource associations part to an EGL build file

A resource associations part relates a file name with a system resource name on the target platform where you intend to deploy the generated code. You can add a resource associations part to an EGL build file. For details, see *Resource associations and file types*. To add a resource associations part, do the following:

1. To open an EGL build file with the EGL build parts editor, do as follows in the Project Explorer:
   a. Right-click on the EGL build file
   b. Select **Open With** > **EGL Build Parts Editor**

2. If the Outline view is not displayed, open that view by selecting **Show View** > **Outline** from the Window menu.
3. In the Outline view, right-click on the build file, then click **Add Part**.
4. Click **Resource Associations**, then click **Next**.
5. Choose a name for your resource associations part that conforms to EGL part name conventions. In the Name field, type the name of your resource associations part.
6. In the Description field, type a description of your part.
7. Click **Finish**. The resource associations part is added to the EGL build file and the resource associations part page is opened in the EGL build parts editor.

Related concepts
"Build descriptor part" on page 275
"Resource associations and file types" on page 286

Related tasks
"Editing a resource associations part in an EGL build file"
"Removing a resource associations part from an EGL build file" on page 291

Related reference
"EGL build-file format" on page 358
"Naming conventions" on page 652

Editing a resource associations part in an EGL build file

A resource associations part relates a file name with a system resource name on the target platform where you intend to deploy the generated code.

To edit a resource associations part, do as follows:

1. To open an EGL build file with the EGL build parts editor, do as follows in the Project Explorer:
   a. Right-click on the EGL build file
   b. Select **Open With** > **EGL Build Parts Editor**
2. If the Outline view is not displayed, open that view by selecting **Show View** > **Outline** from the Window menu.
3. In the Outline view, right-click a resource associations part and click **Open**. The editor displays the current part definition.
4. To add a new Association element to the part, click **Add Association** or press the Insert key, and type the logical file name or select a logical file name.
5. To change the default system name associated with your logical file name, you can either:
   - Select the corresponding row in the Association elements list, then click the name once to put the field into edit mode. Select the new system name from the System drop-down list.
   - In the Properties of selected system entries list, click the system property once to put the Value field associated with that property into edit mode. Select the new system name from the Value drop-down list.
6. To change the default file type associated with your logical file name, you can either:
   - Select the row in the Association elements list that corresponds to your logical file name, then click the name once to put the field into edit mode. Select the new file type from the File Type drop-down list.

- Select the row in the Association elements list that corresponds to your logical file name. In the Properties of selected system entries list, click the fileType property once to put the Value field associated with that property into edit mode. Select the file type from the Value drop-down list.

7. Modify the resource associations as needed.
   - To associate more than one system and set of related properties with a logical file name, select the row in the Association elements list that corresponds to your logical file name. At the bottom of the Association elements list, click **Add System**. The added row is now selected and available for editing.
   - To remove a system and related properties from an associated logical file name, select the row in the Association elements list that corresponds to your logical file name. At the bottom of the Association elements list, click **Remove** or press the Delete key.
   - To remove a logical file name and any associated systems, select the row in the Association elements list that corresponds to your logical file name. At the bottom of the Association elements list, click **Remove** or press the Delete key.

Related concepts
"Build descriptor part" on page 275
"Resource associations and file types" on page 286

Related tasks
"Adding a resource associations part to an EGL build file" on page 289
"Removing a resource associations part from an EGL build file"

Related reference
"EGL build-file format" on page 358

Removing a resource associations part from an EGL build file

To remove a resource associations part from an EGL build file, do as follows:

1. To open an EGL build file with the EGL build parts editor, do as follows in the Project Explorer:
   a. Right-click on the EGL build file
   b. Select **Open With** > **EGL Build Parts Editor**
2. If the Outline view is not displayed, open that view by selecting **Show View** > **Outline** from the Window menu
3. In the Outline view, right-click on the resource associations part, then click **Remove**

Related concepts
"Resource associations and file types" on page 286

Related tasks
"Adding a resource associations part to an EGL build file" on page 289
"Editing a resource associations part in an EGL build file" on page 290

Setting up call and transfer options

Linkage options part

A *linkage options* part specifies details on the following issues:

- How a generated Java program or wrapper calls other generated code
- How a generated COBOL program calls and is called by other generated code

- How a generated Java or COBOL program transfers asynchronously to another generated program
- How a generated COBOL program transfers control and ends processing.

Specifying when linkage options are final: For a generated COBOL program, the linkage options specified at generation time are in effect at run time. For generated Java code, you can choose between two alternatives:

- The linkage options specified at generation time are in effect at run time; or
- The linkage options specified in a linkage properties file at deployment time are in effect at run time. Although you can write that file by hand, EGL generates it in this situation:
  - You set the linkage options property **remoteBind** to RUNTIME; and
  - You generate a Java program or wrapper with the build descriptor option **genProperties** set to GLOBAL or PROGRAM.

  For details on using the file, see Deploying a linkage properties file. For details on customizing the file, see Linkage properties file (reference).

Elements of a linkage options part: The linkage options part is composed of a set of elements, each of which has a set of properties and values. The following types of elements are available:

- A **callLink** element specifies the linkage conventions that EGL uses for a given call.

  If you are generating a COBOL program, the following relationships are in effect:
  - If the callLink element refers to the generated program, that element determines aspects of the program's own parameters; for example, whether the program expects pointers to data or expects the data itself. Also, that element helps determine whether to generate a Java wrapper that allows access to the COBOL code from native Java code; for an overview, see Java wrapper.
  - If the callLink element refers to a program being called by the generated program, that element specifies how the call is implemented; for example, whether the call is local or remote.

  If you are generating Java code, the callLink element always applies to a called program. The following relationships are in effect:
  - If the callLink element refers to the program that you are generating, that element helps determine whether to generate a Java wrapper that allows access to the program from native Java code; for an overview, see *Java wrapper*. If you indicate that the Java wrapper accesses the program by way of an EJB session bean, the callLink element also causes generation of an EJB session bean.
  - If you are generating a Java program and if the callLink element refers to a program that is called by that program, the callLink element specifies how the call is implemented; for example, whether the call is local or remote. If you indicate that the calling Java program makes the call through an EJB session bean, the callLink element causes generation of an EJB session bean.
- An *asynchLink* element specifies how a generated Java or COBOL program transfers asynchronously to another program, as occurs when the transferring program invokes the system function sysLib.startTransaction.

- A *transferToProgram* element specifies how a generated COBOL program transfers control to a program and ends processing. This element is not used for Java output and is meaningful only for a main program that issues a **transfer** statement of the type *transfer to program*.
- A *transferToTransaction* element specifies how a generated program transfers control to a transaction and ends processing. This element is meaningful only for a main program that issues a **transfer** statement of the type *transfer to transaction*. The element is unnecessary, however, when the target program is generated with VisualAge Generator or (in the absence of an alias) with EGL.

Identifying the programs or records to which an element refers: In each element, a property (for example, **pgmName**) identifies the programs or records to which the element refers; and unless otherwise stated, the value of that property can be a valid name, an asterisk, or the beginning of a valid name followed by an asterisk. The asterisk is the wild-card equivalent of one or more characters and provides a way to identify a set of names.

Consider a callLink element that includes the following value for the **pgmName** property:

```
myProg*
```

That element pertains to any EGL program part that begins with the letters *myProg*.

If multiple elements are valid, EGL uses the first element that applies. A series of callLink elements, for example, might be characterized by these **pgmName** values, in order:

```
YourProgram
YourProg*
*
```

Consider the element associated with the last value, where the value of **pgmName** is only an asterisk. Such an element could apply to any program; but in relation to a particular program, the last element applies only if the previous elements do not. If your program calls YourProgram01, for instance, the linkage specified in the second element (YourProg*) supersedes the third element (*) to define how EGL handles the call.

In most cases, elements with more specific names should precede those with more general names. In the previous example, the element with the asterisk is appropriately positioned to provide the default linkage specifications.

Related concepts
"Java wrapper" on page 282
"Parts" on page 17

Related tasks
"Adding a linkage options part to an EGL build file" on page 294
"Deploying a linkage properties file" on page 342
"Editing the asynchLink element of a linkage options part" on page 296
"Editing the callLink element of a linkage options part" on page 294

"Editing the transfer-related elements of a linkage options part" on page 297
"Setting up the J2EE run-time environment for EGL-generated code" on page 333

Related reference
"asynchLink element" on page 355
"call" on page 547
"callLink element" on page 395

"linkage" on page 378
"Linkage properties file (details)" on page 637

"startTransaction()" on page 883
"transfer" on page 627
"transferToProgram element" on page 926
"transferToProgram element" on page 926

Adding a linkage options part to an EGL build file

A linkage options part describes how a generated EGL program implements calls and transfers and how the program accesses files. To add this type of part, do as follows:

1. To open an EGL build file with the EGL build parts editor, do as follows in the Project Explorer:
   a. Right-click on the EGL file
   b. Select **Open With** > **EGL Build Parts Editor**
2. If the Outline view is not displayed, open that view by selecting **Show View** > **Outline** from the Window menu.
3. In the Outline view, right-click on the build file, then click **Add Part**.
4. Click **Linkage Options**, then click **Next**.
5. Choose a name for your linkage options part that adheres to EGL part name conventions. In the Name field, type the name of your linkage options part.
6. In the Description field, type a description of your part.
7. Click **Finish**. The linkage options part is added to the EGL file and the linkage options part page is opened in the EGL build parts editor.

Related concepts
"Linkage options part" on page 291

Related tasks
"Editing the asynchLink element of a linkage options part" on page 296
"Editing the callLink element of a linkage options part"

"Editing the transfer-related elements of a linkage options part" on page 297
"Removing a linkage options part from an EGL build file" on page 298

Related reference
"EGL build-file format" on page 358
"Naming conventions" on page 652

Editing the callLink element of a linkage options part

A linkage options part describes how a generated EGL program implements calls and transfers and how the program accesses files. To edit the part's callLink element, do as follows:

1. To open an EGL build file with the EGL build parts editor, do as follows in the Project Explorer:
   a. Right-click on the EGL build file
   b. Select **Open With** > **EGL Build Parts Editor**

2. If the Outline view is not displayed, open that view by selecting **Show View** > **Outline** from the Window menu.
3. In the Outline view, right-click a linkage options part and click **Open**. The EGL part editor displays the current part declaration.
4. Click the Show CallLink Elements button on the editor toolbar.
5. To add a new CallLink element, click **Add** or press the Insert key, and type the Program Name (pgmName) or select a program name from the Program Name drop-down list.
6. To change the default call type associated with your program name, you can either:
   - Select the corresponding row in the CallLink elements list, then click the Type field (localCall, remoteCall, ejbCall) once to put the field into edit mode. Select the new call type from the Type drop-down list.
   - In the Properties of selected callLink elements list, click the type property once to put the Value field associated with that property into edit mode. Select the new call type from the Value drop-down list.
7. Other properties associated with your program name are listed in the Properties of selected callLink elements list based on the call type. To change the value of one of these properties, select the program name. In the Properties of selected callLink elements list, click the property you want to define once to put the Value field associated with that property into edit mode. Define the new value by selecting an option in the Value drop-down list, or by typing the new value in the Value field. For some properties, you can only select an option in a drop-down list. For other properties, you can only type a value in the Value field.
8. Modify the CallLink elements list as needed:
   - To reposition a callLink element, select an element and click either **Move Up** or **Move Down**.
   - To remove a callLink element, select the element and click **Remove** or press the Delete key.

Related concepts
"Linkage options part" on page 291

Related tasks
"Adding a linkage options part to an EGL build file" on page 294
"Editing the asynchLink element of a linkage options part" on page 296

"Editing the transfer-related elements of a linkage options part" on page 297
"Removing a linkage options part from an EGL build file" on page 298

Related reference
"asynchLink element" on page 355
"callLink element" on page 395
"EGL build-file format" on page 358

"Linkage properties file (details)" on page 637
"transferToProgram element" on page 926

Enterprise JavaBean (EJB) session bean: An EJB session bean comprises the following components:

- Home interface, which gives a client access to the EJB session bean at run time

- Remote bean interface, which lists the methods that are directly available to that client
- Bean implementation, which contains the logic that is indirectly available to that client

An EJB session bean is an intermediary between one program and another or between an EGL Java wrapper and a program. Generation of the EJB session bean largely depends on settings in the linkage options part that is used at generation time. For details, see *Linkage options part*; in particular, the overview of the **callLink** element.

For details on the output file names, see *Generated output (reference)*.

Related concepts
"Generated output" on page 515
"Linkage options part" on page 291

Related reference
"Generated output (reference)" on page 516

Editing the asynchLink element of a linkage options part

A linkage options part describes how a generated EGL program implements calls and transfers and how it accesses files. To edit the part's asynchLink element, do as follows:

1. To open an EGL build file with the EGL build parts editor, do as follows in the Project Explorer:
   a. Right-click on the EGL build file
   b. Select **Open With** > **EGL Build Parts Editor**
2. If the Outline view is not displayed, open that view by selecting **Show View** > **Outline** from the Window menu.
3. In the Outline view, right-click a linkage options part and click **Open**. The EGL build parts editor displays the current part declaration.
4. Click the Show AsynchLink Elements button on the editor toolbar.
5. To add a new AsynchLink element, click **Add** or press the Insert key, and type the Record Name (recordName) or select a record name from the Record Name drop-down list.
6. To change the default linkage type associated with your record name, you can either:
   - Select the corresponding row in the AsynchLink Elements list, then click the Type field (localAsynch, remoteAsynch) once to put the field into edit mode. Select the new linkage type from the Type drop-down list.
   - In the Properties of selected asynchLink elements list, click the type property once to put the Value field associated with that property into edit mode. Select the new linkage type from the Value drop-down list.
7. Other properties associated with your record name are listed in the Properties of selected asynchLink elements list based on the linkage type. To change the value of one of these properties, select the record name. In the Properties of selected aynchLink elements list, click the property you want to define once to put the Value field associated with that property into edit mode. Define the new value by selecting an option in the Value drop-down list, or by typing the new value in the Value field. For some properties, you can only select an option in a drop-down list. For other properties, you can only type a value in the Value field.

8. Modify the asynchLink elements list as needed:
   - To reposition an asynchLink element, select an element and click either **Move Up** or **Move Down**.
   - To remove an asynchLink element, select an element and click **Remove** or press the Delete key.

Related concepts
"Linkage options part" on page 291

Related tasks
"Adding a linkage options part to an EGL build file" on page 294
"Editing the callLink element of a linkage options part" on page 294

"Editing the transfer-related elements of a linkage options part"
"Removing a linkage options part from an EGL build file" on page 298

Related reference
"asynchLink element" on page 355
"EGL build-file format" on page 358
"Linkage properties file (details)" on page 637
"startTransaction()" on page 883

Editing the transfer-related elements of a linkage options part

A linkage options part describes how a generated EGL program implements calls and transfers and how the program accesses files. To edit the part's transfer-related elements, do as follows:

1. To open an EGL build file with the EGL build parts editor, do as follows in the Project Explorer:
   a. Right-click on the EGL build file
   b. Select **Open With** > **EGL Build Parts Editor**
2. If the Outline view is not displayed, open that view by selecting **Show View** > **Outline** from the Window menu.
3. In the Outline view, right-click a linkage options part and click **Open**. The EGL build parts editor displays the current part declaration.
4. Click the Show TransferLink Elements button on the editor toolbar. The Transfer to Program and Transfer to Transaction lists display.
5. To edit the Transfer to Program list, do as follows:
   a. At the bottom of the Transfer to Program list, click **Add** or press the Insert key, and type the From Program (fromPgm) name or select a program name from the From Program name drop-down list.
   b. To edit the To Program (toPgm) name, select the corresponding row in the Transfer to Program list, then click the To Program field once to put the field into edit mode. Type the program name or select a program name from the To Program drop-down list.
   c. If an alias name is needed, select the corresponding row in the Transfer to Program list, then click the Alias field once to put the field into edit mode. Type the alias name.
   d. To change the default linkage type associated with your program name, select the corresponding row in the Transfer to Program list, then click the Link Type (linkType) field once to put the field into edit mode. Select the new linkage type from the Link Type drop-down list.
   e. Modify the Transfer to Program list as needed:

- To reposition a transferToProgram element, select an element and click either **Move Up** or **Move Down**.
- To remove a transferToProgram element, select an element and click **Remove** or press the Delete key.

6. To edit the Transfer to Transaction list, do as follows:
   a. At the bottom of the Transfer to Transaction list, click **Add** or press the Insert key, and type the To Program (toPgm) name or select a program name from the To Program name drop-down list.
   b. If an alias name is needed, select the corresponding row in the Transfer to Transaction list, then click the Alias field once to put the field into edit mode. Type the alias name.
   c. To edit the Externally Defined property associated with your program name, select the corresponding row in the Transfer to Transaction list, then click the Externally Defined field once to put the field into edit mode. Select the externally defined property from the Externally Defined property drop-down list.
   d. Modify the Transfer to Transaction list as needed:
      - To reposition a transferToTransaction element, select an element and click either **Move Up** or **Move Down**.
      - To remove a transferToTransaction element, select an element and click **Remove** or press the Delete key.

Related concepts
"Linkage options part" on page 291

Related tasks
"Adding a linkage options part to an EGL build file" on page 294
"Editing the asynchLink element of a linkage options part" on page 296
"Editing the callLink element of a linkage options part" on page 294

"Removing a linkage options part from an EGL build file"

Related reference
"EGL build-file format" on page 358
"transferToProgram element" on page 926
"transferToProgram element" on page 926

Removing a linkage options part from an EGL build file

To remove a linkage options part from an EGL build file, do as follows:

1. To open an EGL build file with the EGL build parts editor, do as follows in the Project Explorer:
   a. Right-click on the EGL build file
   b. Select **Open With** > **EGL Build Parts Editor**
2. If the Outline view is not displayed, open that view by selecting **Show View** > **Outline** from the Window menu
3. In the Outline view, right-click on the linkage options part, then click **Remove**

Related concepts
"Linkage options part" on page 291

Related tasks
"Adding a linkage options part to an EGL build file" on page 294
"Editing the asynchLink element of a linkage options part" on page 296

"Editing the callLink element of a linkage options part" on page 294

"Editing the transfer-related elements of a linkage options part" on page 297

Setting up references to other EGL build files

Adding an import statement to an EGL build file

Import statements allow EGL build files to reference parts in other build files. See *Import* for more information on the import feature.

To add an import statement to an EGL build file, do as follows:

1. Open an EGL build file with the EGL build parts editor. If you do not have a file open, do this in the Project Explorer:
   a. Right-click on the build file in the Project Explorer
   b. Select **Open With** > **EGL Build Parts Editor**
2. Click the **Imports** tab in the build parts editor.
3. Click the **Add** button.
4. Type or select the name of the file or folder to import, then click **OK**.

Related concepts
"Import" on page 30

Related tasks
"Editing an import statement in an EGL build file"
"Removing an import statement from an EGL build file"

Editing an import statement in an EGL build file

To edit an import statement in an EGL build file, do as follows:

1. Open the EGL build file with the EGL build parts editor. If you do not have a file open, do this in the Project Explorer:
   a. Right-click on the build file in the Project Explorer
   b. Select **Open With** > **EGL Build Parts Editor**
2. Click the **Imports** tab in the build parts editor. The import statements are displayed.
3. Select the import statement you want to change, then click the **Edit** button.
4. Type or select the name of the file or folder to import, then click **OK**.

Related concepts
"Import" on page 30

Related tasks
"Adding an import statement to an EGL build file"
"Removing an import statement from an EGL build file"

Removing an import statement from an EGL build file

To remove an import statement in an EGL build file, do as follows:

1. Open the EGL build file with the EGL build parts editor. If you do not have a file open, do this in the Project Explorer:
   a. Right-click on the build file in the Project Explorer
   b. Select **Open With** > **EGL Build Parts Editor**

2. Click the **Imports** tab in the build parts editor. The import statements are displayed.
3. Select the import statement you want to remove, then click the **Remove** button.

Related concepts
"Import" on page 30

Related tasks
"Adding an import statement to an EGL build file" on page 299
"Editing an import statement in an EGL build file" on page 299

Editing an EGL build path

For overview material, see these topics:

- *References to parts*
- *EGL build path and eglpath*

To include projects in the EGL project path, follow these steps:

1. In the Project Explorer, right-click on a project that you want to link to other projects, then click **Properties**.
2. Select the **EGL Build Path** properties page.
3. A list of all other projects in your workspace is displayed in the **Projects** tab. Click the check box beside each project you want to reference.
4. To put the projects in a different order or to export any of them, click the **Order and Export** tab and do as follows--
   - To change the position of a project in the build-path order, select the project and click the **Up** and **Down** buttons.
   - To export a project, select the related check box. To handle all the projects at once, click the **Select All** or **Deselect All** button.
5. Click **OK**.

Related concepts
"EGL projects, packages, and files" on page 13
"References to parts" on page 20
"Import" on page 30
"Parts" on page 17

Related reference
"EGL build path and eglpath" on page 465
"References to parts" on page 20
"Import" on page 30
"Parts" on page 17

Generating, preparing, and running EGL output

Generation

Generation is the creation of output from EGL parts.

You can generate output in the Workbench, from the Workbench batch interface, or from the EGL Software Development Kit (EGL SDK). The EGL SDK provides a batch interface for file-based generation that is independent of the Workbench.

Generation uses the saved versions of your EGL files.

Related concepts
"Development process" on page 8
"Generated output" on page 515

Related tasks
"Generating for COBOL" on page 309
"Generating Java wrappers" on page 282

Related reference
"Build descriptor options" on page 359
"Generated output (reference)" on page 516

Generation of Java code into a project

If you are generating a Java program or wrapper, it is recommended (and in some cases required) that you set build descriptor option **genProject**, which causes generation into a project.

EGL provides various services for you when you generate into a project. The services vary by project type, as do your next tasks:

Application client project

When you generate into an application client project, EGL does as follows:

- Provides preparation-time access to EGL jar files (fda6.jar and fdaj6.jar) by adding the following entries to the project's Java build path:

  ```
  EGL_GENERATORS_PLUGINDIR/runtime/fda6.jar
  EGL_GENERATORS_PLUGINDIR/runtime/fdaj6.jar
  ```

  For details on the variable at the beginning of each entry, see *Setting the variable EGL_GENERATORS_PLUGINDIR*.
- Provides run-time access to the EGL jar files:
  - Imports the jar files into each enterprise application project that references the application client project
  - Updates the manifest in the application client project so that the jar files in an enterprise application project are available
- Puts run-time values into the deployment descriptor so that you can avoid cutting and pasting entries from a generated J2EE environment file; for an overview of this subject, see *Setting deployment-descriptor values*

Your next tasks are as follows:

1. If you are calling the generated program by way of TCP/IP, provide run-time access to a listener, as described in *Setting up the TCP/IP listener*

2. Provide access to non-EGL jar files
3. Now that you have placed output files in a project, continue setting up the J2EE run-time environment

EJB project

When you generate into an EJB project, EGL does as follows:

- Provides preparation-time access to EGL jar files (fda6.jar and fdaj6.jar) by adding the following entries to the project's Java build path:

  ```
  EGL_GENERATORS_PLUGINDIR/runtime/fda6.jar
  EGL_GENERATORS_PLUGINDIR/runtime/fdaj6.jar
  ```

  For details on the environment variable at the beginning of each entry, see *Setting the variable EGL_GENERATORS_PLUGINDIR*.
- Provides run-time access to the EGL jar files:
  - Imports fda6.jar and fdaj6.jar into each enterprise application project that references the EJB project
  - Updates the manifest in the EJB project so that fda6.jar and fdaj6.jar in an enterprise application project are available at run time
- Assigns the JNDI name automatically so that the EGL run-time code can access the EJB code; but this step occurs only when you generate an EJB session bean.
- In most cases, puts run-time values into the deployment descriptor so that you can avoid cutting and pasting entries from a generated J2EE environment file; for an overview of this subject, see *Setting deployment-descriptor values*.

  EGL does not put run-time values into the deployment descriptor if EGL cannot find the necessary session element in the deployment descriptor. This situation occurs, for example, when the Java program is generated before the wrapper or when the build descriptor option **sessionBeanID** is set to a value that is not found in the deployment descriptor. For details on session elements, see *sessionBeanID*.

Your next tasks are as follows:

1. Provide access to non-EGL jar files
2. Generate deployment code
3. Now that you have placed output files in a project, continue setting up the J2EE run-time environment

J2EE Web project

EGL does as follows:

- Provides access to EGL jar files by importing fda6.jar and fdaj6.jar into the project's Web Content/WEB-INF/lib folder
- Puts run-time values into the deployment descriptor so that you can avoid cutting and pasting entries from a generated J2EE environment file; for an overview of this subject, see *Setting deployment-descriptor values*

Your next tasks are as follows:

1. Providing access to non-EGL jar files
2. Now that you have placed output files in a project, continue as described in *Setting up the J2EE run-time environment for EGL-generated code*

Java project

If you are generating into a non-J2EE Java project for debugging or production purposes, EGL does as follows:

- Provides access to EGL jar files (fda6.jar and fdaj6.jar) by adding the following entries to the project's Java build path:

  ```
  EGL_GENERATORS_PLUGINDIR/runtime/fda6.jar
  EGL_GENERATORS_PLUGINDIR/runtime/fdaj6.jar
  ```

  For details on the variable at the beginning of each entry, see *Setting the variable EGL_GENERATORS_PLUGINDIR*.
- Generates a properties file, but only if the build descriptor includes the following option values:
  - **genProperties** is set to GLOBAL or PROGRAM; and
  - **J2EE** is set to NO.

  If you request a global properties file (**rununit.properties**), EGL places that file in the Java source folder, which is the folder that contains the Java packages. (The Java source folder may be either a folder within the project or the project itself.) If you request a program properties file instead, EGL places that file in the folder that contains the program.

  At run time, values in the program properties file are used to set up a standard JDBC connection. For details, see *Understanding how a standard JDBC connection is made*.

Now that you have placed output files in a project, do as follows:

- If your program accesses a relational database, make sure that your Java build path includes the directory where the driver is installed. For DB2, for example, specify the directory that contains db2java.zip.
- If your code accesses MQSeries, provide access to non-EGL jar files
- Place a linkage properties file in the module

For details on the consequence of generating into a non-existent project, see *genProject*.

Related tasks
"Generating deployment code for EJB projects" on page 319
"Deploying a linkage properties file" on page 342
"Setting deployment-descriptor values" on page 334
"Providing access to non-EGL jar files" on page 343
"Setting the variable EGL_GENERATORS_PLUGINDIR" on page 319
"Setting up the J2EE run-time environment for EGL-generated code" on page 333
"Understanding how a standard JDBC connection is made" on page 245

Related reference
"genDirectory" on page 372
"genProject" on page 374
"sessionBeanID" on page 381

Build

When you are working in an EGL or EGL Web project, the word *build* does not (in general) refer to code generation.

The following menu options have a distinct meaning:

Build project

Builds a subset of the project--

1. Validates all EGL files that have changed in the project since the last build

2. Generates PageHandlers that were changed since the prior PageHandler generation
3. Compiles any Java source that changed since the last compile

The menu option **Build Project** is available only if have not set the Workbench preference **Perform build automatically on resource modification**. If you *have* set that preference, the actions described earlier occur whenever you save an EGL file.

Build all
Conducts the same actions as **Build project**, but for every open project in the workspace.

Rebuild project
Acts as follows--

1. Validates all the EGL files in the project
2. Generates all PageHandlers in the project
3. Compiles any Java source that changed since the last compile

Rebuild all
Conducts the same actions as **Rebuild project**, but for every open project in the workspace.

When you generate code into a project (as is possible only for Java output), a Java compile occurs locally in the following situations:

- When you build or rebuild the project; or
- When you generate the source files; but only if you checked the workbench preference **Perform build automatically on resource modification**.

When you generate code into a directory, EGL optionally creates a *build plan*, which is an XML file that includes the following details:

- The location of any files that will be transferred to another machine;
- Other information needed for the transfer, which occurs by way of TCP/IP; and
- A Java compile statement (if appropriate).

In the case of COBOL generation for iSeries, the iSeries build server calls a build script (FDAPREP) that resides in the iSeries library QEGL/QREXSRC. That build script specifies how to prepare output, and a system administrator must customize that build script, as described in the document *EGL Server Guide for iSeries*, on your installation CD.

Preparation of generated output on a remote platform requires that a build server be running on that platform.

You may wish to create a build plan and to invoke that plan at a later time. For details, see *Invoking a build plan after generation*.

Related concepts
"Build descriptor part" on page 275
"Build plan" on page 305

"Build server" on page 323
"Development process" on page 8

Related tasks
"Creating a build file" on page 275
"Invoking a build plan after generation" on page 315

Related reference
"Build descriptor options" on page 359
"Build scripts delivered with EGL" on page 392

Building EGL output

To build EGL output , complete the following steps:

Steps to build output

Java programs or wrappers	COBOL programs
Generate Java source code into a project or directory: • If you generate into a project (as is recommended) and if your Eclipse preferences are set to build automatically on resource modification, the workbench prepares the output. • If you generate into a directory, the generator's distributed build function prepares the output.	Generate into a directory. This step produces COBOL source code. COBOL preparation occurs on a remote host system.
Prepare the generated output. This step is done automatically unless you set build descriptor option **buildPlan** or **prep** to `no`.	Prepare the generated output. This step compiles and links the source code to produce executable code.

Related concepts
"Build" on page 303
"Development process" on page 8
"EGL projects, packages, and files" on page 13
"Generation of Java code into a project" on page 301
"Generated output" on page 515
"Generation" on page 301

Related tasks
"Creating an EGL source file" on page 120
"Processing Java code that is generated into a directory" on page 315
"Setting up the J2EE run-time environment for EGL-generated code" on page 333

Related reference
"Generated output (reference)" on page 516

Build plan

The build plan is an XML file that makes the following details available at preparation time:

- What files need to be processed on the build machine
- What build scripts are needed to process them
- Where outputs are to be placed

The build plan resides on the development platform and informs the build client of all the build steps. For each step a request is made of the build server.

EGL produces a build plan whenever you generate a COBOL program, Java program, or Java wrapper, unless you set the build descriptor option **buildPlan** to NO.

For details on the name of the build plan, see *Generated output (reference)*.

Related concepts
"Build script" on page 322
"Generated output" on page 515

Related reference
"buildPlan" on page 364
"Generated output (reference)" on page 516

Java program, PageHandler, and library

When you request that a program part be generated as a Java program, or when you request that a pageHandler or Java-related library part be generated, EGL produces a class and a file for each of the following:

- The program, pageHandler, or library part
- Each record declared either in that part itself or in any function that is invoked directly or indirectly by that part
- Each data table, form group, and form that is used

For details on the class names, see *Output of Java program generation*.

Related concepts
"Library (generated output)" on page 629
"PageHandler" on page 180

Related tasks
"Setting up the J2EE run-time environment for EGL-generated code" on page 333

Related reference
"Generated output (reference)" on page 516
"Output of Java program generation" on page 655

COBOL program

A COBOL program can be generated to run on the iSeries platform.

When you request that a program part be generated as a COBOL program, EGL produces a source file. For details on the file name, see *Generated output (reference)*.

Note: If you are generating for COBOL, the following restrictions are in effect--

- When you invoke a function, you cannot pass an argument that includes the invocation of an EGL system function. The workaround is to pass a variable that includes the value of the function invocation.
- When you invoke a function in a conditional expression, you cannot pass an argument that is itself an expression other than a literal or variable. For example, the first of these is valid, but the second is not:

```
// valid
if (myFunction(a) > 1)
  ;
end
```

```
// not valid
if (myFunction(a + b) > 1)
  ;
end
```

The workaround is to pass a variable that includes the value of the expression.

- A function cannot return an expression such as A + B. The workaround is to return a variable that includes the value of the expression.
- Non-fixed records are not supported.
- Set-value blocks are supported only for setting property values in part definitions and variable declarations and for initializing field values in variable declarations.
- The property **validValues** supports only a single range of numbers.
- In relation to text and print forms, the following statements apply:
  - You cannot include fields of type HEX, FLOAT, SMALLFLOAT, TIME, or TIMESTAMP
  - If you are using a field as a date, you specify a date-format string or constant in the field property **dateFormat**, and these restrictions are in effect--
    - If you specify the form field as type DATE, NUM(8) or NUM(10) and intend to present a date in Gregorian format, you must specify the field length as 10 and may use your own 8 or 10-character format (either *"yy/MM/dd"* or *"yyyy/MM/dd"*, with a separator of your choice); or any of the following date formats:
      - usaDateFormat
      - eurDateFormat
      - isoDateFormat
      - jisDateFormat
      - systemGregorianDateFormat
    - If you specify the form field as type NUM(8) and intend to present a date in Julian format, you must specify the field length as 8 and can use either the Julian format *"yy/ddd"* or *"yyyy/ddd"* (with a separator of your choice) or the date format systemJulianDateFormat
    - If you specify the form field as type NUM(6) and intend to present a date in Julian format, you must specify the field length as 6 and can use either the Julian format *"yy/ddd"* (with a separator of your choice) or the date format systemJulianDateFormat
    - If you specify the form field as type CHAR(8), you must specify the field length as 8 and may use either the Gregorian format *"yy/MM/dd"* (with a separator of your choice) or either of the following date formats:
      - systemGregorianDateFormat
      - systemJulianDateFormat
    - If you specify the form field as type CHAR(10), you must specify the field length as 10 and may use your own 8 or 10-character format (either *"yy/MM/dd"* or *"yyyy/MM/dd"*, with a separator of your choice); or any of the following date formats:
      - usaDateFormat
      - eurDateFormat
      - isoDateFormat
      - jisDateFormat

    - systemGregorianDateFormat
    - systemJulianDateFormat
- The following capabilities are not supported:
  - array literals, arrayDictionaries, Dictionaries, EGL libraries, report processing, and consoleUI
  - The system variable **sysVar.currentException**; you cannot identify which exception was thrown most recently in the run unit, although use of the **OnException** block is supported
  - Multidimensional arrays or the array-specific functions resizeAll and setMaxSizes
  - The primitive types ANY, CLOB, LOB, and STRING; but literals and the substring syntax are supported
- The following system functions are not supported:
  - currentArrayCount
  - currentArrayDataLine
  - currentArrayScreenLine
  - fieldInputLength
  - getCmdLineArg
  - getCmdLineArgCount
  - getKey
  - getKeyCode
  - getKeyName
  - getMessage
  - getProperty
  - getBlobLen
  - getSubStrFromClob
  - getStrFromClob
  - isCurrentField
  - isCurrentFieldByName
  - isFieldModified
  - isFieldModifiedByName
  - lastKeyTyped
- The linkage options part, callLink element, type property does not support use of remoteCall
- Tracing of runtime statements is not supported; specifically, these build descriptor options are not supported:
  - sqlErrorTrace
  - sqlIOTrace
  - statementTrace

Related concepts
"Generated output" on page 515
"Java program, PageHandler, and library" on page 306
"Java wrapper" on page 282
"Run-time configurations" on page 9

Related tasks
"Generating for COBOL" on page 309

Related reference
"Generated output (reference)" on page 516
"Output of COBOL generation" on page 655

Generating for COBOL

The steps required to create a COBOL load module are as follows:

1. Create one or more EGL source files and one or more EGL build files
2. Generate COBOL source code.
3. Prepare the COBOL code by compiling and linking it. This happens automatically unless you set build descriptor option **buildPlan** or **prep** to *no*.

The build descriptor option **distLibrary** specifies the iSeries library into which the code is placed.

Related concepts

"COBOL program" on page 306
"EGL projects, packages, and files" on page 13
"Generated output" on page 515
"Generation" on page 301

Related tasks
"Building EGL output" on page 305

Related reference
"Build scripts delivered with EGL" on page 392
"destLibrary" on page 369
"Generated output (reference)" on page 516
"Output of COBOL generation" on page 655

Results file

The results file contains status information on the code-preparation steps that were done on the target environment. You receive the file only if EGL attempts to prepare generated output. If you are preparing a COBOL program, you also receive a file for each step in preparation.

Preparation occurs automatically when you generate into a directory and use the following build descriptor options:

- **prep** is set to YES
- **buildPlan** is set to YES

For details on the name of the results file and on the additional files provided to you after a COBOL program is prepared, see *Generated output (reference)*.

Related concepts
"Build descriptor part" on page 275
"Generated output" on page 515

Related tasks
"Processing Java code that is generated into a directory" on page 315

Related reference
"Generated output (reference)" on page 516
"buildPlan" on page 364
"prep" on page 380

Generating in the workbench

Generating in the workbench is accomplished with either the generation wizard or the generation menu item. When you select the generation menu item, EGL uses the default build descriptor. If you have not selected a default build descriptor, use the generation wizard. For details on selecting the default build descriptor, see *Setting the default build descriptors*.

To generate in the workbench by invoking the generation wizard, do as follows:

1. Right-click on a resource name (project, folder, or file) in the Project Explorer.
2. Select the **Generate With Wizard...** option.

The generation wizard includes four pages:

1. The first page displays a list of parts to generate based on the selection that the user made to start the generation process. You must select at least one part from the list before you can continue to the next page. The interface provides buttons to let you select or deselect all of the parts in the list.
2. The second page lets you choose a build descriptor or build descriptors to be used to generate the parts selected on the first page. You have two options:
   - Choose from a drop-down list of all the build descriptors that are in the workspace, and use that build descriptor to generate all of the parts.
   - Select a build descriptor for each of the parts selected on the first page. You use a table to select the build descriptors for each part. The first column of the table displays the part names; the second column displays a drop-down list of build descriptors for each part.
3. The third page lets you set user IDs and passwords for both the destination machine and the SQL database used in the generation process, if IDs and passwords are necessary. These user IDs and passwords, if any, override the ones listed in the specified build descriptor for each part being generated. You may want to set user IDs and passwords on this page rather than the build descriptor to avoid keeping sensitive information in persistent storage.
4. The fourth page lets you create a command file that you can use for generating an EGL program outside of the workbench. You can reference the command file in the workbench batch interface (by using the command EGLCMD) or in the EGL SDK (by using the command EGLSDK).

   To create a command file, do as follows:

   a. Select the Create a Command File checkbox
   b. Specify the name of the output file, either by typing the fully qualified path or by clicking **Browse** and using the standard Windows procedure for selecting a file
   c. Select or clear the Automatically Insert EGL Path (eglpath) check box to specify whether you want to include the EGL project path in the command file, as the initial value for eglpath; for details, see *EGL command file*
   d. Select a radio button to indicate whether to avoid generating output when creating the command file
5. Click **Finish**.

To generate in the workbench using the generation menu item, do one of the following sets of steps:

1. Select one or more resource names (project, folder, or file) in the Project Explorer. To select multiple resource names, hold down the **Ctrl** key as you click.
2. Right-click, then select the **Generate** menu option.

or

1. Double-click on a resource name (project, folder, or file) in the Project Explorer. The file opens in the EGL editor.
2. Right-click inside the editor pane, then select **Generate**.

Related concepts
"Generation in the workbench"

Related tasks
"Setting the default build descriptors" on page 109

Related reference
"EGLCMD" on page 466
"EGLSDK" on page 476
"Generated output (reference)" on page 516

Generation in the workbench

To generate output in the Workbench, do the following:

- Load parts to generate, along with any parts that are referenced during generation.
- Select parts to generate. If you invoke the generation process for a file, folder, package, or project, EGL creates output for every *primary part* (every program, PageHandler, form group, data table, or library) that is in the container you selected.
- Initiate generation.
- Monitor progress.

To make generation easier, it is recommended that you first select a default build descriptor from these types:

- Debug build descriptor (as appropriate when you are using the EGL debugger)
- Target system build descriptor (as used for generating parts and deploying them in a run-time environment)

For details on how to select a default build descriptor, see *Setting the default build descriptors*.

You can identify each of two build descriptors (debug and target system) in these ways:

- As a property at the file, folder, package, and project level
- As a Workbench preference

A lower-level build descriptor of a particular kind takes precedence over any higher-level build descriptor of the same kind. For example, a target system build descriptor that was assigned to the current package takes precedence over a target

system build descriptor that was assigned to the project. However, a master build descriptor takes precedence over all others, as described in *Build descriptor part*.

The following precedence rules are in effect for every file that contains a primary part:

- A property that is specific to the file takes precedence over all others
- The related folder property takes precedence over the package, project, or Workbench property
- The related package property takes precedence over the project or Workbench property
- The related project property takes precedence over the Workbench property
- The Workbench property is used if no others are specified

For details on initiating generation, see *Generating in the workbench*.

Related concepts
"Build descriptor part" on page 275
"Development process" on page 8
"Generated output" on page 515

Related tasks
"Generating in the workbench" on page 310
"Setting the default build descriptors" on page 109

Related reference
"Generated output (reference)" on page 516

Generating from the workbench batch interface

To generate from the Workbench batch interface, do as follows:

1. Make sure that your Java classpath provides access to these jar files--
   - startup.jar, which is in the following directory:

     installationDir\eclipse

     installationDir
     The product installation directory, such as C:\Program Files\IBM\RSPD\6.0. If you installed and kept a Rational Developer product before installing the product that you are using now, you may need to specify the directory that was used in the earlier install.
   - eglutil.jar, which is in the following directory:

     installationDir\egl\eclipse\plugins\
     com.ibm.etools.egl.utilities_*version*\runtime

     installationDir
     The product installation directory, such as C:\Program Files\IBM\RSPD\6.0. If you installed and kept a Rational Developer product before installing the product that you are using now, you may need to specify the directory that was used in the earlier install.

     version
     The installed version of the plugin; for example, 6.0.0
2. Make sure that a workspace contains the projects and EGL parts that are required for generation.
3. Develop an EGL command file.

4. Invoke the command EGLCMD, possibly in a larger batch job that generates, runs, and tests the code. You specify the workspace of interest when invoking EGLCMD.

Related concepts
"Generation from the workbench batch interface"

Related reference
"EGLCMD" on page 466
"EGL command file" on page 469

Generation from the workbench batch interface

The workbench batch interface is a feature that lets you generate EGL output from a batch environment that can access the workbench. The workbench does not need to be running. The generation of EGL code can access only projects and EGL parts that were previously loaded into a workspace.

To invoke the interface, use the batch command EGLCMD, which references both a workspace and an EGL command file.

Related concepts
"Development process" on page 8
"Generated output" on page 515

Related tasks
"Generating from the workbench batch interface" on page 312

Related reference
"EGLCMD" on page 466

Generating from the EGL Software Development Kit (SDK)

To generate from the EGL SDK, do as follows:

1. Make sure that Java 1.3.1 (or a higher level) is on the machine where you will generate code. An appropriate level of Java code is installed automatically on the machine where you install EGL. The Java levels on the generation and target machines must be compatible.
2. Make sure that eglbatchgen.jar is in your Java classpath. The jar file is in the following directory:

   installationDir\bin

   installationDir
   The product installation directory, such as C:\Program Files\IBM\RSPD\6.0. If you installed and kept a Rational Developer product before installing the product that you are using now, you may need to specify the directory that was used in the earlier install.
3. If you wish to enable COBOL generation for iSeries, make sure that the run-time jar file eglwdsc.jar is in your class path. The jar file is in the following directory:

   installationDir\egl\eclipse\plugins\
   com.ibm.etools.egl.wdsc_*version*\runtime

   installationDir
   The product installation directory, such as C:\Program Files\IBM\RSPD\6.0. If you installed and kept a Rational Developer

product before installing the product that you are using now, you may need to specify the directory that was used in the earlier install.

version
The installed version of the plugin; for example, 6.0.0

4. Make sure that the EGL SDK can access the EGL files that are required for generation
5. Optionally, develop an EGL command file
6. Invoke the command EGLSDK, possibly in a larger batch job that generates, runs, and tests the code

Related concepts
"Generation from the EGL Software Development Kit (SDK)"
"EGL projects, packages, and files" on page 13

Related reference
"EGL build path and eglpath" on page 465
"EGLCMD" on page 466
"EGL command file" on page 469
"EGLSDK" on page 476

Generation from the EGL Software Development Kit (SDK)

The EGL software development kit (SDK) is a feature that lets you generate output in a batch environment, even when you lack access to the following aspects of the Rational Developer product:

- The graphical user interface
- The details on how projects are organized

You can use the EGL SDK to trigger generation from a software configuration management (SCM) tool such as Rational ClearCase®, perhaps as part of a batch job that is run after normal working hours.

To invoke the EGL SDK, you use the command EGLSDK in a batch file or at a command prompt. The command invocation itself can take either of two forms:

- It can specify an EGL file and build descriptor. In this case, if you want to cause multiple generations you write multiple commands.
- Alternatively, the invocation can reference an EGL command file that includes the information necessary to cause one or more generations.

However you organize your work, you can specify a value for *eglpath*, which is a list of directories that are searched when the EGL SDK uses an import statement to resolve a part reference. Also, you must specify the build descriptor option **genDirectory** instead of **genProject**.

The prerequisites and process for using EGLSDK are described in *Generating from the EGL SDK*. For details on the command invocation, see *EGLSDK*.

Related concepts
"Development process" on page 8
"Generated output" on page 515

Related tasks
"Generating from the EGL Software Development Kit (SDK)" on page 313

Related reference
"genDirectory" on page 372
"EGLCMD" on page 466
"EGL build path and eglpath" on page 465
"EGLSDK" on page 476

Invoking a build plan after generation

You may wish to create a build plan and to invoke that plan at a later time. This case might occur, for example, if a network failure prevents you from preparing code on a remote machine at generation time.

To invoke a build plan in this case, complete the following steps:

1. Make sure that eglbatchgen.jar is in your Java classpath, as happens automatically on the machine where you install EGL. The jar file is in the following directory:

   ```
   installationDir\egl\eclipse\plugins\
   com.ibm.etools.egl.batchgeneration_version
   ```

   installationDir
   The product installation directory, such as C:\Program Files\IBM\RSPD\6.0. If you installed and kept a Rational Developer product before installing the product that you are using now, you may need to specify the directory that was used in the earlier install.

   version
   The installed version of the plugin; for example, 6.0.0
2. Similarly, make sure that your PATH variable includes that directory.
3. From a command line, enter the following command:

   ```
   java com.ibm.etools.egl.distributedbuild.BuildPlanLauncher bp
   ```

   bp The fully qualified path of the build plan file. For details on the name of the generated file, see *Generated output (reference)*.

Related concepts
"Build plan" on page 305
"Generation" on page 301

Related tasks
"Building EGL output" on page 305

Related reference
"Build descriptor options" on page 359
"Generated output (reference)" on page 516

Generating Java; miscellaneous topics

Processing Java code that is generated into a directory

This page describes how to process Java code that is generated into a directory. It is recommended, however, that you avoid generating Java code into a directory; for details see *Generation of Java code into a project*.

To generate Java code into a directory, specify the build descriptor option **genDirectory** and avoid specifying the build descriptor option **genProject**.

Your next tasks depend on the project type:

Application client project

For an application client project, do as follows:

1. Provide preparation-time access to EGL jar files by adding the following entries to the project's Java build path:

   ```
   EGL_GENERATORS_PLUGINDIR/runtime/fda6.jar
   EGL_GENERATORS_PLUGINDIR/runtime/fdaj6.jar
   ```

   For details on the variable at the beginning of each entry, see *Setting the variable EGL_GENERATORS_PLUGINDIR*.
2. Provide run-time access to fda6.jar, fdaj6.jar, and (if you are calling the generated program by way of TCP/IP) EGLTcpipListener.jar:
   - Access the jar files from the following directory:

     installationDir\egl\eclipse\plugins\
     com.ibm.etools.egl.generators_*version*\runtime

     installationDir
     The product installation directory, such as C:\Program Files\IBM\RSPD\6.0. If you installed and kept a Rational Developer product before installing the product that you are using now, you may need to specify the directory that was used in the earlier install.

     version
     The installed version of the plugin; for example, 6.0.0

     Copy those files into each enterprise application project that references the application client project.
   - Update the manifest in the application client project so that the jar files (as stored in an enterprise application project) are available.
3. Provide access to non-EGL jar files (an optional task)
4. Import your generated output into the project, in keeping with these rules:
   - The folder *appClientModule* must include the top-level folder of the package that contains your generated output
   - The hierarchy of folder names beneath *appClientModule* must match the name of your Java package

   If you are importing generated output from package *my.trial.package*, for example, you must import that output into a folder that resides in the following location:

   ```
   appClientModule/my/trial/package
   ```
5. If you generated a J2EE environment file, update that file
6. Update the deployment descriptor
7. Now that you have placed output files in a project, continue setting up the J2EE run-time environment

EJB project

For an EJB project, do as follows:

1. Provide preparation-time access to EGL jar files (fda6.jar and fdaj6.jar) by adding the following entries to the project's Java build path:

   ```
   EGL_GENERATORS_PLUGINDIR/runtime/fda6.jar
   EGL_GENERATORS_PLUGINDIR/runtime/fdaj6.jar
   ```

   For details on the variable at the beginning of each entry, see *Setting the variable EGL_GENERATORS_PLUGINDIR*.
2. Provide run-time access to the EGL jar files:
   - Access fda6.jar and fdaj6.jar from the following directory:

installationDir\egl\eclipse\plugins\
com.ibm.etools.egl.generators_*version*\runtime

installationDir
The product installation directory, such as C:\Program Files\IBM\RSPD\6.0. If you installed and kept a Rational Developer product before installing the product that you are using now, you may need to specify the directory that was used in the earlier install.

version
The installed version of the plugin; for example, 6.0.0

Copy those files into each enterprise application project that references the EJB project.

- Update the manifest in the EJB project so that fda6.jar and fdaj6.jar (as stored in an enterprise application project) are available.

3. Provide access to non-EGL jar files (an optional task)
4. Import your generated output into the project, in keeping with these rules:
   - The folder *ejbModule* must include the top-level folder of the package that contains your generated output
   - The hierarchy of folder names beneath *ejbModule* must match the name of your Java package

   If you are importing generated output from package *my.trial.package*, for example, you must import that output into a folder that resides in the following location:

   ```
   ejbModule/my/trial/package
   ```
5. If you generated a J2EE environment file, update that file.
6. Update the deployment descriptor
7. Set the JNDI name
8. Generate deployment code
9. Now that you have placed output files in a project, continue setting up the J2EE run-time environment

J2EE Web project

For a Web project, do as follows:

1. Provide access to EGL jar files by copying fda6.jar and fdaj6.jar into your Web project folder. To do so, import the external jars found in the following directory:

   installationDir\egl\eclipse\plugins\
   com.ibm.etools.egl.generators_*version*\runtime

   installationDir
   The product installation directory, such as C:\Program Files\IBM\RSPD\6.0. If you installed and kept a Rational Developer product before installing the product that you are using now, you may need to specify the directory that was used in the earlier install.

   version
   The installed version of the plugin; for example, 6.0.0

   The destination for the files is the following project folder:

   ```
   WebContent/WEB-INF/lib
   ```
2. Provide access to non-EGL jar files (an option)
3. Import your generated output into the project, in keeping with these rules:
   - The folder *WebContent* must include the top-level folder of the package that contains your generated output

- The hierarchy of folder names beneath *WebContent* must match the name of your Java package

If you are importing generated output from package *my.trial.package*, for example, you must import that output into a folder that resides in the following location:

```
WebContent/my/trial/package
```

4. Update the deployment descriptor
5. Now that you have placed output files in a project, continue setting up the J2EE run-time environment

Java project

If you are generating code for use in a non-J2EE environment, you generate a properties file if you use the following combination of build descriptor options:

- **genProperties** is set to GLOBAL or PROGRAM; and
- **J2EE** is set to NO.

If you request a global properties file (**rununit.properties**), EGL places that file in the top-level directory. If you request a program properties file instead, EGL places that file with the program, either in the folder that corresponds to the last qualifier in the package name or in the top-level directory. (The top-level directory is used if the package name is not specified in the EGL source file.)

At run time, values in the program properties file are used to set up a standard JDBC connection. For details, see *Understanding how a standard JDBC connection is made*.

For a Java project, your tasks are as follows:

1. Provide access to EGL jar files by adding the following entries to the project's Java build path:

   ```
   EGL_GENERATORS_PLUGINDIR/runtime/fda6.jar
   EGL_GENERATORS_PLUGINDIR/runtime/fdaj6.jar
   ```

   For details on the variable at the beginning of each entry, see *Setting the variable EGL_GENERATORS_PLUGINDIR*.
2. If your program accesses a relational database, make sure that your Java build path includes the directory where the driver is installed. For DB2, for example, specify the directory that contains db2java.zip.
3. If your generated code accesses MQSeries, provide access to non-EGL jar files
4. Make sure that the program properties file (if present) is in the top-level project folder and that the global properties file (**rununit.properties**, if present) is either in the folder that corresponds to the last qualifier in the package name or in the top-level project folder. (The top-level folder is used if the package name is not specified in the EGL source file.)
5. Place a linkage properties file in the project (an optional task)

Related concepts

"Generation of Java code into a project" on page 301

Related tasks

"Generating deployment code for EJB projects" on page 319
"Generating for COBOL" on page 309
"Deploying a linkage properties file" on page 342
"Setting deployment-descriptor values" on page 334
"Providing access to non-EGL jar files" on page 343
"Setting the JNDI name for EJB projects" on page 337

"Setting the variable EGL_GENERATORS_PLUGINDIR"
"Setting up the J2EE run-time environment for EGL-generated code" on page 333
"Understanding how a standard JDBC connection is made" on page 245
"Updating the deployment descriptor manually" on page 336
"Updating the J2EE environment file" on page 335

Related reference
"genDirectory" on page 372
"genProject" on page 374

Generating deployment code for EJB projects

After you generate into an EJB project and specify the deployment descriptor properties, you can generate the stubs and skeletons that allow for remote access of the EJB:

1. In the Project Explorer, right-click on the project name; then click **Deploy**
2. Follow the directions specified in the help page on Generating EJB deployment code from the workbench

Related tasks
"Setting up the J2EE run-time environment for EGL-generated code" on page 333

Setting the variable EGL_GENERATORS_PLUGINDIR

The workbench classpath variable EGL_GENERATORS_PLUGINDIR contains the fully qualified path to the EGL plug-in in the Workbench. The variable is used in the Java build path when you generate an EGL program into a project of type Java, application client, or EJB.

If you encounter a classpath error that refers to EGL_GENERATORS_PLUGINDIR, the variable may not be set. The problem occurs, for example, if you check out an EGL-related project from a software configuration management system like Concurrent Versions System (CVS) before you ever work with an EGL part.

You can set the variable by creating an EGL part, by generating EGL code, or by following these steps:

1. Select **Window**, then **Preferences**
2. On the Preferences page, select **Java**, then **Classpath Variables**
3. Select **New...**
4. At the New Variable Entry page, type **EGL_GENERATORS_PLUGINDIR** and specify the following directory:

   ```
   installationDir\egl\eclipse\plugins\
   com.ibm.etools.egl.generators_version
   ```

   installationDir
   The product installation directory, such as C:\Program Files\IBM\RSPD\6.0. If you installed and kept a Rational Developer product before installing the product that you are using now, you may need to specify the directory that was used in the earlier install.

   version
   The installed version of the plugin; for example, 6.0.0

After you set the variable, rebuild the project.

Related concepts
"Generation of Java code into a project" on page 301

Related reference
"genProject" on page 374

Running EGL-generated Java code on the local machine

Starting a basic or text user interface Java application on the local machine

To start an EGL-generated basic (batch) or text user interface (TUI) Java application on your local machine, do the following:

1. Generate Java source code from your EGL source code; for details see *Generating in the workbench*.
2. In the Project Explorer, expand the **JavaSource** folder and select the Java source file for the application you want to run.
3. On the Workbench menu, select **Run** > **Run As** > **Java Application**; or on the Workbench toolbar, click the down arrow next to the **Run** button, then select **Run As** > **Java Application**.

Related concepts
"Generation in the workbench" on page 311

Related tasks
"Generating in the workbench" on page 310

Starting a Web application on the local machine

If you are working with an EGL-based Web application that accesses a JNDI data source, you cannot follow the instructions in the current topic unless you previously configured a Web application server. For background information that is specific to WebSphere, see *WebSphere Application Server and EGL*.

To start a Web application, follow these steps:

1. Generate Java source code from your EGL source code; for details see *Generating in the workbench*.
2. In the Project Explorer, expand the **WebContent** and **WEB-INF** folders. Right-click the JSP you want to run, then select **Run > Run on Server**. The Server Selection dialog is displayed.
3. If you have already configured a server for this Web project, select **Choose an existing server**, then select a server from the list. Click **Finish** to start the server, to deploy the application to the server, and to start the application.
4. If you have not configured a server for this Web project, you can proceed as follows, but only if your application does not access a JNDI data source--
   a. Select **Manually define a server**.
   b. Specify the host name, which (for the local machine) is **localhost**.
   c. Select a server type that is similar to the Web application server on which you intend to deploy your application at run time. Choices include **WebSphere v5.1 Test Environment** and **WebSphere v6.0 Server**.
   d. If you do not intend to change your choices as you work on the current project, select the check box for **Set server as project default**.

e. In most cases, you can avoid this step; but if you wish to specify settings that are different from the defaults, click **Next** and make your selections.

f. Click **Finish** to start the server, to deploy the application to the server, and to start the application.

Related concepts
"Generation in the workbench" on page 311
"WebSphere Application Server and EGL"
"Web support" on page 173

Related tasks
"Generating in the workbench" on page 310

WebSphere Application Server and EGL

When you run or debug an EGL-written, J2EE application in the Workbench, you are likely to use one of these IBM run-time environments:

- WebSphere v5.1 Test Environment, which supports Java servlet version 2.3 (and earlier) and EJB version 2.0 (and earlier)
- WebSphere Application Server v6.0, which supports Java servlet version 2.4 (and earlier) and EJB version 2.1 (and earlier)

If you are debugging or running code that does not use a J2EE data source, the processes for running code in the two environments are similar and require only a few mouse clicks.

If you require access to a J2EE data source, however, the situation is as follows:

- If you are working with the WebSphere v5.1 Test Environment, do the following two steps in any order:
  1. Identify the data source when you define the server configuration.
  2. Make sure that your application refers to the server-configuration entry for that data source.

     This second step involves specifying the JNDI name in the deployment descriptor that is specific to your project. You specify the JNDI name in either of these ways--

     – When you create the project; or

     – When you update the deployment descriptor.

     For details on server configuration, see *Configuring WebSphere Application Server v5.x*.
- If you are working with WebSphere Application Server v6.0, do the following two steps in any order:
  1. Identify the data source to the server, as is possible in either of these ways--

     – When you update the application deployment descriptor (application.xml), as is recommended; or

     – When you configure the server at the Administrative Console.

     For details on updating the application deployment descriptor, see *Setting up a server to test data sources for WebSphere Application Server v6.0*. For details on using the Administrative Console, see *Configuring WebSphere Application Server v6.x*.
  2. Make sure that your application refers to the server-configuration entry for that data source.

This second step involves specifying the JNDI name in the deployment descriptor that is specific to your project. You specify the JNDI name in either of these ways--

- When you create the project; or
- When you update the deployment descriptor.

The benefits of updating the application deployment descriptor rather than working at the Administrative Console are as follows:

- You can deploy the enterprise application to any Web application server that supports J2EE version 1.4, with no additional server configuration necessary for identifying the data source.
- You can update the application deployment descriptor regardless of whether the server is running.
- You gain convenience because your actions are within the development component of your Rational Developer product rather than within the WebSphere Application Server component.

However you update the data-source information, your change is available to the server almost immediately.

Related concepts
"Web support" on page 173

Build script

A build script is a file that is invoked by a build plan and that prepares output from generated files. Examples are as follows:

- A Java compiler or other .exe (binary) file or a .bat (text) file is available to a build server on the development system or is sent to a build server on a remote Windows 2000/NT/XP.
- A script (.scr file) or some binary code is sent to a USS build server.

You specify the address of a build machine by setting the build descriptor option **destHost**.

COBOL build script for iSeries

The build script for iSeries is a REXX program named FDAPREP and is described in the *EGL Server Guide for iSeries*, which is available in the help system.

Java build script

To prepare Java code for execution, EGL puts the javac (Java compiler) command and its parameters in the build plan and sends to the build machine the javac command and the input required by the command.

Related concepts
"Build" on page 303
"Build plan" on page 305
"Build server" on page 323

Related reference
"Build descriptor options" on page 359
"Build scripts delivered with EGL" on page 392
"destDirectory" on page 368
"destHost" on page 368
"destPassword" on page 369

"destUserID" on page 370
"Output of COBOL generation" on page 655
"Output of Java program generation" on page 655
"Output of Java wrapper generation" on page 656

Build server

A build server receives requests from a client system to create executable files from source code sent from that client. A build server must be started prior to sending any requests from a build client. A build server typically services requests from multiple clients. Multiple threads may be started if concurrent build requests are received.

In a generator environment you start a build server on a machine whose operating system is the target generation system, for example, Windows 2000. The generator produces Java source code. Java code is sent to a specified build server where the Java compiler is invoked.

If you are generating Java code for Windows, you can build the Java outputs on the same machine as the machine where generation was performed. This is called a local build. In this case you do not have to start a build server. If you want to perform a local build, omit the **destHost** option from the build descriptor.

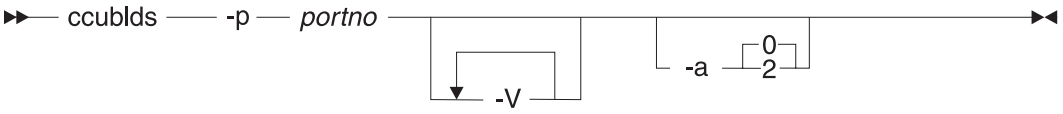
Related concepts
"Build" on page 303
"Build script" on page 322

Related tasks
"Starting a build server on AIX, Linux, or Windows 2000/NT/XP"

Related reference
"Build descriptor options" on page 359

Starting a build server on AIX, Linux, or Windows 2000/NT/XP

To start a remote build server on AIX, Linux, or Windows 2000/NT/XP, enter the ccublds command in a Command Prompt window. The syntax is as follows:



where

-p Specifies the port number (*portno*) that the server listens to, to communicate with the clients.

-V Specifies the verbosity level of the server. You may specify this parameter up to three times (maximum verbosity).

-a Specifies the authentication mode:
- **0** The server performs builds requested by any client. This mode is recommended only in an environment where security is not a concern.
- **2** The server requires the client to provide a valid user ID and password before accepting a build. The user ID and password are first configured by the owner of the host machine where the build server runs. You do the configuration by using the Security Manager described below.

Setting the language of messages returned from the build server

The build server on Windows returns messages in any of the languages listed in the next table, and the default is English.

Language	Code
Brazilian Portugese	ptb
Chinese, simplified	chs
Chinese, traditional	cht
English, USA	enu
French	fra
German	deu
Italian	ita
Japanese	jpn
Korean	kor
Spanish	esp

To specify a language other than English, make sure that before you start the build server, the environment variable CCU_CATALOG is set to a non-English message catalog. The needed value is in the following format (on a single line):

```
installationDir\egl\eclipse\plugins
\com.ibm.etools.egl.distributedbuild\executables
\ccu.cat.xxx
```

installationDir
: The product installation directory, such as C:\Program Files\IBM\RSPD\6.0. If you installed and kept a Rational Developer product before installing the product that you are using now, you may need to specify the directory that was used in the earlier install.

xxx
: The language code being supported by the build server; one of the codes listed in the previous table

Security Manager

The Security Manager is a server program that the build server uses to authenticate clients that send build requests.

Setting the environment for the Security Manager: The Security Manager uses the following Windows environment variables:

CCUSEC_PORT
: Sets the number of the port to which the Security Manager listens. The default value is 22825.

CCUSEC_CONFIG
: Sets the path name of the file in which configuration data is saved. The default is C:\temp\ccuconfig.bin. If this file is not found, the Security Manager creates it.

CCU_TRACE
: Initiates tracing of the Security Manager for diagnostics purposes, if this variable is set to *.

Starting the Security Manager: To start the Security Manager, issue the following command:

```
java com.ibm.etools.egl.distributedbuild.security.CcuSecManager
```

Configuring the Security Manager: To configure the Security Manager, use the Configuration Tool, which has a graphical interface. You can run the tool by issuing the following command:

```
java com.ibm.etools.egl.distributedbuild.security.CCUconfig
```

When Configuration Tool is running, select the **Server Items** tab. Using the button 'Add...', To add the user that you want the build server to support, click the **Add ...** button. You must define a password for the user ID. You can define the following restrictions and privileges for the user:

- The locations, that is, the values of the -la parameter to ccubldc command, that this user can specify. Different locations are separated by semicolons.
- The name of the build script that this user can specify. (The EGL build plan only uses the javac command as a build script.)
- Whether or not this user can send build scripts from client, that is, use the -ft parameter of ccubldc command. (The EGL generator does not use the -ft parameter. You would specify this parameter if they were using the build for purposes other than preparing Java-generation outputs.)

These definitions are kept in persistent storage, in the file specified by CCUSEC_CONFIG, and are remembered across sessions.

Related concepts
"Build script" on page 322
"Build server" on page 323

Related tasks

"Syntax diagram for EGL statements and commands" on page 733

Starting a build server on iSeries

Related concepts

Related tasks

Related reference

Deploying EGL-generated Java output

Java runtime properties

An EGL-generated Java program uses a set of runtime properties that provide information such as how to access the databases and files that are used by the program.

In a J2EE environment

In relation to a generated Java program that will run in a J2EE environment, these situations are possible:

- EGL can generate the runtime properties directly into a J2EE deployment descriptor. In this case, EGL overwrites properties that already exist and appends properties that do not exist. The program accesses the J2EE deployment descriptor at run time.
- Alternatively, EGL can generate the runtime properties into a J2EE environment file. You can customize the properties in that file, then copy them into the J2EE deployment descriptor.
- You can avoid generating the runtime properties at all, in which case you must write any needed properties by hand.

In a J2EE module, every program has the same runtime properties because all code in the module shares the same deployment descriptor.

In WebSphere Application Server, properties are specified as env-entry tags in the web.xml file that is associated with the Web project, as in these examples:

```
<env-entry>
  <env-entry-name>vgj.nls.code</env-entry-name>
  <env-entry-value>ENU</env-entry-value>
  <env-entry-type>java.lang.String</env-entry-type>
</env-entry>

<env-entry>
  <env-entry-name>vgj.nls.number.decimal</env-entry-name>
  <env-entry-value>.</env-entry-value>
  <env-entry-type>java.lang.String</env-entry-type>
</env-entry>
```

In a non-J2EE Java environment

In relation to a generated Java program that runs outside of a J2EE environment, you can generate the runtime properties into a program properties file or code that file by hand. (The program properties file provides the kind of information that is available in the deployment descriptor, but the format of the properties is different.)

In a non-J2EE Java environment, properties can be specified in any of several properties files, which are searched in this order:

- **user.properties**
- A file named as follows--

  programName.properties

  programName
  The first program in the run unit

- **rununit.properties**

Use of **user.properties** is appropriate when you specify properties that are specific to a user. EGL does not generate content for this file.

Use of **rununit.properties** is especially appropriate when the first program of a run unit does not access a file or database but calls programs that do:

- When generating the caller, you can generate a properties file named for the program, and the content might include no database- or file-related properties
- When you generate the called program, you can generate **rununit.properties**, and the content would be available for both programs

None of those files is mandatory, and simple programs do not need any.

At deployment time, these rules apply:

- The user properties file (**user.properties**, if present) is in the user home directory, as determined by the Java system property *user.home*.
- The location of a program properties file (if present) depends on whether the program is in a package. The rules are best illustrated by example:
  - If program P is in package x.y.z and is deployed to MyProject/JavaSource, the program properties file must be in MyProject/JavaSource/x/y/z
  - If program P is not in a package and is deployed to myProject/JavaSource, the program properties file (like the global properties file) must be in MyProject/JavaSource

  In either case, MyProject/JavaSource must be in the classpath.
- The global properties file (**rununit.properties**, if present) must be with the program, in a directory that is specified in the classpath.

If you generate output to a Java project, EGL places the properties files (other than **user.properties**) in the appropriate folders.

If you are generating Java code for use in the same run-unit as Java code generated with an earlier version of EGL or VisualAge Generator, the rules for deploying properties file depends on whether the first program in the run unit was generated with EGL 6.0 or later (in which case the rules described here apply) or whether the first program was generated with an earlier version of EGL or VisualAge Generator (in which case the properties files can be in any directory in the classpath, and the global file is called **vgj.properties**).

Finally, if the first program was generated with the earlier software, you can specify an alternate properties file, which is used throughout the run unit in place of any non-global program properties files. For details, see the description of property **vgj.properties.file** in *Java runtime properties (details)*.

Build descriptors and program properties

Choices are submitted to EGL as build-descriptor-option values:

- To generate properties into a J2EE deployment descriptor, set **J2EE** to YES; set **genProperties** to PROGRAM or GLOBAL; and generate into a J2EE project.
- To generate properties into a J2EE environment file, set **J2EE** to YES; set **genProperties** to PROGRAM or GLOBAL; and do either of these:
  - Generate into a directory (in which case you use the build descriptor option **genDirectory** rather than **genProject**); or

  - Generate into a non-J2EE project.
- To generate a program properties file with the same name as the program being generated, set **J2EE** to NO; set **genProperties** to PROGRAM; and generate into a project other than a J2EE project.
- To generate a program properties file **rununit.properties**, set **J2EE** to NO; set **genProperties** to GLOBAL; and generate into a project other than a J2EE project.
- To avoid generating properties, set **genProperties** to NO.

For additional information

For details on generating properties into a deployment descriptor or into a J2EE environment file, see *Setting deployment-descriptor values*.

For details on the meaning of the runtime properties, see *Java runtime properties (details)*.

For details on accessing runtime properties in your EGL code, see *sysLib.getProperty*.

Related concepts
"EGL debugger" on page 261
"Generation of Java code into a project" on page 301
"J2EE environment file" on page 336
"Program properties file"
"Run unit" on page 721

Related tasks
"Processing Java code that is generated into a directory" on page 315
"Setting up the J2EE run-time environment for EGL-generated code" on page 333
"Setting deployment-descriptor values" on page 334
"Updating the deployment descriptor manually" on page 336
"Updating the J2EE environment file" on page 335

Related reference
"genProperties" on page 375
"J2EE" on page 377
"Java runtime properties (details)" on page 525
"getProperty()" on page 876

Setting up the non-J2EE runtime environment for EGL-generated code

Program properties file

The *program properties file* contains Java run-time properties in a format that is accessible only to a Java program that runs outside of a J2EE environment. For overview information, see *Java run-time properties*.

The program properties file is a text file. Each entry other than comments has the following format:

```
propertyName = propertyValue
```

propertyName
One of the properties described in *Java run-time properties (details)*

propertyValue
The property value that is available to your program at run time

A comment is any line where the first non-text character is a pound sign (#).

A portion of an example file is as follows:

```
# This file contains properties for generated
# Java programs that are being debugged in a
# non-J2EE Java project

vgj.nls.code = ENU
vgj.datemask.gregorian.long.ENU = MM/dd/yyyy
```

For details on the name given to the generated file, see *Generated output (reference)*.

Related concepts
"EGL debugger" on page 261
"Java runtime properties" on page 327

Related tasks
"Generated output (reference)" on page 516

Related reference
"genProperties" on page 375
"J2EE" on page 377
"Java runtime properties (details)" on page 525

Deploying Java applications outside of J2EE

To deploy a Java application outside of J2EE, do as follows:

1. Follow the procedure detailed in *Installing the EGL run-time code for Java*
2. Export the EGL-generated code into jar files, remembering to include generated output files that have extensions other than java; for example, jasper, properties, and tab files
3. Export any manually written Java code into jar files
4. Include the exported files in the classpath of the target machine

Related tasks
"Installing the EGL run-time code for Java"

Installing the EGL run-time code for Java

The EGL run-time code for generated Java applications is available in a zip file on the following Web site:

http://www3.software.ibm.com/ibmdl/pub/software/rationalsdp/rad/60/redist

The supported distributed platforms are AIX, HP-UX, Linux (Intel®), iSeries, Solaris, and Windows 2000/NT/XP. (See product prerequisites for supported versions.) EGL provides 32- and 64-bit support for AIX, HP-UX, and Solaris.

The zip file you download from the previously mentioned Web site includes the following:

- Jar files which contain Java code that is common to all supported distributed platforms
- Platform-specific code

Do as follows:

1. Extract the files in the EGLRuntimes directory to each machine on which deployed EGL applications are to be run outside of a J2EE application server. (These files are already included in any Enterprise Archive (EAR) file used to deploy J2EE applications.)
2. Include the jar files in the classpath of the deployment machines.
3. Copy any platform-specific code to a directory on each deployment machine; and set environment variables for each of those machines as appropriate:

   For AIX (32- or 64-bit support)
   The files of interest are in the directory **Aix** or (for 64-bit support) **Aix64**. Change the PATH and LIBPATH environment variables so they reference the directory that contains the platform-specific code you copied from the Web site.

   For HP-UX (32- or 64-bit support)
   The files of interest are in the directory **hpux** or (for 64-bit support) **hpux64**. Change the PATH and LIBPATH environment variables so they reference the directory that contains the platform-specific code you copied from the Web site.

   For iSeries
   The files of interest are in the directory **Iseries**. In qshell, change to the directory you just uploaded the files to and run the setup.sh script with the ″install″ option:

   > setup.sh install

   In addition, some other environment variables must be set. For information on how to set these environment variables, run the script with the ″envinfo″ option:

   > setup.sh envinfo

   If for some reason you delete a symlink that is created for you during install, you can recreate it with the ″link″ option:

   > setup.sh link

   For Linux
   The files of interest are in the directory **Linux**. Change the PATH and LIBPATH environment variables so they reference the directory that contains the platform-specific code you copied from the Web site.

   For Solaris (32- or 64-bit support)
   The files of interest are in the directory **Solaris** or (for 64-bit support) **Solaris64**. Change the PATH and LIBPATH environment variables so they reference the directory that contains the platform-specific code you copied from the Web site.

   For Windows 2000/NT/XP
   The files of interest are in the directory **Win32**. Change the PATH environment variable so it references the directory that contains the platform-specific code you copied from the Web site.

Related tasks
“Deploying Java applications outside of J2EE” on page 330

Including JAR files in the CLASSPATH of the target machine

You must include any JAR files that contain EGL-generated code or manually written Java code in the CLASSPATH of the target machine. The steps for this process are system dependent. See your operating system documentation for details.

Setting up the UNIX curses library for EGL run time

When you deploy an EGL text program on AIX or Linux, the EGL run time tries to use the UNIX curses library. If the environment is not set up for the UNIX curses library or if that library is not supported, the EGL run time tries to use the Java Swing technology; and if that technology is also not available, the program fails.

The UNIX curses library is required when the user runs an EGL program from a terminal emulator window or a character terminal.

To enable the EGL run time to access the UNIX curses terminal library on AIX or Linux, you must fulfill several steps in the UNIX shell environment. In each of the first two steps, *installDir* refers to the run-time installation library:

1. Modify the LD_LIBRARY_PATH environment variable to include the shared object libCursesCanvas6.so, which is provided in the run-time installation library--

   ```
   export LD_LIBRARY_PATH=$LD_LIBRARY_PATH: /installDir/bin
   ```

2. Modify the CLASSPATH environment variable to add fda6.jar and fdaj6.jar--

   ```
   export CLASSPATH=$CLASSPATH:
   /installDir/lib/fda6.jar: /installDir/lib/fdaj6.jar
   ```

   The previous information must be typed on a single line.

3. Set the TERM environment variable to the appropriate terminal setting, as in the following example:

   ```
   export TERM=vt100
   ```

   If terminal exceptions occur, try various terminal settings such as xterm, dtterm, or vt220.

4. Run your EGL Java program from the UNIX shell, as in the following example:

   ```
   java myProgram
   ```

   Make sure that the CLASSPATH environment variable identifies the directory in which your program resides.

For additional details on using the Curses library on UNIX, refer to the UNIX man pages.

Setting up the TCP/IP listener for a called non-J2EE application

If you want a caller to use TCP/IP to exchange data with a called non-J2EE Java program, you must set up a TCP/IP listener for the called program.

If you are using TCP/IP to communicate with a called non-J2EE Java program, you must configure a standalone Java program called CSOTcpipListener for that program. Specifically, you must do as follows:

- Make sure that the classpath used when running CSOTcpipListener contains fda6.jar, fdaj6.jar, and the directories or archives that contain the called programs; and

- Set the Java run-time property **tcpiplistener.port** to the number of the port at which CSOTcpipListener receives data.

You can start the standalone TCP/IP listener in either of two ways:

- To start the listener from the workbench, use the launch configuration for a Java application. In this case, you can specify the name of the properties file in the program arguments of the launch configuration. Alternatively, if you are using the file tcpiplistener.properties as a default, that file should not be in a folder, but should be directly under the project that you specified when you created the launch configuration.
- To start the listener from the command line, run the program as follows:

  ```
  java CSOTcpipListener propertiesFile
  ```

  propertiesFile
  The fully qualified path to the properties file used by the TCP/IP listener. If you do not specify a properties file, the listener attempts to open the following file in the current directory:

  ```
  tcpiplistener.properties
  ```

Related tasks
"Providing access to non-EGL jar files" on page 343

Setting up the J2EE run-time environment for EGL-generated code

EGL-generated Java programs and wrappers run on a J2EE 1.4 server such as WebSphere Application Server v6.0, on the platforms listed in *Run-time configurations*.

Your primary tasks when embedding generated Java classes into a J2EE module are as follows:

1. Place output files in a project, in either of two ways:
   - Generate into a project, as is the preferred technique; or
   - Generate into a directory, then import files into a project.
2. Place a linkage properties file in the module (see *Deploying a linkage properties file*).
3. Eliminate duplicate jar files.
4. Export an enterprise archive (.ear) file, which may include Web application archive (.war) files and other .ear files; for details on the procedure, see the help pages on export.
5. Import the .ear file into the J2EE server that will host your application; for details on the procedure, see the documentation for your J2EE server.

You may need to fulfill these tasks as well:

- "Setting up a J2EE JDBC connection" on page 341
- "Setting up the J2EE server for CICSJ2C calls" on page 337
- "Setting up the TCP/IP listener for a called appl in a J2EE appl client module" on page 338
- "Setting up the TCP/IP listener for a called non-J2EE application" on page 332

Related concepts
"Development process" on page 8
"Generation of Java code into a project" on page 301
"Java program, PageHandler, and library" on page 306

"Linkage options part" on page 291
"Linkage properties file" on page 343
"Run-time configurations" on page 9

Related tasks
"Deploying Java applications outside of J2EE" on page 330
"Deploying a linkage properties file" on page 342
"Eliminating duplicate jar files"
"Generating deployment code for EJB projects" on page 319
"Processing Java code that is generated into a directory" on page 315
"Providing access to non-EGL jar files" on page 343
"Setting deployment-descriptor values"
"Setting the JNDI name for EJB projects" on page 337
"Setting up a J2EE JDBC connection" on page 341
"Setting up the J2EE server for CICSJ2C calls" on page 337
"Setting up the TCP/IP listener for a called appl in a J2EE appl client module" on page 338
"Understanding how a standard JDBC connection is made" on page 245
"Updating the deployment descriptor manually" on page 336
"Updating the J2EE environment file" on page 335

Related reference
"Java runtime properties (details)" on page 525
"Linkage properties file (details)" on page 637

Eliminating duplicate jar files

If you place multiple J2EE modules into a single ear file, eliminate the duplicate jar files as follows:

1. Move a copy of each duplicate jar file to the top level of the ear
2. Delete the duplicate jar files from the J2EE modules
3. Ensure that the build path for each of the affected J2EE modules points to the jar files in the ear; specifically, do as follows for each of those J2EE modules:
   a. Right-click on the module from within the Project Explorer or J2EE view
   b. Select **Edit Module Dependencies**
   c. When the Module Dependencies dialog is displayed, select the jar files to access from the top level of the ear, then click **Finish**.

Related tasks
"Setting up the J2EE run-time environment for EGL-generated code" on page 333

Setting deployment-descriptor values

An important task is to place run-time values (similar to environment-variable values) into the deployment descriptor of your J2EE module. You can interact with a workbench editor listed in the next table, for example; and in any case, the editors are available if you wish to reassign a value.

Project type	Name of deployment descriptor	How to assign values
application client	application-client.xml	Use the XML editor, Design tab
EJB	ejb-jar.xml	Use the EJB editor, Beans tab
J2EE Web	web.xml	Use the web.xml editor, Environment tab

The recommended way to update the deployment descriptor is to add content automatically, as happens if all of the following conditions apply:

- You are generating a Java program or wrapper
- The build descriptor option **genProperties** is set to GLOBAL or PROGRAM
- You are generating for J2EE run time by setting **J2EE** to YES
- You set **genProject** to a valid J2EE project

EGL never deletes a property from an existing deployment descriptor, but does as follows:

- Overwrites properties that already exist
- Appends properties that do not exist

Another method of updating the deployment descriptor is to paste values from the J2EE environment file, which is an output of generation if all of the following conditions apply:

- You are generating a Java program
- The build descriptor option **genProperties** is set to GLOBAL or PROGRAM
- You are generating for J2EE run time by setting **J2EE** to YES
- You do not set **genProject** to a valid J2EE project, as when you generate into a directory instead

Before you paste entries from a J2EE environment file into the deployment descriptor of an application client or EJB project, you need to change the order of entries in the file, as described in *Updating the J2EE environment file*. You do not need to change the order of entries if you are working with a J2EE Web project.

For details on deployment descriptor properties, see *Java run-time properties (details)*.

Related concepts
"J2EE environment file" on page 336
"Generation of Java code into a project" on page 301
"Program properties file" on page 329

Related tasks
"Processing Java code that is generated into a directory" on page 315
"Setting up the J2EE run-time environment for EGL-generated code" on page 333
"Updating the J2EE environment file"
"Updating the deployment descriptor manually" on page 336

Related reference
"genDirectory" on page 372
"genProperties" on page 375
"J2EE" on page 377
"Java runtime properties (details)" on page 525

Updating the J2EE environment file

The J2EE environment file contains a series of entries like the following example:

```
<env-entry>
  <env-entry-name>vgj.nls.code</env-entry-name>
  <env-entry-value>ENU</env-entry-value>
  <env-entry-type>java.lang.String</env-entry-type>
</env-entry>
```

The order of sub-elements is name, value, type. This is correct for J2EE Web projects; however, for application client and EJB projects, you need to change the order to name, type, value. For the example above, change the order of the sub-elements to:

```
<env-entry>
  <env-entry-name>vgj.nls.code</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>ENU</env-entry-value>
</env-entry>
```

This step can be avoided if you generate directly into a project instead of into a directory. When you generate into a project, EGL can determine the type of project you are using and generate the environment entries in the appropriate order.

Related tasks
"Setting up the J2EE run-time environment for EGL-generated code" on page 333
"Setting deployment-descriptor values" on page 334

Related reference
"Java runtime properties (details)" on page 525

J2EE environment file

A *J2EE environment file* is a text file that contains property-and-value pairs that are derived from information that you specify when you generate a Java program. The sources of information are the build descriptor, the resource associations part, and the linkage options part.

When you configure the environment of the Java program, you can use the J2EE environment file as the basis of information that you place in the run-time deployment descriptor.

For details on the name of the J2EE environment file, see *Generated output (reference)*.

For details on the different ways that you can set deployment-descriptor values, see *Setting deployment-descriptor values*.

Related concepts
"Run-time configurations" on page 9

Related tasks
"Setting up the J2EE run-time environment for EGL-generated code" on page 333
"Setting deployment-descriptor values" on page 334

Related reference
"Generated output (reference)" on page 516
"genProperties" on page 375
"sqlDB" on page 384

Updating the deployment descriptor manually

If you are updating a deployment descriptor from a generated J2EE environment file, do as follows:

1. Read the overview information in *Setting deployment-descriptor values*.
2. If you worked on an application client or EJB project, you must make sure that the order of the sub-elements in the generated environment entries is correct, as described in *Updating the J2EE environment file*.

3. Copy the environment entries into your project's deployment descriptor as follows:
   a. Make a backup copy of the deployment descriptor.
   b. Open the J2EE environment file, which is called *programName*-env.txt file. Copy the environment entries into the clipboard.
   c. Double-click on the deployment descriptor.
   d. Click on the Source tab.
   e. Paste the entries at a proper location.

For details on deployment descriptors, see *Java run-time properties (details)*.

Related tasks
"Setting deployment-descriptor values" on page 334
"Setting up the J2EE run-time environment for EGL-generated code" on page 333
"Updating the J2EE environment file" on page 335

Related reference
"Java runtime properties (details)" on page 525

Setting the JNDI name for EJB projects

To set the JNDI name for an EJB project, do as follows:

1. Right click on ejb-jar.xml (the deployment descriptor) to open the context menu.
2. Use the EJB Editor to open the following file in the project--
   ```
   \ejbModule\META-INF\ejb-jar.xml
   ```
3. Click on the Beans tab.
4. On the list, click on the name of the EJB you just generated.
5. Enter the JNDI name under WebSphere Bindings. The JNDI name must be as follows for use by the EGL run-time code:
   - First character of the program name, in upper case
   - Subsequent characters of the program name, in lower case
   - The letters EJB in upper case.

Related tasks
"Setting up the J2EE run-time environment for EGL-generated code" on page 333

Setting up the J2EE server for CICSJ2C calls

You must set up a ConnectionFactory in the J2EE server for each CICS transaction accessed through protocol CICSJ2C.

If a generated Java wrapper is making the CICSJ2C call, you can handle security in any of the following ways (where a wrapper-specified value overrides that of the J2EE server):

- Set the userid and password in the wrapper's CSOCallOptions object; or
- Set the userid and password in the ConnectionFactory configuration in the J2EE server; or
- Set up the CICS region so that user authentication is not required.

When calling a program from WebSphere 390, the following restrictions apply:

- If the callLink element property **luwControl** is set to CLIENT, the call fails. The WebSphere 390 connect implementation does not support an extended unit of work.
- The setting of deployment descriptor property **cso.cicsj2c.timeout** has no effect.

  By default, timeouts never occur. In the EXCI options table generated by the macro DFHXCOPT, however, you can set the parameter TIMEOUT, which lets you specify the time that EXCI will wait for a DPL command (an ECI request) to complete. A setting of 0 means to wait indefinitely.

For details, see *Java Connectors for CICS: Featuring the J2EE Connector Architecture* (SG24-6401-00), which is available from web site *http://www.redbooks.ibm.com*.

Related tasks
"Setting up the J2EE run-time environment for EGL-generated code" on page 333

Setting up the TCP/IP listener for a called appl in a J2EE appl client module

If you want a caller to use TCP/IP to exchange data with a called program in a J2EE application client module, you must set up a TCP/IP listener for the called program.

You need to make sure that the following situation is in effect:

- An EGL-specific TCP/IP listener is the main class for the module, as specified in the manifest (.MF) file of the module
- A port is assigned to the listener, as specified in the deployment descriptor (application-client.xml) of the module

If you are working with projects at the level of J2EE 1.2, it is recommended that you set up an application client project that is initialized with the listener, before you generate any EGL code into that project. If you fail to follow that sequence (listener first, EGL code second) or if you are working with projects at the level of J2EE 1.3, you need to follow the procedure described in *Providing access to the listener from an existing application client project*.

Setting up an application client project that is initialized with the listener

To set up an application client project that is initialized with the listener, do as follows:

1. Click **File > Import**.
2. At the Select page, double-click **App Client JAR file**.
3. At the Application Client Import page, specify several details--
   a. In the Application Client file field, specify the jar file that sets up access to (but does not include) the TCP/IP listener:

      ```
      installationDir\egl\eclipse\plugins\
      com.ibm.etools.egl.generators_version\runtime\EGLTcpipListener.jar
      ```

      installationDir
      The product installation directory, such as C:\Program Files\IBM\RSPD\6.0. If you installed and kept a Rational Developer product before installing the product that you are using now, you may need to specify the directory that was used in the earlier install.

      version
      The latest version of the plugin; for example, 6.0.0.

The TCP/IP listener itself resides in fdaj6.jar, which is placed in the application client project when you first generate EGL code into that project.

b. Click the **New** radio button, which follows the label **Application Client project**.

c. Type the name of the application client project into the **New Project Name** field; then set or unset the **Use default** check box. If you set the check box, the project is stored in a workspace directory that is named with the name of the project. If you unset the check box, specify the project name in the **New project location** field.

d. Specify the name of the enterprise application project that contains the application client project:

- If you are using an existing J2EE 1.2 enterprise application project, click the **Existing** radio button, which follows the label **Enterprise application project**. In this case, specify the project name in the **Existing project name** field.
- If you are creating a new enterprise application project, do as follows:
  1) Click the **New** radio button, which follows the label **Enterprise application project**.
  2) Type the name of the enterprise application project into the **New Project Name** field.
  3) Set or unset the **Use default** check box.
  4) If you set the check box, the project is stored in a workspace directory that is named with the name of the project. If you unset the check box, specify the project name in the **New project location** field.

4. Click **Finish**.
5. Ignore the two warning messages that refer to the jar files (fda6.jar, fdaj6.jar) that will be added automatically when you generate EGL output into the project.

In the application client project, the deployment descriptor property **tcpiplistener.port** is set to the number of the port at which the listener receives data. By default, that port number is 9876. To change the port number, do as follows:

1. In the Project Explorer view, expand your application client project, then appClientModule, then META-INF
2. Click **application-client.xml > Open With > Deployment Descriptor editor**
3. The deployment descriptor editor includes a source tab; click that tab and change the 9876 value, which is the content of the last tag in a grouping like this:

```
<env-entry-name>tcpiplistener.port</env-entry-name>
<env-entry-type>java.lang.Integer</env-entry-name>
<env-entry-value>9876</env-entry-value>
```

4. To save the deployment descriptor, press **Ctrl-S**.

Providing access to the listener from an existing application client project

If you generate EGL code into an application client project that was not initialized with the listener, you need to update the deployment descriptor (application-client.xml) and the manifest file (MANIFEST.MF):

1. In the Project Explorer view, expand your application client project, then appClientModule, then META-INF
2. Click **application-client.xml > Open With > Deployment Descriptor Editor**

3. The deployment descriptor editor includes a Source tab. Click that tab. In the text, immediately below the line that holds the tag <display-name>, add the following entries (however, if port 9876 is already in use on your machine, substitute a different number for 9876):

```
<env-entry>
  <env-entry-name>tcpiplistener.port</env-entry-name>
  <env-entry-type>java.lang.Integer</env-entry-name>
  <env-entry-value>9876</env-entry-value>
</env-entry>
```

4. To save the deployment descriptor, press **Ctrl-S**.
5. In the Project Explorer view, click **MANIFEST.MF > Open With > JAR Dependency Editor**.
6. The JAR Dependency Editor includes a Dependencies tab. Click that tab.
7. Review the Dependencies section to make sure that fda6.jar and fdaj6.jar are selected.
8. In the Main Class section, in the Main-Class field, type the following value or use the Browse mechanism to specify the following value:

   ```
   CSOTcpipListenerJ2EE
   ```

9. To save the manifest file, press **Ctrl-S**.

Deploying the application client project

To start the TCP/IP listener, follow either of two procedures:

- Start the listener from the Workbench by using the launch configuration for a WebSphere application client:
  1. Switch to a J2EE perspective
  2. Click **Run > Run**
  3. At the Launch Configurations page, click either **WebSphere v5 Application Client** (as is necessary if you are working with a project at the level of J2EE 1.3) or **WebSphere v4 Application Client**
  4. Select an existing configuration. Alternatively, click **New** and set up a configuration:
     a. In the Application tab, select the enterprise application project
     b. In the Arguments tab, add an argument:

        ```
        -CCjar=myJar.jar
        ```

        myJar.jar
        The name of the application client jar file. This argument is only necessary when you have multiple client jar files in the ear file. In most cases, the value is the name of the application client project, followed by the extension .jar.

        If you wish to confirm the relationship of project name to jar-file name, do as follows:

        1) In the Project Explorer view, expand your enterprise application project, then META-INF
        2) Click **application.xml > Open With > Deployment Descriptor Editor**.
        3) The Deployment Descriptor editor includes a Module tab. Click that tab.
        4) At the leftmost part of the page, click the jar file and see (at the rightmost part of the page) the project name associated with that jar file.

- If you have on WebSphere Application Server (WAS) installed, you can use launchClient.bat, which is in the WAS installation directory, subdirectory bin.

  You can invoke launchClient as follows from a command prompt:

  ```
  launchClient myCode.ear -CCjar=myJar.jar
  ```

  myCode.ear
  The name of the enterprise archive

  myJar.jar
  The name of the application client jar file, as described in relation to the Workbench procedure

  For details on launchClient.bat, see the WebSphere Application Server documentation.

Related tasks
"Providing access to non-EGL jar files" on page 343
"Setting up the J2EE run-time environment for EGL-generated code" on page 333

Setting up a J2EE JDBC connection

If you are connecting to a relational database at run time, you need to define a data source for use with your program. The directions are in the help system of the WebSphere Sever administrative console.

When you define a data source, assign values to the following properties:

JNDI name
Specify a value that matches the name to which the database is bound in the JNDI registry:

- If you are defining a data source which connects to a database that your J2EE module uses by default, make sure that the JNDI name specified in the data source definition matches the value of the **vgj.jdbc.default.database** property in the J2EE deployment descriptor used at run time
- If you are defining a data source that will be accessed when the system function VGLib.connectionService runs, make sure that the JNDI name specified in the data source definition matches the value of the appropriate **vgj.jdbc.database.*SN*** property in the J2EE deployment descriptor used at run time

Database name
Specify the name of your database, as known to the database management system

User ID
Specify the user name for connecting to the database.

If the data source definition refers to the default database, the value you specify in the User ID field is overridden by any value set in the **vgj.jdbc.default.userid** property of the J2EE deployment descriptor used at run time, but only if you have specified values for both **vgj.jdbc.default.userid** and **vgj.jdbc.default.password**. Similarly, if the data source definition refers to a database that is accessed by way of the system function sysLib.connect or VGLib.connectionService, the value you specify in the User ID field is overridden by any user ID that you specify in the call to that system function, but only if the call passes both a user ID and password.

You specify the name when setting up the authentication alias. To reach the display where you can define that alias, follow this sequence in the

Administrative Console: **Security** > **GlobalSecurity** > **Authentication** > **JAAS Configuration** > **J2C Authentication Data**.

Password
Specify the password for connecting to the database. If the data source definition refers to the default database, the value you specify in the Password field is overridden by any value set in the **vgj.jdbc.default.password** property of the J2EE deployment descriptor used at run time, but only you have specified values for both **vgj.jdbc.default.userid** and **vgj.jdbc.default.password**. Similarly, if the data source definition refers to a database that is accessed by way of the system function VGLib.connectionService, the value you specify in the Password field is overridden by any password that you specify in the call to that system function, but only if the call passes both a user ID and password.

You specify the password when setting up the authentication alias. To reach the display where you can define that alias, follow this sequence in the Administrative Console: **Security** > **GlobalSecurity** > **Authentication** > **JAAS Configuration** > **J2C Authentication Data**.

You may define multiple data sources, in which case you use the system function VGLib.connectionService to switch between them.

For details on the meaning of the deployment descriptor properties, including details on how the generated values are derived, see *Java run-time properties (reference)*.

Related tasks
"Setting up the J2EE run-time environment for EGL-generated code" on page 333
"Understanding how a standard JDBC connection is made" on page 245

Related reference
"Java runtime properties (details)" on page 525
"JDBC driver requirements in EGL" on page 543
"connectionService()" on page 888

Deploying a linkage properties file

The linkage properties file must be in the same J2EE application as the Java program that uses the file. If the file is in the top-level directory of the application, set the Java run-time property **cso.linkageOptions.*LO*** to the file name, without path information. If the file is under the top-level directory of the application, use a path that starts at the top-level directory and includes a virgule (/) for each level, even if the application is running on a Windows platform.

When you are developing a J2EE project, the top-level directory corresponds to the appClientModule, ejbModule, or Web Content directory of the project in which the module resides. When you are developing a Java project, the top-level directory is the project directory.

For additional details on how a linkage properties file is formatted and identified, see *Linkage properties file (reference)*.

Related concepts
"Java runtime properties" on page 327
"Linkage options part" on page 291
"Linkage properties file" on page 343

Related tasks
"Setting up the J2EE run-time environment for EGL-generated code" on page 333
"Setting deployment-descriptor values" on page 334

Related reference
"callLink element" on page 395
"Exception handling" on page 89
"Linkage properties file (details)" on page 637

Linkage properties file

A *linkage properties file* file is a text file that is used at Java run time to give details on how a generated Java program or wrapper calls a generated Java program in a different process.

The file is applicable only if you specified that linkage options for a Java program or wrapper are set at run time instead of at generation time. You may generate the file or create one from scratch.

For details on when the file is generated and on the file format, see *Linkage properties file (details)*. For details on the name of the generated file, see *Generated output (reference)*. For details on deployment, see *Deploying a linkage properties file*.

Related concepts
"Generated output" on page 515

Related tasks
"Deploying a linkage properties file" on page 342
"Setting up the J2EE run-time environment for EGL-generated code" on page 333

Related reference
"Generated output (reference)" on page 516
"genProperties" on page 375
"Linkage properties file (details)" on page 637

Providing access to non-EGL jar files

You may need to provide access to non-EGL jar files to debug and run your EGL-generated Java code. The process for providing access to those files varies by project type:

Application client project

Before using the interpretive debugger, reference the non-EGL jar files in the CLASSPATH variable, as described in *Setting preferences for the EGL debugger*.

Before running your code (with or without the EGL Java debugger), do as follows:

1. For each enterprise application project that references the application client project, import jar files of interest from a directory in the file system:
   a. In the Project Explorer view, right-click an enterprise application project and click **Import**
   b. At the Select page, click **File System**
   c. At the File System page, specify the directory in which the jar files reside
   d. At the right of the page, select the jar files of interest
   e. Click **Finish**

2. Update the manifest in the application client project so that the jar files in the enterprise application project are available at run time:
   a. In the Project Explorer view, right-click your application client project and click **Properties**
   b. At the left of the Properties page, click **Java JAR Dependencies**
   c. When the section called Java JAR Dependencies is displayed at the right of the page, set each check box that corresponds to a jar file of interest
   d. Click **OK**

EJB project

Before using the interpretive debugger, reference the non-EGL jar files in the CLASSPATH variable, as described in *Setting preferences for the EGL debugger*.

Before running your code (with or without the EGL Java debugger), do as follows:

1. For each enterprise application project that references the EJB project, import jar files of interest from a directory in the file system:
   a. In the Project Explorer view, right-click an enterprise application project and click **Import**
   b. At the Select page, click **File System**
   c. At the File System page, specify the directory in which the jar files reside
   d. At the right of the page, select the jar files of interest
   e. Click **Finish**
2. Update the manifest in the EJB project so that the jar files in the enterprise application project are available at run time:
   a. In the Project Explorer view, right-click your EJB project and click **Properties**
   b. At the left of the Properties page, click **Java JAR Dependencies**
   c. When the section called Java JAR Dependencies is displayed at the right of the page, set each check box that corresponds to a jar file of interest
   d. Click **OK**

Java project

Before running your code with the interpretive debugger, reference the non-EGL jar files in the CLASSPATH variable, as described in *Setting preferences for the EGL debugger*.

Before running your code with the EGL Java debugger, add entries to the project's Java build path:

1. In the Project Explorer view, right-click your Java project and click **Properties**
2. At the left of the Properties page, click **Java Build Path**
3. When the section called Java Build Path is displayed at the right of the page, click the Libraries tab
4. For each jar file to be added, click **Add External Jars** and use the Browse mechanism to select the file
5. To close the Properties page, click **OK**

J2EE Web project

Before using the interpretive debugger, reference the non-EGL jar files in the CLASSPATH variable, as described in *Setting preferences for the EGL debugger*.

Before running your code (with or without the EGL Java debugger), import the jar files from the file system to the following Web project folder:

```
Web Content/WEB-INF/lib
```

The import process is as follows for a set of jar files in a directory:

1. In the Project Explorer view, expand the Web project, expand **Web Content**, expand **WEB-INF**, right-click **lib**, and click **Import**
2. At the Select page, click **File System**
3. At the File System page, specify the directory in which the jar files reside
4. At the right of the page, select the jar files of interest
5. Click **Finish**

The following jar-file requirements are in effect:

- A generated Java program that accesses MQSeries in any way requires MQ Series Classes for Java; in particular, the Java program needs the following jar files (although not at preparation time):
  - com.ibm.mq.jar
  - com.ibm.mqbind.jar

  If you have WebSphere MQ V5.2, the software is in IBM WebSphere MQ SupportPac™ MA88, which you can find by going to the IBM web site (www.ibm.com) and searching for MA88. Download and install the software; then you can access the jar files from the Java\lib subdirectory of the directory where you installed that software.

  If you have WebSphere MQ V5.3, you can get the equivalent software by doing a custom install and selecting Java Messaging. Then you can access the jar files from the Java\lib subdirectory of the MQSeries installation directory.
- A generated Java program or wrapper that uses the protocol CICSJ2C to access CICS for z/OS requires access to connector.jar and cicsj2ee.jar, but only at run time. Those files are available to you when you install the CICS Transaction Gateway.

  Note: Access of CICS is possible when the EGL Java debugger runs in J2EE. Calls to CICS are attempted but fail, however, when that debugger runs outside of J2EE or when you are using the EGL interpretive debugger, which always runs outside of J2EE.
- A generated Java program that accesses an SQL table requires a file that is installed with the database management system--
  - For DB2 UDB, the file is one of the following:

    ```
    sqllib\java\db2java.zip
    sqllib\java\db2jcc.jar
    ```

    The second of those files is available with DB2 UDB Version 8 or higher, as described in the DB2 UDB documentation.
  - For Informix, the files are as follows:

    ```
    ifxjdbc.jar
    ifxjdbcx.jar
    ```
  - For Oracle, consult the Oracle documentation.

  The database file is required at run time, and can be used to validate SQL statements at preparation time.

Related tasks

"Setting preferences for the EGL debugger" on page 108
"Setting up the J2EE run-time environment for EGL-generated code" on page 333

EGL reference

Assignment compatibility in EGL

The assignment-compatibility rules (as described later) apply in these situations:

- Your code assigns one non-reference variable to another; or
- EGL transfers data between an argument and the related parameter in a function invocation, but only if the parameter in the receiving function has the modifier IN (in which case the argument is the source) or OUT (in which case the parameter is the source). The assignment-compatibility rules do not apply, however, if the parameter is in the onPageLoad function of a PageHandler; for details on that case, see *Reference Compatibility in EGL*.

Assignment compatibility is based on the following type classification:

- Text types are CHAR, MBCHAR, STRING, and UNICODE
- Numeric types are BIN, INT, BIGINT, SMALLINT, DECIMAL, NUM, NUMBER, FLOAT, SMALLFLOAT, MONEY
- Datetime types are DATE, INTERVAL, TIME, TIMESTAMP
- HEX is in its own category
- VisualAge Generator legacy types are DBCHAR, NUMC, and PACF, each of which follows VisualAge Generator rules

The assignment-compatibility rules are as follows:

- A field of any text type can be assigned to a field of any text type
- A field of any numeric type can be assigned to a field of any numeric type
- A field of any datetime type can be assigned to a field of any text or numeric type
- A field of type STRING or CHAR can be assigned to or from a field of type HEX
- A field of type CHAR can be assigned to a field of type NUM
- To assign a field of a numeric type to a field of a text type, use the system function **StrLib.formatNumber**
- To assign a field of type DATE, TIME, or TIMESTAMP to a formatted field of a text type, use the appropriate system function:
  - **StrLib.formatDate** (for dates)
  - **StrLib.formatTime** (for times)
  - **StrLib.formatTimestamp** (for timestamps)
- To assign a field of a text type to a field of type DATE, TIME, or TIMESTAMP, use the appropriate system function:
  - **ConverseLib.dateValue** (for dates)
  - **ConverseLib.timeValue** (for times)
  - **ConverseLib.timestampValue** (for timestamps)

Assignment across numeric types

A value of any of the numeric types (including NUMC and PACF) can be assigned to a field of any numeric type and size, and EGL does the conversions necessary to retain the value in the target format.

Non-significant zeros are added or truncated as needed. (Initial zeros in the integer part of a value are non-significant, as are trailing digits in the fraction part of a value.)

For any of the numeric types, you can use the system variable sysVar.overflowIndicator to test whether an assignment or arithmetic calculation resulted in an arithmetic overflow, and you can set the system variable VGVar.handleOverflow to specify the consequence of such an overflow.

If an arithmetic overflow occurs, the value in the target field is unchanged. If an arithmetic overflow does not occur, the value assigned to the target field is aligned in accordance with the declaration of the target field.

Let's assume that you are copying a field of type NUM to another and that the run-time value of the source field is 108.314:

- If the target field allows seven digits with one decimal place, the target field receives the value 000108.3, and a numeric overflow is *not* detected. (A loss of precision in a fractional value is not considered an overflow.)
- If the target field allows four digits with two decimal places, a numeric overflow is detected, and the value in the target field is unchanged

When you assign a floating-point value (type FLOAT or SMALLFLOAT) to a field of a fixed-point type, the target value is truncated if necessary. If a source value is 108.357 and the fixed-point target has one decimal place, for example, the target receives 108.3.

Other cross-type assignments

Details on other cross-type assignments are as follows:

- The assignment of a value of type NUM to a target of type CHAR is valid only if the source declaration has no decimal places. This operation is equivalent to a CHAR-to-CHAR assignment.

  If the source length is 4 and value is 21, for example, the content is equivalent to ″0021″, and a length mismatch does not cause an error condition:

  – If the length of the target is 5, the value is stored as ″0021 ″ (a single-byte space was added on the right)
  – If the length of the target is 3, the value is stored as ″002″ (a digit was truncated on the right)

  If the value of type NUM is negative and assigned to a value of type CHAR, the last byte copied into the field is an unprintable character.
- The assignment of a value of type CHAR to a target of type NUM is valid only in the following case:
  – The source (a field or text expression) has digits with no other characters
  – The target declaration has no decimal place

  This operation is equivalent to a NUM-to-NUM assignment.

  If the source length is 4 and value is ″0021″, for example, the content is equivalent to a numeric 21, and the effect of a length mismatch is shown in these examples:

  – If the length of the target is 5, the value is stored as 00021 (a numeric zero was padded on the left)
  – If the length of the target is 3, the value is stored as 021 (a non-significant digit was truncated)
  – If the length of the target is 1, the value is stored as 1

- The assignment of a value of type NUMC to a target of type CHAR is possible in two steps, which eliminates the sign if the value is positive:
  1. Assign the NUMC value to a target of type NUM
  2. Assign the NUM value to a target of type CHAR

  If the value of the target of type NUMC is negative, the last byte copied into the target of type CHAR is an unprintable character.
- The assignment of a value of type CHAR to a target of type HEX is valid only if the characters in the source are within the range of hexadecimal digits (0-9, A-F, a-f).
- The assignment of a value of type HEX to a target of type CHAR stores digits and uppercase letters (A-F) in the target.
- The assignment of a value of type MONEY to a target of type CHAR is not valid. The best practice for converting from MONEY to CHAR is to use the system function **strLib.formatNumber**.
- The assignment of a value of type NUM or CHAR to a target of type DATE is valid only if the source value is a valid date in accordance with the mask *yyyyMMdd*; for details, see the topic *DATE*.
- The assignment of a value of type NUM or CHAR to a target of type TIME is valid only if the source value is a valid time in accordance with the mask *hhmmss*; for details, see the topic *TIME*.
- The assignment of a value of type CHAR to a target of type TIMESTAMP is valid only if the source value is a valid timestamp in accordance with the mask of the TIMESTAMP field. An example is as follows:

  ```
  // NOT valid because February 30 is not a valid date
  myTS timestamp("yyyyMMdd");
  myTS = "20050230";
  ```

  If characters at the beginning of a full mask are missing (for example, if the mask is ″dd″), EGL assumes that the higher-level characters (″yyyyMM″, in this case) represent the current moment, in accordance with the machine clock. The following statements cause a run-time error in February:

  ```
  // NOT valid if run in February
  myTS timestamp("dd");
  myTS = "30";
  ```
- The assignment of a value of type TIME or DATE to a target of type NUM is equivalent to a NUM-to-NUM assignment.
- The assignment of a value of type TIME, DATE, or TIMESTAMP to a target of type CHAR is equivalent to a CHAR-to-CHAR assignment.

Padding and truncation with character types

If the target is of a non-STRING character type (including DBCHAR and HEX) and has more space than is required to store a source value, EGL pads data on the right:

- Uses single-byte blanks to pad a target of type CHAR or MBCHAR
- Uses double-byte blanks to pad a target of type DBCHAR
- Uses Unicode double-byte blanks to pad a target of type UNICODE
- Uses binary zeros to pad a target of type HEX, which means (for example) that a source value ″0A″ is stored in a two-byte target as ″0A00″ rather than as ″000A″

EGL truncates values on the right if the target of a character type has insufficient space to store the source value. No error is signaled. A special case can occur in the following situation:

- The run-time platform supports the EBCDIC character set

- The assignment statement copies a literal of type MBCHAR or an item of type MBCHAR to a shorter item of type MBCHAR
- A byte-by-byte truncation would remove a final shift-in character or split a DBCHAR character

In this situation, EGL truncates characters as needed to ensure that the target item contains a valid string of type MBCHAR, then adds (if necessary) terminating single-byte blanks.

Assignment between timestamps

If you assign an item of type TIMESTAMP to another field of type TIMESTAMP, the following rules apply:

- If the mask of the source field is missing relatively high-level entries that are required by the target field, those target entries are assigned in accordance with the clock on the machine at the time of the assignment, as shown by these examples:

  —
  ```
  sourceTimeStamp timestamp ("MMdd");
  targetTimeStamp timestamp ("yyyyMMdd");

  sourceTimeStamp = "1201";

  // if this code runs in 2004, the next statement
  // assigns 20041201 to targetTimeStamp
  targetTimeStamp = sourceTimeStamp;
  ```
  —
  ```
  sourceTimeStamp02 timestamp ("ssff");
  targetTimeStamp02 timestamp ("mmssff");

  sourceTimeStamp02 = "3201";

  // the next assignment includes the minute
  // that is current when the assignment statement runs
  targetTimeStamp02 = sourceTimeStamp02;
  ```

  – If the mask of the source item is missing relatively low-level entries that are required by the target field, those target entries are assigned the lowest valid values, as shown by these examples:

    -
    ```
    sourceTimeStamp timestamp ("yyyyMM");
    targetTimeStamp timestamp ("yyyyMMdd");

    sourceTimeStamp = "200412";

    // regardless of the day, the next statement
    // assigns 20041201 to targetTimeStamp
    targetTimeStamp = sourceTimeStamp;
    ```
    -
    ```
    sourceTimeStamp02 timestamp ("hh");
    targetTimeStamp02 timestamp ("hhmm");

    sourceTimeStamp02 = "11";

    // regardless of the minute, the next statement
    // assigns 1100 to targetTimeStamp02
    targetTimeStamp02 = sourceTimeStamp02;
    ```

Assignment to or from substructured fields in fixed structures

You can assign a substructured field to a non-substructured field or the reverse, and you can assign values between two substructured fields. Assume, for example, that variables named *myNum* and *myRecord* are based on the following parts:

```
DataItem myNumPart
  NUM(12)
end
```

```
Record myRecordPart type basicRecord
  10 topMost CHAR(4);
    20 next01 HEX(4);
    20 next02 HEX(4);
end
```

The assignment of a value of type HEX to an item of type NUM is not valid outside of the mathematical system variables; but an assignment of the form **myNum = topMost** is valid because **topMost** is of type CHAR. In general terms, the primitive types of the fields in the assignment statement guide the assignment, and the primitive types of subordinate items are not taken into account.

The primitive type of a substructured item is CHAR by default. If you assign data to or from a substructured field and do not specify a different primitive type at declaration time, the rules described earlier for fields of type CHAR are in effect during the assignment.

Assignment of a fixed record

An assignment of one fixed record to another is equivalent to assigning one substructured item of type CHAR to another. A mismatch in length adds single-byte blanks to the right of the received value or removes single-byte characters from the right of the received value. The assignment does not consider the primitive types of subordinate structure fields.

The following exceptions apply:

- The content of a record can be assigned to a record or to a field of type CHAR, HEX, or MBCHAR, but not to a field of any other type
- A record can receive data from a record or from a string literal or from a field of type CHAR, HEX, or MBCHAR, but not from a numeric literal or from a field of a type other than CHAR, HEX, or MBCHAR

Finally, if you assign an SQL record to or from a record of a different type, you must ensure that the non-SQL record has space for the four-byte area that precedes each structure field.

Related concepts
"PageHandler" on page 180

Related reference
"Assignments" on page 352
"DATE" on page 38
"EGL statements" on page 83
"formatNumber()" on page 851
"Function parameters" on page 508
"Function part in EGL source format" on page 513
"handleOverflow" on page 921
"move" on page 592
"overflowIndicator" on page 906
"PageHandler part in EGL source format" on page 659
"Primitive types" on page 31
"Program parameters" on page 706
"Program part in EGL source format" on page 707"Substrings" on page 731
"TIME" on page 40

Assignments

An EGL assignment copies data from one area of memory to another and can copy the result of a numeric or text expression into a source field.

target A field, record, fixed record, or system variable.

You can specify a substring on the left side of an assignment statement if the target field is of type CHAR, DBCHAR, or UNICODE. The substring area is filled (padded with blanks, if necessary), and the assigned text does not extend beyond the substring area but is truncated, if necessary. For syntax details, see *Substrings*.

source A record, fixed record, or a numeric or character expression

Examples of assignments are as follows:

```
z = a + b + c;
myDate = VGVar.currentShortGregorianDate;
myUser = sysVar.userID;
myRecord01 = myRecord02;
myRecord02 = "USER";
```

The behavior of an EGL assignment statement is different from that of a **move** statement, which is described in *move*.

The assignment rules are described in *Assignment compatibility in EGL*.

Related concepts
"Syntax diagram for EGL statements and commands" on page 733

Related reference
"Assignment compatibility in EGL" on page 347
"move" on page 592
"Substrings" on page 731

Association elements

As described in *Resource associations*, the resource associations part is composed of association elements. Each element is specific to a file name (property "fileName" on page 353) and contains a set of entries, each with these properties:

- "system" on page 354
- "fileType" on page 353

The values of the **system** and **fileType** properties determine what additional properties are available to you from the following list:

- "commit" on page 353
- "conversionTable" on page 353
- "formFeedOnClose" on page 354
- "duplicates" on page 353
- "replace" on page 354
- "systemName" on page 355
- "text" on page 355

commit

Indicates (for an EGL-generated Java program on iSeries) whether to enable commitment control.

Select one of these values:

NO (the default)
Use of sysLib.commit or sysLib.rollback has no effect.

YES
You can use sysLib.commit and sysLib.rollback to define the end of a logical unit of work.

conversionTable

Specifies the name of the conversion table used by a generated Java program during access of an MQSeries message queue.

For additional information, see *Data conversion*.

duplicates

Specifies (for an EGL-generated COBOL program on iSeries) whether an accessed VSAM file is allowed to contain duplicate keys. Valid values are NO (the default) and YES.

The value of **duplicates** must be consistent with the use of the keyword UNIQUE in the data description specification (DDS) that describes the physical file on iSeries. If the value of **duplicates** is YES, for example, you must not specify UNIQUE.

The next table shows the consequence of an inconsistency in the two values.

DDS keyword	Value of duplicates in the association element	COBOL return code after a file open	EGL return code after a file open	EGL I/O error value
UNIQUE	YES	95	00000220	format
no UNIQUE	NO	95	00000220	format

fileType

Specifies the file organization on the target system. You can select an explicit type like *seqws*. Alternatively, you can select the value *default*, which is itself the default value of the property **fileType**. Use of the default means that a file type is selected automatically:

- For a particular combination of target system and EGL record type; or
- For print output, when the file name is *printer*.

Record and file type cross-reference shows the explicit **fileType** values, as well as the value used if you select *default*.

fileName

Refers to a logical file name, as specified in one or more records. You are creating an association element that relates this name to a physical resource on one or more target systems. (For print output, specify the value *printer*.)

You can use an asterisk (*) as a global substitution character in a logical file name; however, that character is valid only as the last character. For details, see *Resource associations and file types*.

formFeedOnClose

Indicates whether a form feed is issued when the output of a print form ends. (A print form is produced when your code issues a **print** statement.)

This property is available only if the **fileName** value is *printer* in one of the following cases:

- The **system** value is *aix*, *iSeriesj*, or *linux*, and the **fileType** value is *seqws* or *spool*; or
- The **system** value is *win*, and the **fileType** value is *seqws*.

Select one of these values:

YES
A form feed occurs (the default)

NO
A form feed does not occur

replace

Specifies whether adding a record to the file replaces the file rather than appending to the file. This entry is used only in these cases:

- You are generating Java code; and
- The record is of file type **seqws**.

Select one of these values:

NO
Append to the file (the default)

YES
Replace the file

system

Specifies the target platform. Select one of the following values:

aix
AIX

iseriesj
iSeries

linux
Linux

win
Windows 2000/NT/XP

any
Any target platform; for details, see *Resource associations and file types*

systemName

Specifies the system resource name of the file or data set associated with the file name. Enclose the value in single or double quote marks if a space or any of the following characters is in the value:

```
% = , ( ) /
```

text

Specifies whether to cause a generated Java program to do the following when accessing a file by way of a serial record:

- Append end-of-line characters during the **add** operation. On non-UNIX platforms, those characters are a carriage return and linefeed; on UNIX platforms, the only character is a linefeed.
- Remove end-of-line characters during the **get** or **get next** operation.

Select one of these values:

NO
The default is not to append or remove the end-of-line characters

YES
Make the changes, as is useful if the generated program is exchanging data with products that expect records to end with the end-of-line characters

Related concepts
"Resource associations and file types" on page 286

Related task
"Adding a resource associations part to an EGL build file" on page 289
"Editing a resource associations part in an EGL build file" on page 290
"Removing a resource associations part from an EGL build file" on page 291

Related reference
"Data conversion" on page 454
"I/O error values" on page 522
"Record and file type cross-reference" on page 716

asynchLink element

An *asynchLink* element of a linkage options part specifies how a generated Java or COBOL program invokes another program asynchronously, as occurs when the originating program invokes the system function sysLib.startTransaction.

You can avoid specifying an asynchLink element if you accept the default behavior, which assumes that the created transaction is to be started from the same Java package.

Each element includes the property recordName, which references a record that is also referenced in the specific **sysLib.startTransaction** function whose action is being modified.

For Java programs, the other property is **package**, which is needed only if the source for the invoked program is in a package that is different from the invoker's package.

Related concepts
"Linkage options part" on page 291

Related reference

"package in asynchLink element"
"recordName in asynchLink element"

package in asynchLink element

The linkage options part, asynchLink element, property **package** is valid only for Java output and specifies the name of the package that contains the program being invoked. The default is the package of the invoking program.

The package name that is used in generated Java programs is the package name of the EGL program, but in lower case; and when EGL generates output from the asynchLink element, the value of **package** is changed (if necessary) to lower case.

Related concepts
"Linkage options part" on page 291

Related reference
"asynchLink element" on page 355

"recordName in asynchLink element"

recordName in asynchLink element

The linkage options part, asynchLink element, property **recordName** specifies the name of the record that is used in the system function sysLib.startTransaction. In this case, the record name is used to identify which program or transaction is associated with the asynchLink element.

You can use an asterisk (*) as a global substitution character in the record name; however, that character is valid only as the last character. For details, see *Linkage options part*.

Related concepts
"Linkage options part" on page 291

Related reference
"asynchLink element" on page 355

"package in asynchLink element"
"startTransaction()" on page 883

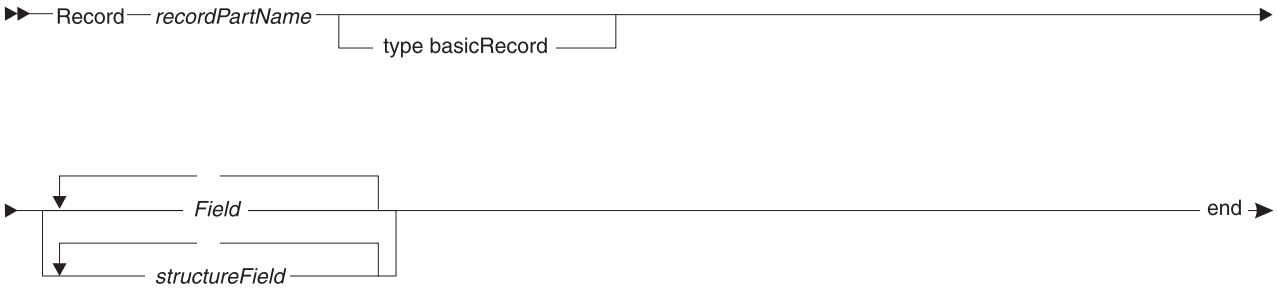
Basic record part in EGL source format

You declare a record part of type basicRecord in an EGL file, which is described in *EGL source format*.

An example of a basic record part is as follows:

```
Record myBasicRecordPart type basicRecord
  10 myField01 CHAR(2);
  10 myField02 CHAR(78);
end
```

The syntax diagram for a basic record is as follows:



Record *recordPartName* **basicRecord**
Identifies the part as being of type basicRecord and specifies the name. For rules, see *Naming conventions*.

field
A variable appropriate in a record, as described in *Record parts*. End each variable declaration with a semicolon.

structureField
A fixed-structure field, as described in *Structure field in EGL source format*.

Related concepts
"EGL projects, packages, and files" on page 13
"Fixed record parts" on page 125
"References to parts" on page 20
"Parts" on page 17
"Record parts" on page 124
"References to variables in EGL" on page 55
"Typedef" on page 25

Related tasks
"Syntax diagram for EGL statements and commands" on page 733

Related reference

"DataItem part in EGL source format" on page 461
"EGL source format" on page 478
"Function part in EGL source format" on page 513
"Indexed record part in EGL source format" on page 520
"MQ record part in EGL source format" on page 642
"Naming conventions" on page 652
"Primitive types" on page 31
"Program part in EGL source format" on page 707

Build parts

EGL build-file format

The structure of a .eglbld file is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE EGL PUBLIC "-//IBM//DTD EGL 5.1//EN" "">
<EGL>
  <!-- place your import statements here -->
  <!-- place your parts here -->
</EGL>
```

Your task is to place import statements and parts inside the <EGL> element.

You specify <import> elements to reference the file containing the next build descriptor in a chain or to reference any of the build parts referenced by a build descriptor. An example of an import statement is as follows:

```
<import file="myBldFile.eglbld"/>
```

You declare parts from this list:

- <BuildDescriptor>
- <LinkageOptions>
- <ResourceAssociations>

A simple example is as follows:

```
<EGL>
  <import file="myBldFile.eglbld"/>
  <BuildDescriptor name="myBuildDescriptor"
    genProject="myNextProject"
    system="WIN"
    J2EE="NO"
    genProperties="GLOBAL"
    genDataTables="YES"
    dbms="DB2"
    sqlValidationConnectionURL="jdbc:db2:SAMPLE"
    sqlJDBCDriverClass="COM.ibm.db2.jdbc.app.DB2Driver"
    sqlDB="jdbc:db2:SAMPLE"
  </BuildDescriptor>
</EGL>
```

You can review the build-file DTD, which is in the following subdirectory:

installationDir\egl\eclipse\plugins\
com.ibm.etools.egl_*version*\dtd

installationDir
The product installation directory, such as C:\Program Files\IBM\RSPD\6.0. If you installed and kept a Rational Developer product before installing the product that you are using now, you may need to specify the directory that was used in the earlier install.

version
The installed version of the plugin; for example, 6.0.0

The file name (like egl_wssd_6_0.dtd) begins with the letters *egl* and an underscore. The characters *wssd* refer to Rational Web Developer and Rational Application Developer; the characters *wsed* refer to Rational Application Developer for z/OS; and the characters *wdsc* refer to Rational Application Developer for iSeries.

Related concepts
"Import" on page 30
"Parts" on page 17

Related tasks
"Creating an EGL source file" on page 120

Related reference
"EGL editor" on page 471

Build descriptor options

The next table lists all the build descriptor options.

Build descriptor option	Build option filter(s)	Description
bidiConversionTable	• iSeriesc	Identifies a conversion table, but only when you generate a COBOL program that contains literals with Arabic or Hebrew characters
buildPlan	• Java target • iSeriesc	Specifies whether a build plan is created
checkNumericOverflow	• iSeriesc	Specifies whether to support numeric overflow checking in a generated COBOL program
checkType	• iSeriesc	Specifies the degree to which EGL checks at validation time for primitive-type conflicts within structures and records
cicsj2cTimeout	• Debug • Java target • Java iSeries	Assigns a value to the Java run-time property **cso.cicsj2c.timeout**, which specifies the number of milliseconds before a timeout occurs during a call that uses protocol CICSJ2C
clientCodeSet	• iSeriesc	Specifies the coded character set name used when you generate COBOL source code on the workstation
commentLevel	• Java target • Java iSeries • iSeriesc	Specifies the extent to which EGL system comments are included in output source code
currencySymbol	• Debug • Java target	Specifies a currency symbol that is composed of one to three characters

Build descriptor option	Build option filter(s)	Description
dbms	• Debug • Java target • Java iSeries	Specifies the type of database accessed by the generated program
debugTrace	• iSeriesc	Indicates whether EGL puts trace information at the end of a generated COBOL source file
decimalSymbol	• Debug • Java target • Java iSeries	Assigns a character to the Java run-time property **vgj.nls.number. decimal**, which indicates what character is used as a decimal symbol
destDirectory	• Java target	Specifies the name of the directory that stores the output of preparation, but only when you generate Java
destHost	• Java target • iSeriesc	Specifies the name or numeric TCP/IP address of the target machine where the build server resides
destLibrary	• iSeriesc	Specifies the 1- to 10-character name of the iSeries library that receives the objects created during generation and contains the objects used at run time
destPassword	• Java target • iSeriesc	Specifies the password that EGL uses to log on to the machine where preparation occurs
destPort	• Java target • iSeriesc	Specifies the port on which a remote build server is listening for build requests
destUserID	• Java target • iSeriesc	Specifies the user ID that EGL uses to log on to the machine where preparation occurs
eliminateSystemDependentCode	• Java target • Java iSeries • iSeriesc	Indicates whether, at validation time, EGL ignores code that will never run in the target system.
enableJavaWrapperGen	• Java target • Java iSeries • iSeriesc	Specifies whether to allow generation of Java wrapper classes
fillWithNulls	• iSeriesc	Indicates whether to fill print-form fields with null characters
genDataTables	• Java target • Java iSeries • iSeriesc	Indicates whether you want to generate data tables

Build descriptor option	Build option filter(s)	Description
genDDSFile	• iSeriesc	Indicates whether you want to create iSeries data description specification (DDS) files from the record declarations with which your program does input or output.
genDirectory	• Java target • iSeriesc	Specifies the fully qualified path of the directory into which EGL places generated output and preparation-status files
genFormGroup	• Java target • iSeriesc	Indicates whether you want to generate the form group that is referenced in the use declaration of the program you are generating
genHelpFormGroup	• Java target • iSeriesc	Indicates whether you want to generate the help form group that is referenced in the use declaration of the program you are generating.
genProject	• Java target • Java iSeries • iSeriesc	Places the output of Java generation into a workbench project and automates tasks that are required for Java run-time setup
genProperties	• Java target • Java iSeries	Specifies what kind of Java run-time properties to generate (if any) and, in some cases, whether to generate a linkage properties file
initIORecords	• iSeriesc	Specifies whether the generated COBOL program initializes global records other than basic records
initNonIOData	• iSeriesc	Specifies whether the generated COBOL program initializes global, basic records
J2EE	• Debug • Java target • Java iSeries	Specifies whether a Java program is generated to run in a J2EE environment
leftAlign	• iSeriesc	Indicates whether to left-justify the output data on print-form fields
linkage	• Debug • Java target • Java iSeries • iSeriesc	Contains the name of the linkage options part that guides aspects of generation

Build descriptor option	Build option filter(s)	Description
math	• iSeriesc	Specifies whether to do arithmetic calculations based on *CSP math*, which is used in some COBOL programs that were written either with IBM Cross System Product (CSP) or with VisualAge Generator
nextBuildDescriptor (see Build descriptor part)	• Debug • Java target • Java iSeries • iSeriesc	Identifies the next build descriptor in chain
oneFormItemCopybook	• iSeriesc	Allows your program to pass and receive text forms that are formatted as VisualAge Generator maps
positiveSignIndicator	• iSeriesc	Specifies the character that the iSeries-based ILE COBOL compiler uses as the positive sign for numeric data of types DECIMAL, NUM, NUMC, and PACF
prep	• Java target • iSeriesc	Specifies whether EGL begins preparation when generation completes with a return code <= 4
reservedWord	• iSeriesc	Specifies a fully qualified path name for a text file that contains reserved words other than the EGL reserved words
resourceAssociations	• Debug • Java target • Java iSeries • iSeriesc	Contains the name of a resource associations part, which relates record parts to files and queues on the target platforms
serverCodeSet	• iSeriesc	Specifies the name of the coded character set used by the z/OS build server
sessionBeanID	• Java target • Java iSeries	Identifies the name of an existing session element in the J2EE deployment descriptor
setFormItemFull	• iSeriesc	Indicates whether to display asterisks (*) in every empty print field for which you specified *set field full*
spacesZero	• iSeriesc	Specifies whether a generated COBOL program includes extra code to process numeric items that are filled with spaces

Build descriptor option	Build option filter(s)	Description
sqlDB	• Debug • Java target • Java iSeries • iSeriesc	Specifies the default database used by a generated program
sqlErrorTrace	• iSeriesc	Specifies whether a generated COBOL program includes the code necessary to trace errors that occur during I/O operations against a relational database
sqlID	• Debug • Java target • Java iSeries • iSeriesc	Specifies a user ID that is used to connect to a database during generation-time validation of SQL statements or at run time
sqlIOTrace	• iSeriesc	Specifies whether a generated COBOL program includes the code necessary to trace the I/O operations done against a relational database
sqlJDBCDriverClass	• Debug • Java target • Java iSeries	Specifies a driver class that is used to connect to a database during generation-time validation of SQL statements or during a non-J2EE Java debugging session
sqlJNDIName	• Debug • Java target • Java iSeries	Specifies the default database used by a generated Java program that runs in J2EE
sqlPassword	• Debug • Java target • Java iSeries • iSeriesc	Specifies a password that is used to connect to a database during generation-time validation of SQL statements or at run time
sqlValidationConnectionURL	• Debug • Java target • Java iSeries • iSeriesc	Specifies a URL that is used to connect to a database during generation-time validation of SQL statements
sysCodes	• iSeriesc	Determines the source of the code that is placed in the system variable **sysVar.errorCode** in response to a file I/O error in a COBOL program
system	• Debug • Java target • Java iSeries • iSeriesc	Specifies a category of generation output

Build descriptor option	Build option filter(s)	Description
targetNLS	• Debug • Java target • Java iSeries • iSeriesc	Specifies the target national language code used for run-time output
templateDir	• iSeriesc	Specifies the directory that contains templates used to produce run-time JCL
VAGCompatibility	• Debug • Java target • Java iSeries • iSeriesc	Indicates whether the generation process allows use of special program syntax
validateMixedItems	• iSeriesc	Specifies whether a generated COBOL program validates items that are of type MIX
validateOnlyIfModified	• iSeriesc	Specifies whether to validate only those text-form fields for which the modified data tag is set
validateSQLStatements	• Java target • Java iSeries • iSeriesc	Indicates whether SQL statements are validated against a database

Related concepts
"Build descriptor part" on page 275
"Java runtime properties" on page 327

Related tasks
"Adding a build descriptor part to an EGL build file" on page 279
"Editing general options in a build descriptor" on page 280

Related reference
"Java runtime properties (details)" on page 525

bidiConversionTable

The build descriptor option **bidiConversionTable** identifies a conversion table, but only when you generate a COBOL program that contains literals with Arabic or Hebrew characters. For details, see *Bidirectional text conversion*.

Related reference
"Bidirectional language text" on page 458
"Build descriptor options" on page 359
"Data conversion" on page 454

buildPlan

The build descriptor option **buildPlan** specifies whether a build plan is created. Valid values are YES and NO, and the default is YES.

The build plan is placed in the directory identified by build descriptor option **genDirectory**.

A special case is in effect when you generate Java code into a project. Then, no build plan is created regardless of the setting of **buildPlan**, but preparation occurs in either of two situations:

- Whenever you rebuild the project
- Whenever you generate the source files; but only if you checked the workbench preference **Perform build automatically on resource modification**

You may wish to create a build plan and to invoke that plan at a later time. For details, see *Invoking a build plan after generation*.

Related concepts
"Build plan" on page 305

Related tasks
"Invoking a build plan after generation" on page 315

Related reference
"Build descriptor options" on page 359

checkNumericOverflow

The build descriptor option **checkNumericOverflow** specifies whether to support numeric overflow checking in a generated COBOL program. Valid values are YES (the default) and NO.

If you specify NO, the system function VGVar.handleOverflow is ignored. Division by zero results in an abend with a message. In other overflow conditions, the result is truncated, causing the significant digits to be lost, but the generated program gives no indication that truncation has occurred.

Setting **checkNumericOverflow** to NO may result in smaller programs with better performance.

Related reference
"Build descriptor options" on page 359
"handleOverflow" on page 921

checkType

The build descriptor option **checkType** specifies the degree to which EGL checks at validation time for primitive-type conflicts within records. You can receive an information message, for example, if a structure item that is of type CHAR is substructured with structure items of type DECIMAL. Such conflicts can cause run-time errors.

Valid values are as follows:

NONE
Specify NONE (the default) if you do not want to check for potential conflicts in the primitive types of substructured items.

LOW
Specify LOW to check for conflicting primitive types only in the items that are subordinate to the highest level of the structure. Consider the following example:

```
10 ItemA
  15 ItemB
    20 ItemC
      30 ItemD
```

If you specify LOW, EGL will not compare ItemA to ItemB, but will compare ItemB to ItemC, and ItemC to ItemD.

ALL
Specify ALL to check for conflicting primitive types in all levels of a substructured data item.

Specifying a value other than NONE increases both the time needed for validation and the number of messages issued.

Related reference
"Build descriptor options" on page 359

cicsj2cTimeout

When you are generating Java code, the build descriptor option **cicsj2cTimeout** assigns a value to the Java run-time property **cso.cicsj2c.timeout**. That property specifies the number of milliseconds before a timeout occurs during a call that uses protocol CICSJ2C.

The default value of the run-time property is 30000, which represents 30 seconds. If the value is set to 0, no timeout occurs. The value must be greater than or equal to 0.

The property **cso.cicsj2c.timeout** has no effect on calls when the called program is running in WebSphere 390; for details, see *Setting up the J2EE server for CICSJ2C calls*. Also, the build descriptor option **cicsj2cTimeout** has no effect when you are generating COBOL code.

Related concepts
"Java runtime properties" on page 327

Related tasks
"Setting up the J2EE server for CICSJ2C calls" on page 337

Related reference
"Build descriptor options" on page 359
"Java runtime properties (details)" on page 525

clientCodeSet

The build descriptor option **clientCodeSet** specifies the name of the coded character set that is in effect when you generate COBOL source on the workstation.

The coded character set identified in clientCodeSet must be the one defined to the ICONV conversion service on the machine where the build server is started. The default is IBM-850, which is a standard character set for Latin-1 countries.

Related reference
"Build descriptor options" on page 359
"serverCodeSet" on page 381

commentLevel

The build descriptor option **commentLevel** specifies the extent to which EGL system comments are included in output source code.

Valid values are as follows:

0 Minimal comments are in the output, which includes comments on any name aliases that EGL generates

1 In addition to the comments included with level 0, scripting statements are placed immediately before the code that is generated to implement those statements.

The default is 1.

Raising the comment level has no effect on the size or performance of the prepared code, but increases the size of the output and the time needed to generate, transfer, and prepare the output.

Related reference
"Build descriptor options" on page 359

currencySymbol

The build descriptor option **currencySymbol** is available only for Java output and specifies a currency symbol that is composed of one to three characters. If you do not specify this option, the default value is derived from the locale of the system on which you generate output.

To specify a character that is not on your keyboard, hold down the **Alt** key and use the numeric key pad to type the character's decimal code. The decimal code for the Euro, for example, is 0128 on Windows 2000/NT/XP.

Related concepts
"Build descriptor part" on page 275

Related reference
"Build descriptor options" on page 359

dbms

The build descriptor option **dbms** specifies the type of database accessed by the generated program. Select one of the following values:

- DB2 (the default value)
- INFORMIX
- ORACLE

Related reference
"Build descriptor options" on page 359
"Informix and EGL" on page 235

debugTrace

The build descriptor option **debugTrace** indicates whether EGL puts trace information at the end of a generated COBOL source file. The valid values are YES and NO. The default is NO.

It is recommended that you use this option only when you are providing debugging information to IBM service personnel.

Related reference
"Build descriptor options" on page 359

decimalSymbol

When you are generating Java code, the build descriptor option **decimalSymbol** assigns a character to the Java run-time property **vgj.nls.number.decimal**, which indicates what character is used as a decimal symbol. If you do not specify the build descriptor option **decimalSymbol**, the character is determined by the locale associated with the Java run-time property **vgj.nls.code**.

The build descriptor option **decimalSymbol** has no effect when you are generating COBOL code. Also, the value can be no more than one character.

Related concepts
"Java runtime properties" on page 327

Related reference
"Build descriptor options" on page 359
"Java runtime properties (details)" on page 525

destDirectory

The build descriptor option **destDirectory** specifies the directory that stores the output of preparation, but only when you generate Java. This option is meaningful only when you generate into a directory rather than into a project. A similar option for COBOL generation is **projectID**.

When you specify a fully qualified file path, all but the last directory must exist. If you specify c:\buildout on Windows 2000, for example, EGL creates the buildout directory if it does not exist. If you specify c:\interim\buildout and the interim directory does not exist, however, preparation fails.

If you specify a relative directory (such as myid/mysource on USS), the output is placed in the bottom-most directory, which is relative to the default directory, as described next.

The default value of **destDirectory** is affected by the status of build descriptor option **destHost**:

- If **destHost** is specified, the default value of **destDirectory** is the directory in which the build server was started
- If **destHost** is not specified, preparation occurs on the machine where generation occurs, and the default value of **destDirectory** is given by build descriptor option **genDirectory**

The user specified by build descriptor option **destUserID** must have the authority to write to the directory that receives the output of preparation.

You cannot use a UNIX variable ($HOME, for example) to identify part of a directory structure on USS.

Related reference
"Build descriptor options" on page 359
"destHost"
"genProject" on page 374

destHost

The build descriptor option **destHost** specifies the name or numeric TCP/IP address of the target machine where the build server resides. No default is available.

If you are preparing a generated COBOL program, the following statements apply:

- **destHost** is required
- A build server must be started on the remote machine before generation begins

If you are preparing Java output, the following statements apply:

- **destHost** is optional
- **destHost** is meaningful only if you generate into a directory rather than into a project
- If you specify **destHost** without specifying **destDirectory**, the directory in which the build server was started is the one that receives source and preparation outputs
- If you do not specify **destHost**, preparation occurs on the machine where generation occurs; and if **destDirectory** is not specified, the directory that is specified by build descriptor option **genDirectory** is the one that receives source and preparation outputs
- The UNIX environments are case sensitive

You can type up to 64 characters for the name or TCP/IP address. If you are developing on Windows NT®, you must specify a name rather than a TCP/IP address.

Two example values for **destHost** are as follows:

```
abc.def.ghi.com

9.99.999.99
```

Related reference
"Build descriptor options" on page 359
"destDirectory" on page 368
"destPassword"
"destPort" on page 370

destLibrary

Specifies the 1- to 10-character name of the iSeries library that receives the objects created during generation and contains the objects used at run time.

The default is QGPL.

Related concepts
"Build descriptor part" on page 275

Related reference
"Build descriptor options" on page 359

destPassword

The build descriptor option **destPassword** specifies the password that EGL uses to log on to the machine where preparation occurs.

This option and the description on this page are meaningful only if you are generating into a directory rather than into a project and only if you specify a value for build descriptor option **destHost**.

The password provides access for the userid specified in build descriptor option **destUserID**. The value of the password is case sensitive for all target systems.

No default is available.

Use of **destPassword** means that a password is stored in an EGL build file. You can avoid the security risk by not setting the build descriptor option. When you start generation, you can set the password in an interactive generation dialog or on the command line.

Related reference
“Build descriptor options” on page 359
“destHost” on page 368
“destUserID”

destPort

The build descriptor option **destPort** specifies the port on which a remote build server is listening for build requests.

This option is meaningful only if you are generating into a directory rather than into a project and only if you specify a value for build descriptor option **destHost**.

No default value is available.

Related reference
“Build descriptor options” on page 359
“destHost” on page 368

destUserID

The build descriptor option **destUserID** specifies the userid that EGL uses to log on to the machine where preparation occurs.

This option and the description on this page are meaningful only if you are generating into a directory rather than into a project and only if you specify a value for build descriptor option **destHost**.

The user specified by **destUserID** must have the authority to write to the directory. The option value is case sensitive for all target systems.

No default is available.

Related reference
“Build descriptor options” on page 359
“destHost” on page 368
“destPassword” on page 369

eliminateSystemDependentCode

The build descriptor option **eliminateSystemDependentCode** indicates whether, at validation time, EGL ignores code that will never run in the target system. Valid values are *yes* (the default) and *no*. Specify *no* only if the output of the current generation will run in multiple systems.

The option **eliminateSystemDependentCode** is meaningful only in relation to the system function **sysVar.systemType**. That function does not itself affect what code is validated at generation time. For example, the following **add** statement may be validated even if you are generating for Windows:

```
if (sysVar.systemType IS AIX)
  add myRecord;
end
```

To avoid validating code that will never run in the target system, take either of the following actions:

- Set the build descriptor option **eliminateSystemDependentCode** to *yes*. In the current example, the **add** statement is not validated if you set that build descriptor option to *yes*. Be aware, however, that the generator can eliminate system-dependent code only if the logical expression (in this case, `sysVar.systemType IS AIX`) is simple enough to evaluate at generation time.
- Alternatively, move the statements that you do not want to validate to a second program; then, let the original program call the new program conditionally:

```
if (sysVar.systemType IS AIX)
  call myAddProgram myRecord;
end
```

Related concepts
"Build descriptor part" on page 275

Related reference
"Build descriptor options" on page 359

enableJavaWrapperGen

When you issue the commands to generate a program, the build descriptor option **enableJavaWrapperGen** allows you to choose from three alternatives:

YES (the default)
Generate the program and allow generation of the related Java wrapper classes and (if appropriate) the related EJB session bean

ONLY
Do not generate the program, but allow generation of the related Java wrapper classes and (if appropriate) the related EJB session bean

NO
Generate the program, but not the Java wrapper classes or the related EJB session bean, if any

Actual generation of the Java wrapper classes and EJB session bean requires appropriate settings in the linkage options part that is used at generation time. For an overview, see *Java wrapper*.

Related concepts
"Java wrapper" on page 282

Related reference
"Java wrapper classes" on page 535

fillWithNulls

The build descriptor option **fillWithNulls** is used only when you generate a form group that includes print forms. The option indicates whether to fill print-form fields with null characters. The affected fields have these characteristics:

- The item property **fillCharacter** is set to an empty string; and
- The field is of one of these types: CHAR, DBCHAR, MBCHAR, or NUM.

Valid values are *yes* (the default) and *no*. If you specify *no*, the affected fields are filled with spaces.

Related concepts
"Build descriptor part" on page 275

Related reference
"Build descriptor options" on page 359

genDataTables

The build descriptor option **genDataTables** indicates whether you want to generate the data tables that are referenced in the program you are generating. The references are in the program's use declaration and in the program property **msgTablePrefix**.

Valid values are *yes* (the default) and *no*.

Set the value to *no* in the following case:

- The data tables referenced in the program were previously generated; and
- Those tables have not changed since they were last generated.

For other details, see *DataTable part*.

Related concepts
"Build descriptor part" on page 275
"DataTable" on page 137

Related reference
"Build descriptor options" on page 359
"Program part in EGL source format" on page 707
"Use declaration" on page 930

genDDSFile

The build descriptor option **genDDSFile** indicates whether you want to create iSeries data description specification (DDS) files from the record declarations with which your program does input or output. Valid values are *no* (the default) and *yes*.

If you are creating DDS files and if you accept the default value of the build descriptor option **prep**, EGL uploads the DDS files to the host system.

Related concepts
"Build descriptor part" on page 275

Related reference
"Build descriptor options" on page 359

genDirectory

The build descriptor option **genDirectory** specifies the fully qualified path of the directory into which EGL places generated output and preparation-status files.

When you are generating in the workbench or from the workbench batch interface, the following rules apply:

For Java generation
You must specify either **genProject** or **genDirectory**, but an error results if you specify both. Also, you must specify **genProject** if you generate Java code for iSeries.

For COBOL generation
You must specify **genDirectory**, and in most cases EGL ignores any setting for **genProject**.

If you are generating from the EGL SDK, the following rules apply:

- You must specify **genDirectory**
- An error results if you specify **genProject**
- You cannot generate Java code for iSeries

For details on deploying Java code, see *Processing Java code that is generated into a directory*.

Related concepts
"Generation from the EGL Software Development Kit (SDK)" on page 314
"Generation from the workbench batch interface" on page 313
"Generation in the workbench" on page 311

Related tasks
"Processing Java code that is generated into a directory" on page 315

Related reference
"Build descriptor options" on page 359
"genDirectory" on page 372
"genProject" on page 374

genFormGroup

The build descriptor option **genFormGroup** indicates whether you want to generate the form group that is referenced in the use declaration of the program you are generating. Valid values are *yes* (the default) and *no*.

The help form group, if any, is not affected by this option, but by the build descriptor option genHelpFormGroup.

Related concepts
"Build descriptor part" on page 275

Related reference
"Build descriptor options" on page 359
"genHelpFormGroup"
"Use declaration" on page 930

genHelpFormGroup

The build descriptor option **genHelpFormGroup** indicates whether you want to generate the help form group that is referenced in the use declaration of the program you are generating. Valid values are *yes* (the default) and *no*.

The main form group is not affected by this option, but by the build descriptor option genFormGroup.

Related concepts
"Build descriptor part" on page 275

Related reference
"Build descriptor options" on page 359
"genFormGroup" on page 373
"Use declaration" on page 930

genProject

The build descriptor option **genProject** places the output of Java generation into a Workbench project and automates tasks that are required for Java run-time setup. For details on that setup and on the benefits of using **genProject**, see *Generation of Java code into a project*.

To use **genProject**, specify the project name. EGL then ignores the build descriptor options **buildPlan**, **genDirectory**, and **prep**, and preparation occurs in either of two cases:

- Whenever you rebuild the project
- Whenever you generate the source files; but only if you checked the workbench preference **Perform build automatically on resource modification**

If you set the option **genProject** to the name of a project that does not exist in the workbench, EGL uses the name to create a Java project, except in these cases:

- If you are generating a PageHandler and specify a project different from the one that contains the related JSP and if that other project does not exist, EGL creates an EGL Web project. (However, it is recommended that you generate the PageHandler into the project that contains the related JSP.)
- A second exception concerns EJB processing and occurs if you are generating a Java wrapper when the linkage options part, callLink element, **type** property is ejbCall (for the call from the wrapper to the EGL-generated program). In that case, EGL uses the value of **genProject** to create an EJB project and creates a new enterprise application project (if necessary) with a name that is the same as the EJB project name plus the letters EAR.

In addition to creating a project, EGL does as follows:

- EGL creates folders in the project. The package structure begins under the top-level folder JavaSource. You may change the name JavaSource by right-clicking on the folder name and selecting **Refactor**.
- If a JRE definition is specified in the preferences page for Java (installed JREs), EGL adds the classpath variable JRE_LIB. That variable contains the path to the run-time JAR files for the JRE currently in use.

When you are generating in the Workbench or from the Workbench batch interface, the following rules apply:

For Java generation
You are not required to specify either **genProject** or **genDirectory**. If neither is specified, Java output is generated into the project that contains the EGL source file being generated.

If you are generating a PageHandler, and the project specified exists, then the project must be an EGL Web project. If you are generating a session EJB, and the project specified exists, then the project must be an EJB project.

For COBOL generation
You must specify **genDirectory**, and EGL ignores any setting for **genProject**.

If you are generating from the EGL SDK, the following rules apply:

- You must specify **genDirectory**

- An error results if you specify **genProject**
- You cannot generate Java code for iSeries

Related concepts
"Generation from the EGL Software Development Kit (SDK)" on page 314
"Generation from the workbench batch interface" on page 313
"Generation in the workbench" on page 311
"Generation of Java code into a project" on page 301

Related tasks

Related reference
"Build descriptor options" on page 359
"buildPlan" on page 364
"genDirectory" on page 372
"prep" on page 380
"type in callLink element" on page 412

genProperties

The build descriptor option **genProperties** specifies what kind of Java run-time properties to generate (if any) and, in some cases, whether to generate a linkage properties file. This build descriptor option is meaningful only when you are generating a Java program (which can use either kind of output) or a wrapper (which can use only the linkage properties file).

Valid values are as follows:

NO (the default)
EGL does not generate run-time or linkage properties.

PROGRAM
The effects are as follows:

- If you are generating a program to run outside of J2EE, EGL generates a properties file that is specific to the program being generated. The name of that file is as follows:

  pgmAlias.properties

  pgmAlias
  The name of the program at run time.
- The other effects occur whether you specify **PROGRAM** or **GLOBAL**:
  - If you are generating a program that runs in J2EE, EGL generates a J2EE environment file or into a deployment descriptor; for details, see *Understanding alternatives for setting deployment-descriptor values*.
  - If you are generating a Java wrapper or calling program, EGL may generate a linkage properties file; for details on the situation in which this file is generated, see *Linkage properties file (reference)*.

GLOBAL
The effects are as follows:

- If you are generating a program to run outside of J2EE, EGL generates a properties file that is used throughout the run unit but is not named for the initial program in the run unit. The name of that properties file is **rununit.properties**.

  This option is especially useful when the first program of a run unit does not access a file or database but calls programs that do.

When generating the caller, you can generate a properties file named for the program, and the content might include no database-related properties. When you generate the called program, you can generate **rununit.properties**, and the content would be available for both programs.

- The other effects occur whether you specify **GLOBAL** or **PROGRAM**:
  - If you are generating a program that runs in J2EE, EGL generates a J2EE environment file or into a deployment descriptor; for details, see *Understanding alternatives for setting deployment-descriptor values*.
  - If you are generating a Java wrapper or calling program, EGL may generate a linkage properties file; for details on the situation in which this file is generated, see *Linkage properties file (reference)*.

For further details, see *Java run-time properties* and *Linkage properties file*.

Related concepts
"J2EE environment file" on page 336
"Java runtime properties" on page 327
"Linkage options part" on page 291
"Linkage properties file" on page 343

Related tasks
"Setting deployment-descriptor values" on page 334

Related reference
"Build descriptor options" on page 359
"Java runtime properties (details)" on page 525

initIORecords

The build descriptor option **initIORecords** specifies whether the generated COBOL program initializes global records that are used in I/O operations. Valid values are YES and NO. The default is YES.

The option **initIORecords** is meaningful only when you are generating a COBOL program. If you specify the **initialized** property when declaring a global record, the property takes precedence over the build descriptor option. Also, this build descriptor option has no effect on records that are not used in I/O operations.

For details on initialization, see *Data initialization*.

Related reference
"Build descriptor options" on page 359
"Data initialization" on page 459
"initNonIOData"

initNonIOData

The build descriptor option **initNonIOData** specifies whether the generated COBOL program initializes global basic records. Valid values are YES and NO. The default is YES.

The option **initNonIOData** is meaningful only when you are generating a COBOL program. If you specify the **initialized** property when declaring a global basic record, the property takes precedence over the build descriptor option.

For details on initialization, see *Data initialization*.

Related reference
"Build descriptor options" on page 359
"Data initialization" on page 459
"initIORecords" on page 376

itemsNullable

The build descriptor option **itemsNullable** specifies the circumstance in which your code can set primitive fields to NULL.

Valid values are as follows:

NO
You cannot set primitive fields to NULL except in this case--

- The field is in an SQL record; and
- The SQL item property **isNullable** is set to yes.

This setting of **itemsNullable** is the default, and the behavior is consistent with previous versions of EGL.

YES
You can set to NULL any primitive field in any record other than a fixed record. The behavior is consistent with the Informix product I4GL.

The next table shows the effect of your decision.

Table 9. Effect of **itemsNullable**

Operation	ItemsNullable = NO	ItemsNullable = YES
Assign a null field to another field	The value of the source is 0 or blank, and the assignment copies both a value and (if the target is nullable) the NULL state.	If the target is nullable, the target is set to NULL. Otherwise, the target is set to 0 or blank.
Use a null field in a numeric expression	The field is treated as if it contained a 0	The expression evaluates to NULL
Use a null field in a text expression	The field is treated as if it contained a space	The field is treated as if it were an empty string
Use a null field in a logical expression	The expression is treated as if the value of the field were 0 or blank, with the next example evaluating to TRUE: `0 == null`	The expression evaluates to TRUE only if null is compared with null, as is not the case in the next example, which evaluates to FALSE: `0 == null`
SET field empty	Null state is not set	Null state is set
SET record empty	Null state is not set	Null state is set

Related reference
"Build descriptor options" on page 359

J2EE

The build descriptor option **J2EE** specifies whether a Java program is generated to run in a J2EE environment. Valid values are as follows:

NO (the default)
Generates a program that will not run in a J2EE environment. The program connects to databases directly, and the environment is defined by a properties file.

YES
Generates a program to run in a J2EE environment. The program connects to databases using a data source, and the environment is defined by a deployment descriptor.

When you generate a PageHandler, J2EE is always set to YES regardless of what is specified in this option.

Related concepts
"EGL debugger" on page 261

Related reference
"Build descriptor options" on page 359

leftAlign

The build descriptor option **leftAlign** is used only when you generate a form group that includes print forms. The option indicates whether to left-justify the output data on print-form fields that have the following characteristics:

- The item property **align** is set to *left*; and
- The field is of one of these types: CHAR, DBCHAR, or MBCHAR.

Valid values are *yes* (the default) and *no*. If you do not need left alignment during output, specify *no* to give better performance and to reduce the code size.

Left alignment strips leading spaces and places them at the end of the field.

Related concepts
"Build descriptor part" on page 275

Related reference
"Build descriptor options" on page 359

linkage

The build descriptor option **linkage** contains the name of the linkage options part that guides aspects of generation. This option is not required for generation, and no default value is available.

Related concepts
"Linkage options part" on page 291

Related reference
"Build descriptor options" on page 359
"callLink element" on page 395

math

The build descriptor option **math** specifies whether to do arithmetic calculations based on *CSP math*, which is used in some COBOL programs that were written either with IBM Cross System Product (CSP) or with VisualAge Generator. You might choose CSP math if your EGL-generated program interacts with an older application. Otherwise, accept the default, which is COBOL.

This option is meaningful only if you are generating a COBOL program.

Valid values are as follows:

COBOL
Use COBOL truncation algorithms, which may provide faster performance, smaller load module size, and better accuracy.

CSPAE
Truncate intermediate arithemetic values to a number of significant digits. The number is equal to the number of significant digits for the memory area that holds the final result.

Related reference
"Build descriptor options" on page 359

nextBuildDescriptor

The build descriptor option **nextBuildDescriptor** identifies the next build descriptor in chain, if any. For details, see *Build descriptor part*.

Related concepts
"Build descriptor part" on page 275

Related reference
"Build descriptor options" on page 359

oneFormItemCopybook

The build descriptor option **oneFormItemCopybook** indicates how EGL-generated COBOL code accesses the values of form-item properties. Values are as follows:

no (the default)
EGL generates a COBOL copybook into the definition of each form item, in the Data Section of the COBOL program. Access is direct, not requiring use of COBOL SET statements.

yes
EGL places a single copybook in the Linkage Section, and access is by COBOL SET statements.

If possible, accept the default value, which maximizes performance. If your program uses many forms or if the forms contain many items, however, EGL generates a large number of COBOL variable names, and the COBOL compiler symbol table can become so large that compilation fails.

If you need to avoid the compilation problem just described, set **oneFormItemCopybook** to *yes*; then, the EGL-generated code will invoke a COBOL SET statement whenever a form-item property value is accessed.

Related concepts
"Build descriptor part" on page 275

Related reference
"Build descriptor options" on page 359

positiveSignIndicator

The build descriptor option **positiveSignIndicator** specifies the character that the iSeries-based ILE COBOL compiler uses as the positive sign for numeric data of types DECIMAL, NUM, NUMC, and PACF. You can specify the value *F* or *C*.

The default value is *F*. If your code includes more occurrences of items that are of type NUMC and DECIMAL (as compared to items of type NUM and PACKF), you can improve performance by setting **positiveSignIndicator** to *C*.

Related concepts
"Build descriptor part" on page 275

Related reference
"Build descriptor options" on page 359
"Primitive types" on page 31

prep

The build descriptor option **prep** specifies whether EGL begins preparation when generation completes with a return code <= 4. Valid values are YES and NO, and the default is YES.

Even if you set **prep** to NO, you can prepare code later. For details, see *Invoking a build plan after generation*.

Consider these cases:

- When you generate a COBOL program, EGL writes preparation messages to the directory specified in build descriptor option **genDirectory**, to the results file, and to additional files that are each specific to a preparation step
- When you generate Java code into a directory, EGL writes preparation messages to the directory specified in build descriptor option **genDirectory**, to the results file
- When you generate Java code into a project (option **genProject**), the option **prep** has no effect, and preparation occurs in either of two situations:
  - Whenever you rebuild the project
  - Whenever you generate the source files; but only if you checked the workbench preference **Perform build automatically on resource modification**

If you wish to customize the generated build plan, do as follows:

- Set option **prep** to NO
- Set option **buildPlan** to YES (as is the default)
- Generate the output
- Customize the build plan
- Invoke the build plan, as described in *buildPlan*

Related concepts
"Results file" on page 309

Related tasks
"Invoking a build plan after generation" on page 315

Related reference
"Build descriptor options" on page 359
"buildPlan" on page 364

"Generated output (reference)" on page 516
"genDirectory" on page 372
"genProject" on page 374

reservedWord

The build descriptor option **reservedWord** specifies a fully qualified path name for a text file that contains reserved words other than the EGL reserved words.

This option has no default and is meaningful only when you are generating a COBOL program. For details, see *COBOL reserved-word file*.

Related concepts
"COBOL reserved-word file" on page 426

Related reference
"Build descriptor options" on page 359
"EGL reserved words" on page 474
"Format of COBOL reserved-word file" on page 427

resourceAssociations

The build descriptor option **resourceAssociations** contains the name of a resource associations part, which relates record parts to files and queues on the target platforms. This option is not required for generation, and no default value is available.

Related concepts
"Resource associations and file types" on page 286

Related tasks
"Adding a resource associations part to an EGL build file" on page 289

Related reference
"Build descriptor options" on page 359
"Association elements" on page 352
"Record and file type cross-reference" on page 716

serverCodeSet

The build descriptor option **serverCodeSet** specifies the name of the coded character set that is used by the iSeries build server. This option (along with the build descriptor option **clientCodeSet**) helps to cause a particular data conversion to occur when file content, file-path information, and environment variables are transferred from the workstation to the build server.

The coded character set specified for **serverCodeSet** must be the one defined to the ICONV conversion service on the machine where the build server is started. The default is IBM-037, which is a character set that is used for English (U.S.).

Related reference
"Build descriptor options" on page 359
"clientCodeSet" on page 366

sessionBeanID

The build descriptor option **sessionBeanID** identifies the name of an existing session element in the J2EE deployment descriptor. The environment entries are placed into the session element when you act as follows:

- Generate a program for a Java platform (by setting **system** to AIX, WIN, or USS)

- Generate into an EJB project (by setting **genProject** to an EJB project)
- Request that environment properties be generated (by setting **genProperties** to GLOBAL or PROGRAM)

The option **sessionBeanID** is useful in the following case:

1. You generate a Java wrapper, along with an EJB session bean. In the EJB project deployment descriptor (file ejb-jar.xml), EGL creates a session element, without environment entries.

   Both the EJB session bean and the session element are named as follows:

   *Programname*EJBBean

   Programname is the name of the run-time program that receives data by way of the EJB session bean. The first letter in the name is uppercase, the other letters are lowercase.

   In this example, the name of the program is ProgramA, and the name of the session element and the EJB session bean is ProgramaEJBBean.
2. After you generate the EJB session bean, you generate the Java program itself. Because the build descriptor option **genProperties** is set to YES, EGL generates J2EE environment entries into the deployment descriptor, into the session element established in step 1.
3. You generate ProgramB, which is a Java program that is used as a helper class for ProgramA. The values of **system** and **genProject** are the same as those used in step 2; also, you generate environment entries and set **sessionBeanID** to the name of the session element.

   Your use of **sessionBeanID** causes EGL to place the environment entries for the second program into the session element that was created in step 2; specifically, into the session element ProgramaEJBBean.

In the portion of the deployment descriptor that follows, EGL created the environment entries **vgj.nls.code** and **vgj.nls.number.decimal** during step 2, when ProgramA was generated; but the entry **vgj.jdbc.default.database** is used only by ProgramB and was created during step 3:

```
<ejb-jar id="ejb-jar_ID">
  <display-name>EJBTest</display-name>
  <enterprise-beans>
    <session id="ProgramaEJBBean">
      <ejb-name>ProgramaEJBBean</ejb-name>
      <home>test.ProgramaEJBHome</home>
      <remote>test.ProgramaEJB</remote>
      <ejb-class>test.ProgramaEJBBean</ejb-class>
      <session-type>Stateful</session-type>
      <transaction-type>Container</transaction-type
      <env-entry>
       <env-entry-name>vgj.nls.code</env-entry-name>
       <env-entry-type>java.lang.String</env-entry-type>
       <env-entry-value>ENU</env-entry-value>
      </env-entry>
      <env-entry>
       <env-entry-name>vgj.nls.number.decimal</env-entry-name>
       <env-entry-type>java.lang.String</env-entry-type>
       <env-entry-value>.</env-entry-value>
      </env-entry>
      <env-entry>
       <env-entry-name>vgj.jdbc.default.database</env-entry-name>
       <env-entry-type>java.lang.String</env-entry-type>
       <env-entry-value>jdbc/Sample</env-entry-value>
```

```
      </env-entry>
   </session>
  </enterprise-beans>
</ejb-jar>
```

A session element must be in the deployment descriptor before you can add environment entries. Because session elements are created during Java wrapper generation, it is recommended that you generate the Java wrapper before generating the related programs.

In the following cases, you generate a program into an EJB project, but the environment entries are placed into a J2EE environment file rather than into the deployment descriptor:

- **sessionBeanID** is set, but the session element that matches the value of **sessionBeanID** is not found in the deployment descriptor; or
- **sessionBeanID** is not set, and the session element that is named for the program is not found in the deployment descriptor. This case occurs when the program is generated before the wrapper.

For EJB projects, an environment entry name (like **vgj.nls.code**) can appear only once for each session element. If an environment entry already exists, EGL updates the entry type and value instead of creating a new entry.

EGL never deletes an environment entry from a deployment descriptor.

No default value is available for **sessionBeanID**.

Related reference
"Build descriptor options" on page 359

setFormItemFull

The build descriptor option **setFormItemFull** is used only when you generate a form group that includes print forms. The option indicates whether to display asterisks (*) in every empty field for which you specified a **set** statement of the form *set field full*. Valid values are *yes* (the default) and *no*.

If you specify *no*, the **set** statement of format *set field full* is ignored for the print forms, which may result in better performance and smaller load module size, especially if the form group contains print forms with many variables. Specify *no* if the programs that use the form group do not use a **set** statement of format *set field full*.

spacesZero

The build descriptor option **spacesZero** specifies whether a generated COBOL program includes extra code to process numeric items that are filled with spaces. The specific situation concerns items that have the following characteristics:

- Were declared in EGL with primitive type NUM or NUMC
- May receive spaces, as when the item is subordinate to a structure item of type CHAR

Valid values are as follows:

NO
Do not include the extra code necessary to interpret the spaces as zeros. NO is

appropriate if you are sure that no item in the program will ever receive spaces. NO is the default because the lack of code is more efficient at run time.

YES
Include extra code to avoid an abend when a numeric item that contains spaces is processed in a program or function script.

The option **spacesZero** has no effect on items that receive a combination of spaces and other characters.

Related reference
"Build descriptor options" on page 359
"CHAR" on page 36
"NUM" on page 48
"NUMC" on page 49

sqlCommitControl

The build descriptor option **sqlCommitControl** allows generation of a Java runtime property that specifies whether a commit occurs after every change to the default database.

The property (vgj.jdbc.default.database.autoCommit) is generated only if the build descriptor option **genProperties** is also set to PROGRAM or GLOBAL. You can set the Java runtime property at deployment time regardless of your decision at generation time.

Valid values of **sqlCommitControl** are as follows:

NOAUTOCOMMIT
The commit is not automatic; the behavior is consistent with previous versions of EGL; and the Java runtime property is set to false, as is the default.

For details on the rules of commit and rollback in this case, see *Logical unit of work*.

AUTOCOMMIT
The commit is automatic; the behavior is consistent with previous versions of the Informix product I4GL; and the Java runtime property is set to true.

The option **sqlCommitControl** is ignored when you generate output in COBOL.

Related concepts
"Build descriptor options" on page 359
"Java runtime properties" on page 327

Related reference
"Build descriptor options" on page 359
"Default database" on page 234
"genProperties" on page 375

sqlDB

The build descriptor option **sqlDB** specifies the default database used by a generated Java program that runs outside of J2EE. The value is a connection URL; for example, jdbc:db2:MyDB.

The option **sqlDB** is case-sensitive, has no default value, and is used only when you are generating a non-J2EE Java program. The option assigns a value to the Java run-time property **vgj.jdbc.default.database**, but only if option **genProperties** is set to GLOBAL or PROGRAM.

To specify the database used for validation (in relation to Java or COBOL generation), set **sqlValidationConnectionURL**.

Related concepts
"Java runtime properties" on page 327
"SQL support" on page 213

Related reference
"Build descriptor options" on page 359
"genProperties" on page 375
"Java runtime properties (details)" on page 525
"sqlPassword" on page 387
"sqlValidationConnectionURL" on page 387
"sqlJDBCDriverClass" on page 386
"validateSQLStatements" on page 391

sqlErrorTrace

The build descriptor option **sqlErrorTrace** specifies whether a generated COBOL program includes the code necessary to trace errors that occur during I/O operations against a relational database. The valid values are YES and NO. The default is NO.

This option is intended for use by support personnel and should be used only when a trace is requested as part of a support effort.

Related reference
"Build descriptor options" on page 359
"sqlIOTrace" on page 386

sqlID

The build descriptor option **sqlID** specifies a userid that is used to connect to a database during generation-time validation of SQL statements. You specify the database by setting **sqlValidationConnectionURL**.

When you generate a Java program, EGL also assigns the value of **sqlID** to the Java run-time property **vgj.jdbc.default.userid**. That property identifies the userid for connecting to the default database at run time, and you can specify the default database in **sqlDB**.

The option **sqlID** is case-sensitive and has no default value.

Related reference
"Build descriptor options" on page 359
"Java runtime properties (details)" on page 525
"sqlDB" on page 384
"sqlPassword" on page 387
"sqlValidationConnectionURL" on page 387
"sqlJDBCDriverClass" on page 386
"validateSQLStatements" on page 391

sqlIOTrace

The build descriptor option **sqlIOTrace** specifies whether a generated COBOL program includes the code necessary to trace the I/O operations done against a relational database. The valid values are YES and NO. The default is NO.

This option is intended for use by support personnel and should be used only when a trace is requested as part of a support effort.

Related reference
"Build descriptor options" on page 359

sqlJDBCDriverClass

The build descriptor option **sqlJDBCDriverClass** specifies a driver class for connecting to the database that EGL uses to validate SQL statements at generation time. You specify the database by setting **sqlValidationConnectionURL**. Database access is through JDBC.

In the following cases EGL also assigns the value of **sqlJDBCDriverClass** to the Java run-time property **vgj.jdbc.drivers** in the program properties file:

- **genProperties** is set to GLOBAL or PROGRAM
- **J2EE** is set to NO

No default is available for the driver class, and the format varies by driver:

- For IBM DB2 APP DRIVER for Windows, the driver class is as follows--
  ```
  COM.ibm.db2.jdbc.app.DB2Driver
  ```
- For IBM DB2 NET DRIVER for Windows, the driver class is as follows--
  ```
  COM.ibm.db2.jdbc.net.DB2Driver
  ```
- For IBM DB2 UNIVERAL DRIVER for Windows, the driver class is as follows (with com in lower case)--
  ```
  com.ibm.db2.jcc.DB2Driver
  ```
- For the Oracle JDBC thin client-side driver, the driver class is as follows--
  ```
  oracle.jdbc.driver.OracleDriver
  ```
- For the IBM Informix JDBC driver, the driver class is as follows--
  ```
  com.informix.jdbc.IfxDriver
  ```

For other driver classes, refer to the documentation for the driver.

To specify more than one driver class, separate each class name from the next with a colon (:). You might do this if one Java program makes a local call to another but accesses a different database management system.

Related reference
"Build descriptor options" on page 359
"Informix and EGL" on page 235
"sqlDB" on page 384
"sqlID" on page 385
"sqlPassword" on page 387
"sqlValidationConnectionURL" on page 387
"validateSQLStatements" on page 391

sqlJNDIName

The build descriptor option **sqlJNDIName** specifies the default database used by a generated Java program that runs in J2EE. The value is the name to which the default datasource is bound in the JNDI registry; for example, jdbc/MyDB.

The option **sqlJNDIName** is case-sensitive, has no default value, and is used only when you are generating a Java program for J2EE. The option assigns a value to the Java run-time property **vgj.jdbc.default.database**, but only if option **genProperties** is set to GLOBAL or PROGRAM.

To specify the database used for validation (in relation to Java or COBOL generation), set **sqlValidationConnectionURL**.

Related concepts
"Java runtime properties" on page 327
"SQL support" on page 213

Related reference
"Build descriptor options" on page 359
"genProperties" on page 375
"Java runtime properties (details)" on page 525
"sqlPassword"
"sqlValidationConnectionURL"
"sqlJDBCDriverClass" on page 386
"validateSQLStatements" on page 391

sqlPassword

The build descriptor option **sqlPassword** specifies a password that is used to connect to a database during generation-time validation of SQL statements. You specify the database by setting **sqlValidationConnectionURL**.

When you generate a Java program, EGL also assigns the value of **sqlPassword** to the Java run-time property **vgj.jdbc.default.password**. That property identifies the password for connecting to the default database at run time, and you can specify the default database in **sqlDB**.

The option **sqlPassword** is case-sensitive and has no default value.

Related concepts
"Java runtime properties" on page 327

Related reference
"Build descriptor options" on page 359
"Java runtime properties (details)" on page 525
"sqlDB" on page 384
"sqlID" on page 385
"sqlValidationConnectionURL"
"sqlJDBCDriverClass" on page 386
"validateSQLStatements" on page 391

sqlValidationConnectionURL

The build descriptor option **sqlValidationConnectionURL** specifies a URL for connecting to the database that EGL uses to validate SQL statements at generation time. Database access is through JDBC.

No default is available for the URL, and the format varies by driver:

- For IBM DB2 APP DRIVER for Windows, the URL is as follows--

  ```
  jdbc:db2:dbName
  ```

  dbName
  Database name
- For the Oracle JDBC thin client-side driver, the URL varies by database location. If the database is local to your machine, the URL is as follows--

  ```
  jdbc:oracle:thin:dbName
  ```

  If the database is on a remote server, the URL is as follows--

  ```
  jdbc:oracle:thin:@host:port:dbName
  ```

  host
  Host name of the database server

  port
  Port number

  dbName
  Database name
- For the IBM Informix JDBC driver, the URL is as follows (with the lines combined into one)--

  ```
  jdbc:informix-sqli://host:port
    /dbName:informixserver=servername;
    user=userName;password=passWord
  ```

  host
  Name of the machine on which the database server resides

  port
  Port number

  dbName
  Database name

  serverName
  Name of the database server

  userName
  Informix user ID

  passWord
  Password associated with the user ID
- For other drivers, refer to the documentation for the driver.

Related reference
"Build descriptor options" on page 359
"Informix and EGL" on page 235
"sqlDB" on page 384
"sqlID" on page 385
"sqlPassword" on page 387
"sqlJDBCDriverClass" on page 386
"validateSQLStatements" on page 391

sysCodes

The build descriptor option **sysCodes** determines the source of the code that is placed in the system word **sysVar.errorCode** in response to a file I/O error. Values are as follows:

NO
sysVar.errorCode receives codes that are returned from EGL run-time services. NO is the default.

YES
sysVar.errorCode receives code that are returned from the operating system.

The code is specific to the type of resource being accessed (VSAM rather than a transient data queue, for example). For details on the meaning of specific error codes, see the reference material provided for the operating system or for the subsystem (like VSAM).

For additional details, including specific error values, see *sysVar.errorCode*.

Related reference
"Build descriptor options" on page 359
"Exception handling" on page 89
"errorCode" on page 903

system

The build descriptor option **system** specifies the target platform for generation. This option is required; no default value is available. Valid values are as follows:

AIX
Indicates that generation produces a Java program that can run on AIX

ISERIESC
Indicates that generation produces a COBOL program that can run on iSeries

ISERIESJ
Indicates that generation produces a Java program that can run on iSeries

LINUX
Indicates that generation produces a Java program that can run on Linux (with an Intel processor)

USS
Indicates that generation produces a Java program that can run on z/OS UNIX System Services

WIN
Indicates that generation produces a Java program that can run on Windows 2000/NT/XP

Related concepts
"Generated output" on page 515
"Linkage options part" on page 291
"Run-time configurations" on page 9

Related reference
"Build descriptor options" on page 359
"callLink element" on page 395
"Generated output (reference)" on page 516
"Informix and EGL" on page 235

targetNLS

The build descriptor option **targetNLS** specifies the national language code used to identify run-time messages.

The next table lists the supported languages. The code page for the language you specify must be loaded on your target platform.

Code	Languages
CHS	Simplified Chinese
CHT	Traditional Chinese
DES	Swiss German
DEU	German
ENP	Uppercase English (not supported on Windows 2000, Windows NT, and z/OS UNIX System Services)
ENU	US English
ESP	Spanish
FRA	French
ITA	Italian
JPN	Japanese
KOR	Korean
PTB	Brazilian Portuguese

EGL determines if the Java locale on the development machine is associated with one of the supported languages. If the answer is "yes," the default value of **targetNLS** is the supported language. Otherwise, **targetNLS** has no default value.

Related reference
"Build descriptor options" on page 359

templateDir

The build descriptor option **templateDir** specifies the directory containing templates that the iSeries build server uses to create the CL program identified by the symbolic parameter %EZEMBR%_R. That program is used at run time only if it is called from a client program running on a workstation.

This option is meaningful only when you are generating a program of type *iSeriesc*.

Related concepts
"Build descriptor part" on page 275
"Run-time configurations" on page 9

Related reference
"Build descriptor options" on page 359

VAGCompatibility

The build descriptor option **VAGCompatibility** indicates whether the generation process allows use of special program syntax, as described in *Compatibility with VisualAge Generator*. Valid values are *no* and *yes*.

The setting of the EGL preference **VAGCompatibility** determines the default value of the build descriptor. If you are generating in the EGL SDK, no preferences are available, and the default value of **VAGCompatibility** is *no*.

Specify *yes* only if your program or PageHandler uses the special syntax.

Related concepts
"Build descriptor part" on page 275
"Compatibility with VisualAge Generator" on page 428

Related reference
"Build descriptor options" on page 359

validateMixedItems

The build descriptor option **validateMixedItems** specifies whether a generated COBOL program validates the integrity of DBCHAR strings when an item of type MBCHAR is assigned to an item of type MBCHAR. Valid values are YES and NO. YES is the default.

The value YES means that EGL run-time services raises an error if a DBCHAR string is truncated as a result of a MBCHAR-to-MBCHAR assignment on the mainframe. If the error occurs in an invoked function, the function returns control, and the result depends on code aspects that are described in *Exception handling*. If the error occurs in the main function, the program ends with an error message.

If your code is meant for the mainframe and assigns values to items of type MBCHAR, and if the error situation is not possible, set **validateMixedItems** to NO for better run-time performance.

Related reference
"Build descriptor options" on page 359
"CHAR" on page 36
"DBCHAR" on page 36
"Exception handling" on page 89
"MBCHAR" on page 37

validateOnlyIfModified

The build descriptor option **validateOnlyIfModified** specifies whether to validate only those text-form fields for which the modified data tag is set. Valid values are as follows:

no (the default)
Validate all variable fields.

yes
Validate only fields for which the modified data tag is set.

Related concepts
"Build descriptor part" on page 275
"Modified data tag and modified property" on page 150

Related reference
"Build descriptor options" on page 359

validateSQLStatements

The build descriptor option **validateSQLStatements** indicates whether SQL statements are validated against a database. Successful use of **validateSQLStatements** requires that you specify option **sqlValidationConnectionURL** and, in most cases, other options that begin with the letters **sql**, as listed later.

Valid values are YES and NO, and the default is NO. Validation of SQL statements increases the time required to generate your code.

When you request SQL validation, the database manager accessed from the generation platform prepares the SQL statements dynamically.

SQL statement validation has these restrictions:

- No validation is possible for SQL statements that use dynamic SQL and are based on SQL records
- The validation process may indicate errors that are found by the database manager in the generation environment but that will not be found by the database manager on the target platform
- Validation occurs only if your JDBC driver supports validation of SQL prepare statements and (in some cases) only if you have configured the driver to do such validation; for details, see the documentation for your JDBC driver

Related reference
"Build descriptor options" on page 359
"sqlID" on page 385
"sqlPassword" on page 387
"sqlValidationConnectionURL" on page 387
"sqlJDBCDriverClass" on page 386

Build scripts

Build scripts delivered with EGL

On iSeries, the EGL build server invokes the build script FDAPREP. The script normally resides in the QEGL/REXSRC file but can be copied to another location and customized.

Related concepts
"Build script" on page 322

Options required in EGL build scripts

In EGL build scripts, certain preparation options are required if you are using DB2 UDB.

Required options for DB2 precompiler

The following options are required for DB2 usage and are included in the fdaptcl build script:

- HOST(COB2)
- APOSTSQL
- QUOTE

Symbolic parameters

A symbolic parameter is a variable that is substituted in certain build descriptor options and in iSeries build scripts to affect placement and preparation of generated outputs.

Some symbolic parameters are predefined by the generator. EZEGTIME, for example, contains the time at which generation occurs. For a list of these, see *Predefined symbolic parameters for EGL generation*. Users may also define their own symbolic parameters.

You can use symbolic parameters in the values for the **genDirectory** and the **destDirectory** generation options and in build scripts.

When specifying symbolic parameters with **genDirectory** and **destDirectory**, you reference the value of a symbolic parameter by delimiting the parameter name with percentage signs (%). If you want to refer to the time at which generation occurs, for example, specify %EZEGTIME%.

You can also use more than one symbolic parameter to assign a value. The following symbols represent date and time, separated by a space:

```
%EZEGDATE% %EZEGTIME%
```

When specifying symbolic parameters in build scripts you reference the value by preceding the symbolic parameter name with an ampersand (&) and appending a period (.).

```
&EZEGDATE.
```

For example, if **genDirectory** is set to `C:\MyProject\%EZEENV%`, and the build descriptor option is set to ZOSCICS, then generation outputs will be written to C:\MyProject\ZOSCICS. (The EZEENV predefined symbolic parameter is populated from the option **system**.)

You also can specify your own symbolic parameters, and for each such parameter you assign a value; **MYDIR**, for example, might contain the name of a directory. It is not valid to define the same symbolic parameter (like MYDIR) twice in the same build descriptor.

Note: User-defined symbols cannot start with the prefix EZE.

If the build descriptor you are using for generation uses the **nextBuildDescriptor** option to chain multiple build descriptors, and you define the same-named symbolic parameter in multiple build descriptors that are chained together, the value in use at generation time is determined by the precedence rules described in the page on build descriptors.

The value assigned to the symbolic parameter MYDIR in the master build descriptor, for example, takes precedence over the value assigned to MYDIR in any other build descriptor.

Both predefined and user-defined symbolic parameters are available as substitution variables in the build scripts used to prepare COBOL output.For details, see the *EGL Server Guide for iSeries*.

Predefined symbols

The substitution values for predefined symbols are automatically set at generation. You do not define values for these symbols. There are two categories of predefined symbols.

- Symbolic parameters for EGL generation
-

User-defined symbols cannot start with the prefix EZE.

Related concepts
"Build descriptor part" on page 275
"Sources of additional information on EGL" on page 12

Related reference
"Build descriptor options" on page 359
"Predefined symbolic parameters for EGL generation"

Predefined symbolic parameters for EGL generation

All of the EGL generator's symbolic parameters, whether predefined or user-defined, are passed as environment variables to the build server when you build a generated COBOL program. Environment variables passed to the build server supply values for build script substitution variables of the same name. The environment variable values override any default values defined for the substitution variables.

The next table shows the predefined symbolic parameters.

Name	Description
BUILD_SCRIPT_LIBRARY	Allows overriding the name of the PDS from which the build server reads the build scripts. This symbolic parameter is useful when exception build processing is needed. For example, you can use a separate build script PDS if a special test system is needed with a separate database, COBOL libraries, or CICS libraries. The build scripts could have different default substitution variables or different compile options. An alternative approach is to start a build server on a different port and to allocate a different build script PDS.
DATA	A flag that specifies whether you want to allocate working storage with 24- or 31- bit addresses. The value supplied for this symbolic parameter is passed in the DATA parameter for the COBOL compiler and the z/OS linkage editor. The parameter's value is taken from the build descriptor option **data**.
EZEALIAS	The member name used to store the currently generated member in its associated PDS. If an alias property was specified for the currently generated member, the value of that property, truncated to 8 characters if necessary, is used. If no alias property is specified, then the part name, truncated to 8 characters if necessary, is used. When a form group is the current member, and the form group has print forms included, the format module name is truncated to 6 characters rather than 8, and the characters FM are appended.
EZEGDATE	The date on which a program is generated. The format is *mm/dd/yy*, where *mm* is the two-digit month, *dd* is the two-digit day of the month, and *yy* is the last two digits of the year.
EZEGMBR	The name of the program part that was specified to start generation.

Name	Description
EZEMBR	The name of the program that was generated. The value will be the same as the value of the MBR parameter unless you specified the **alias** property for the program generated, or the name of the generated program is longer than 8 characters.
EZEPID	The high-level qualifier that is used for the PDSs that receive the generated and built outputs. The parameter's value is taken from the build descriptor option **projectID**.
EZESQL	An indicator as to whether or not the generated part performs SQL I/O. ″Y″ indicates yes; ″N″ indicates no.
EZETIME	The time at which a program is generated. The format is *hh*:*mm*:*ss*, where *hh* is the hour, *mm* is the minute, and *ss* is the seconds portion of the time.
EZEEXTNM	The external name, if any, that is specified in the program part's **alias** property. This symbolic parameter is available only during generation of bind control and link edit files. It is not available when build scripts are being executed. If the external name is not specified, the name of the part is used but is truncated (if necessary) to the maximum number of characters allowed in the run-time environment.
MBR	The external name given to the generated source code. The external name is the same as the EGL program name unless an **alias** property was specified for the program or the program name is longer than 8 characters. If the **alias** property was specified, its value is placed in the MBR environment variable. Otherwise, if the program name is longer than 8 characters, the program name is truncated to 8 characters and the result is placed in the MBR environment variable.
SYSTEM	The target system for which the EGL program was generated; for example, ZOSCICS or ISERIESC. The parameter's value is taken from the build descriptor option **system**.

In addition to these predefined symbolic parameters, you can define your own symbolic parameters that are passed to the build server as environment variables. If the build script contains a substitution variable whose name matches the symbolic parameter name, the build server uses the value of the symbolic parameter in the build script, in place of the substitution variable.

Related concepts

"Generation" on page 301

callLink element

The callLink element of a linkage options part specifies the type of linkage used in a call. Each element includes these properties:

- pgmName
- type

The value of the **type** property determines what additional properties are available, as shown in the next sections:

- "If callLink type is localCall (the default)"
- "If callLink type is remoteCall"
- "If callLink type is ejbCall"

If callLink type is localCall (the default)

Set property **type** to localCall when you are generating a Java program that calls a generated Java program that resides in the same thread. In this case, EGL middleware is not in use, and the following properties are meaningful for a callLink element in which **pgmName** identifies the called program--

- "alias in callLink element" on page 397
- "package in callLink element" on page 404
- "pgmName in callLink element" on page 406
- "type in callLink element" on page 412

You do not need to specify a callLink element for the call if the called program is in the same package as the caller and if either of these conditions is in effect:

- You do not specify an external name for the called program; or
- The external name for the called program is identical to the part name for that program.

The value of **type** cannot be localCall when you are generating a Java wrapper.

If callLink type is remoteCall

Set property **type** to remoteCall when you are generating a Java program or wrapper, and the Java code calls a program that runs in a different thread. The call is not by way of a generated EJB session bean. In this case, EGL middleware is in use, and the following properties are meaningful for a callLink element in which **pgmName** identifies the called program--

- "alias in callLink element" on page 397
- "conversionTable in callLink element" on page 398
- "location in callLink element" on page 402
- "package in callLink element" on page 404 (used only if the generated code is calling a Java program that is stored in another package)
- "pgmName in callLink element" on page 406
- "remoteBind in callLink element" on page 407
- "remoteComType in callLink element" on page 408
- "remotePgmType in callLink element" on page 410
- "serverID in callLink element" on page 411
- "type in callLink element" on page 412

If callLink type is ejbCall

Set property **type** to ejbCall when a callLink element is required to handle either of the following situations:

- You are generating a Java wrapper and intend to call the related, generated program by way of a generated EJB session bean
- You are generating a Java program and intend to call another generated program by way of a generated EJB session bean

In this case, EGL middleware is in use, and the following properties are meaningful for a callLink element in which **pgmName** identifies the called program:

- "alias in callLink element"
- "conversionTable in callLink element" on page 398
- "location in callLink element" on page 402
- "package in callLink element" on page 404 (used only if the generated Java code is calling a Java program that is stored in a package other than the package in which the EJB session bean resides)
- "parmForm in callLink element" on page 405 (used only if the generated Java code is calling a program that runs on CICS)
- "pgmName in callLink element" on page 406
- "providerURL in callLink element" on page 406
- "remoteBind in callLink element" on page 407
- "remoteComType in callLink element" on page 408
- "remotePgmType in callLink element" on page 410
- "serverID in callLink element" on page 411
- "type in callLink element" on page 412

Related concepts
"Linkage options part" on page 291
"Run-time configurations" on page 9

Related tasks
"Editing the callLink element of a linkage options part" on page 294

Related reference
"alias in callLink element"
"conversionTable in callLink element" on page 398

"linkType in callLink element" on page 401
"location in callLink element" on page 402

"package in callLink element" on page 404

"pgmName in callLink element" on page 406
"providerURL in callLink element" on page 406
"remoteBind in callLink element" on page 407
"remoteComType in callLink element" on page 408
"remotePgmType in callLink element" on page 410
"serverID in callLink element" on page 411
"type in callLink element" on page 412

alias in callLink element

The linkage options part, callLink element, property **alias** specifies the run-time name of the program identified in property **pgmName**. The property is meaningful only when **pgmName** refers to a program that is called by the program being generated.

The value of this property must match the alias (if any) you specified when declaring the program. If you did not specify an alias when declaring the program, either set the callLink element property **alias** to the name of the program part or do not set the property at all.

Related concepts
"Linkage options part" on page 291

Related tasks
"Editing the callLink element of a linkage options part" on page 294

Related reference
"callLink element" on page 395
"pgmName in callLink element" on page 406

conversionTable in callLink element

The linkage options part, callLink element, property **conversionTable** specifies the name of the conversion table that is used to convert data on a call. The property is meaningful only when **pgmName** identifies a program that is called by the generated program or wrapper.

When you generate a COBOL program, these details are in effect:

- The property **conversionTable** is useful only in this case--
  - The call is to a non-EGL-generated program
  - The called program runs on a platform that supports the ASCII character set
- The property is available only if the value of property **type** is remoteCall
- The default is that no conversion occurs

When you generate a Java program or wrapper, the following details are in effect:

- When the call is to a non-Java program, a default conversion occurs in accordance with the character set (ASCII or EBCDIC) used on the calling platform. You must specify a value for **conversionTable** in the following case--
  - The caller is Java code and is on a machine that supports one character set (EBCDIC or ASCII); and
  - The called program is non-Java and is on a machine that supports the other character set.
- An attempt to specify a conversion table has no effect when EGL-generated Java code calls a Java program, except in the case of bidirectional text.
- The property **conversionTable** is available only if the value of property **type** is ejbCall or remoteCall.

Select one of the following values:

conversion table name
The caller uses the conversion table specified. For a list of tables, see *Data conversion*.

* Uses the default conversion table. For a COBOL client, the name of that table is ELA*xxx*, where the value of *xxx* is the value of build descriptor option targetNLS. For a Java client, the selected table is based either on the locale of the client machine or (if the client is running on a Web application server) on the locale of that server. If an unrecognized locale is found, English is assumed.

For a list of tables, see *Data conversion*.

programControlled
The caller uses the conversion table name that is in the system item sysVar.callConversionTable at run time. If sysVar.callConversionTable contains blanks, no conversion occurs.

Related concepts
"Linkage options part" on page 291

Related tasks
"Editing the callLink element of a linkage options part" on page 294

Related reference
"Bidirectional language text" on page 458
"callLink element" on page 395
"Data conversion" on page 454
"pgmName in callLink element" on page 406
"convert()" on page 870
"targetNLS" on page 389
"type in callLink element" on page 412

ctgKeyStore in callLink element

The linkage options part, callLink element, property **ctgKeyStore** is the name of the key store generated with the Java tool keytool.exe or with the CICS Transaction Gateway tool IKEYMAN. This property is required when the value of property **remoteComType** is set to CICSSSL.

Related concepts
"Linkage options part" on page 291

Related reference
"callLink element" on page 395
"ctgKeyStorePassword in callLink element"
"remoteComType in callLink element" on page 408

ctgKeyStorePassword in callLink element

The linkage options part, callLink element, property **ctgKeyStorePassword** is the password used when generating the key store.

Related concepts
"Linkage options part" on page 291

Related reference
"callLink element" on page 395
"ctgKeyStore in callLink element"
"remoteComType in callLink element" on page 408

ctgLocation in callLink element

The linkage options part, callLink element, property **ctgLocation** is the URL for accessing a CICS Transaction Gateway (CTG) server, as is used if the value of property **remoteComType** is CICSECI or CICSSSL. Specify the related port by setting the property **ctgPort**.

Related concepts
"Linkage options part" on page 291

Related reference
"callLink element" on page 395
"remoteComType in callLink element" on page 408

ctgPort in callLink element

The linkage options part, callLink element, property **ctgPort** is the port for accessing a CICS Transaction Gateway (CTG) server, as is used if the value of property **remoteComType** is CICSECI or CICSSSL. Specify the related URL by setting the property **ctgLocation**.

If the case of CICSSSL, the value of **ctgPort** is the TCP/IP port on which a CTG JSSE listener is listening for requests; and if **ctgPort** is not specified, the CTG default port of 8050 is used.

Related concepts
"Linkage options part" on page 291

Related reference
"callLink element" on page 395
"ctgLocation in callLink element"
"remoteComType in callLink element" on page 408

JavaWrapper in callLink element

The linkage options part, **callLink** element, property **javaWrapper** indicates whether to allow generation of Java wrapper classes that can invoke the program being generated.

Valid values are as follows:

No (the default)
Do not allow generation of Java wrapper classes.

Yes
Allow that generation to occur. The generation occurs only if the build descriptor option **enableJavaWrapperGen** is set to **yes** or **only**.

Your choice for **javaWrapper** property has an effect only when you are setting up a remote call, as occurs when the value of the **callLink** property **type** is **remoteCall**. In contrast, if you are setting up a call to the program by way of an EJB, the value of **javaWrapper** is always **yes**; and if you are setting up a local call, the value of **javaWrapper** is always **no**.

If you are generating in the workbench or from the workbench batch interface, the build descriptor option genProject identifies the project that receives the classes. If

genProject is not specified (or if you are generating in the EGL SDK), the wrapper classes are placed in the directory specified by the build descriptor option genDirectory.

Related concepts
"Linkage options part" on page 291

Related reference
"callLink element" on page 395
"genDirectory" on page 372
"genProject" on page 374

linkType in callLink element

The linkage options part, callLink element, property **linkType** specifies the type of linkage when the value of property **type** is localCall.

If you are generating a COBOL or Java program, **linkType** is meaningful when property **pgmName** refers to a program that is called by the program being generated. If you are generating a Java wrapper, property type must be remoteCall or ejbCall, and **linkType** is not available.

Select a value from this list:

DYNAMIC
If you are generating a COBOL program, specifies that the call is a dynamic COBOL call.

If you are generating a Java program, specifies that the call is to a Java program in the same thread. DYNAMIC is the default value when you are generating a Java program.

STATIC
If you are generating a COBOL program, specifies that a static COBOL call occurs, which means that you must link-edit the called program with the calling program.

If you are generating a Java program, STATIC is equivalent to DYNAMIC.

Related concepts
"Linkage options part" on page 291

Related tasks
"Editing the callLink element of a linkage options part" on page 294

Related reference
"callLink element" on page 395
"pgmName in callLink element" on page 406
"type in callLink element" on page 412

library in callLink element

The linkage options part, callLink element, property **library** specifies the DLL or library that contains the called program when the value of the **type** property is ejbCall or remoteCall:

- If your program is calling a remote COBOL program on iSeries, the **library** property refers to the iSeries library that contains the program to be called.

- If your EGL-generated Java program is calling a remote, non-EGL generated program on iSeries (for example, a C or C++ service program), the called program belongs to an iSeries library, and the **library** property refers to the name of the program that contains the entry point to be called. Set the other callLink properties as follows:
  - Set the **pgmName** property to the name of the entry point
  - Set the **remoteComType** property to direct or distinct
  - Set the **remotePgmType** property to externallyDefined
  - Set the **location** property to the name of the iSeries library
- Otherwise, if the calling program is an EGL-generated Java program not on iSeries, the **library** property refers to the name of a DLL that contains an entry point to be called locally as a native program. The entry point is identified by the **pgmName** property; but you need to specify the **library** property only if the names of the entry point and DLL are different.

  To call a native DLL, set the other callLink properties as follows:
  - Set the **remoteComType** property to direct
  - Set the **remotePgmType** property to externallyDefined
  - Set the **type** property to remoteCall because EGL middleware is used even though the DLL is called on the machine where the Java program is running.

Related concepts
"Linkage options part" on page 291

Related reference
"callLink element" on page 395

location in callLink element

The linkage options part, callLink element, property **location** specifies how the location of a called program is determined at run time. The property **location** is applicable in the following situation:

- The value of property **type** is ejbCall or remoteCall;
- The value of property **remoteComType** is JAVA400, CICSECI, CICSSSL, CICSJ2C, or TCPIP; and
- One of these statements applies:
  - If you are generating a COBOL orJava program, property **pgmName** refers to a program that is called by the program being generated
  - If you are generating a Java wrapper, **pgmName** refers to a program that is called by way of the Java wrapper

Select a value from this list:

programControlled
Specifies that the location of the called program is obtained from the system function sysVar.remoteSystemID when the call occurs.

system name
Specifies the location where the called program resides.

If you are generating a Java program or wrapper, the meaning of this property depends on property **remoteComType**:

- If the value of **remoteComType** is JAVA400, **location** refers to the iSeries system identifier

- If the value of **remoteComType** is CICSECI or CICSSSL, **location** refers to the CICS system identifier
- If the value of **remoteComType** is CICSJ2C, **location** refers to the JNDI name of the ConnectionFactory object that you establish for the CICS transaction invoked by the call. You establish that ConnectionFactory object when setting up the J2EE server, as described in *Setting up the J2EE server for CICSJ2C calls*. By convention, the name of the ConnectionFactory object begins with `eis/`, as in the following example:

  ```
  eis/CICS1
  ```
- If the value of **remoteComType** is TCPIP, **location** refers to the TCP/IP hostname, and no default value exists
- If all the next conditions apply, **location** refers to the library of the called program--
  - The called program is an EGL-generated Java program that runs locally on iSeries
  - The value of **remoteComType** is DIRECT or DISTINCT
  - The value of **remotePgmType** is EXTERNALLYDEFINED

Related concepts
"Linkage options part" on page 291

Related tasks
"Editing the callLink element of a linkage options part" on page 294
"Setting up the J2EE server for CICSJ2C calls" on page 337

Related reference
"callLink element" on page 395
"remoteSystemID" on page 906
"pgmName in callLink element" on page 406
"remoteComType in callLink element" on page 408
"type in callLink element" on page 412

luwControl in callLink element

The linkage options part, callLink element, property **luwControl** specifies whether the caller or called program controls the unit of work. This property is applicable only in the following situation:

- The value of property **type** is remoteCall; and
- You are generating a Java program or wrapper--
  - If you are generating a Java program, property **pgmName** refers to a CICS-based program that is called by the program being generated
  - If you are generating a Java wrapper, **pgmName** refers to a CICS-based program that is called by way of the Java wrapper

Select one of the following values:

CLIENT
Specifies that the unit of work is under the caller's control. Updates by the called program are not committed or rolled back until the caller requests commit or rollback. If the called program issues a commit or rollback, a run-time error occurs.

CLIENT is the default value, unless a caller-controlled unit of work is not supported on the platform where the called program resides.

SERVER
Specifies that a unit of work started by the called program is independent of any unit of work controlled by the calling program. In the called program, these rules apply:

- The first change to a recoverable resource begins a unit of work
- Use of the system functions sysLib.commit and sysLib.rollback are valid

On a call from EGL-generated Java code to a VisualAge Generator COBOL program, a commit (or rollback on abnormal termination) is issued automatically when the called program returns. That command affects only the changes that were made by the called program.

When the property **type** is ejbCall, the run-time behavior is as described for SERVER.

Related concepts
"Linkage options part" on page 291
"Logical unit of work" on page 288

Related tasks
"Editing the callLink element of a linkage options part" on page 294

Related reference
"callLink element" on page 395
"commit()" on page 866
"rollback()" on page 878
"pgmName in callLink element" on page 406
"type in callLink element" on page 412

package in callLink element

The linkage options part, callLink element, property **package** identifies the Java package in which a called Java program resides. The property is useful whether property **type** is ejbcall, localCall, or remoteCall.

If you are generating a Java program, **package** is meaningful when property **pgmName** refers to a program that is called by the program being generated. If you are generating a Java wrapper, **package** is meaningful when property **pgmName** refers to the program that is called by way of the Java wrapper.

If the **package** property is not specified, the called program is assumed to be in the same package as the caller.

The package name that is used in generated Java programs is the package name of the EGL program, but in lower case; and when EGL generates output from the callLink element, the value of **package** is changed (if necessary) to lower case.

Related concepts
"Linkage options part" on page 291

Related tasks
"Editing the callLink element of a linkage options part" on page 294
"Setting up the J2EE run-time environment for EGL-generated code" on page 333

Related reference
"callLink element" on page 395
"pgmName in callLink element" on page 406
"type in callLink element" on page 412

parmForm in callLink element

The linkage options part, callLink element, property **parmForm** specifies the format of call parameters.

If you are generating a COBOL program, **parmForm** is applicable in this situation:

- Property **pgmName** refers to the program being generated or to a CICS-based program that is called by the program being generated; and
- Property **type** is localCall or remoteCall--
  - If the **type** is localCall, the valid **parmForm** values (as described later) are COMMDATA, COMMPTR (the default), and OSLINK
  - If the **type** is remoteCall, the valid **parmForm** values are COMMDATA (the default) and (if you are referring to a COBOL program called from Java code) COMMPTR.

If you are generating a Java program, **parmForm** is applicable in this situation:

- Property **pgmName** refers to a CICS-based program that is called by the program being generated; and
- Property **type** is ejbCall or remoteCall; in either case, the valid **parmForm** values (as described later) are COMMDATA (the default) and COMMPTR.

If you are generating a Java wrapper, **parmForm** is applicable in this case:

- Property **pgmName** refers to a generated COBOL program that is called by way of the Java wrapper; and
- Property **type** is ejbCall or remoteCall; in either case, the valid **parmForm** values (as described later) are COMMDATA (the default) or COMMPTR.

Select a value from this list:

COMMDATA
Specifies that the caller places business data (rather than pointers to data) in the COMMAREA.

Each argument value is moved to the buffer adjoining the previous value without regard for boundary alignment.

COMMDATA is the default value if the property **type** is ejbCall or remoteCall.

COMMPTR
Specifies that the caller acts as follows:

- Places a series of 4-byte pointers in the COMMAREA, one pointer per argument passed
- Sets the high-order bit of the last pointer to 1

COMMPTR is the default value if the value of property **type** is localCall.

OSLINK
Specifies that the standard COBOL parameter-passing conventions are in effect, with the called program expecting pointers to data, but without the CICS EIB or COMMAREA.

OSLINK is available only when **type** is localCall, **linkType** is DYNAMIC or STATIC, and you are generating a COBOL program.

Related concepts
"Linkage options part" on page 291

Related tasks
"Editing the callLink element of a linkage options part" on page 294

Related reference
"callLink element" on page 395
"linkType in callLink element" on page 401
"parmForm in callLink element" on page 405
"pgmName in callLink element"
"type in callLink element" on page 412

pgmName in callLink element

The linkage options part, callLink element, property **pgmName** specifies the name of the program part to which the callLink element refers.

You can use an asterisk (*) as a global substitution character in the program name; however, that character is valid only as the last character. For details, see *Linkage options part*.

Related concepts
"Linkage options part" on page 291

Related tasks
"Editing the callLink element of a linkage options part" on page 294

Related reference
"callLink element" on page 395

providerURL in callLink element

The linkage options part, callLink element, property **providerURL** specifies the host name and port number of the name server used by an EGL-generated Java program or wrapper to locate an EJB session bean that in turn calls an EGL-generated Java program. The property must have the following format:

```
iiop://hostName:portNumber
```

hostName
The IP address or host name of the machine on which the name server runs

portNumber
The port number on which the name server listens

The property **providerURL** is applicable only in the following situation:

- The value of property **type** is ejbCall; and
- Property **pgmName** refers to the program being called from the Java program or wrapper being generated.

Enclose the URL in double quote marks to avoid a problem either with periods or with the colon that precedes the port number.

A default is used if you do not specify a value for **providerURL**. The default directs an EJB client to look for the name server that is on the local host and that listens on port 900. The default is equivalent to the following URL:

```
"iiop:///"
```

The following **providerURL** value directs an EJB client to look for a remote name server that is called *bankserver.mybank.com* and that listens on port 9019:

```
"iiop://bankserver.mybank.com:9019"
```

The following property value directs an EJB client to look for a remote name server that is called *bankserver.mybank.com* and that listens on port 900:

```
"iiop://bankserver.mybank.com"
```

Related concepts
"Linkage options part" on page 291

Related tasks
"Editing the callLink element of a linkage options part" on page 294
"Setting up the J2EE run-time environment for EGL-generated code" on page 333

Related reference
"callLink element" on page 395
"pgmName in callLink element" on page 406
"type in callLink element" on page 412

refreshScreen in callLink element

The linkage options part, callLink element, property **refreshScreen** indicates whether an automatic screen refresh is to occur when the called program returns control. Valid values are *yes* (the default) and *no*.

Set **refreshScreen** to *no* if the caller is in a run unit that presents text forms to a screen and either of these situations applies:

- The called program does not present a text form; or
- The caller writes a full-screen text form after the call.

The property **refreshScreen** applies only in these cases:

- The callLink **type** property is localCall; or
- The callLink **type** property is remoteCall when the remoteComType property is direct or distinct.

The property is ignored if you include the **noRefresh** indicator on the call statement.

Related reference
"call" on page 547

remoteBind in callLink element

The linkage options part, callLink element, property **remoteBind** specifies whether linkage options are determined at generation time or at run time. This property is applicable only in the following situation:

- The value of property **type** is ejbCall or remoteCall; and

- You are generating a Java program or wrapper. The property **pgmName** may refer to a program that is called by the program being generated, in which case the entry refers to the call from program to program. Alternatively, the property may refer to the program being generated, in which case the entry refers to the call from wrapper to program.

Select one of these values:

GENERATION
The linkage options specified at generation time are necessarily in use at run time. GENERATION is the default value.

RUNTIME
The linkage options specified at generation time can be revised at deployment time. In this case, you must include a linkage properties file in the run-time environment.

EGL generates a linkage properties file in the following situation:

- You are generating a Java program or wrapper;
- You set the property **remoteBind** to RUNTIME; and
- You generate with the build descriptor option **genProperties** set to GLOBAL or PROGRAM.

Related concepts
"Linkage options part" on page 291
"Linkage properties file" on page 343

Related tasks
"Deploying a linkage properties file" on page 342
"Editing the callLink element of a linkage options part" on page 294

Related reference
"callLink element" on page 395
"genProperties" on page 375
"Linkage properties file (details)" on page 637
"pgmName in callLink element" on page 406
"type in callLink element" on page 412

remoteComType in callLink element

The linkage options part, callLink element, property **remoteComType** specifies the communication protocol used in the following case:

- The value of property **type** is ejbCall or remoteCall; and
- You are generating a Java program or wrapper--
  - If you are generating a Java program, property **pgmName** refers to a program that is called by the program being generated
  - If you are generating a Java wrapper, **pgmName** refers to a program that is called by way of the Java wrapper

Select one of the following values.

DEBUG
Causes the called program to run in the EGL debugger, even when the calling program is running in a Java run-time or Java debug environment. You might use this setting in the following cases:

- You are running a Java program that uses an EGL Java wrapper to call a program written with EGL; or
- You are running an EGL-generated calling program that calls a program written with EGL.

The preceding situations can occur outside the WebSphere Test Environment, but can also occur within that environment, as when a JSP invokes a program written with EGL. In any case, the effect is to invoke the EGL source, not an EGL-generated program.

If you are using the WebSphere Test Environment, the caller and called programs must both be running there; the call cannot be from a remote machine.

When you use DEBUG, you set the following properties in the same **callLink** element--

- **library**, which names the project that contains the called program
- **package**, which identifies the package that contains the called program; but you do not need to set this property if the caller and called programs are in the same package

If the caller is not running in the EGL debugger and is not running in the WebSphere Test Environment, you must set these properties of the callLink element:

- **serverid**, which should specify the listener's port number if it's not 8346; and
- **location**, which must contain the hostname of the machine where the Eclipse workbench is running.

DIRECT

Specifies that the calling program or wrapper uses a direct local call, which means that the calling and called code run in the same thread. No TCP/IP listener is involved, and the value of property **location** is ignored. DIRECT is the default.

A calling Java program does not use the EGL middleware, but a calling wrapper uses that middleware to handle data conversion between EGL and Java primitive types.

If the EGL-generated Java code is calling a non-EGL-generated dynamic link library (DLL) or a C or C++ program, it is recommended that you use the **remoteComType** value DISTINCT.

DISTINCT

Specifies that a new run unit is started when calling a program locally. The call is still considered to be remote because EGL middleware is involved.

You can use this value for an EGL-generated Java program that calls a dynamic link library (DLL) or a C or C++ program.

CICSECI

Specifies use of the CICS Transaction Gateway (CTG) ECI interface, as is needed when you are debugging or running non-J2EE code that accesses CICS.

CTG Java classes are used to implement this protocol. To specify the URL and port for a CTG server, assign values to the callLink element, properties ctgLocation and ctgPort. To identify the CICS region where the called program resides, specify the location property.

CICSJ2C
Specifies use of a J2C connector for the CICS Transaction Gateway.

CICSSSL
Specifies use of the Secure Socket Layer (SSL) features of CICS Transaction Gateway (CTG). The JSSE implementation of SSL is supported.

CTG Java classes are used to implement this protocol. To specify additional information for a CTG server, assign values to the following callLink element properties:

- ctgKeyStore
- ctgKeyStorePassword
- ctgLocation
- ctgPort, which in this case is the TCP/IP port on which a CTG JSSE listener is listening for requests. If ctgPort is not specified, the CTG default port of 8050 is used.

To identify the CICS region where the called program resides, specify the location property.

JAVA400
Specifies use of the IBM Toolbox for Java to communicate between a Java wrapper or program and a COBOL program that was generated (by EGL or VisualAge Generator) for iSeries.

TCPIP
Specifies that the EGL middleware uses TCP/IP.

Related concepts
"Linkage options part" on page 291

Related tasks
"Editing the callLink element of a linkage options part" on page 294

Related reference
"ctgKeyStore in callLink element" on page 399
"ctgKeyStorePassword in callLink element" on page 399
"ctgLocation in callLink element" on page 400
"ctgPort in callLink element" on page 400
"Editing the callLink element of a linkage options part" on page 294
"Setting up the J2EE server for CICSJ2C calls" on page 337
"Setting up the TCP/IP listener for a called appl in a J2EE appl client module" on page 338
"Setting up the TCP/IP listener for a called non-J2EE application" on page 332

remotePgmType in callLink element

The linkage options part, callLink element, property **remotePgmType** specifies the kind of program being called. The property is applicable in the following situation:

- The value of property **type** is ejbCall or remoteCall; and
- One of these statements applies:
  - If you are generating a program (rather than a wrapper), property **pgmName** refers to a program that is called by the program being generated.

    The called program is one of the following kinds:
    - An EGL-generated Java program
    - A non-EGL-generated dynamic link library (DLL) or C or C++ program
    - A program that runs on CICS and has CICS commands

– If you are generating a Java wrapper, **pgmName** refers to the program that is called by way of the Java wrapper.

EGL
The called program is a COBOL or Java program that was generated by EGL or by VisualAge Generator; in this case, the caller is a COBOL program, Java program, or Java wrapper. This value is the default.

Related concepts
"Linkage options part" on page 291
"Run-time configurations" on page 9

Related tasks
"Editing the callLink element of a linkage options part" on page 294

Related reference
"callLink element" on page 395
"library in callLink element" on page 401
"pgmName in callLink element" on page 406
"type in callLink element" on page 412

serverID in callLink element

The linkage options part, callLink element, property **serverID** specifies one of the following values:

- The TCP/IP port number of a called program's listener; but only if the TCP/IP protocol is in use. In this case, no default exists.
- The ID of a CICS transaction being invoked, but only when access to CICS is by the ECI interface or Secure Socket Layer features of the CICS Transaction Gateway. In this case, the default is the CICS server system mirror transaction.

The property is used only in the following situation:

- The value of property **type** is ejbCall or remoteCall;
- The value of **remoteComType** is TCPIP, CICSECI, or CICSSSL; and
- You are generating a Java program or wrapper--
  - If you are generating a Java program, property **pgmName** refers to a program that is called by the program being generated
  - If you are generating a Java wrapper, **pgmName** refers to a program that is called by way of the Java wrapper

Related concepts
"Linkage options part" on page 291

Related tasks
"Editing the callLink element of a linkage options part" on page 294
"Setting up the TCP/IP listener for a called appl in a J2EE appl client module" on page 338
"Setting up the TCP/IP listener for a called non-J2EE application" on page 332

Related reference
"callLink element" on page 395
"pgmName in callLink element" on page 406
"remoteComType in callLink element" on page 408
"type in callLink element" on page 412

type in callLink element

The linkage options part, callLink element, property **type** specifies the kind of call. Select one of the following values:

ejbCall
Indicates that the generated Java program or wrapper will implement the program call by using an EJB session bean and that the EJB session bean will access the COBOL or Java program identified in the property **pgmName**. The value ejbCall is applicable in either of two cases:

- You are generating a Java wrapper for a COBOL or Java program, and the wrapper calls that program by way of an EJB session bean. In this case, the property **pgmName** refers to the program called from the wrapper, and your use of ejbCall causes generation of the EJB session bean.
- You are generating a Java program that calls a generated COBOL or Java program by way of an EJB session bean. In this case, the property **pgmName** refers to the called program, and an EJB session bean is not generated.

In either case, if you are using an EJB session bean, you must generate a Java wrapper, if only to generate the EJB session bean.

The generated session bean must be deployed on an enterprise Java server, and one of the following statements must be true:

- The name server used to locate the EJB session bean resides on the same machine as the code calling that session bean; or
- The property **providerURL** identifies where the name server resides.

If you wish to use an EJB session bean, you must generate the calling program or wrapper with a linkage options part in which the value of property **type** for the called program is ejbCall. You cannot make the decision to use a session bean at deployment time. If you set the property **remoteBind** to RUNTIME, however, you can decide at deployment time *how* the EJB session bean accesses the generated program, although making this decision at generation time is more efficient.

localCall
Specifies that the call does *not* use EGL middleware. The called program in this case is in the same process as the caller.

If the caller is a COBOL program, the situation is further defined by other properties. Most important are **linkType** and (for CICS COBOL programs) **parmForm**. Those properties have default values that you can accept or override.

localCall is the default value

remoteCall
Specifies that the call uses EGL middleware, which adds 12 bytes to the end of the data passed. Those bytes allow the caller to receive a return value from the called program.

If the caller is Java code, communication is handled by the protocol specified in property **remoteComType**; the protocol choice indicates whether the called program is in the same or a different thread.

If variable length records are passed on a call, these statements apply:

- Space is reserved for the maximum length specified for the record
- If the value of callLink property **type** is remoteCall or ejbCall, the variable-length item (if any) must be inside the record

Related concepts
"Linkage options part" on page 291

Related tasks
"Editing the callLink element of a linkage options part" on page 294

Related reference
"callLink element" on page 395
"linkType in callLink element" on page 401
"location in callLink element" on page 402
"parmForm in callLink element" on page 405
"pgmName in callLink element" on page 406
"providerURL in callLink element" on page 406
"remoteComType in callLink element" on page 408

C functions with EGL

EGL programs can invoke C functions.

To invoke a C function from EGL:

After you have identified the C functions to use in your EGL program, you must:

1. Download the EGL stack library and application object file from the IBM website to your computer.
2. Compile all C code into one shared library and link it with the appropriate platform-specific stack library.
3. Create a function table.
4. Compile the function table and the appropriate platform-specific application object file into a shared library, and link this shared library with the shared library created in Step 2 and the stack library.

1. Download the EGL stack library and application object file

To download the EGL stack library and application object file:

1. Locate the EGL Support website.
   - The URL for Rational Application Developer is:
     `http://www3.software.ibm.com/ibmdl/pub/software/rationalsdp/rad/60/redist`
   - The URL for Rational Web Developer is:
     `http://www3.software.ibm.com/ibmdl/pub/software/rationalsdp/rwd/60/redist`

   .
2. Download the **EGLRuntimesV60IFix001.zip** file to your preferred directory.
3. Unzip **EGLRuntimesV60IFix001.zip** to identify the following files:

For the platform-specific stack libraries:

- AIX: EGLRuntimes/Aix/bin/libstack.so
- Linux: EGLRuntimes/Linux/bin/libstack.so
- Win32:
  EGLRuntimes/Win32/bin/stack.dll
  EGLRuntimes/Win32/bin/stack.lib

.

For the platform-specific application object files:

- AIX: EGLRuntimes/Aix/bin/application.o
- Linux: EGLRuntimes/Linux/bin/application.o
- Win32: EGLRuntimes/Win32/bin/application.obj

.

2. Compile all C code into a shared library

Your C code receives values from EGL using pop external functions and returns values to EGL using return external functions. The pop external functions are described in *Receiving values from EGL*; the return external functions are described in *Returning values to EGL*.

To compile all C code into a shared library:

1. Using standard methods, compile all of your C code into one shared library and link it with the appropriate platform-specific EGL stack library.
2. In the following platform-specific examples, **file1.c** and **file2.c** are C files containing functions invoked from EGL.

   On AIX (the ld command must be on a single line):

   ```
   cc -c -Iincl_dir  file1.c file2.c
   ld -G -b32 -bexpall -bnoentry
      -brtl file1.o file2.o -Lstack_lib_dir
      -lstack -o lib1_name -lc
   ```

   On Linux (the gcc command must be on a single line):

   ```
   cc -c -Iincl_dir  file1.c file2.c
   gcc -shared file1.o file2.o -Lstack_lib_dir
       -lstack -o lib1_name
   ```

   On Windows (the link command must be on a single line):

   ```
   cl /c -Iincl_dir file1.c file2.c
   link /DLL file1.obj file2.obj
        /LIBPATH:stack_lib_dir
        /DEFAULTLIB:stack.lib /OUT:lib1_name
   ```

   incl_dir
   the directory location for the header files.

   stack_lib_dir
   the directory location for the stack library.

   lib1_name
   the name of the output library.

Note: If your C code is using any of the IBM Informix ESQL/C library functions (BIGINT, DECIMAL, DATE, INTERVAL, DATETIME), then the ESQL/C library must also be linked.

3. Create a function table

The function table is a C source file which includes the names of all C functions to be invoked from the EGL program. In the following function table example, **c_fun1** and **c_fun2** are names of the C functions. All of the functions identified in the code must have been exported from the C shared library created in Step 2 above.

```
#include <stdio.h>
struct func_table {

      char *fun_name;
```

```
      int (*fptr)(int);
};

extern int c_fun1(int);
extern int c_fun2(int);
/* Similar prototypes for other functions */

struct func_table ftab[] =
            {
                 "c_fun1", c_fun1,
                 "c_fun2", c_fun2,
                 /* Similarly for other functions */
                 "", NULL
            };
```

Create a function table based on the example above, and populate the function table with the appropriate C functions. Indicate the end of the function table with *""*, **NULL**.

4. Compile the function table and the platform-specific application object file into a shared library

The application object file is the interface between the EGL code and the C code.

The following two artifacts must be compiled into one shared library and linked with the stack library and the library created in Step 2 above:

- function table
- application object file

Compile the new shared library using the following example, where **ftable.c** is the name of the function table and **mylib** is the name of the C shared library created in Step 2 and **lib_dir** is the directory location for **mylib**. Specify **lib2_name** by using the *dllName* property or the *vgj.defaultI4GLNativeLibrary* Java runtime property.

On AIX (the ld command must be on a single line):

```
cc -c  ftable.c
ld -G -b32 -bexpall -bnoentry
   -brtl ftable.o application.o
   -Lstack_lib_dir -lstack -Llib_dir
   -lmylib -o lib2_name -lc
```

On Linux (the gcc command must be on a single line):

```
cc -c  ftable.c
gcc -shared ftable.o application.o
    -Lstack_lib_dir -lstack -Llib_dir
    -lmylib -o lib2_name
```

On Windows (the link command must be on a single line):

```
cl /c ftable.c
link /DLL ftable.obj application.obj
     /LIBPATH:stack_lib_dir
     /DEFAULTLIB:stack.lib
     /LIBPATH:lib_dir
     /DEFAULTLIB:mylib.lib /OUT:lib2_name
```

Link the three libraries together.

With your C shared library, function table, and stack library linked, you are now ready to invoke the C functions from your EGL code. For information on how to invoke a C function in EGL, see *Invoking a C function from an EGL program*.

Related concept
"Linkage options part" on page 291

Related reference
"BIGINT functions for C"
"C data types and EGL primitive types" on page 417
"DATE functions for C" on page 418
"DATETIME and INTERVAL functions for C" on page 418
"DECIMAL functions for C" on page 420
"Invoking a C Function from an EGL Program" on page 421
"Return functions for C" on page 425
"Stack functions for C" on page 422

BIGINT functions for C

Note: The following BIGINT functionality is available only to users of IBM Informix ESQL/C. To use these functions, ESQL/C users will need to manually link their C code to the ESQL/C libraries.

The BIGINT data type is a machine-independent method for representing numbers in the range of $-2^{63}-1$ to $2^{63}-1$. ESQL/C provides routines that facilitate the conversion from the BIGINT data type to other data types in the C language.

The BIGINT data type is internally represented with the **ifx_int8_t** structure. Information about the structure can be found in the header file **int8.h**, which is included in the ESQL/C product. Include this file in all C source files that use any of the BIGINT functions.

All operations on **int8** type numbers must be performed using the following ESQL/C library functions for the **int8** data type. Any other operations, modifications, or analyses can produce unpredictable results. The ESQL/C library provides the following functions that allow you to manipulate **int8** numbers and convert **int8** type numbers to and from other data types.

Function Name	Description
ifx_int8add( )	Adds two BIGINT type values
ifx_int8cmp( )	Compares two BIGINT type numbers
ifx_int8copy( )	Copies an **ifx_int8_t** structure
ifx_int8cvasc( )	Converts a C **char** type value to a BIGINT type number
ifx_int8cvdbl( )	Converts a C **double** type number to a BIGINT type number
ifx_int8cvdec( )	Converts a **decimal** type value into a BIGINT type value
ifx_int8cvflt( )	Converts a C **float** type value into a BIGINT type value
ifx_int8cvint( )	Converts a C **int** type number into a BIGINT type number
ifx_int8cvlong( )	Converts a C **long** (**int** on 64 bit machine) type value to a BIGINT type value
ifx_int8cvlong_long( )	Converts a C **long long** type (8-byte value, **long long** in 32 bit and **long** in 64 bit) value into a BIGINT type value
ifx_int8div( )	Divides two BIGINT numbers

Function Name	Description
ifx_int8mul( )	Multiples two BIGINT numbers
ifx_int8sub( )	Subtracts two BIGINT numbers
ifx_int8toasc( )	Converts a BIGINT type value to a C **char** type value
ifx_int8todbl( )	Converts a BIGINT type value to a C **double** type value
ifx_int8todec( )	Converts a BIGINT type number into a **decimal** type number
ifx_int8toflt( )	Converts a BIGINT type number into a C **float** type number
ifx_int8toint( )	Converts a BIGINT type value to a C **int** type value
ifx_int8tolong( )	Converts a BIGINT type value to a C **long** (**int** on 64 bit machine) type value
ifx_int8tolong_long( )	Converts a C **long long** (**long** on 64 bit machine) type to a BIGINT type value

Related reference
For more information about the individual functions, see the following: IBM Informix ESQL/C Programmer's Manual.
"DATE functions for C" on page 418
"DATETIME and INTERVAL functions for C" on page 418
"DECIMAL functions for C" on page 420
"Invoking a C Function from an EGL Program" on page 421

C data types and EGL primitive types

.

The following table shows the mapping between C data types, I4GL data types, and EGL primitive types.

C data types	Equivalent I4GL data type	Equivalent EGL primitive type
char	CHAR or CHARACTER	UNICODE(1)
char	NCHAR	UNICODE(size)
char	NVARCHAR	STRING
char	VARCHAR	STRING
int	INT or INTEGER	INT
short	SMALLINT	SMALLINT
ifx_int8_t	BIGINT	BIGINT
dec_t	DEC or DECIMAL(p,s,) or NUMERIC(p)	DECIMAL(p)
dec_t	MONEY	MONEY
double	FLOAT	FLOAT
float	SMALLFLOAT	SMALLFLOAT
loc_t	TEXT	CLOB
loc_t	BYTE	BLOB
int	DATE	DATE
dtime_t	DATETIME	TIMESTAMP
intvl_t	INTERVAL	INTERVAL

Related reference
"BIN and the integer types" on page 47
"BLOB" on page 46
"CLOB" on page 45
"DATE" on page 38
"DECIMAL" on page 47
"FLOAT" on page 48
"INTERVAL" on page 39
"Invoking a C Function from an EGL Program" on page 421
"MBCHAR" on page 37
"MONEY" on page 48
"NUM" on page 48
"Primitive types" on page 31
"SMALLFLOAT" on page 50
"TIME" on page 40
"TIMESTAMP" on page 41

DATE functions for C

Note: The following DATE functionality is available only to users of IBM Informix ESQL/C. To use these functions, ESQL/C users will need to manually link their C code to the ESQL/C libraries.

The following date-manipulation functions are in the ESQL/C library. They convert dates between a string format and the internal DATE format.

Function Name	Description
rdatestr( )	Converts an internal DATE to a character string format
rdayofweek( )	Returns the day of the week of a date in internal format
rdefmtdate( )	Converts a specified string format to an internal DATE
rfmtdate( )	Converts an internal DATE to a specified string format
rjulmdy( )	Returns month, day, and year from a specified DATE
rleapyear( )	Determines whether the specified year is a leap year
rmdyjul( )	Returns an internal DATE from month, day, and year
rstrdate( )	Converts a character string format to an internal DATE
rtoday( )	Returns a system date as an internal DATE

Related reference

For more information about the individual functions, see the following: IBM Informix ESQL/C Programmer's Manual.

"BIGINT functions for C" on page 416
"DATETIME and INTERVAL functions for C"
"DECIMAL functions for C" on page 420
"Invoking a C Function from an EGL Program" on page 421

DATETIME and INTERVAL functions for C

Note: The following DATETIME and INTERVAL functionality is available only to users of IBM Informix ESQL/C. To use these functions, ESQL/C users will need to manually link their C code to the ESQL/C libraries.

The DATETIME and INTERVAL data types are internally represented with the **dtime_t** and **intrvl_t** structures, respectively. Information about these structures can be found in the header file **datetime.h**, which is included in the ESQL/C product. Include this file in all C source files that use any of the DATETIME and INTERVAL functions.

You must use the following ESQL/C library functions for the **datetime** and **interval** data types to perform all operations on those types of values.

Function Name	Description
dtaddinv( )	Adds an interval value to a datetime value
dtcurrent( )	Gets the current date and time
dtcvasc( )	Converts an ANSI-compliant character string to a datetime value
dtcvfmtasc( )	Converts a character string with a specified format to a datetime value
dtextend( )	Changes the qualifier of a datetime value
dtsub( )	Subtracts one datetime value from another
dsubinv()	Subtracts an interval value from a datetime value
dttoasc( )	Converts a datetime value to an ANSI-compliant character string
dttofmtasc( )	Converts a datetime value to a character string with a specified format
incvasc( )	Converts an ANSI-compliant character string to an interval value
incvfmtasc( )	Converts a character string with a specified format to an interval value
intoasc( )	Converts an interval value to an ANSI-compliant character string
intofmtasc( )	Converts an interval value to a character string with a specified format
invdivdbl( )	Divides an interval value by a numeric value
invdivinv( )	Divides an interval value by another interval value
invextend( )	Extends an interval value to a different interval qualifier
invmuldbl( )	Multiples an interval value by a numeric value

Related reference

For more information about the individual functions, see the following: IBM Informix ESQL/C Programmer's Manual.

"BIGINT functions for C" on page 416
"DATE functions for C" on page 418
"DECIMAL functions for C" on page 420
"Invoking a C Function from an EGL Program" on page 421

DECIMAL functions for C

Note: The following DECIMAL functionality is available only to users of IBM Informix ESQL/C. To use these functions, ESQL/C users will need to manually link their C code to the ESQL/C libraries.

The data type DECIMAL is a machine-independent method for representing numbers of up to 32 significant digits, with or without a decimal point, and with exponents in the range -128 to +126. ESQL/C provides routines that facilitate the conversion of DECIMAL-type numbers to and from every data type allowed in the C language. DECIMAL-type numbers consist of an exponent and a mantissa (or fractional part) in base 100. In normalized form, the first digit of the mantissa must be greater than zero.

The DECIMAL date type is internally represented with the **dec_t** structure. The **decimal** structure and the type definition **dec_t** can be found in the header file **decimal.h**, which is included in the ESQL/C product. Include this file in all C source files that use any of the decimal functions.

All operations on **decimal** type numbers must be performed using the following ESQL/C library functions for the **decimal** data type. Any other operations, modifications or analyses can produce unpredictable results.

Function Name	Description
deccvasc( )	Converts C **int1** type to DECIMAL type
dectoasc( )	Converts DECIMAL type to C **int1** type
deccvint( )	Converts C **int** type to DECIMAL type
dectoint( )	Converts DECIMAL type to C **int** type
deccvlong( )	Converts C **int4** type to DECIMAL type
dectolong( )	Converts DECIMAL type to C **int4** type
deccvflt( )	Converts C **float** type to DECIMAL type
dectoflt( )	Converts DECIMAL type to C **float** type
deccvdbl( )	Converts C **double** type to DECMAL type
dectodbl( )	Converts DECIMAL type to C **double** type
decadd( )	Adds two DECIMAL numbers
decsub( )	Subtracts two DECIMAL numbers
decmul( )	Multiplies two DECIMAL numbers
decdiv( )	Divides two DECIMAL numbers
deccmp( )	Compares two DECIMAL numbers
deccopy( )	Copies a DECIMAL number
dececvt( )	Converts DECIMAL value to ASCII string
decfcvt( )	Converts DECIMAL value to ASCII string

Related reference

For more information about the individual functions, see the following:
IBM Informix ESQL/C Programmer's Manual.
"BIGINT functions for C" on page 416
"DATE functions for C" on page 418

"DATETIME and INTERVAL functions for C" on page 418
"Invoking a C Function from an EGL Program"

Invoking a C Function from an EGL Program

You can invoke (or call) a C function from an EGL program. Prior to following the instructions below, you must compile and link your C code as identified in *C functions with EGL*.

To invoke a C function from an EGL program:

1. Using the *function invocation* statement, specify the following:
   - The name of the C function
   - Any arguments to pass to the C function
   - Any variables to return to the EGL program
2. Create an EGL native *library part* containing the function definition.
3. With the USE statement, specify the EGL native library part in the calling module.

For example, the following function invocation statement calls the C function **sendmsg( )**

```
sendmsg(chartype, 4, msg_status, return_code);
```

It passes two arguments (**chartype** and **4**, respectively) to the function and expects two arguments to be passed back (**msg_status** and **return_code**, respectively). This is made clear by defining the function in a native library as follows:

```
Library I4GLFunctions type nativeLibrary
   {callingConvention = "I4GL", dllName = "mydll"}
   Function sendmsg(chartype char(10) in, i int in, msg_status int out, return_code int out)
   end
end
```

The arguments passed are specified using the "in" parameter and the arguments to be returned are specified using the "out" parameter.

callingConvention
specifies that the arguments will be passed between functions and the calling code using the argument stack mechanism.

dllName
specifies the C shared library in which this function exists.

Note: The C shared library name can also be specified using the *vgj.defaultI4GLNativeLibrary* system property. If both *dllName* and the system property have been specified, the *dllName* will be used. For more information about the EGL nativeLibrary, see the *Library part of type nativeLibrary* help topic.

The C function receives an integer argument that specifies how many values were pushed on the argument stack (in this case, two arguments). This is the number of values to be popped off the stack in the C function. The function also needs to return values for the **msg_status** and **return_code** arguments before passing control back to the EGL program. The pop external functions are described in *Receiving values from EGL*; the return external functions are described in *Returning values to EGL*.

The C function should not assume it has been passed the correct number of stacked values. The C function should test its integer argument to see how many EGL arguments were stacked for it.

This example shows a C function that requires exactly one argument:

```
int nxt_bus_day(int nargs);
{
    int theDate;
    if (nargs != 1)
    {
       fprintf(stderr,
          "nxt_bus_day: wrong number of parms (%d)\n",
          nargs );
       ibm_lib4gl_returnDate(0L);
       return(1);
    }
    ibm_lib4gl_popDate(&theDate);
    switch(rdayofweek(theDate))
    {
    case 5: /* change friday -> monday */
          ++theDate;
    case 6: /* saturday -> monday*/
          ++theDate;
    default: /* (sun..thur) go to next day */
          ++theDate;
    }
    ibm_lib4gl_returnDate(theDate); /* stack result */
    return(1) /* return count of stacked */
 }
```

The function returns the date of the next business day after a given date. Because the function must receive exactly one argument, the function checks for the number of arguments passed. If the function receives a different number of arguments, it terminates the program (with an identifying message).

Related reference
"BIGINT functions for C" on page 416
"C data types and EGL primitive types" on page 417
"Creating an EGL library part" on page 132
"DATE functions for C" on page 418
"DATETIME and INTERVAL functions for C" on page 418
"DECIMAL functions for C" on page 420
"Function invocations" on page 504
"Library part of type basicLibrary" on page 133
"Stack functions for C"
"Return functions for C" on page 425
"C functions with EGL" on page 413

Stack functions for C

To call a C function, EGL uses an *argument stack*, a mechanism that passes arguments between the functions and the calling code. The EGL calling function pushes its arguments onto the stack and the called C function pops them off of the stack to use the values. The called function pushes its return values onto the stack and the caller pops them off to retrieve the values. The pop and return external functions are provided with the argument stack library. The pop external functions are described below according to the data type of the value that each pops from the argument stack. The return external functions are described in *Return functions for C*.

Note: The pop functions were originally used with IBM Informix 4GL (I4GL); hence the inclusion of ″4gl″ in the function names.

Library functions for returning values

You can call the following library functions from a C function to pop number values from the argument stack:

- extern void ibm_lib4gl_popMInt(int *iv)
- extern void ibm_lib4gl_popInt2(short *siv)
- extern void ibm_lib4gl_popInt4(int *liv)
- extern void ibm_lib4gl_popFloat(float *fv)
- extern void ibm_lib4gl_popDouble(double *dfv)
- extern void ibm_lib4gl_popDecimal(dec_t *decv)
- extern void ibm_lib4gl_popInt8(ifx_int8_t *bi)

The following table and similar tables below map the return function names between I4GL pre-Version 7.31 and Version 7.31 and later:

Pre-Version 7.31 name	Version 7.31 and later name
popint	ibm_lib4gl_popMInt
popshort	ibm_lib4gl_popInt2
poplong	ibm_lib4gl_popInt4
popflo	ibm_lib4gl_popFloat
popdub	ibm_lib4gl_popDouble
popdec	ibm_lib4gl_popDecimal

Each of these functions, like all library functions for popping values, performs the following actions:

1. Removes one value from the argument stack.
2. Converts its data type if necessary. If the value on the stack cannot be converted to the specified type, an error occurs.
3. Copies the value to the designated variable.

The structure types **dec_t** and **ifx_int8_t** are used to represent DECIMAL and BIGINT data in a C program. For more information about the **dec_t** and **ifx_int8_t** structure types and library functions for manipulating and printing DECIMAL and BIGINT variables, see the *IBM Informix ESQL/C Programmer's Manual*.

Library Functions for Popping Character Strings

You can call the following library functions to pop character values:

- extern void ibm_lib4gl_popQuotedStr(char *qv, int len)
- extern void ibm_lib4gl_popString(char *qv, int len)
- extern void ibm_lib4gl_popVarChar(char *qv, int len)

Pre-Version 7.31 name	Version 7.31 and later name
popquote	ibm_lib4gl_popQuotedStr
popstring	ibm_lib4gl_popString
popvchar	ibm_lib4gl_popVarChar

Both **ibm_lib4gl_popQuotedStr( )** and **ibm_lib4gl_popVarChar( )** copy exactly **len** bytes into the string buffer ***qv**. Here **ibm_lib4gl_popQuotedStr( )** pads with spaces as necessary, but **ibm_lib4gl_popVarChar( )** does not pad to the full length. The final byte copied to the buffer is a null byte to terminate the string, so the maximum string data length is **len**-1. If the stacked argument is longer than **len**-1, its trailing bytes are lost.

The **len** argument sets the maximum size of the receiving string buffer. Using **ibm_lib4gl_popQuotedStr( )**, you receive exactly **len** bytes (including trailing blank spaces and the null), even if the value on the stack is an empty string. To find the true data length of a string retrieved by **ibm_lib4gl_popQuotedStr( )**, you must trim trailing spaces from the popped value.

Note: The functions **ibm_lib4gl_popString( )** and **ibm_lib4gl_popQuotedStr( )** are identical, except that **ibm_lib4gl_popString( )** automatically trims any trailing blanks.

Library Functions for Popping Time Values

You can call the following library functions to pop DATE, INTERVAL, and DATETIME (TIMESTAMP) values:

- extern void ibm_lib4gl_popDate(int *datv)
- extern void ibm_lib4gl_popInterval(intrvl_t *iv, int qual)

You can call the following library function to pop TIMESTAMP values:

- extern void ibm_lib4gl_popDateTime(dtime_t *dtv, int qual)

Pre-Version 7.31 name	Version 7.31 and later name
popdate	ibm_lib4gl_popDate
popdtime	ibm_lib4gl_popDateTime
popinv	ibm_lib4gl_popInterval

The structure types **dtime_t** and **intrvl_t** are used to represent DATETIME and INTERVAL data in a C program. The **qual** argument receives the binary representation of the DATETIME or INTERVAL qualifier. For more information about the **dtime_t** and **intrvl_t** structure types and library functions for manipulating and printing DATE, DATETIME, and INTERVAL variables, see the *IBM Informix ESQL/C Programmer's Manual*.

Library Functions for Popping BYTE or TEXT Values

You can call the following function to pop a BYTE or TEXT argument:

- extern void ibm_lib4gl_popBlobLocator(loc_t **blob)

Pre-Version 7.31 name	Version 7.31 and later name
poplocator	ibm_lib4gl_popBlobLocator

The structure type **loc_t** defines a BYTE or TEXT value, and is discussed in the *IBM Informix ESQL/C Programmer's Manual*.

Any BYTE or TEXT argument must be popped as BYTE or TEXT because EGL provides no automatic data type conversion.

Related reference
"BIGINT functions for C" on page 416
"C data types and EGL primitive types" on page 417
"C functions with EGL" on page 413
"DATE functions for C" on page 418
"DATETIME and INTERVAL functions for C" on page 418
"DECIMAL functions for C" on page 420
"Invoking a C Function from an EGL Program" on page 421
IBM Informix ESQL/C Programmer's Manual
"Return functions for C"

Return functions for C

To call a C function, EGL uses an *argument stack*, a mechanism that passes arguments between the functions and the calling code. The EGL calling function pushes its arguments onto the stack and the called C function pops them off of the stack to use the values. The called function pushes its return values onto the stack and the caller pops them off to retrieve the values. The pop and return external functions are provided with the argument stack library. The return external functions are described below; the pop external functions used are described in *Stack functions for C*.

The external return functions copy their arguments to storage allocated outside the calling function. This storage is released when the returned value is popped. This situation makes it possible to return values from local variables of the function.

Note: The return functions were originally used with IBM Informix 4GL (I4GL); hence the inclusion of "4gl" in the function names.

Library functions for returning values

The following library functions are available to return values:

- extern void ibm_lib4gl_returnMInt(int iv)
- extern void ibm_lib4gl_returnInt2(short siv)
- extern void ibm_lib4gl_returnInt4(int lv)
- extern void ibm_lib4gl_returnFloat(float *fv)
- extern void ibm_lib4gl_returnDouble(double *dfv)
- extern void ibm_lib4gl_returnDecimal(dec_t *decv)
- extern void ibm_lib4gl_returnQuotedStr(char *str0)
- extern void ibm_lib4gl_returnString(char *str0)
- extern void ibm_lib4gl_returnVarChar(char *vc)
- extern void ibm_lib4gl_returnDate(int date)
- extern void ibm_lib4gl_returnDateTime(dtime_t *dtv)
- extern void ibm_lib4gl_returnInterval(intrvl_t *inv)
- extern void ibm_lib4gl_returnInt8(ifx_int8_t *bi)

The following table maps the return function names between I4GL pre-Version 7.31 and Version 7.31 and later:

Pre-Version 7.31 name	Version 7.31 and later name
retint	ibm_lib4gl_returnMInt
retshort	ibm_lib4gl_returnInt2
retlong	ibm_lib4gl_returnInt4
retflo	ibm_lib4gl_returnFloat
retdub	ibm_lib4gl_returnDouble
retdec	ibm_lib4gl_returnDecimal
retquote	ibm_lib4gl_returnQuotedStr
retstring	ibm_lib4gl_returnString
retvchar	ibm_lib4gl_returnVarChar
retdate	ibm_lib4gl_returnDate
retdtime	ibm_lib4gl_returnDateTime
retinv	ibm_lib4gl_returnInterval

The argument of **ibm_lib4gl_returnQuotedStr( )** is a null-terminated string. The **ibm_lib4gl_returnString( )** function is included only for symmetry; it internally calls **ibm_lib4gl_returnQuotedStr( )**.

The C function can return data in whatever form is convenient. If conversion is possible, EGL converts the data type as required when popping the value. If data type conversion is not possible, an error occurs.

C functions called from EGL must always exit with the statement **return(***n***)**, where *n* is the number of return values pushed onto the stack. A function that returns nothing must exit with **return(0)**.

Related reference
"BIGINT functions for C" on page 416
"C data types and EGL primitive types" on page 417
"Invoking a C Function from an EGL Program" on page 421
"C functions with EGL" on page 413
"DATE functions for C" on page 418
"DATETIME and INTERVAL functions for C" on page 418
"DECIMAL functions for C" on page 420
"Stack functions for C" on page 422

COBOL reserved-word file

The COBOL reserved word file is a text file that contains reserved words other than EGL reserved words. When EGL generates a COBOL program and finds one of the listed words used as a function name, record name, structure name, or variable name, the name is aliased. If the name of an EGL item would cause the COBOL generator to create a COBOL variable matching a word in the file, the generator instead uses an alias that does not conflict with other variable names and does not match a word in the reserved-word file.

COBOL reserved words documented at the time the generator was developed are automatically reserved by the COBOL generator. You do not need to use a COBOL reserved-word file unless a new keyword is introduced by the COBOL compiler or unless you just have some words that you do not want used as COBOL variable names.

If you have added a new word to the COBOL reserved-word file, set the build descriptor option **reservedWord** to the fully qualified path name of that file.

Related concepts
"COBOL program" on page 306

Related reference
"Format of COBOL reserved-word file"
"How COBOL names are aliased" on page 648
"EGL reserved words" on page 474
"reservedWord" on page 381

Format of COBOL reserved-word file

A reserved-word file contains a list of reserved words that are in addition to the built-in COBOL reserved words. The format of the file is one reserved word per line. An EGL reserved-word file applies only to COBOL generation.

If you use a reserved word as the name of a program, the EGL generator exits with an error. If you use a reserved word as the name of another part, the generator aliases it.

Related concepts
"COBOL reserved-word file" on page 426

Related reference
"How COBOL names are aliased" on page 648

Comments

A *comment* in an EGL file is created in either of the following ways:

- Double right slashes (//) indicate that the subsequent characters are a comment, up to and including the end-of-line character
- A single or multiline comment is delimited by a right slash and asterisk at the start (/*) and by an asterisk and right slash at the end (*/); this form of comment is valid anywhere that a white-space character is valid

You may place a comment inside or outside of an executable statement, as in this example:

```
/* the assignment e = f occurs if a == b or if c == d */
if (a == b         // one comparison
  || /* OR; another comparison */ c == d)
  e = f;
end
```

EGL does not support embedded comments, so the following entries cause an error:

```
/* this line starts a comment /* and
   this line ends the comment, */
   but this line is not inside a comment at all */
```

The comment in the first two lines includes a second comment delimiter (/*). An error results only when EGL tries to interpret the third line as source code.

The following is valid:

```
a = b;   /* this line starts a comment // and
   this line ends the comment */
```

The double right slashes (//) in the last example are themselves part of a larger comment.

Between the symbols #sql{ and }, the EGL comments described earlier are not valid. The following statements apply:

- An SQL comment begins with a double hyphen (--) at the beginning of a line or after white space and continues until the end of the line
- Comments are not available inside a string literal. A series of characters in that literal is interpreted as text even in these contexts:
  - A prepare statement
  - The **defaultSelectCondition** property of a record of type SQLRecord

Related concepts
"EGL projects, packages, and files" on page 13

Related reference
"EGL source format" on page 478
"EGL statements" on page 83

Compatibility with VisualAge Generator

EGL is the replacement for VisualAge Generator 4.5 and includes some syntax primarily to enable the migration of existing programs to the new development environment. This syntax is supported in the development environment if the EGL preference **VAGCompatibility** is selected or (at generation or debug time) if the build descriptor option **VAGCompatibility** is set to *yes*. The setting of the preference also establishes the default value of the build descriptor option.

The following statements apply when VisualAge Generator compatibility is in effect:

- Three otherwise invalid characters (- @ #) are valid in identifiers, although the hypen (-) and pound sign (#) are each invalid as an initial character in any case; for details, see *Naming conventions*
- If you refer to a static, single-dimension array of structure items without specifying an index, the array index defaults to 1; for details, see *Arrays*
- The primitive types NUMC and PACF are available, as described in *Primitive types*
- If you specify an even length for an item of primitive type DECIMAL, EGL increments the length by one except when the item is used as an SQL host variable.
- The SQL item property **SQLDataCode** is available, as described in *SQL item properties*
- A set of call options are available in the call statement
- The option **externallyDefined** is in the statements show and transfer
- The following system variables are available:
  - VGVar.handleSysLibraryErrors
  - ConverseVar.segmentedMode
- The following system functions are available:
  - VGLib.getVAGSysType
  - VGLib.connectionService
- You can issue a statement of the following form:

```
display printForm
```

printForm
Name of a print form that is visible to the program.

In that case, **display** is equivalent to print.

- The following program properties are available in all cases and are especially useful for code that was written in VisualAge Generator:
  - **allowUnqualifiedItemReferences**
  - **handleHardIOErrors** (when set to *no*)
  - **includeReferencedFunctions**
  - **localSQLScope** (when set to *yes*)
  - **throwNrfEofExceptions** (when set to *yes*)

  For details, see *Program part in EGL source format*.
- If you set the text-form property **value**, the content of that property is available in the program only after the user has returned the form. For this reason, the value that you set in the program does not need to be valid for the item in the program.

For access to full details on migrating VisualAge Generator programs to EGL, see *Sources of additional information on EGL*.

Related concepts
"Sources of additional information on EGL" on page 12

Related reference
"Arrays" on page 69
"call" on page 547
"Input form" on page 715
"Input record" on page 715
"Naming conventions" on page 652
"pfKeyEquate" on page 666
"Primitive types" on page 31
"print" on page 613
"Program part in EGL source format" on page 707
"show" on page 626
"SQL item properties" on page 63
"connectionService()" on page 888
"getVAGSysType()" on page 892
"handleSysLibraryErrors" on page 922
"segmentedMode" on page 898
"transfer" on page 627

ConsoleUI

ConsoleField properties and fields

The following properties are required in a variable of type ConsoleField:

- **fieldLen** (unless the ConsoleField is a constant field)
- **position**

The **name** field is also required, although not in a constant ConsoleField.

The properties of ConsoleField are as follows:

fieldLen
Specifies the number of positions needed to display the largest value of interest. For constant consoleFields, you do not set this property: **fieldLen** is the number of characters occupied by the displayed value, as included in the **value** property.

Type: *INT*

Example: *fieldLen = 20*

Default: *none*

position
The location of the console field within the form. The property contains an array of two positive integers: the line number followed by the column number. The line number is calculated from the top of the form. Similarly, the column number is calculated from the left of the form.

Type: *INT[]*

Example: *position = [2, 3]*

Default: *[1,1]*

segments
Specifies the row, column, and length of each *field segment*, which is a consoleField subsection that can have delimiters. To create the appearance of a multiline text box, you stack one field segment on successive lines at the same form column, and the collection of segments acts as one field.

Type: *INT[3][]*

Example: *segments = [[5,1,10],[6,1,10]]*

Default: *none*

If a value is specified for **segments**, the value for **position** is ignored, and **fieldLen** should be set to the length of all segments combined.

If you specify multiple segments, the behavior of the ConsoleField is also affected by the value of the **lineWrap** field.

validValues
Specifies the list of values that are valid for user input.

Type: *Array literal of singular and two-value elements*

Example: *validValues = [[1,3], 5, 12]*

Default: *none*

For details, see *validValues*.

The properties of a consoleField array include the previous ones (except for **segments**), as well as these:

columns
Specifies the number of columns in which to display the elements in an array of type ConsoleField. If the array has five elements and the value of the **columns** property is two, for example, the first line of the form shows two elements; the second line shows two elements; and the third line shows one element.

Type: *INT*

Example: *columns = 3*

Default: *1*

This property is meaningful only for arrays of type ConsoleField. The distribution of array elements on screen (whether across or up and down) is affected by the property **orientIndexAcross**.

linesBetweenRows
Specifies the number of blank lines between each line that contains an array element.

Type: *INT*

Example: *linesBetweenRows = 3*

Default: *0*

This property is meaningful only for arrays of type ConsoleField.

orientIndexAcross
Indicates whether the distribution of array elements is across the screen, as shown in a later example.

Type: *Boolean*

Example: *orientIndexAcross = yes*

Default: *yes*

This property is meaningful only for arrays of type consoleField.

If the property **orientIndexAcross** is set to *yes*, successive elements of the array are displayed from left to right. In the following, 2-column example, each successive element displays an integer that is equivalent to the element index:

```
1   2
3   4
5
```

If the property **orientIndexAcross** is set to *no*, the successive elements are displayed from top to bottom:

```
1   4
2   5
3
```

spacesBetweenColumns
Specifies the number of spaces separating each column of fields.

Type: *INT*

Example: *spacesBetweenColumns = 3*

Default: *1*

This property is valid only for arrays of type consoleField.

The fields of ConsoleField are as follows:

align
The **align** field specifies the position of data in a variable field when the length of the data is smaller than the length of the field.

Type: *AlignKind*

Example: *align = left*

Default: *left for character or timestamp data, right for numeric*

Updatable at run time? *Yes*

Values are as follows:

left
Place the data at the left of the field. Initial spaces are stripped and placed at the end of the field.

none
Do not justify the data. This setting is valid only for character data.

right
Place the data at the right of the field. Trailing spaces are stripped and placed at the beginning of the field.

autonext
Indicates whether, after the user fills the current ConsoleField, the cursor goes to the next field.

Type: *Boolean*

Example: *autonext = yes*

Default: *None*

Updatable at run time? *Yes*

The tab order determines which ConsoleField is next, as described in *ConsoleUI parts and related variables*.

binding
Specifies the name of the variable to which the ConsoleField is bound by default.

Type: *String*

Example: *binding = "myVar"*

Default: *None*

Updatable at run time? *No*.

For on overview of binding, see *ConsoleUI parts and related variables*.

caseFormat
Specifies how to treat input and output in relation to case sensitivity.

Type: *CaseFormatKind*

Example: *caseFormat = lowerCase*

Default: *defaultCase*

Updatable at run time? *Yes*

Values are as follows:

defaultCase (the default)
Has no effect on case

lowerCase
Transforms characters to lowercase, as possible

upperCase
Transforms characters to uppercase, as possible

color
Specifies the color of the text in the ConsoleField.

Type: *ColorKind*

Example: *color = red*

Default: *white*

Updatable at run time? *Yes, but the update has a visual effect only if the ConsoleField is displayed (or obtains focus) after the field is updated*

Values are as follows:

defaultColor or white (the default)
White

black
Black

blue
Blue

cyan
Cyan

green
Green

magenta
Magenta

red
Red

yellow
Yellow

comment
Specifies the *comment*, which is the text displayed in the Window-specific comment line (if any) when the cursor is in the ConsoleField.

Type: *String*

Example: *"Employee name"*

Default: *Empty string*

Updatable at run time? *No*

commentKey
Specifies a key used to search the resource bundle that includes the *comment*, which is the text displayed in the Window-specific comment line (if any) when the cursor is in the ConsoleField. If you specify both **comment** and **commentKey**, **comment** is used.

Type: *String*

Example: *commentKey = "myKey"*

Default: *Empty string*

Updatable at run time? *No*

The resource bundle is identified by the system variable **ConsoleLib.messageResource**, as described in *messageResource*.

dataType
Specifies a string to identify a data type. The value is used to validate that user input (such as = *1.5*) is compatible with a particular kind of SQL column. The field is meaningful only when the **openUI** statement for the ConsoleField (or related ConsoleForm) includes the statement property **isConstruct**.

Type: *String*

Example: *dataType = "NUMBER"*

Default: *Empty string*

Updatable at run time? *No*

In relation to numeric input, specify the value *"NUMBER"* if you allow the user to specify a floating point value (in which case, *> 1.5* is valid user input); otherwise, specify the string equivalent of an integer; for example, *"INT"*.

dateFormat

Indicates how to format output; but specify **dateFormat** only if the ConsoleField accepts a date.

Type: *a String or date-related system constant*

Example: *dateFormat = isoDateFormat*

Default: *none*

Updatable at run time? *No*

Valid values are as follows:

"pattern"
The value of *pattern* consists of a set of characters, as described in *Date, time, and timestamp format specifiers*.

Characters may be dropped from the beginning or end of a complete date specification, but not from the middle.

defaultDateFormat
The date format given in the run-time Java locale.

isoDateFormat
The pattern "yyyy-MM-dd", which is the date format specified by the International Standards Organization (ISO).

usaDateFormat
The pattern "MM/dd/yyyy", which is the IBM USA standard date format.

eurDateFormat
The pattern "dd.MM.yyyy", which is the IBM European standard date format.

jisDateFormat
The pattern "yyyy-MM-dd", which is the Japanese Industrial Standard date format.

systemGregorianDateFormat
An 8- or 10-character pattern that includes dd (for numeric day of the month), MM (for numeric month), and yy or yyyy (for numeric year), with characters other than d, M, y, or digits used as separators.

The format is in this Java run-time property:

```
vgj.datemask.gregorian.long.NLS
```

NLS
The NLS (national language support) code that is specified in the Java run-time property **vgj.nls.code**. The code is one of those listed in targetNLS. Uppercase English (code ENP) is not supported.

For additional details on **vgj.nls.code**, see *Java run-time properties (details)*.

systemJulianDateFormat
A 6- or 8-character pattern that includes DDD (for numeric day of the year) and yy or yyyy (for numeric year), with characters other than D, y, or digits used as separators.

The format is in this Java run-time property:

```
vgj.datemask.julian.long.NLS
```

NLS
The NLS (national language support) code that is specified in the Java run-time property **vgj.nls.code**. The code is one of those listed in targetNLS. Uppercase English (code ENP) is not supported.

For additional details on **vgj.nls.code**, see *Java run-time properties (details)*.

editor
Specifies the program for user interaction with the data; but is meaningful only if the ConsoleField is bound to a variable of type LOB.

Type: *String*

Example: *editor = "/bin/vi"*

Default: *none*

Updatable at run time? *Yes*

You can specify the name of an executable found in the PATH or LIBPATH; alternatively, you can specify the fully qualified path of that executable.

help
Specifies the text to display when the following situation is in effect:

- The cursor is in the ConsoleField; and
- The user presses the key identified in **ConsoleLib.key_help**.

Type: *String*

Example: help = *"Update the value"*

Default: *Empty string*

Updatable at run time? *Yes*

helpKey
Specifies an access key for searching the resource bundle that contains text for display when the following situation is in effect:

- The cursor is in the ConsoleField; and
- The user presses the key identified in **ConsoleLib.key_help**.

If you specify both **help** and **helpKey**, **help** is used.

Type: *String*

Example: *helpKey = "myKey"*

Default: *Empty string*

Updatable at run time? *Yes*

The resource bundle is identified by the system variable **ConsoleLib.messageResource**, as described in *messageResource*.

highlight
Specifies the special effects (if any) that are used when displaying the ConsoleField.

Type: *HighlightKind[]*

Example: *highlight = [reverse, underline]*

Default: *[noHighLight]*

Updatable at run time? *Yes, but the update has a visual effect only if the ConsoleField is displayed (or obtains focus) after the* ***highlight*** *field is updated*

Values are as follows:

noHighlight (the default)
Causes no special effect. Use of this value overrides any other.

blink
Has no effect.

reverse
Reverses the text and background colors so that (for example) if the display has a black background with white letters, the background becomes white and the text becomes black.

underline
Places an underline under the affected areas. The color of the underline is the color of the text, even if the value **reverse** is also specified.

initialValue
Specifies the initial value for display.

Type: *String*

Example: *initialValue = "200"*

Default: *Empty string*

Updatable at run time? *Yes*

If the **setInitial** property in the **openUI** statement is set to true, the value of the **initialValue** property in the consoleField is used. If that openUI property is false, however, current values of bound variables are shown instead, and the value of the **initialValue** property is ignored.

initialValueKey
Specifies an access key for searching the resource bundle that contains the initial value for display. If you specify both **initialValue** and **initialValueKey**, **initialValue** is used.

Type: *String*

Example: *initialValueKey = "myKey"*

Default: *Empty string*

Updatable at run time? *Yes*

The resource bundle is identified by the system variable **ConsoleLib.messageResource**, as described in *messageResource*.

inputRequired
Indicates whether the user will be prevented from navigating away from the field without entering a value.

If you specify both **initialValue** and **initialValueKey**, **initialValue** is used.

Type: *String*

Example: *initialValueKey = "myKey"*

Default: *Empty string*

Updatable at run time? *No*

The resource bundle is identified by the system variable **ConsoleLib.messageResource**, as described in *messageResource*.

intensity
Specifies the strength of the displayed font.

Type: *IntensityKind[]*

Example: *intensity = [bold]*

Default: *[normalIntensity]*

Updatable at run time? *Yes, but the update has a visual effect only if the ConsoleField is displayed (or obtains focus) after the **intensity** field is updated*

Values are as follows:

normalIntensity (the default)
Causes no special effect. Use of this value overrides any other.

bold
Causes the text to appear in a bold-weight font.

dim
Has no effect at this time. In future, may cause the text to appear with a lessened intensity, as appropriate when an input field is either disabled or should be deemphasized.

invisible
Removes any indication that the field is on the form.

isBoolean
Indicates whether the ConsoleField represents a Boolean value. The field **isBoolean** restricts the valid ConsoleField values and is useful for input or output.

The value of a numeric field is 0 (for false) or 1 (for true).

The value of a character field is represented by a word or subset of a word that is national-language dependent, and the specific values are determined by the locale. In English, for example, a boolean field of three or more characters has the value *yes* (for true) or *no* (for false), and a one-character boolean field value has the truncated value *y* or *n*.

Type: *Boolean*

Example: *isBoolean = yes*

Default: *no*

Updatable at run time? *No*

lineWrap
Indicates how to wrap text onto a new line whenever wrapping is necessary to avoid truncating text.

Type: *LineWrapType*

Example: *value = compress*

Default: *character*

Updatable at run time? *Yes*

Values are as follows:

character (the default)
The text in a field will not be split at a white space, but at the character position where the boundary of the field segment is.

compress
If possible, the text will be split at a white space. When the user leaves the consoleField (either by navigating to another consoleField or by pressing **Esc**), the value is assigned to the bound variable, and any additional spaces that are used to wrap text are removed.

word
If possible, the text in a field will be split at a white space. When the value is assigned to the bound variable, additional spaces are included to reflect how the value was padded to wrap at word boundaries.

The **lineWrap** field is meaningful only for a ConsoleField that has multiple segments, as is controlled by the **segments** property.

masked
Indicates whether each character in the ConsoleField is displayed as an asterisk (*), as is appropriate when the user types a password.

Type: *Boolean*

Example: *masked = yes*

Default: *no*

Updatable at run time? *Yes*

minimumInput
Indicates the minimum number of characters in valid input.

Type: *INT*

Example: *minimumInput = 4*

Default: *no*

Updatable at run time? *No*

name
ConsoleField name, as used in a programming context in which the name is resolved at run time. It is strongly recommended that the value of the name field be the same as the name of the variable.

Type: *String*

Example: *name = "myField"*

Default: *none*

Updatable at run time? *No*

numericFormat

Indicates how to format output; but specify **numericFormat** only if the ConsoleField accepts a number.

Type: *String*

Example: *numericFormat = "-###@"*

Default: *none*

Updatable at run time? *No*

Valid characters are as follows:

A placeholder for a digit.

***** Use an asterisk (*) as the fill character for a leading zero.

& Use a zero as the fill character for a leading zero.

Use a space as the fill character for a leading zero.

< Left justify the number.

, Use a locale-dependent numeric separator unless the position contains a leading zero.

. Use a locale-dependent decimal point.

- Use a minus sign (-) for values less than 0; use a space for values greater than or equal to 0.

+ Use a minus sign for values less than 0; use a plus sign (+) for values greater than or equal to 0.

(Precede negative values with a left parenthesis, as appropriate in accounting.

) Place a right parenthesis after a negative value, as appropriate in accounting.

$ Precede the value with the locale-dependent currency symbol.

@ Place the locale-dependent currency symbol after the value.

pattern

Specifies the pattern for input and output formatting if the ConsoleField content is of a character type.

Type: *String*

Example: *pattern = "(###) ###-####"*

Default: *none*

Updatable at run time? *No*

These control characters are available:

- *A* is a placeholder for letters, and the subset of characters that are considered to be letters is dependent on the locale
- # is a placeholder for numeric digits
- *X* is a placeholder for a required character of any kind

Characters other than the preceding three are included in the input or output; but for output, any overlaid characters are lost:

- If the output pattern is "(###) ###-####", the value "6219655561212" is shown as follows:

  ```
  (219) 555-1212
  ```

  Each 6 in the original value is unavailable to the user and is lost if the data store is updated.
- For input, the cursor skips the literal charcters and only allows typing where the placeholder characters occur. In the current example, if the user types 2195551212, the string "(219) 555-1212" becomes the value within the ConsoleField and is the value placed in the bound variable.

protect

Specifies whether the ConsoleField is protected from user update.

Type: *Boolean*

Example: *protect = yes*

Default: *no*

Updatable at run time? *No*

Values are as follows:

No (the default)
Sets the field so that the user can overwrite the value in it.

Yes
Sets the consoleField so that the user cannot overwrite the value in it. In addition, the cursor skips the consoleField whenever the user attempts to navigate to it, as in these cases:

- The user is working on the previous consoleField in the tab order and either (a) presses **Tab** or (b) fills that previous consoleField with content when field **autonext** is set to yes.
- The user is working on the next consoleField in the tab order and presses **Shift Tab**.
- The user uses arrow keys to move to the next or previous consoleField.

You can bind a variable to a consoleField that is protected or not. The setting of the openUI property **setInitial** determines whether the value of the bound variable is displayed.

A runtime error occurs if the program tries to move to a consoleField that is protected.

SQLColumnName
Specifies the name of the database table column that is associated with the ConsoleField. The name is used to create search criteria when the **openUI** statement for the ConsoleField (or related ConsoleForm) includes the statement property **isConstruct**.

Type: *String*

Example: *SQLColumnName = "ID"*

Default: *none*

Updatable at run time? *Yes*

timeFormat

Indicates how to format output; but specify **timeFormat** only if the ConsoleField accepts a time.

Type: *a String or time-related system constant*

Example: *timeFormat = isoTimeFormat*

Default: *none*

Updatable at run time? *No*

Valid values are as follows:

"pattern"
The value of *pattern* consists of a set of characters, as described in *Date, time, and timestamp format specifiers*.

Characters may be dropped from the beginning or end of a complete time specification, but not from the middle.

defaultTimeFormat
The time format given in the run-time Java locale.

isoTimeFormat
The pattern "HH.mm.ss", which is the time format specified by the International Standards Organization (ISO).

usaTimeFormat
The pattern "hh:mm AM", which is the IBM USA standard time format.

eurTimeFormat
The pattern "HH.mm.ss", which is the IBM European standard time format.

jisTimeFormat
The pattern "HH:mm:ss", which is the Japanese Industrial Standard time format.

timestampFormat

Indicates how to format output; but specify **timestampFormat** only if the ConsoleField accepts a timestamp.

Type: *a String or timestamp-related system constant*

Example: *timestampFormat = odbcTimestampFormat*

Default: *none*

Updatable at run time? *No*

Valid values are as follows:

"pattern"
The value of *pattern* consists of a set of characters, as described in *Date, time, and timestamp format specifiers*.

Characters may be dropped from the beginning or end of a complete timestamp specification, but not from the middle.

defaultTimestampFormat
The timestamp format given in the run-time Java locale.

db2TimeStampFormat
The pattern "yyyy-MM-dd-HH.mm.ss.ffffff", which is the IBM DB2 default timestamp format.

odbcTimestampFormat
The pattern "yyyy-MM-dd HH:mm:ss.ffffff", which is the ODBC timestamp format.

value
The current value displayed in the consoleField. Your code can set this value so that invocation of **ConsoleLib.displayForm** displays the specified value in the consoleField.

Type: *String*

Example: *value = "View"*

Default: *none*

Updatable at run time? *Yes*

verify
Indicates whether the user is prompted to retype the same value after trying to exit the ConsoleField.

Type: *String*

Example: *value = "View"*

Default: *none*

Updatable at run time? *Yes*

Values are as follows:

No (the default)
EGL run time does not issue a special prompt.

Yes
When the user tries to leave the ConsoleField, EGL run time acts as follows:

- Clears the consoleField, keeping the cursor there
- Displays a message for the user to repeat the entry
- Compares the two input values when the user tries to leave the consoleField again

If the values match, the bound variable receives that value and processing continues as usual. If the values do not match, the consoleField content reverts to the value that preceded the first of the two user inputs, and the cursor remains in the field.

Related concepts
"Console user interface" on page 165

Related reference
"ConsoleUI parts and related variables" on page 167
"Date, time, and timestamp format specifiers" on page 42
"Java runtime properties (details)" on page 525"openUI" on page 602
"validValues" on page 701

Related task
"Creating an interface with consoleUI" on page 166

ConsoleForm properties in EGL consoleUI

The properties of a record part of type ConsoleForm are as follows, and only formSize is required:

delimiters
Specifies the characters that are displayed before and after input fields. The characters are displayed only if the value of property **showBrackets** is *yes*.

Type: *String literal*

Example: delimiters = *"<>/"*

Default: *"[]|"*

Wherever possible, the first character is displayed before each non-constant ConsoleField, and the second character is displayed after each non-constant ConsoleField. However, the third character is displayed between two non-constant ConsoleFields that are separated by a single position.

If you specify fewer than three characters, a default character is in effect for each unspecified character. If you specify more than three characters, the fourth and subsequent characters are ignored.

formSize
The dimensions of the form. The field must contain an array of two positive integers: the number of lines followed by the number of columns.

Type: *INT[2]*

Example: *size = [24, 80]*

Default: *none*

If either dimension exceeds the size of the window in which the form is displayed, the form size is reduced to fit the window dimensions. However, if a ConsoleField cannot fit into the window, the program ends.

name
Form name, as used in a programming context in which the name is resolved at run time. It is recommended that the value of the name field, if any, be the same as the name of the variable.

Type: *String*

Example: *name = "myForm"*

Default: *none*

The name field is used in system functions such as **ConsoleLib.displayFormByName**.

showBrackets
Indicates whether the non-constant ConsoleFields are delimited by a pair of characters such as brackets.

Type: *Boolean*

Example: *showBrackets = no*

Default: *yes*

For other details, see the property **delimiters**.

Related concepts
"Console user interface" on page 165

Related reference
"ConsoleUI parts and related variables" on page 167
"openUI" on page 602

Related task
"Creating an interface with consoleUI" on page 166

Menu fields in EGL consoleUI

The following list defines the fields in a variable of type Menu. You must specify the field **labelText** or **labelTextKey**.

labelText
The label that is displayed to the left of the list of menuItems.

Type: *String literal*

Example: *labelText = "Options: "*.

Default: *none*.

Updatable at run time? *No*

labelKey
Specifies a key for searching the resource bundle that contains the menu label. If you specify both **labelText** and **labelKey**, **labelText** is used.

Type: *String*

Example: *labelKey = "myKey"*

Default: *Empty string*

Updatable at run time? *No*

The resource bundle is identified by the system variable **ConsoleLib.messageResource**, as described in *messageResource*.

menuItems
An array of menu items, each of which is declared in the program or is created dynamically with the keyword **new**. For details on the second option, see *Use of new in consoleUI*.

Type: *MenuItem[]*

Example: *menuItems = [myItem, new MenuItem {name = "Remove", labelText = "Delete all"}]*.

Default: *none*.

Updatable at run time? *No*

You can add a menuItem in the program by using the following syntax:

```
myMenu.MenuItems.addElement(myMenuItem)
```

myMenu
Name of the variable of type Menu.

myMenuItem
Name of the variable of type MenuItem.

The program ends if you issue an **openUI**statement for a menu on which no menuItems exist.

Related concepts
"Console user interface" on page 165

Related reference
"Arrays" on page 69
"ConsoleUI parts and related variables" on page 167
"openUI" on page 602
"MenuItem fields in EGL consoleUI"
"Use of new in ConsoleUI" on page 170

Related task
"Creating an interface with consoleUI" on page 166

MenuItem fields in EGL consoleUI

The following list defines the consoleFields in a variable of type MenuItem. None of the consoleFields is required; you can determine the user's selection by setting any of three fields: **accelerators**, **labelText**, or **labelKey**.

accelerators
Indicates keystrokes that are equivalent to the user's selection of the menuItem. Each of those keystrokes causes execution of the **openUI** statement's OnEvent clause that corresponds to the menuItem selection.

Type: *String[]*

Example: *accelerators = ["F1", "ALT_F1"]*

Default: *none*

Updatable at run time? *No*

comment
Specifies the *comment*, which is the text displayed in the menuItem-specific comment line when the menuItem is selected.

Type: *String*

Example: *"Delete the record"*

Default: *Empty string*

Updatable at run time? *Yes*

The comment line is one beneath the menu line.

commentKey
Specifies a key used to search the resource bundle that includes the *comment*, which is text displayed in the menuItem-specific comment line (if any) when the menuItem is selected. If you specify both **comment** and **commentKey**, **comment** is used.

Type: *String*

Example: *commentKey = "myKey"*

Default: *Empty string*

Updatable at run time? *Yes*

The resource bundle is identified by the system variable **ConsoleLib.messageResource**, as described in *messageResource*.

help
Specifies the text to display when the following situation is in effect:

- The menuItem is selected; and
- The user presses the key identified in **ConsoleLib.key_help**.

Type: *String*

Example: help = *"Deletion is permanent"*

Default: *Empty string*

Updatable at run time? *Yes*

helpKey
Specifies an access key for searching the resource bundle that contains text for display when the following situation is in effect:

- The menuItem is selected; and
- The user presses the key identified in **ConsoleLib.key_help**.

If you specify both **help** and **helpKey**, **help** is used.

Type: *String*

Example: *helpKey = "myKey"*

Default: *Empty string*

Updatable at run time? *Yes*

The resource bundle is identified by the system variable **ConsoleLib.messageResource**, as described in *messageResource*.

labelText
The label that represents the menuItem.

Type: *String literal*

Example: *labelText = "Delete"*.

Default: *none*.

Updatable at run time? *No*

labelKey
Specifies a key for searching the resource bundle that contains the menuItem label. If you specify both **labelText** and **labelKey**, **labelText** is used.

Type: *String*

Example: *labelKey = "myKey"*

Default: *Empty string*

Updatable at run time? *No*

The resource bundle is identified by the system variable **ConsoleLib.messageResource**, as described in *messageResource*.

name
MenuItem name, as used in a programming context in which the name is resolved at run time. In particular, the name is used in the **openUI** statement that responds to the menuItem selection.

It is recommended that the value of the name field be the same as the name of the variable.

Type: *String*

Example: *name = "myItem"*

Default: *none*

Updatable at run time? *No*

Related concepts
"Console user interface" on page 165

Related reference
"ConsoleUI parts and related variables" on page 167
"Menu fields in EGL consoleUI" on page 443
"openUI" on page 602

Related task
"Creating an interface with consoleUI" on page 166

PresentationAttributes fields in EGL consoleUI

The following list defines the fields that you can set or retrieve in any system variable of type PresentationAttributes:

color
Specifies a color.

Type: *ColorKind*

Example: *color = red*

Default: *white*

Updatable at run time? *Yes, but the update has a visual effect only for output that is displayed after the color field is updated*

Values are as follows:

defaultColor or white (the default)
White

black
Black

blue
Blue

cyan
Cyan

green
Green

magenta
Magenta

red
Red

yellow
Yellow

highlight
Specifies the special effects (if any) that are used when displaying output.

Type: *HighlightKind[]*

Example: *highlight = [reverse, underline]*

Default: *[noHighLight]*

Updatable at run time? *Yes, but the update has a visual effect only for output that is displayed after the highlight field is updated*

Values are as follows:

noHighlight (the default)
Causes no special effect. Use of this value overrides any other.

blink
Has no effect at this time.

reverse
Reverses the text and background colors so that (for example) if the display has a black background with white letters, the background becomes white and the text becomes black.

underline
Places an underline under the affected areas. The color of the underline is the color of the text, even if the value **reverse** is also specified.

intensity
Specifies the strength of the displayed font.

Type: *IntensityKind[]*

Example: *intensity = [bold]*

Default: *[normalIntensity]*

Updatable at run time? *Yes, but the update has a visual effect only for output that is displayed after the intensity field is updated*

Values are as follows:

normalIntensity (the default)
Causes no special effect. Use of this value overrides any other.

bold
Causes the text to appear in a bold-weight font.

dim
Has no effect at this time. In future, may cause the text to appear with a lessened intensity, as appropriate when all input fields are disabled.

invisible
Removes any indication that the text is on the form.

Related concepts
"Console user interface" on page 165

Related reference
"currentDisplayAttrs" on page 746
"currentRowAttrs" on page 746
"defaultDisplayAttributes" on page 747
"defaultInputAttributes" on page 747
"ConsoleUI parts and related variables" on page 167
"openUI" on page 602

Related task
"Creating an interface with consoleUI" on page 166

Prompt fields in EGL consoleUI

The following list defines the fields in a variable of type Prompt. None of the fields is required.

isChar
Indicates whether, after the prompt is displayed, the user's first keystroke ends the operation.

Type: *Boolean*

Example: *isChar = yes*

Default: *no*

Updatable at run time? *Yes*

Values are as follows:

no (the default)
The operation ends when the user presses **Enter** or presses a key associated with an OnEvent clause of the **openUI** statement that displays the prompt. The variable to which the prompt is bound receives the input characters.

yes
The user's first keystroke ends the operation. The variable to which the prompt is bound receives the character, if the character is printable.

In either case, you can respond to a particular keystroke by setting an OnEvent clause of type ON_KEY.

message
Specifies the text that prompts the user.

Type: *String*

Example: *message = "Type here: "*

Default: *Empty string*

Updatable at run time? *Yes, before your code issues the **openUI** statement*

messageKey
Specifies a key used to search the resource bundle that includes the prompt text. If you specify both **message** and **messageKey**, **message** is used.

Type: *String*

Example: *messageKey = "promptText"*

Default: *Empty string*

Updatable at run time? *Yes*

The resource bundle is identified by the system variable **ConsoleLib.messageResource**, as described in *messageResource*.

responseAttr
Specifies the presentation attributes that are used when displaying user input.

Type: *PresentationAttributes literal*

Example: *responseAttr {color = green, highlight = [underline], intensity = [bold]}*

Default: *no*

Updatable at run time? *Yes*

This field has an effect only if the field **isChar** is set to *no*.

For details on **responseAttr** values, see *PresentationAttributes fields in EGL consoleUI*.

Related concepts
"Console user interface" on page 165

Related reference
"ConsoleUI parts and related variables" on page 167
"Java runtime properties (details)" on page 525
"messageResource" on page 759
"openUI" on page 602
"PresentationAttributes fields in EGL consoleUI" on page 446

Related task
"Creating an interface with consoleUI" on page 166

Window fields in EGL consoleUI

The following list defines the fields in a variable of type Window. None of the fields is required, but **size** is needed in practice.

color
Specifies the color that is used when displaying the following kinds of output in the window:

- Labels in consoleForms
- Input fields in prompts
- Window border
- Output of system functions such as **ConsoleLib.displayAtPosition**

  Type: *ColorKind*

  Example: *color = red*

  Default: *white*

  Updatable at run time? *Yes, but the update has a visual effect only if you open the window after the field is updated*

Values are as follows:

defaultColor or white (the default)
White

black
Black

blue
Blue

cyan
Cyan

green
Green

magenta
Magenta

red
Red

yellow
Yellow

commentLine
Sets the number of the line at which a comment (if any) is displayed if the Window field **hasCommentLine** is set to *yes*. The line number is calculated from the top of the screen window's content area (in which case the first line is 1) or (if the value is negative) from the bottom of that area (in which case the last line is -1, the second-to-last is -2, and so on).

Type: *INT*

Example: *commentLine = 10*

Default: *Last line of the window (although if only the screen window is open, the comment is on the second to last line of that window)*

Updatable at run time? *Yes, but the update has a visual effect only if you open the window after the field is updated*

The validity of the value is determined only at run time.

formLine

Sets the number of the line at which forms are displayed. The line number is calculated from the top of the screen window's content area (in which case the first line is 1) or (if the value is negative) from the bottom of that area (in which case the last line is -1, the second-to-last is -2, and so on).

Type: *INT*

Example: *formLine = 8*

Default: *3*

Updatable at run time? *Yes, but the update has a visual effect only if the window is displayed after the field is updated*

The validity of the value is determined only at run time.

hasBorder

Indicates whether the window is surrounded by a border. If the value is *yes*, the color of the border is specified in the Window field *color*.

Type: *Boolean*

Example: *hasBorder = yes*

Default: *no*

Updatable at run time? *Yes, but the update has a visual effect only if you open the window after the field is updated*

hasCommentLine

Indicates whether the window reserves a line for *comments*, which are text entries that are displayed when the cursor enters a consoleField. If the value is *yes*, the line number is specified in the Window field *commentLine*.

Type: *Boolean*

Example: *hasCommentLine = yes*

Default: *no*

Updatable at run time? *Yes, but the update has a visual effect only if you open the window after the field is updated*

highlight

Specifies the special effects (if any) that are used when displaying the following kinds of output in the window:

- Labels in consoleForms
- Input fields in prompts
- Window border
- Output of system functions such as **ConsoleLib.displayAtPosition**

Type: *HighlightKind[]*

Example: *highlight = [reverse, underline]*

Default: *[noHighLight]*

Updatable at run time? *Yes, but the update has a visual effect only if the window is displayed after the field is updated*

Values are as follows:

noHighlight (the default)
Causes no special effect. Use of this value overrides any other.

blink
Has no effect at this time.

reverse
Reverses the text and background colors so that (for example) if the display has a black background with white letters, the background becomes white and the text becomes black.

underline
Places an underline under the affected areas. The color of the underline is the color of the text, even if the color of the text has been reversed because you also specified the value **Reverse**.

intensity
Specifies the strength of the displayed font that is used when displaying the following kinds of output in the window:

- Labels in consoleForms
- Input fields in prompts
- Window border
- Output of system functions such as **ConsoleLib.displayAtPosition**

Type: *IntensityKind[]*

Example: *intensity = [bold]*

Default: *[normalIntensity]*

Updatable at run time? *Yes, but the update has a visual effect only if you open the window after the field is updated*

Values are as follows:

normalIntensity (the default)
Causes no special effect. Use of this value overrides any other.

bold
Causes the text to appear in a bold-weight font.

dim
Has no effect at this time. In future, may cause the text to appear with a lessened intensity, as appropriate when all input fields are disabled.

invisible
Removes any indication that the field is on the form.

menuLine
Sets the number of the line at which a menu (if any) is displayed in the Window. The line number is calculated from the top of the screen window's content area (in which case the first line is 1) or (if the value is negative) from the bottom of that area (in which case the last line is -1, the second-to-last is -2, and so on).

Type: *INT*

Example: *menuLine = 2*

Default: *1*

Updatable at run time? *Yes, but the update has a visual effect only if you open the window after the field is updated*

The validity of the value is determined only at run time.

messageLine
Sets the number of the line at which a message (if any) is displayed in the Window. The line number is calculated from the top of the screen window's content area (in which case the first line is 1) or (if the value is negative) from the bottom of that area (in which case the last line is -1, the second-to-last is -2, and so on).

Type: *INT*

Example: *messageLine = 3*

Default: *2*

Updatable at run time? *Yes, but the update has a visual effect only if you open the window after the field is updated*

The validity of the value is determined only at run time.

name
Window name, as used in a programming context in which the name is resolved at run time. It is recommended that the value of the name field be the same as the name of the variable.

Type: *String*

Example: *name = "myWindow"*

Default: *none*

Updatable at run time? *No*

position
The location of the top left corner of the window within the content area of the screen window. The field contains an array of two integers: the line number followed by the column number. The line number is calculated from the top of the screen window's content area (in which case the first line is 1) or (if the value is negative) from the bottom of that area (in which case the last line is -1, the second-to-last is -2, and so on). The column number is calculated from the left of the console window's content area, and the first column is 1.

Type: *INT[2]*

Example: *position = [2, 3]*

Default: *[1,1]*

Updatable at run time? *No*

promptLine
Sets the number of the line at which a prompt (if any) is displayed in the Window. The line number is calculated from the top of the console window's content area or (if the value is negative) from the bottom of that area.

Type: *INT*

Example: *promptLine = 4*

Default: *1*

Updatable at run time? *Yes, but the update has a visual effect only if you open the window after the field is updated*

The validity of the value is determined only at run time.

size
An array of two positive integers that represent window dimensions: the number of lines followed by the number of columns.

Type: *INT[2]*

Example: *size = [24, 80]*

Default: *none*

Updatable at run time? *No*

A value is required for practical purposes. If you display a window that lacks a value for **size**, the run time presents a window that is too small for content.

If either dimension exceeds the size available in the content area of the screen window, an error occurs at run time.

Related concepts
"Console user interface" on page 165

Related reference
"ConsoleUI parts and related variables" on page 167
"openUI" on page 602

Related task
"Creating an interface with consoleUI" on page 166

containerContextDependent

The function part property **containerContextDependent** allows you to extend the name space that is used to resolve function references from within the function part that includes the property. Valid values are *no* (the default) and *yes*.

It is recommended that you avoid using this capability when you develop new code. The property is primarily available for migrating programs from VisualAge Generator. If you set this property to *yes*, however, the implications are as follows:

- If the usual steps of a name search do not resolve a reference at editing time, the EGL editor does not flag the unresolved references as errors.
- If the usual steps of a name search do not resolve a reference at generation time, the search continues by reviewing the name space of the program, library, or PageHandler that contains the function part.
- If you have declared a function at the top level of an EGL source file rather than physically inside a container (a program, PageHandler, or library), that function can invoke library functions only if the following situation is in effect:
  - The container includes a use statement that refers to the library
  - In the invoking function, the property **containerContextDependent** is set to *yes*

Related concepts
"References to parts" on page 20

Related reference
"Function part in EGL source format" on page 513
]"Use declaration" on page 930

Database authorization and table names

An authorization ID is a character string that is passed to the database manager when a connection is established between the database manager and a program, whether the program prepares another program or allows end-user access to SQL tables. The character string is the user identifier that is required to check the database-access authorization held by the preparer or end user.

The source of the authorization ID depends on the system where database access occurs.

For iSeries COBOL programs, the authorization ID is the user ID under which the programs runs.

The situation for EGL-generated Java programs is as follows:

- The authorization ID is one obtained by the database manager when a connection was established between the database manager and the program:
  - In relation to the default database, the authorization ID is the value specified for the Java run-time property **vgj.jdbc.default.userid**
  - When you invoke the system function **sysLib.connect** or **VGLib.connectionService**, the authorization ID is the value specified for the userID parameter

The authorization ID may be used when you specify a table name. In that case, you can specify a table-name qualifier, in accordance with this syntax:

```
tableOwner.myTable
```

tableOwner
A qualifier that is known to the database manager and that is necessary to identify the table. The qualifier at table creation is the authorization ID of the person who created the table.

myTable
The table name.

If you do not include a table-name qualifier when you specify a table name, the table owner is resolved at run time. The qualifier is set to the authorization ID.

For more information on authorization IDs, consult your database manager documentation.

Related concepts

"Dynamic SQL" on page 224
"Java runtime properties" on page 327
"SQL support" on page 213

Related reference
"Java runtime properties (details)" on page 525
"SQL record part in EGL source format" on page 726
"connect()" on page 867
"connectionService()" on page 888

Data conversion

Because of differences in how data is interpreted in different run-time environments, your program may need to convert the data that passes from one environment to another. Data conversion occurs at COBOL preparation time and at COBOL or Java run time.

The COBOL preparation process converts file content, file-path information, and values of environment variables when transferring workstation-based files to a build server. The steps needed to establish a data conversion table in this case are described later.

Your code also uses a conversion table in the following run-time situations:

- Your EGL-generated Java code calls a program on CICS for z/OS.

  In this case, you can specify the conversion table in a callLink element that refers to the called program. Alternatively, you can indicate (in that callLink element) that the system variable sysVar.callConversionTable identifies the conversion table at run time.
- Your EGL-generated COBOL program calls a program residing on a remote platform that supports the ASCII character set.

  In this case, you also can specify the conversion table in a callLink element that refers to the called program. Alternatively, you can indicate (in that callLink element) that the system variable sysVar.callConversionTable identifies the conversion table at run time.
- An EGL-generated Java or COBOL program (on a platform that supports the EBCDIC character set) transfers asynchronously to a program on a platform that supports the ASCII character set, as might occur when the transferring program invokes the system function sysLib.startTransaction.

  In this case, you can specify the conversion table in a asynchLink element that refers to the program to which control is transferred. Alternatively, you can indicate (in that asynchLink element) that the system variable sysVar.callConversionTable identifies the conversion table at run time.
- An EGL-generated Java program shows a text or print form that includes series of Arabic or Hebrew characters; or presents a text form that accepts a series of such characters from the user.

  In these cases, you specify the bidirectional conversion table in the system variable sysVar.formConversionTable.

You would use run-time conversion, for example, if your code places values into one of two redefined records, each of which refers to the same area of memory as a record that is passed to another program. Assume that the characteristics of the data that you pass would be different, depending on the redefined record to which you assign values. In this case, the requirements of data conversion cannot be known at generation time.

The next sections provide the following details:

- "Data conversion when you generate a COBOL program"
- "Data conversion when the invoker is Java code" on page 456
- "Conversion algorithm" on page 457

Data conversion when you generate a COBOL program

When COBOL is generated on a workstation and prepared on an iSeries build server, conversion is handled on the build server in accordance with your specification in build descriptor options clientCodeSet and serverCodeSet. Each of those build descriptor options must identify a code set that is defined to the ICONV conversion service on iSeries, and default settings are used in the absence of a specification.

See also "Bidirectional language text" on page 458.

Data conversion when the invoker is Java code

The following rules pertain to Java code:

- When a generated Java program or wrapper invokes a generated Java program, conversion occurs in the caller, in accordance with a set of EGL classes invoked at run-time. It is sufficient to request no conversion at all in most cases, even if the caller is accessing a remote platform that uses a code page that is different from the code page used by the invoker. You must specify a conversion table, however, in the following situation:
  - The caller is Java code and is on a machine that supports one code page
  - The called program is non-Java and is on a machine that supports another code page

  The table name in this case is a symbol that indicates the kind of conversion that is required at run time.
- When a generated Java program accesses a remote MQSeries message queue, conversion occurs in the invoker, in accordance with a set of EGL classes invoked at run time. If the caller is accessing a remote platform that uses a code page that is different from the code page used by the invoker, specify a conversion table in the association element that refers to the MQSeries message queue.

The next table lists the conversion tables that can be accessed by generated Java code at run time. Each name has the format CSOcx:

c Represents the character set supported on the invoked platform. Select one of these:
- J for Java (if the called program is an EGL-generated Java program)
- E for EBCDIC (if the called platform is an EGL-generated COBOL program)

x Represents the code page number on the invoked platform. Each number is specified in the *Character Data Representation Architecture Reference and Registry*, SC09-2190. The registry identifies the coded character sets supported by the conversion tables.

	Platform of Invoked Program			
Language	**UNIX**	**Windows 2000/NT/XP**	**z/OS UNIX System Services or iSeries Java**	**iSeries COBOL**
Arabic	CSOJ1046	CSOJ1256	CSOJ420	CSOE420
Chinese, simplified	CSOJ1381	CSOJ1386	CSOJ1388	CSOE1388
Chinese, traditional	CSOJ950	CSOJ950	CSOJ1371	CSOE1371
Cyrillic	CSOJ866	CSOJ1251	CSOJ1025	CSOE1025
Danish	CSOJ850	CSOJ850	CSOJ277	CSOE277
Eastern European	CSOJ852	CSOJ1250	CSOJ870	CSOE870
English (UK)	CSOJ850	CSOJ1252	CSOJ285	CSOE285
English (US)	CSOJ850	CSOJ1252	CSOJ037	CSOE037
French	CSOJ850	CSOJ1252	CSOJ297	CSOE297
German	CSOJ850	CSOJ1252	CSOJ273	CSOE273
Greek	CSOJ813	CSOJ1253	CSOJ875	CSOE875

Language	Platform of Invoked Program			
	UNIX	Windows 2000/NT/XP	z/OS UNIX System Services or iSeries Java	iSeries COBOL
Hebrew	CSOJ856	CSOJ1255	CSOJ424	CSOE424
Japanese	CSOJ943	CSOJ943	CSOJ1390 (Katakana SBCS), CSOJ1399 (Latin SBCS)	CSOE1390 (Katakana SBCS), CSOE1399 (Latin SBCS)
Korean	CSOJ1363	CSOJ1363	CSOJ1364	CSOE1364
Portuguese	CSOJ850	CSOJ1252	CSOJ037	CSOE037
Spanish	CSOJ850	CSOJ1252	CSOJ284	CSOE284
Swedish	CSOJ850	CSOJ1252	CSOJ278	CSOE278
Swiss German	CSOJ850	CSOJ1252	CSOJ500	CSOE500
Turkish	CSOJ920	CSOJ1254	CSOJ1026	CSOE1026

If you do not specify a value for the conversion table in the linkage options part when you are calling a program from Java, the default conversion tables are those for English (US).

Conversion algorithm

Data conversion of records and structures is based on the declarations of the structure items that lack a substructure.

Data of type CHAR, DBCHAR, or MBCHAR is converted in accordance with the COBOL or Java conversion tables (for conversion that occurs in an EGL-generated invoker).

No conversion is performed for filler data items (data items that have no name) or for data items of type DECIMAL, PACF, HEX, or UNICODE.

On EBCDIC-to-ASCII conversion for MBCHAR data, the conversion routine deletes shift-out/shift-in (SO/SI) characters and inserts an equivalent number of blanks at the end of the data item. On ASCII-to-EBCDIC conversion, the conversion routine inserts SO/SI characters around double-byte strings and truncates the value at the last valid character that can fit in the field. If the MBCHAR field is in a variable length record and the current record end is in the MBCHAR field, the record length is adjusted to reflect the insertion or deletion of SO/SI characters. The record length indicates where the current record ends.

For data items of type BIN, the conversion routine reverses the byte order of the item if the caller or called platform uses Intel binary format and the other platform does not.

For data items of type NUM or NUMC items, the conversion routine converts all but the last byte using the CHAR algorithm. The sign half-byte (the first half byte of the last byte in the field) is converted according to the hexadecimal values shown in the next table.

EBCDIC for type NUM	EBCDIC for type NUMC	ASCII
F (positive sign)	C	3
D (negative sign)	D	7

Related reference
"Association elements" on page 352
"bidiConversionTable" on page 364
"Bidirectional language text"
"callLink element" on page 395
"clientCodeSet" on page 366

"serverCodeSet" on page 381
"convert()" on page 870
"callConversionTable" on page 902

Bidirectional language text

Bidirectional (bidi) languages such as Arabic and Hebrew are languages in which the text is presented to the user ordered from right to left, but numbers and Latin alphabetic strings within the text are presented left to right. In addition, the order in which characters appear within program variables can vary. In COBOL environments, the text in program variables is usually in *visual* order, which is the same order in which the text appears on the user interface. In Java environments, the text is usually stored in *logical* order, the order in which the characters are entered in the input field.

These differences in ordering and in other associated presentation characteristics require the program to have the ability to convert bi-directional text strings from one format to another. The bidi conversion attributes are specified in a bidi conversion table (.bct) file created separately from the program. The program references the name of the conversion table to indicate how attribute conversion should be performed.

In all cases, the bidi conversion table reference is specified as the 1 to 8 character file name without the .bct extension. For example, if you have created a bidi conversion table named mybct.bct, you can set the value of formConversionTable in a program by adding the following statement at the beginning of the program:

```
sysVar.formConversionTable = "mybct.bct" ;
```

Your tasks are as follows:

- Create bidi conversion tables that specify the transformations that should occur. Note that different tables are needed for converting data being passed between a Java client and a COBOL host and for converting data to be displayed in a text or print form in a Java environment.
- Reference the appropriate table in the appropriate situation:
  - When generating a COBOL program that uses forms, data tables, or literals with bidi language text, set the build descriptor option bidiConversionTable
  - When generating Java programs that call remote COBOL programs, customize the linkage options part so that the property conversionTable is in the callLink element (or, for asynchronous transfers, in the asynchLink element) for the invoked program:
    - You can specify a conversion table as the value of that property; or

- You can set the property to programControlled, which means that the invoking program specifies the conversion table before invoking the other program. The invoker specifies the table by assigning the conversion table name to the system variable sysVar.callConversionTable.

– When generating a Java program that uses text or print forms with bidi language text, add a statement to the program that assigns the conversion table name to the system function sysVar.formConversionTable before showing the form.

You build the bidi conversion table file using the bidi conversion table wizard plugin, which is in the file BidiConversionTable.zip:

1. Download the file from the following web site:
2. Unzip the file into your workbench directory
3. To begin running the wizard, click **File > New > Other > BidiConversionTable**.

   The name of a table used with EGL programs must have eight characters or less and must have the .bct extension.
4. While running the wizard, press F1 for help in choosing the correct options for creating the table.

When creating a bidi conversion table for generating COBOL programs, specify the client encoding and server encoding as shown in the next table.

Language	Client encoding for bidi conversion table	Server encoding for bidi conversion table
Arabic	Cp1256	Cp864
Hebrew	Cp1255	Cp1255

The bidi conversion table controls the transformation of the text from logical to visual order for the COBOL environment, along with any other formatting transformation requested in the table. At program-generation time, a pair of build descriptor options (clientCodeSet and serverCodeSet) control the conversion of the code page from ASCII to EBCDIC, as shown in the next table.

Language	clientCodeSet	serverCodeSet
Arabic	IBM-864	IBM-420
Hebrew	IBM-1255	IBM-424

Related reference
“bidiConversionTable” on page 364
“clientCodeSet” on page 366
“Data conversion” on page 454
“serverCodeSet” on page 381
“callConversionTable” on page 902

Data initialization

If an EGL-generated program initializes a record automatically (as occurs in some cases, described later), each of the lowest-level structure items is set to a value appropriate to the primitive type. Form initialization is similar, except that your form declaration can assign values that override the defaults.

Initialization also occurs in these situations:

- The **initialized** property of a variable (specifically, of an item or record or static array) is set to *yes*.
- Your logic includes certain variations of set

The next table gives details on the initialization values.

Primitive type	Initialization value
ANY	Variable is of an undefined type
BIN and the integer types (BIGINT, INT, and SMALLINT), HEX, FLOAT, SMALLFLOAT	Binary zeros
CHAR, MBCHAR	Single-byte blanks
DATE, TIME, TIMESTAMP	Current® value of the machine clock (for the number of bytes required by the mask, in the case of TIMESTAMP)
DBCHAR	Double-byte blanks
DECIMAL, MONEY, NUM, NUMC, PACF	Numeric zeros
INTERVAL	Numeric zeros (for the number of bytes required by the mask), preceded by a plus sign
UNICODE	Unicode blanks (each of which is hexadecimal 0020)

In a structure, only the lowest-level structure items are considered. If a structure item of type HEX is subordinate to a structure item of type CHAR, for example, the memory area is initialized with binary zeros.

Records or items that are received as program or function parameters are never initialized automatically.

An EGL-generated Java program initializes records, whether local or global.

An EGL-generated COBOL program initializes the input record, which is identified in the program properties. Other record initialization depends on whether you set the *initialized* property for a given variable. If you do not, record initialization depends on how you set two build descriptor options at generation time:

- Setting initNonIOData to YES causes the generated program to initialize global basic records
- Setting initIORecords to YES causes the generated program to initialize other global records

In keeping with the behavior of COBOL programs in general, EGL-generated COBOL programs do *not* initialize *local* records.

If you generate a COBOL program that compares an item of type NUM with an item of type CHAR, make sure that your code initializes the items; otherwise, the comparison may cause the program to fail with an abend (an abnormal end), in which case no exception-handling code is run. A similar, COBOL-specific warning applies to structure items in local structures and records.

Related concepts
"Function part" on page 132

"DataItem part" on page 123
"Program part" on page 130
"Record parts" on page 124
"Fixed structure" on page 24

Related reference
"EGL statements" on page 83
"initNonIOData" on page 376
"initIORecords" on page 376
"set" on page 617

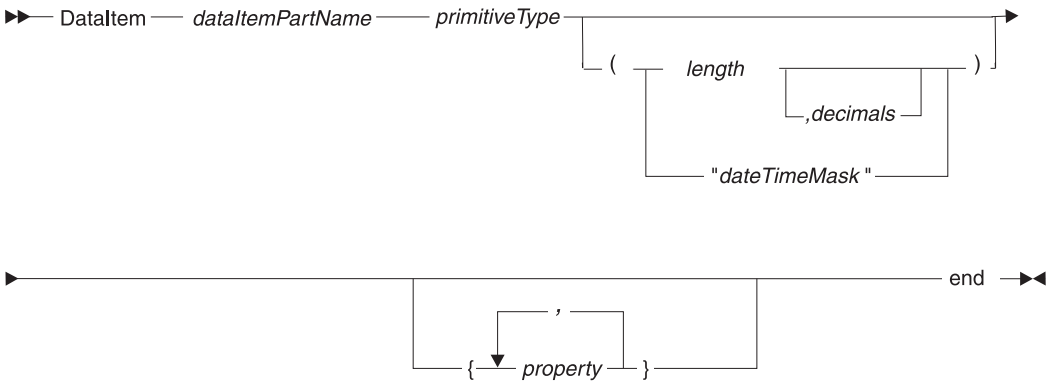
DataItem part in EGL source format

You declare a DataItem part in an EGL file, which is described in *EGL source format*.

An example of a data item part is as follows:

```
DataItem myDataItemPart
  BIN(9,2)
end
```

The syntax diagram for a dataItem part is as follows:



DataItem *dataItemPartName* **... end**
Identifies the part as a dataItem part and specifies the name. For rules, see *naming conventions*.

primitiveType
The primitive type assigned to the dataItem part.

length
An integer that reflects the length of the dataItem part. The value of any variable that is based on the part includes the specified number of characters or digits.

decimals
For a numeric, fixed type other than MONEY (specifically, BIN, DECIMAL, NUM, NUMC, or PACF), you may specify *decimals*, which is an integer that represents the number of places after the decimal point. The maximum number of decimal positions is the smaller of two numbers: 18 or the number of digits declared as *length*. The decimal point is not stored with the data.

"dateTimeMask"
For items of type INTERVAL or TIMESTAMP, you may specify *"dateTimeMask"*,

which assigns a meaning (such as ″year digit″) to a given position in the item value. The mask is not stored with the data.

property
An item property, as described in *Overview of EGL properties and overrides*.

Related concepts
“DataItem part” on page 123
“EGL projects, packages, and files” on page 13
“Overview of EGL properties” on page 60
“References to parts” on page 20
“Parts” on page 17

Related tasks
“Syntax diagram for EGL statements and commands” on page 733

Related reference
“EGL source format” on page 478
“Function part in EGL source format” on page 513
“Indexed record part in EGL source format” on page 520
“MQ record part in EGL source format” on page 642
“Naming conventions” on page 652
“Primitive types” on page 31
“Program part in EGL source format” on page 707
“Relative record part in EGL source format” on page 719
“Serial record part in EGL source format” on page 722
“SQL record part in EGL source format” on page 726

DataTable part in EGL source format

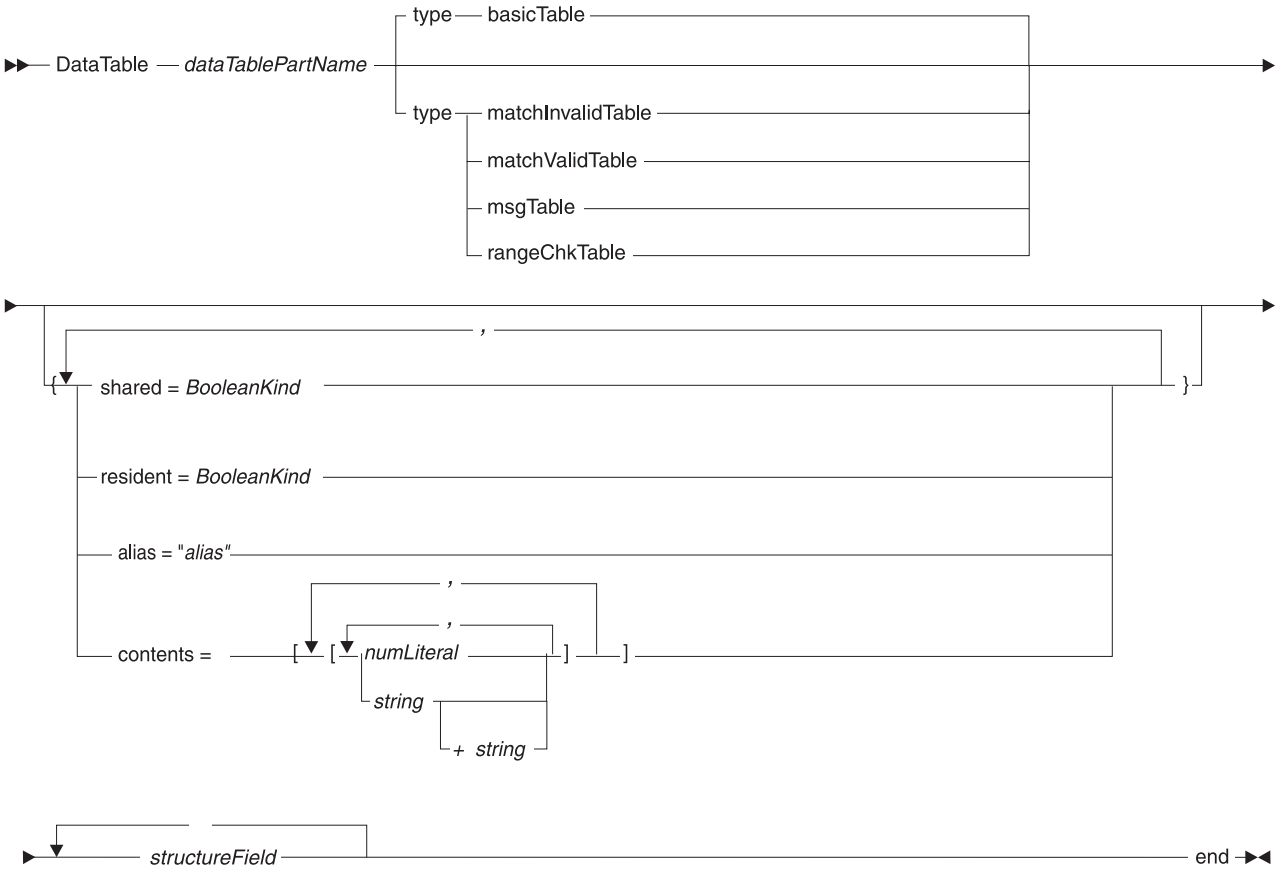
You declare a dataTable part in an EGL file, which is described in *EGL projects, packages, and files*. This part is a generatable part, which means that it must be at the top level of the file and must have the same name as the file.

A dataTable is related to a program by the program’s use declaration or (in the case of the program’s only message table) by the program’s **msgTablePrefix** property. A dataTable is related to a pageHandler by the pageHandler’s use declaration.

An example of a dataTable part is as follows:

```
DataTable myDataTablePart type basicTable
  {
    { shared = yes }
    myColumn1 char(10);
    myColumn2 char(10);
    myColumn3 char(10);


    { contents = [
     [ "row1 col1", "row1 col2", "row1 " + "col3" ] ,
     [ "row2 col1", "row2 col2", "row2 " + "col3" ] ,
     [ "row3 col1", "row3 col2", "row3 col3"      ]
               ]
  }
end
```

The syntax diagram for a dataTable part is as follows:



DataTable *dataTablePartName* **... end**
Identifies the part as a dataTable and specifies the part name. For the rules of naming, see *Naming conventions*.

basicTable (the default)
Contains information that is used in the program logic; for example, a list of countries and related codes.

matchInvalidTable
Is specified in the **validatorDataTable** property of a text field to indicate that the user's input must be different from any value in the first column of the dataTable. The EGL run time acts as follows in response to a validation failure:

- Accesses the table referenced in the **validatorDataTable** property
- Retrieves the message identified by the **validatorDataTableMsgKey** property that is specific to the text field
- Displays the message in the text field identified in the form-specific **msgField** property

matchValidTable
Is specified in the **validatorDataTable** property of a text field to indicate that the user's input must match a value in the first column of the dataTable. The EGL run time acts as follows in response to a validation failure:

- Accesses the table referenced in the **validatorDataTable** property
- Retrieves the message identified by the **validatorDataTableMsgKey** property that is specific to the text field
- Displays the message in the field identified in the form-specific **msgField** property

msgTable
Contains run-time messages. A message is presented in the following circumstance:

- The table is the message table for the program. The association of table to program occurs if the program property **msgTablePrefix** references the *table prefix*, which is the first one to four characters in the name of the dataTable. The rest of the name is one of the national language codes in the next table.

Language	National language code
Brazilian Portugese	PTB
Chinese, simplified	CHS
Chinese, traditional	CHT
English, uppercase	ENP
English, USA	ENU
French	FRA
German	DEU
Italian	ITA
Japanese, Katakana (single-byte character set)	JPN
Korean	KOR
Spanish	ESP
Swiss German	DES

- The program retrieves and presents a message by one of two mechanisms, as described in *ConverseLib.displayMsgNum* and *ConverseLib.validationFailed*.

rangeChkTable
Is specified in the **validatorDataTable** property of a text field to indicate that the user's input must match a value that is between the values in the first and second column of at least one data-table row. (The range is inclusive; the user's input is valid if it matches a value in the first or second column of any row.) The EGL run time acts as follows in response to a validation failure:

- Accesses the table referenced in the **validatorDataTable** property
- Retrieves the message identified by the **validatorDataTableMsgKey** property that is specific to the text field
- Displays the message in the field identified in the form-specific **msgField** property

"**alias**"
A string that is incorporated into the names of generated output. If you do not specify an alias, the dataTable name (or a truncated version)is used instead.

shared
Indicates whether the same instance of a dataTable is used by multiple programs. Valid values are *yes* and *no* (the default).

The property indicates whether the same instance of a dataTable is used by every program in the same run unit. If the value of **shared** is *no*, each program in the run unit has a unique copy of the dataTable.

Changes made at run time are visible to every program that has access to the dataTable, and the changes remain until the dataTable is unloaded. In most cases, the value of the **resident** property (described later) determines when the dataTable is unloaded; for details, see the description of that property.

resident
Indicates whether the dataTable is kept in memory even after every program that accessed the dataTable has ended.

Valid values are *yes* and *no*. The default is *no*.

If you set the **resident** property to *yes*, the dataTable is shared regardless of the value of **shared**.

The benefits of making a dataTable resident are as follows:

- The dataTable retains any values written to it by programs that ran previously
- The table is available for immediate access without additional load processing

A resident dataTable remains loaded until the run unit ends. A non-resident dataTable, however, is unloaded when the program that uses it ends.

Note: A dataTable is loaded into memory (if necessary) at a program's first access, and not when the EGL run time processes a use declaration.

contents
The value of the dataTable cells, each of which is one of the following kinds:

- A numeric literal
- A string literal or a concatenation of string literals

The kind of content in a given row must be compatible with the top-level structure fields, each of which represents a column definition.

structureField
A structure field, as described in *Structure field in EGL source format*.

Related concepts
"DataTable" on page 137
"EGL projects, packages, and files" on page 13
"Run unit" on page 721

Related reference
"Naming conventions" on page 652
"Structure field in EGL source format" on page 730
"displayMsgNum()" on page 766
"validationFailed()" on page 767
"Use declaration" on page 930

EGL build path and eglpath

Each EGL project and EGL Web project is associated with an EGL build path so that the project can reference parts in other projects. For details on when the EGL build path is used and on why the order of build-path entries is important, see *References to parts*.

When you specify the EGL build path, you can choose to *export* one or more of the projects that are listed in the build path. Then, when a project refers to the project being declared, each of the exported projects is made available to the referencing project, as in the following example:

- The EGL build path for project A comprises the following projects, in order:

```
A, B, C, D
```

Projects B and D are exported.

- The EGL build path for project L comprises the following projects, in order:

  ```
  L, J, A, Z
  ```

- The effective build path for project L also includes the projects that were exported from project A. In this case, the EGL build path for project L is effectively as follows:

  ```
  L, J, A, B, D, Z
  ```

  The exported projects are placed after the project that exports them, in the order in which the projects are listed in the build path of the exporting project.

The build path of a project always includes the project itself, which is usually first in build-path order, as is recommended. If you have multiple EGL source folders in your project, all must be listed in the EGL build path for that project, and the order of those folders is used by any project that refers to your project.

It is strongly recommended that you avoid having identically named packages in different projects or in different folders of the same project.

If you generate in the EGL SDK, the situation is as follows:

- Project information is not available.
- The command-line argument *eglpath* replaces the functionality of the EGL build path. *eglpath* is a list of operating-system directories that are searched when the EGL SDK attempts to resolve a part reference.
- The rules for when eglpath is used are equivalent to the rules for when the EGL build path is used; however, you cannot export directories as you can export projects.

When you use the EGL SDK, it is strongly recommended that you avoid having identically named packages in different directories.

Related concepts
"Generation from the EGL Software Development Kit (SDK)" on page 314
"References to parts" on page 20

Related tasks
"Generating from the EGL Software Development Kit (SDK)" on page 313

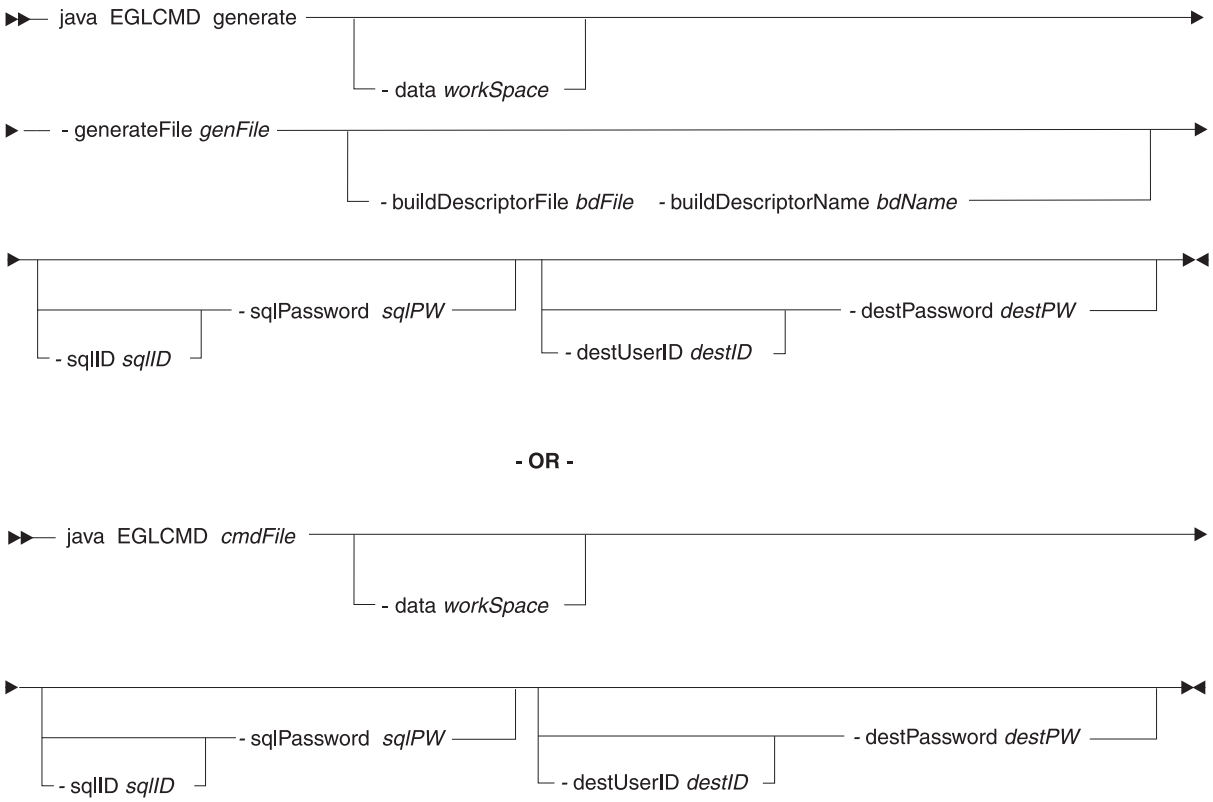
Related reference
"EGLSDK" on page 476

EGLCMD

The command EGLCMD gives you access to the Workbench batch interface, as described in *Generation from the workbench batch interface*.

Syntax

The syntax for invoking EGLCMD is as follows:



generate
Indicates that the command itself references the EGL file and build descriptor part that are used to generate output. In this case, the command EGLCMD does not reference a command file.

–data *workSpace*
Specifies the absolute or relative path of the workspace directory. Relative paths are relative to the directory in which you run the command.

If you do not specify a value, the command accesses the Eclipse default workspace.

Embed the path in double quotes.

cmdFile
Specifies the absolute or relative path of the file described in *EGL command file*. Relative paths are relative to the directory in which you run the command.

Embed the path in double quotes.

The command file must be in your workspace; otherwise, use the Eclipse import process to import the file and then rerun EGLCMD.

–generateFile *genFile*
Specifies the absolute or relative path of the EGL file that contains the part you want to process. Relative paths are relative to the directory in which you run the command.

Embed the path in double quotes.

–buildDescriptorFile *bdFile*
Specifies the absolute or relative path of the build file that contains the build descriptor. Relative paths are relative to the directory in which you run the command.

Embed the path in double quotes.

–buildDescriptorName *bdName*
Specifies the name of a build descriptor part that guides generation. The build descriptor must be at the top level of an EGL build (.eglbld) file.

–sqlID *sqlID*
Sets the value of build descriptor option sqlID.

–sqlPassword *sqlPW*
Sets the value of build descriptor option sqlPassword.

–destUserid *destID*
Sets the value of build descriptor option destUserID.

–destPassword *destPW*
Sets the value of build descriptor option destPassword.

Build descriptor options that you specify when invoking the command EGLCMD take precedence over options in the build descriptor (if any) that is listed in the EGL command file.

Examples

In the commands that follow, each multiline example belongs on a single line:

```
java EGLCMD "commandfile.xml"

java EGLCMD "commandfile.xml" -data "c:\myWorkSpace"

java EGLCMD generate
    -generateFile "c:\myProg.eglpgm"
    -data "myWorkSpace"
    -buildDescriptorFile "c:\myBuild.eglbld"
    -buildDescriptorName myBuildDescriptor

java EGLCMD "myCommand.xml"
    -data "my WorkSpace"
    -sqlID myID -sqlPassword myPW
    -destUserID myUserID -destPassword myPass
```

Related concepts
"Generation from the workbench batch interface" on page 313

Related tasks
"Generating from the workbench batch interface" on page 312
"Syntax diagram for EGL statements and commands" on page 733

Related reference
"destPassword" on page 369
"destUserID" on page 370
"sqlID" on page 385
"sqlPassword" on page 387

EGL command file

An EGL command file indicates what EGL files you wish to process when you generate output outside of the workbench, whether you are using the workbench batch interface (command EGLCMD) or the EGL SDK (command EGLSDK). You can create the file in either of two ways:

- By hand, according to the rules described later; or
- By using the EGL Generation wizard, as described in *Generating in the workbench*.

The command file is an XML file, and the file name must have the extension .xml, in any combination of uppercase and lowercase letters. The file content must conform to the following document type definition (DTD):

```
installationDir\egl\eclipse\plugins\
com.ibm.etools.egl.utilities_version\
dtd\eglcommands_5_1.dtd
```

installationDir
The product installation directory, such as C:\Program Files\IBM\RSPD\6.0. If you installed and kept a Rational Developer product before installing the product that you are using now, you may need to specify the directory that was used in the earlier install.

version
The installed version of the plugin; for example, 6.0.0

The following table shows the elements and attributes supported by the DTD. The element and attribute names are case sensitive.

Element	Attribute	Attribute value
EGLCOMMANDS (required)	**eglpath**	As described in *eglpath*, the eglpath attribute identifies directories to search when EGL uses an import statement to resolve the name of a part. The attribute is optional and if present, references a quoted string that has one or more directory names, each separated from the next by a semicolon. The attribute is used only if the command EGLSDK is referencing the command file. If the command EGLCMD is in use, the value of eglpath is ignored; instead, import statements are resolved in accordance with the EGL project path, as described in *Import*.

Element	Attribute	Attribute value
buildDescriptor (optional; you can avoid specifying this value if you are using a master build descriptor, as described in *Build descriptor part*)	**name**	The name of a build descriptor part that guides generation. The build descriptor must be at the top level of an EGL build (.eglbld) file. Build descriptor options that you specify when invoking EGLCMD or EGLSDK take precedence over options in the build descriptor (if any) that is listed in the EGL command file.
	file	The absolute or relative path of the EGL file that contains the build descriptor. Relative paths specified for EGLCMD are relative to the path name of the Enterprise Developer workspace. Relative paths specified for EGLSDK are relative to the directory in which you run the command. The path must be in double quotes if the path includes a space.
generate (optional)	**file**	The absolute or relative path of the EGL file that contains the part you want to process. Relative paths specified for EGLCMD are relative to the path name of the Enterprise Developer workspace. Relative paths specified for EGLSDK are relative to the directory in which you run the command. The path must be in double quotes if the path includes a space. If you omit the file attribute, no generation occurs.

Examples of command files

This section shows two command files. The results produced by either file are the same whether you use the EGLCMD command or the EGLSDK command, if you run the EGLSDK command in the directory where the EGL program files reside.

The following command file contains a generate command that uses the build descriptor myBDescPart to generate the program myProgram.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE EGLCOMMANDS PUBLIC "-//IBM//DTD EGLCOMMANDS 5.1//EN" "">
<EGLCOMMANDS eglpath="C:\mydata\entdev\workspace\projectinteract">
    <generate file="projectinteract\myProgram.eglpgm">
        <buildDescriptor name="myBDescPart" file="projectinteract\mybdesc.eglbld"/>
    </generate>
</EGLCOMMANDS>
```

The next example contains two generate commands, both of which implicitly use a master build descriptor.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE EGLCOMMANDS PUBLIC "-//IBM//DTD EGLCOMMANDS 5.1//EN" "">
<EGLCOMMANDS eglpath="C:\mydata\entdev\workspace\projecttrade">
    <generate file="projecttrade\program2.eglpgm"/>
    <generate file="projecttrade\program3.eglpgm"/>
</EGLCOMMANDS>
```

Related concepts

"Build descriptor part" on page 275

"Generation from the EGL Software Development Kit (SDK)" on page 314
"Generation from the workbench batch interface" on page 313
"Import" on page 30

Related tasks
"Generating from the EGL Software Development Kit (SDK)" on page 313
"Generating from the workbench batch interface" on page 312
"Generating in the workbench" on page 310

Related reference
"EGLCMD" on page 466
"EGL command file" on page 469
"EGL build path and eglpath" on page 465
"EGLSDK" on page 476

EGL editor

To change an EGL file (extension .egl), work in the EGL source editor, which guides you with content assist.

To change an EGL build file (extension .eglbld), follow one of these procedures:

- Creating a build file
-
- Adding a build descriptor part
- Adding a linkage options part
-
- Adding a resource associations part

Related concepts
"EGL projects, packages, and files" on page 13
"Parts" on page 17

Related reference
"Content assist in EGL"

Content assist in EGL

The EGL editor provides *content assist*, which proposes information that you can add to your source file. With a keystroke or two, you can complete the name of a part, variable, or function or can place a *template* (the outline of a part) into your source file.

The keystroke that activates content assist is **Ctrl + Space**.

Related tasks
"Setting preferences for templates" on page 110
"Using the EGL templates with content assist" on page 121

Enumerations in EGL

In some cases in EGL, the values of a property or field are restricted to the values of a particular *enumeration*, which is a category of predefined values. The property **color**, for example, accepts a value of the enumeration **ColorKind**, and valid values of that enumeration include *white* and *red*.

You can qualify an enumeration value with the enumeration name, so the preceding values can be stated as *ColorKind.white* and *ColorKind.red*. However, you need to qualify the enumeration value only when your code has access to a variable or constant whose name is the same as the enumeration value. If a variable named *red* is in scope, for example, the symbol *red* refers to the variable rather than to the enumeration value.

The following list of enumerations includes the enumeration values; but explanations of those values occur elsewhere, in the context of the property or field in which the enumeration is meaningful:

AlignKind

center
left
none
right

Boolean

yes
no

CallingConventionKind

I4GL
Library

CaseFormatKind

defaultCase
lower
upper

ColorKind

black (as is valid only for console fields)
blue
cyan
defaultColor
green
magenta
red
yellow
white

DataSource

databaseConnection
reportData
sqlStatement

DeviceTypeKind

doubleByte
singleByte

DisplayUseKind

button
hyperlink

input
output
secret
table

EventKind

AFTER_DELETE
AFTER_FIELD
AFTER_OPENUI
AFTER_INSERT
AFTER_ROW
BEFORE_DELETE
BEFORE_FIELD
BEFORE_OPENUI
BEFORE_INSERT
BEFORE_ROW
ON_KEY
MENU_ACTION

ExportFormat

html
pdf
text
xml

HighlightKind

blink
defaultHighlight
noHighlight
reverse
underline

IndexOrientationKind

across
down

IntensityKind

bold
defaultHighlight
dim
invisible
normalIntensity

LineWrapKind

character
compress (as is valid only for console fields)
word

OutlineKind

bottom

left
right
top

Note: sysLib.box is a constant that equates to [left,right,top,bottom]. sysLib.noOutline is a constant that means there is no outlining.

PfKeyKind

pf*n*, where (1 <= *n* <=24)

ProtectKind

skip
no
yes

SelectTypeKind

index
value

SignKind

leading
none
parens
trailing

Related concepts
"Overview of EGL properties" on page 60"References to variables in EGL" on page 55

EGL reserved words

EGL includes two categories of reserved words:

- Words that are reserved for specific uses except when you are working on an SQL statement
- Words that are reserved for specific uses when you are working on an SQL statement

Words that are reserved outside of an SQL statement

Outside of SQL statements, the reserved words are as follows in any combination of upper- and lower-case letters:

- absolute, add, all, any, as
- bigInt, bin, bind, blob, boolean, by, byName, byPosition
- call, case, char, clob, close, const, continue, converse, current
- dataItem, dataTable, date, dbChar, decimal, decrement, delete, display, dliCall
- else, embed, end, escape, execute, exit, externallyDefined
- false, field, first, float, for, forEach, form, formGroup, forUpdate, forward, freeSql, from, function
- get, goto
- handler, hex, hold
- if, import, in, inOut, insert, int, interval, into, is, isa
- label, languageBundle, last, library, like
- matches, mbChar, money, move
- new, next, no, noRefresh, not, nullable, num, number, numc
- onEvent, onException, open, openUI, otherwise, out

- pacf, package, pageHandler, passing, prepare, previous, print, private, program, psb
- record, ref, relative, replace, return, returning, returns
- scroll, self, set, show, singleRow, smallFloat, smallInt, sql, sqlCondition, stack, string
- this, time, timeStamp, to, transaction, transfer, true, try, type
- unicode, update, url, use, using, usingKeys
- when, while, with, withinParent
- yes

Words that are reserved in an SQL statement

In SQL statements, the reserved words are as follows in any combination of upper- and lower-case letters:

- absolute, action, add, alias, all, allocate, alter, and, any, are, as, asc, assertion, at, authorization, avg
- begin, between, bigint, binaryLargeObject, bit, bit_length, blob, boolean, both, by
- call, cascade, cascaded, case, cast, catalog, char, char_length, character, character_length, characterLargeObject, characterVarying, charLargeObject, charVarying, check, clob, close, coalesce, collate, collation, column, comment, commit, connect, connection, constraint, constraints, continue, convert, copy, corresponding, count, create, cross, current, current_date, current_time, current_timestamp, current_user, cursor
- data, database, date, dateTime, day, deallocate, dec, decimal, declare, default, deferrable, deferred, delete, desc, describe, diagnostics, disconnect, distinct, domain, double, doublePrecision, drop
- else, end, endExec, escape, except, exception, exec, execute, exists, explain, external, extract
- false, fetch, first, float, for, foreign, found, from, full
- get, getCurrentConnection, global, go, goto, grant, group
- having, hour
- identity, image, immediate, in, index, indicator, initially, inner, input, insensitive, insert, int, integer, intersect, into, is, isolation
- join
- key
- language, last, leading, left, level, like, local, long, longint, lower, ltrim
- match, max, min, minute, module, month
- national, nationalCharacter, nationalCharacterLargeObject, nationalCharacterVarying, nationalCharLargeObject, nationalCharVarying, natural, nchar, ncharVarying, nclob, next, no, not, null, nullIf, number, numeric
- octet_length, of, on, only, open, option, or, order, outer, output, overlaps
- pad, partial, position, prepare, preserve, primary, prior, privileges, procedure, public
- raw, read, real, references, relative, restrict, revoke, right, rollback, rows, rtrim, runtimeStatistics
- schema, scroll, second, section, select, session, session_user, set, signal, size, smallint, some, space, sql, sqlcode, sqlerror, sqlstate, substr, substring, sum, system_user
- table, tablespace, temporary, terminate, then, time, timestamp, timezone_hour, timezone_minute, tinyint, to, trailing, transaction, translate, translation, trim, true
- uncatalog, union, unique, unknown, update, upper, usage, user, using
- values, varbinary, varchar, varchar2, varying, view
- when, whenever, where, with, work, write
- year
- zone

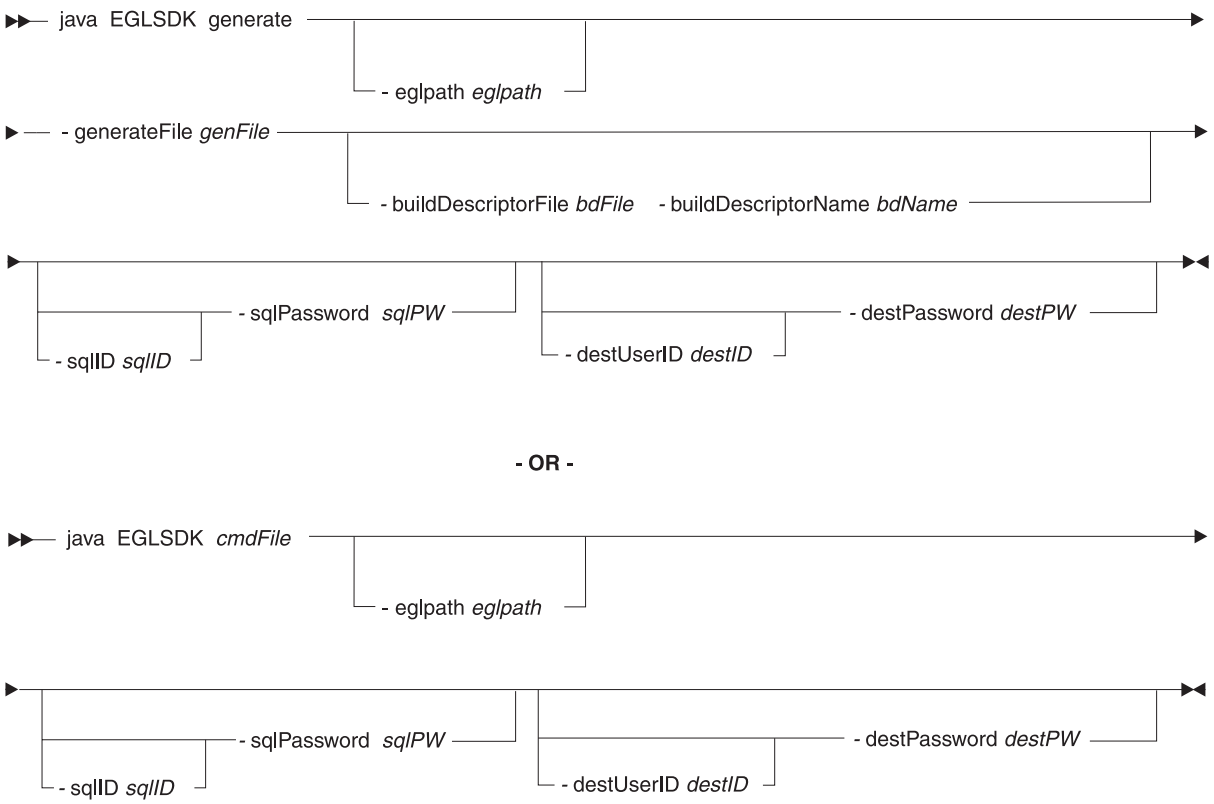
Related reference
"EGL statements" on page 83
"Naming conventions" on page 652
"reservedWord" on page 381

EGLSDK

The command EGLSDK gives you access to the EGL Software Development Kit (EGL SDK), as described in *Generation from the EGL SDK*.

Syntax

The syntax for invoking EGLSDK is as follows:



generate
Indicates that the command itself references the EGL file and build descriptor part that are used to generate output. In this case, the command EGLSDK does not reference a command file.

cmdFile
Specifies the absolute or relative path of the file described in *EGL command file*. Relative paths are relative to the directory in which you run the command.

Embed the path in double quotes.

–eglpath *eglpath*
As described in *eglpath*, the eglpath option identifies directories to search when EGL uses an import statement to resolve the name of a part. You specify a quoted string that has one or more directory names, each separated from the next by a semicolon.

–generateFile *genFile*
The absolute or relative path of the EGL file that contains the part you want to process. Relative paths are relative to the directory in which you run the command.

Embed the path in double quotes.

–buildDescriptorFile *bdFile*
The absolute or relative path of the build file that contains the build descriptor. Relative paths are relative to the directory in which you run the command.

Embed the path in double quotes.

–buildDescriptorName *bdName*
The name of a build descriptor part that guides generation. The build descriptor must be at the top level of an EGL build (.eglbld) file.

–sqlID *sqlID*
Sets the value of build descriptor option sqlID.

–sqlPassword *sqlPW*
Sets the value of build descriptor option sqlPassword.

–destUserid *destID*
Sets the value of build descriptor option destUserID.

–destPassword *destPW*
Sets the value of build descriptor option destPassword.

The eglpath value that you specify when invoking the command EGLSDK takes precedence over any eglpath value in an EGL command file. Similarly, build descriptor options that you specify when invoking the command take precedence over options in any build descriptor that is listed in an EGL command file.

Examples

In the commands that follow, each multiline example belongs on a single line:

```
java EGLSDK "commandfile.xml"

java EGLSDK "commandfile.xml"
    -eglpath "c:\myGroup;h:\myCorp"

java EGLSDK generate
    -eglpath "c:\myGroup;h:\myCorp"
    -generateFile "c:\myProg.eglpgm"
    -buildDescriptorFile "c:\myBuild.eglbld"
    -buildDescriptorName myBuildDescriptor

java EGLSDK "myCommand.xml"
    -sqlID myID -sqlPassword myPW
    -destUserID myUserID -destPassword myPass
```

Related concepts
"Build descriptor part" on page 275
"Generation from the EGL Software Development Kit (SDK)" on page 314
"Import" on page 30
"Master build descriptor" on page 278

Related tasks
"Generating from the EGL Software Development Kit (SDK)" on page 313

Related reference
"destPassword" on page 369
"destUserID" on page 370
"EGL build path and eglpath" on page 465
"sqlID" on page 385
"sqlPassword" on page 387
"Syntax diagram for EGL statements and commands" on page 733

Format of eglmaster.properties file

The eglmaster.properties file is a Java properties file that the EGL SDK uses to specify the name and file path name of the master build descriptor. This properties file must be contained in a directory that is specified in the CLASSPATH variable of the process that invokes the EGLSDK command. The format of the eglmaster.properties file is as follows:

```
masterBuildDescriptorName=desc
masterBuildDescriptorFile=path
```

where:

desc
The name of the master build descriptor

path
The fully qualified path name of the EGL file in which the master build descriptor used by the EGL SDK is declared

The content of this file must follow the rules of a Java properties file. You can use either a slash (/) or two backslashes (\\) to separate file names within a path name.

You must specify both the **masterBuildDescriptorName** and **masterBuildDescriptorFile** keywords in the properties file. Otherwise the eglmaster.properties file is ignored.

Following is an example of the contents of an eglmaster.properties file:

```
# Specify the name of the master build descriptor:
masterBuildDescriptorName=MYBUILDDESCRIPTOR
# Specify the file that contains the master build descriptor:
masterBuildDescriptorFile=d:/egl/builddescriptors/master.egl
```

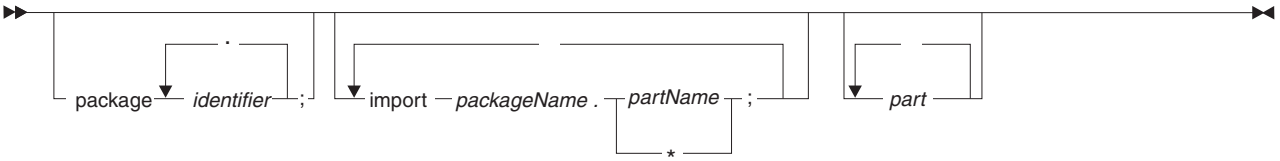
Related concepts
"Master build descriptor" on page 278

Related tasks
"Choosing options for COBOL generation" on page 283
"Choosing options for Java generation" on page 281

Related reference
"Build descriptor options" on page 359
"EGLSDK" on page 476
"Format of master build descriptor plugin.xml file" on page 493

EGL source format

You declare logic, data, and user-interface parts in EGL source files, each of which has the extension *.egl* and is constructed as follows:



package *identifier*
Specifies the name of the package in which the file resides, with each identifier separated from the next by a period.

For an overview, see *EGL projects, packages, and files*.

import *packageName*
Specifies the full name of a package to import. For an overview, see Import.

partName
Specifies a single part to import.

* Indicates that every part in the package is to be imported.

part
One of the EGL logic, data, or user-interface parts.

You may place comments in an EGL file, inside or outside a part.

Related concepts
"EGL projects, packages, and files" on page 13
"Import" on page 30
"References to parts" on page 20
"Parts" on page 17

Related tasks
"Syntax diagram for EGL statements and commands" on page 733

Related reference
"Basic record part in EGL source format" on page 357
"Comments" on page 427
"DataItem part in EGL source format" on page 461
"DataTable part in EGL source format" on page 462
"FormGroup part in EGL source format" on page 494
"Form part in EGL source format" on page 497
"Function part in EGL source format" on page 513
"Indexed record part in EGL source format" on page 520
"Library part in EGL source format" on page 630
"MQ record part in EGL source format" on page 642
"PageHandler part in EGL source format" on page 659
"Program part in EGL source format" on page 707
"Relative record part in EGL source format" on page 719
"Serial record part in EGL source format" on page 722
"SQL record part in EGL source format" on page 726

EGL system exceptions

The EGL system exceptions are available throughout your code, but are most often used in an **onException** block. For an overview, see *Exception handling*.

Each of the EGL system exceptions has at least the following fields:

code
A string that identifies the exception; for example ″com.ibm.egl.InvocationException″ or the equivalent constant, SysLib.InvocationException

description
A string that tells the meaning of the exception

The EGL system exceptions are as follows:

SysLib.FileIOException
Identifies an error that occurs during file access. Errors that occur during relational-database of message queue access do not raise this exception. Exception-specific fields are as follows:

errorCode
The 8-character status code also returned in SysVar.ErrorCode; for details, see *SysVar.ErrorCode*

fileName
The logical name of the file being accessed; for details, see *Resource associations and file types*

SysLib.InvocationException
Identifies an error that occurs in a **call** statement.

Exception-specific fields are as follows:

errorCode
The 8-character status code also returned in SysVar.ErrorCode; for details, see *SysVar.ErrorCode*

name
The name of the program being called.

SysLib.LobProcessingException
Identifies an error that occurred during processing of a field of type LOB or CLOB. Exception-specific fields are as follows:

itemName
Name of the field

operation
Name of the EGL system function that failed

resource
Name of the file (if any) attached to the field

SysLib.MQIOException
Identifies an error that occurs during access of an MQSeries message queue. Exception-specific fields are as follows:

errorCode
The 8-character status code also returned in SysVar.ErrorCode; for details, see *SysVar.ErrorCode*

mqConditionCode
The completion code from an MQSeries API call, as described in *VGVar.mqConditionCode*

name
The logical name of the queue being accessed; for details, see *Resource associations and file types*

SysLib.SQLException
Identifies an error that occurs during access of a relational database. Exception-specific fields are as follows:

sqlca
The SQL communication area; for details, see *SysVar.sqlca*

sqlcode
The SQL return code; for details, see *SysVar.sqlcode*

sqlErrd
A 6-element array, where each element contains the corresponding SQL communication area (SQLCA) value that was returned from the last SQL I/O option; for details, see *VGVar.sqlErrd*

sqlErrmc
The error message associated with sqlcode, for database access other than through JDBC; for details, see *VGVar.sqlErrmc*

sqlState
The SQL state value for the most recently completed SQL I/O operation; for details, see *SysVar.sqlState*

sqlWarn
An 11-element array, where each element contains a warning byte returned in the SQL communications area (SQLCA) for the last SQL I/O operation and where the index is one greater that the warning number in the SQL SQLCA description; for details, see *VGVar.sqlState*

Related concepts
"Resource associations and file types" on page 286

Related reference
"EGL system exceptions" on page 479
"errorCode" on page 903
"sqlca" on page 909
"sqlcode" on page 910
"sqlState" on page 911
"mqConditionCode" on page 922
"sqlerrd" on page 923
"sqlerrmc" on page 924
"sqlWarn" on page 925

EGL system limits

No EGL-defined limits are in effect for the number of parts or the number of hierarchical levels in an EGL file. The following limits apply, however:

- A program can use no more than 32767 variables and literals, including fields within variables.
- A call statement can have no more than 30 arguments; also, these restrictions apply to the size of the arguments in total--
  - Can be no more than 32567 if remoteCall or ejbCall is the value of the **type** property for the call.

  Both properties are in the linkage options part, callLink element.
- A field can be no more than 32767 bytes.

- In most cases, a numeric literal or field can have no more than 32 digits plus a sign, decimal point, or both; but a field that receives the result created by invoking the **mathLib.round** function can be no more than 31 digits plus a sign, decimal point, or both.
- A static array can have no more than 7 dimensions and can have no more than 32767 elements in total.
- The situation for a dynamic array is as follows:
  - A dynamic array can have no more than 14 dimensions (if the array is targeted for Java) or 7 dimensions (for COBOL). The number of dimensions in a dynamic record array is one (for the record array declaration) plus the number of dimensions in the record structure.
  - A dynamic array can have a maximum size no greater than 2,147,483,647 elements for Java environments and no greater than 1,044,472 elements for COBOL. Those numbers are in effect if you do not specify a maximum size, but the size that can be allocated is further limited by the memory available at run time.
  - The total size for all arguments that can be passed on a remote call is limited by the maximum buffer size supported for the protocol.

When a program is generated for CICS for z/OS, additional limits are as follows:

- The maximum number of bytes for an indexed, relative, or serial record that accesses a VSAM file is 32688 for a journaled record, 32763 for a non-journaled record.
- The maximum number of bytes for a record that accesses a transient data queue is 32763.
- The maximum number of bytes for a relative or serial record that accesses a temporary storage queue is 32762.
- The maximum number of bytes for a record that accesses a spool file is 32763.
- The total size for all dynamic-array arguments that can be passed on a remote call is 32 kilobytes for a call that uses the CICS external call interface.

Related reference
"callLink element" on page 395
"round()" on page 827
"Naming conventions" on page 652
"parmForm in callLink element" on page 405
"type in callLink element" on page 412

Expressions

An expression is a series of operands and operators that you specify when you write a program or function script.

Each expression resolves to a particular type of value at run time. A *numeric expression* resolves to a number; a *string expression* resolves to a series of characters; a *logical expression* resolves to true or false; a *datetime expression* resolves to a date, interval, time, or timestamp.

Expressions are evaluated in accordance with a set of precedence rules and (within a given level of precedence) from left to right, but you can use parentheses to force a different ordering. A nested parenthetical subexpression is evaluated before the enclosing parenthetical subexpression, and all parenthetical expressions are evaluated before the expression as a whole.

At a given level of evaluation, the first operand determines the type of expression (or subexpression). Consider this example:

```
"A value = " + 1 + 2
```

The first operand is of a character type, and the expression is a text expression with the following value:

```
"A value = 12"
```

Consider a different text expression:

```
"A value = " + (1 + 2)
```

The value in this case is as follows:

```
"A value = 3"
```

Related reference
"Datetime expressions"
"Logical expressions" on page 484
"Numeric expressions" on page 491
"Text expressions" on page 492

Datetime expressions

A *datetime expression* resolves to a value of type DATE, INT, INTERVAL, TIME, or TIMESTAMP, depending on the context. A datetime expression must include one of these:

- A variable that contains a value of one of those types.
- A function invocation that returns a datetime value. Several system functions create a datetime value from a string literal or constant:
  - **DateTimeLib.dateValue** creates a date
  - **DateTimeLib.intervalValue** creates an interval
  - **DateTimeLib.timeValue** creates a time
  - **DateTimeLib.timeStampValue** creates a timestamp

  Also, the system function **DateTimeLib.extend** returns a timestamp value that is longer or shorter than an input field of type DATE, TIME, or TIMESTAMP.

The next table summarizes the types of arithmetic operations that are valid in a datetime expression. As shown, a datetime expression may include a numeric expression that returns a number, but only in a subset of cases.

Arithmetic operations in a datetime expression

Type of Operand 1	Operator	Type of Operand 2	Type of Result	Comments
DATE	-	DATE	INT	
DATE	+/-	NUMBER	DATE	
NUMBER	+	DATE	DATE	

Arithmetic operations in a datetime expression

Type of Operand 1	Operator	Type of Operand 2	Type of Result	Comments
TIME STAMP	-	TIMESTAMP	INTERVAL	INTERVAL(dd, ss) unless Operand 1 and Operand 2 are both any of the following: • TIMESTAMP(yyyy) • TIMESTAMP(yyyyMM) • TIMESTAMP(MM) In those three cases, the result is INTERVAL(yyyyMM)
DATE	-	TIMESTAMP	INTERVAL	INTERVAL(ddssmmffffff)
TIME STAMP	-	DATE	INTERVAL	INTERVAL(ddHHmmssffffff)
TIME STAMP	+/-	INTERVAL	TIMESTAMP	
INTERVAL	+	TIMESTAMP	TIMESTAMP	
DATE	+/-	INTERVAL	TIMESTAMP	
INTERVAL	+	DATE	TIMESTAMP	
INTERVAL	+/-	INTERVAL	INTERVAL	Operand1 and Operand2 must both have (at most) years and months or both must have (at most) days and a time value
INTERVAL	*//	NUMBER	INTERVAL	

Related reference
"Assignments" on page 352
"dateValue()" on page 771
"extend()" on page 773
"intervalValue()" on page 773
"timeValue()" on page 777
"timeStampValue()" on page 776
"Expressions" on page 482
"Logical expressions"
"Numeric expressions" on page 491
"Operators and precedence" on page 653
"Primitive types" on page 31
"Text expressions" on page 492

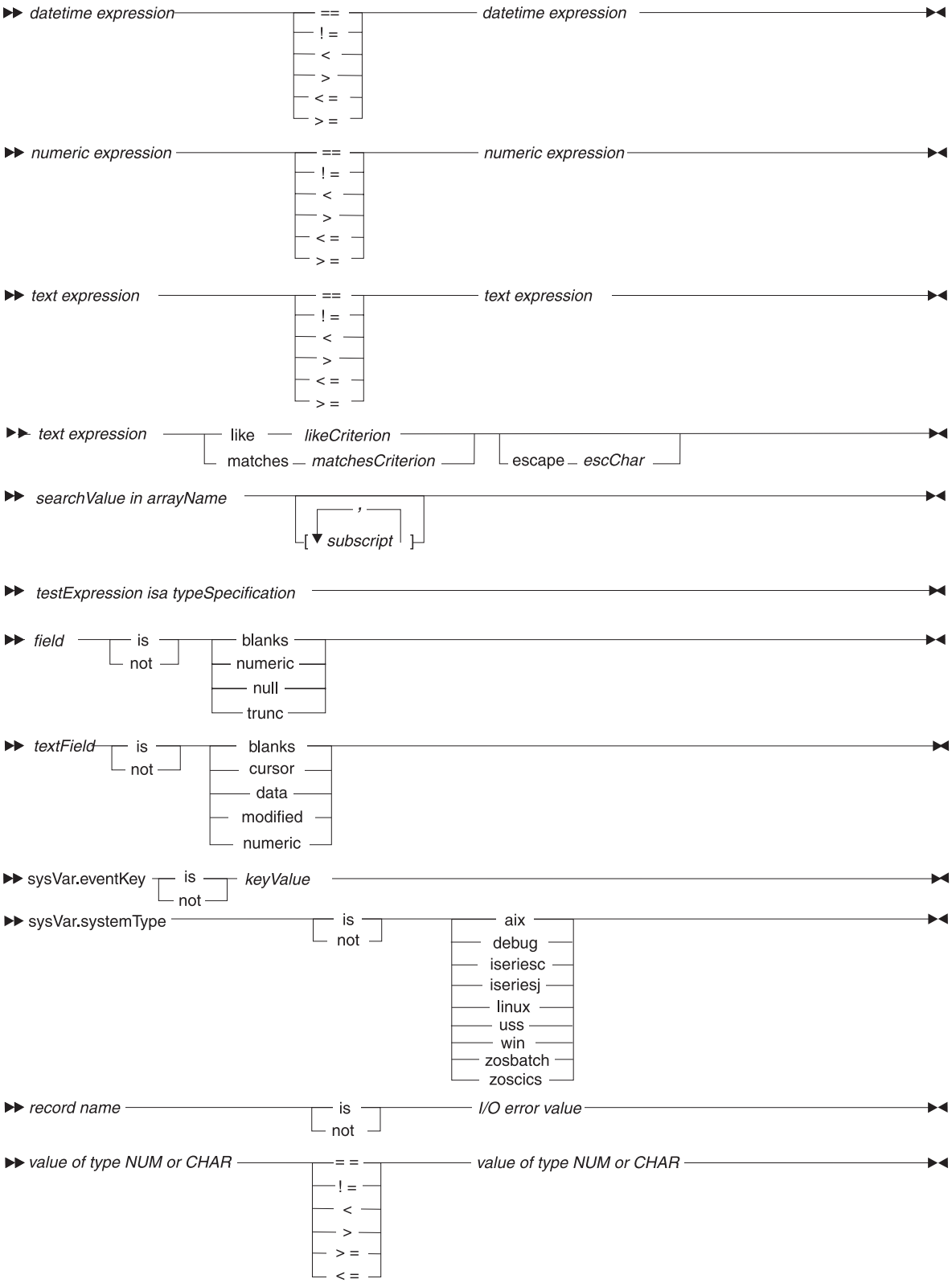
"Substrings" on page 731

Logical expressions

A *logical expression* resolves to true or false and is used as a criterion in an **if** or **while** statement or (in some situations) in a **case** statement.

Elementary logical expressions

An elementary logical expression is composed of an operand, a comparison operator, and a second operator, as shown in this syntax diagram and the subsequent table:


datetime expression
==
! =
<
>
< =
> =
datetime expression
numeric expression
==
! =
<
>
< =
> =
numeric expression
text expression
==
! =
<
>
< =
> =
text expression
text expression
like
matches
likeCriterion
matchesCriterion
escape
escChar
searchValue in arrayName
,
[
subscript
]
testExpression isa typeSpecification
field
is
not
blanks
numeric
null
trunc
textField
is
not
blanks
cursor
data
modified
numeric
sysVar.eventKey
is
not
keyValue
sysVar.systemType
is
not
aix
debug
iseriesc
iseriesj
linux
uss
win
zosbatch
zoscics
record name
is
not
I/O error value
value of type NUM or CHAR
= =
! =
<
>
> =
< =
value of type NUM or CHAR

First operand	Comparison Operator	Second operand
datetime expression	One of these: ==, != , <, >, <=, >=	*datetime expression* The first and second expressions must be of compatible types. In the case of datetime comparisons, the greater than sign (>) means later in time; and the less than (<) sign means earlier in time.
numeric expression	One of these: ==, != , <, >, <=, >=	*numeric expression*
string expression	One of these: ==, != , <, >, <=, >=	*string expression*
string expression	like	*likeCriterion*, which is a character field or literal against which *string expression* is compared, character position by character position from left to right. Use of this feature is similar to the use of keyword **like** in SQL queries. *escChar* is a one-character field or literal that resolves to an escape character. For further details, see *like operator*.
string expression	matches	*matchCriterion*, which is a character field or literal against which *string expression* is compared, character position by character position from left to right. Use of this feature is similar to the use of *regular expressions* in UNIX or Perl. *escChar* is a one-character field or literal that resolves to an escape character. For further details, see *matches operator*.
Value of type NUM or CHAR, as described for the second operand	One of these: ==, != , <, >, <=, >=	*Value of type NUM or CHAR*, which can be any of these: • A field that is of type NUM and has no decimal places • An integer literal • A field or literal of type CHAR
searchValue	in	*arrayName*; for details, see *in*.
field not in SQL record	One of these: • is • not	One of these: • blanks (for testing whether the value of a character field is or is not blanks only) • numeric (for testing whether the value of a field of type CHAR or MBCHAR is or is not numeric)

First operand	Comparison Operator	Second operand
field in an SQL record	One of these: • is • not	One of these: • blanks (for testing whether the value of a character field is or is not blanks only) • null (for testing whether the field was set to null either by a set statement or by reading from a relational database) • numeric (for testing whether the value of a field of type CHAR or MBCHAR is or is not numeric) • trunc (for testing whether non-blank characters were deleted on the right when a single- or double-byte character value was last read from a relational database into the field) The trunc test can resolve to true only when the database column is longer than the field. The value for the test is false after a value is moved to the field or after the field is set to null.
textField (the name of a field in a text form)	One of these: • is • not	One of these: • blanks (for testing whether the value of the text field is or is not limited to blanks or nulls). The test for blanks is based on the user's last input to the form, not on the current contents of the form field; and a test that uses *is* is true in these cases: – The user's last input was blanks or null; or – The user entered no data in the field since the start of the program or since a set statement ran that was of type *set form initial*. • cursor (for testing whether the user left the cursor in the specified text field). • data (for testing whether data other than blanks or nulls is in the specified text field). • modified (for testing whether the field's modified data tag is set, as described in Modified data tag and property). • numeric (for testing whether the value of a field of type CHAR or MBCHAR is or is not numeric).
ConverseVar.eventKey	One of these: • is • not	For details, see *ConverseVar.eventKey*.
sysVar.systemType	One of these: • is • not	For details, see *sysVar.systemType*. You cannot use *is* or *not* to test a value returned by *VGLib.getVAGSysType*.

First operand	Comparison Operator	Second operand
record name	One of these: • is • not	An I/O error value appropriate for the record organization. See *I/O error values*.

The next table lists the comparison operators, each of which is used in an expression that resolves to true or false.

Operator	Purpose
==	The *equality* operator indicates whether two operands have the same value.
!=	The *not equal* operator indicates whether two operands have different values.
<	The *less than* operator indicates whether the first of two operands is numerically less than the second.
>	The *greater than* operator indicates whether the first of two operands is numerically greater than the second.
<=	The *less than or equal to* operator indicates whether the first of two operands is numerically less than or equal to the second.
>=	The *greater than or equal to* operator indicates whether the first of two operands is numerically greater than or equal to the second.
in	The *in* operator indicates whether the first of two operands is a value in the second operand, which references an array. For details, see*in*.
is	The *is* operator indicates whether the first of two operands is in the category of the second. For details, see the previous table.
like	The *like* operator indicates whether the characters in the first of two operands is matched by the second operand, as described in *like operator*.
matches	The *matches* operator indicates whether the characters in the first of two operands is matched by the second operand, as described in *matches operator*
not	The *not* operator indicates whether the first of two operands is not in the category of the second. For details, see the previous table.

The next table and the explanations that follow tell the compatibility rules when the operands are of the specified types.

Primitive type of first operand	Primitive type of second operand
BIN	BIN, DECIMAL, FLOAT, MONEY, NUM, NUMC, PACF, SMALLFLOAT
CHAR	CHAR, DATE, HEX, MBCHAR, NUM, TIME, TIMESTAMP
DATE	CHAR, DATE, NUM, TIMESTAMP
DBCHAR	DBCHAR
DECIMAL	BIN, DECIMAL, FLOAT, MONEY, NUM, NUMC, PACF, SMALLFLOAT
HEX	CHAR, HEX
MBCHAR	CHAR, MBCHAR

Primitive type of first operand	Primitive type of second operand
MONEY	BIN, DECIMAL, FLOAT, MONEY, NUM, NUMC, PACF, SMALLFLOAT
NUM	BIN, CHAR, DATE, DECIMAL, FLOAT, MONEY, NUM, NUMC, PACF, SMALLFLOAT, TIME
NUMC	BIN, DECIMAL, FLOAT, MONEY, NUM, NUMC, PACF, SMALLFLOAT
PACF	BIN, DECIMAL, FLOAT, MONEY, NUM, NUMC, PACF, SMALLFLOAT
TIME	CHAR, NUM, TIME, TIMESTAMP
TIMESTAMP	CHAR, DATE, TIME, TIMESTAMP
UNICODE	UNICODE

Details are as follows:

- A value of any of the numeric types (BIN, DECIMAL, FLOAT, MONEY, NUM, NUMC, PACF, SMALLFLOAT) can be compared to a value of any numeric type and size, and EGL does temporary conversions as appropriate. An equality comparison of equivalent fractions (like 1.4 and 1.40) evaluates to true, even if the decimal places are different.
- A value of type CHAR can be compared to a value of type HEX only if each character of type CHAR is within the range of hexadecimal digits (0-9, A-F, a-f). EGL temporarily converts any lowercase letters to uppercase in the value of type CHAR.
- If a comparison includes two values of character type (CHAR, DBCHAR, HEX, MBCHAR, UNICODE) and one value has fewer bytes than the other, a temporary conversion pads the shorter value on the right:
  - In a comparison with a value of type MBCHAR, a value of type CHAR is padded on the right with single-byte blanks
  - In a comparison with a value of type HEX, a value of type CHAR is padded on the right with binary zeros
  - A value of type DBCHAR is padded on the right with double-byte blanks
  - A value of type UNICODE is padded on the right with Unicode double-byte blanks
  - A value of type HEX is padded on the right with binary zeros, which means (for example) that if a value ″0A″ must be expanded to two bytes, the value for comparison purposes is ″0A00″ rather than ″000A″
- A value of type CHAR can be compared to a value of type NUM only if these conditions apply:
  - The value of type CHAR has single-byte digits, with no other characters
  - The definition of the value of type NUM has no decimal point

  A CHAR-to-NUM comparison works as follows:
  - A temporary conversion puts the NUM value into a CHAR format. The numeric characters are left-justified, with additional single-byte blanks as needed. If a NUM-type field of length 4 has a value of 7, for example, the value is treated as ″7″ with three blanks on the right.
  - If the length of the fields do not match, a temporary conversion pads the shorter value with blanks on the right.
  - The comparison checks the values byte-by-byte. Consider two examples:

- A CHAR-type field of length 2 and value ″7 ″ (including a blank) is equal to a NUM-type field of length 1 and value 7 because the temporary field that is based on the NUM-type field also includes a final blank
- A CHAR-type field of value ″8″ is greater than a NUM-type field of value of 534 because the ″8″ comes after ″5″ in the ASCII or EBCDIC search order

Complex logical expressions

You can build a more complex expression by using either an *and* (&&) or *or* operator (||) to combine a pair of more elementary expressions. In addition, you can use the *not* operator (!), as described later.

If a logical expression is composed of elementary logical expressions that are combined by *or* operators, EGL evaluates the expression in accordance with the rules of precedence, but stops the evaluation if one of the elementary logical expressions resolves to true. Consider an example:

```
field01 == field02 || 3 in array03 || x == y
```

If field01 does not equal field02, evalution proceeds. If the value 3 is in array03, however, the overall expression is proven to be true, and the last elementary logical expression (x == y) is not evaluated.

Similarly, if elementary logical expressions are combined by *and* operators, EGL stops the evaluation if one of the elementary logical expressions resolves to false. In the following example, evaluation stops as soon as field01 is found to be unequal to field02:

```
field01 == field02 && 3 in array03 && x == y
```

You may use paired parentheses in a logical expression for any of these purposes:

- To change the order of evaluation.
- To clarify your meaning.
- To make possible the use of the *not* operator (!), which resolves to a Boolean value (true or false) opposite to the value of a logical expression that immediately follows. The subsequent expression must be in parentheses.

Examples

In reviewing the examples that follow, assume that value1 contains ″1″, value2 contains ″2″, and so on:

```
/* == true */
value5 < value2 + value4

/* == false */
!(value1 is numeric)

/* == true when the generated output runs
          on Windows 2000, Windows NT,
          or z/OS UNIX System Services */
sysVar.systemType is WIN || sysVar.systemType is USS

/* == true */
(value6 < 5 || value2 + 3 >= value5) && value2  == 2
```

Related concepts

“Modified data tag and modified property” on page 150

Related tasks

“Syntax diagram for EGL statements and commands” on page 733

Related reference
"case" on page 549
"Datetime expressions" on page 483
"Exception handling" on page 89
"Expressions" on page 482
"I/O error values" on page 522
"if, else" on page 591
"in operator" on page 518
"like operator" on page 636
"like operator" on page 636
"Numeric expressions"
"Operators and precedence" on page 653
"Primitive types" on page 31
"Text expressions" on page 492
"eventKey" on page 895
"getVAGSysType()" on page 892
"systemType" on page 911
"while" on page 629

Numeric expressions

A *numeric expression* resolves to a number, and you specify such an expression in various situations; for example, on the right side of an assignment statement. A numeric expression may be composed of any of these:

- A numeric operand, which is one of these:
  - A variable that contains a number. The item may be preceded with a sign.
  - A numeric literal, which may begin with a sign, but always has a series of digits and may include a single decimal point.
  - A function invocation that returns a number.

  The type of a numeric literal is implied by the value of that literal:
  - An integer of 4 digits or less is of type SMALLINT
  - An integer of 5 to 8 digits is of type INT
  - An integer of 9 to 18 digits is of type BIGINT
  - A number that includes a decimal point is of type NUM
- A numeric operand, followed by a numeric operator, followed by a second numeric operand.
- A more complex expression formed by using a numeric operator to combine a pair of more elementary expressions.

You may use paired parentheses in a numeric expression to change the order of evaluation or to clarify your meaning.

In reviewing the examples that follow, assume that intValue1 equals 1, intValue2 equals 2, and so on, and that each value has no decimal places:

```
/* == -8, with the parentheses overriding
         the usual precedence of * and + */
intValue2 * (intValue1 - 5)

/* == -2, with a unary minus as the last operator */
intValue2 + -4

/* == 1.4, if the expression is assigned to an
          item with at least one decimal place.  */
intValue7 / intValue5
```

```
/* == 2, which is a remainder
        expressed as an integer value */
intValue7 % intValue5
```

For an example that shows the effect of parentheses on the use of a plus (+) sign, see *Expressions*.

For COBOL output, a numeric expression may give an unexpected result if an intermediate, calculated value has more than 30 or 31 digits; the exact number of digits depends on the ARITH compiler option.

For Java output, a numeric expression may give an unexpected result if an intermediate, calculated value requires more than 128 bits.

Related reference
"Datetime expressions" on page 483
"Expressions" on page 482
"Logical expressions" on page 484
"Operators and precedence" on page 653
"Primitive types" on page 31
"Text expressions"

Text expressions

A *text expression* resolves to a series of characters, and you specify such an expression in various situations; for example, on the right side of an assignment statement. The text expression may be any of these:

- A variable that contains a series of characters.
- A *string literal*, which is a series of characters delimited by double quote marks. The literal is of type STRING.
- A substring of a literal or variable that contains a series of characters. For details, see *Substrings*.
- The invocation of any string-formatting system word that returns a series of characters. For details, see *String formatting (system words)*.
- A series of values of the previous kinds, where each value is separated from the next by the concatenation operator, which is a plus sign (+). The following statement assigns *WebSphere* to myString:

  ```
  myString = "Web" + "Sphere";
  ```

  For an example that shows the effect of parentheses on the use of a plus (+) sign, see *Expressions*.
- Any other function invocation that returns a series of characters.

Any character preceded with the escape character (\) is included in the text expression. In particular, you can use the escape character to include the following characters in a literal, field, or return value:

- A double quote mark (")
- A backslash (\)
- A backspace, as indicated by \b
- A form feed, as indicated by \f
- A newline character, as indicated by \n
- A carriage return, as indicated by \r
- A tab, as indicated by \t

Examples are as follows:

```
myString  = "He said, \"Escape while you can!\"";
myString2 = "Is a backslash (\\) needed?";
```

An error occurs if a literal has no ending quote mark:

```
myString3 = "Escape is impossible\";
```

Each value in the text expression must be valid for the context in which the expression is used. For example, an item of type UNICODE cannot be used in an expression assigned to an item of type CHAR. Additional details are in *Assignments*.

Related reference
"Assignments" on page 352
"Datetime expressions" on page 483
"Expressions" on page 482
"Logical expressions" on page 484
"Numeric expressions" on page 491
"Operators and precedence" on page 653
"Primitive types" on page 31

"Substrings" on page 731

Format of master build descriptor plugin.xml file

The master build descriptor plugin.xml file is an XML file that the workbench uses to specify the name and file path name of the master build descriptor. You need this only if you need a master build descriptor to enforce certain options to be used for generation and you are generating from the workbench or are using the EGLCMD command. You must put this plugin.xml file in a directory in the plugins directory. The format of the file is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
 id="id"
 name="plg"
 version="5.0"
 vendor-name="com">
  <requires />
  <runtime />
  <extension point =
 "com.ibm.etools.egl.generation.base.framework.masterBuildDescriptor">
    <masterBuildDescriptor file = "bfil" name = "mas" />
  </extension>
</plugin>
```

where:

id The identifier for the plug-in

plg
The name of the plug-in

com
The name of your company

bfil
The path name of a file containing a master build descriptor, of the form *project/folder/file*, relative to Enterprise Developer's workspace directory, where:

project
The name of the project directory

folder
The name of a directory within the project directory
file The name of a file that contains a master build descriptor
mas
The name of a master build descriptor

The content of this file must follow the rules of an XML file. To separate file names within a path name you must use the slash (/) character.

You must specify both the name attribute and the file attribute. Otherwise the plugin.xml file is ignored.

Following is an example of the contents of plugin.xml file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Example master BuildDescriptor Plugin -->

<plugin
 id="example.master.BuildDescriptor.plugin"
 name="Example master BuildDescriptor plug-in"
 version="5.0"
 vendor-name="IBM">
  <requires />
  <runtime />
  <!-- ================================================================================ -->
  <!--                                                                                  -->
  <!-- Register the master BuildDescriptor                                              -->
  <!--                                                                                  -->
  <!-- ================================================================================ -->
  <extension point =
 "com.ibm.etools.egl.generation.base.framework.masterBuildDescriptor" >
    <masterBuildDescriptor file
 = "myProject/myFolder/myFile.eglbld" name = "masterBD" />
  </extension>
</plugin>
```

Related concepts
"Build descriptor part" on page 275
"Master build descriptor" on page 278

Related tasks
"Generating from the workbench batch interface" on page 312
"Generating in the workbench" on page 310

Related reference
"Build descriptor options" on page 359
"EGLCMD" on page 466
"Format of eglmaster.properties file" on page 478

FormGroup part in EGL source format

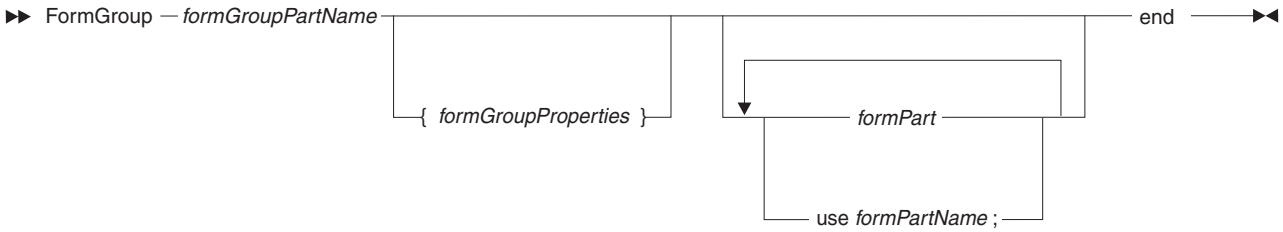
You declare a formGroup part in an EGL file, which is described in *EGL source format*. This part is a primary part, which means that it must be at the top level of the file and must have the same name as the file.

A program can only use forms that are associated with a form group referenced by the program's use declaration.

An example of a formGroup part is as follows:

```
FormGroup myFormGroup
  {
    validationBypassKeys = [pf3],
    helpKey = "pf1",
    pfKeyEquate = yes,
    screenFloatingArea
      {
        screenSize = [24,80],
        topMargin = 0,
        bottomMargin = 0,
        leftMargin = 0,
        rightMargin = 0
      },
    printFloatingArea
      {
        pageSize = [60,80],
        topMargin = 3,
        bottomMargin = 3,
        leftMargin = 5,
        rightMargin = 5
      }
  }
  use myForm01;
  use myForm02;
end
```

The diagram of a formGroup part is as follows:



FormGroup *formGroupPartName* **... end**
Identifies the part as a form group and specifies the part name. For the rules of naming, see *Naming conventions*.

formGroupProperties
A series of properties, each separated from the next by a comma. Each property is described later.

formPart
A text or print form, as described in *Form part in EGL source format*.

use *formPartName*
A use declaration that provides access to a form that is not embedded in the form group.

The form group properties are as follows:

alias
A string that is incorporated into the names of generated output. If you do not specify an alias, the formGroup-part name is used instead.

validationBypassKeys = [*bypassKeyValue***]**
Identifies one or more user keystrokes that causes the EGL run time to skip input-field validations. This property is useful for reserving a keystroke that ends the program quickly. Each *bypassKeyValue* option is as follows:

pf*n*
The name of an F or PF key, including a number between 1 and 24, inclusive.

Note: Function keys on a PC keyboard are often *F* keys such as F1, but EGL uses the IBM *PF* terminology so that (for example) F1 is called PF1.

If you wish to specify more than one key value, delimit the set of values with brackets and separate each value from the next with a comma, as in the following example:

```
validationBypassKeys = [pf3, pf4]
```

helpKey = *"helpKeyValue"*
Identifies a user keystroke that causes the EGL run time to present a help form to the user. The *helpKeyValue* option is as follows:

pf*n*
The name of an f or pf key, including a number between 1 and 24, inclusive.

Note: Function keys on a PC keyboard are often *f* keys such as f1, but EGL uses the IBM *pf* terminology so that (for example) f1 is called pf1.

pfKeyEquate = yes, pfKeyEquate = no
Specifies whether the keystoke that is registered when the user presses a high-numbered function key (PF13 through PF24) is the same as the keystroke that is registered when the user presses a function key that is lower by 12. For details, see *pfKeyEquate*.

screenFloatingArea { *properties* **}**
Defines the floating area used for output to a screen. For an overview of floating areas, see *Form part*. For property details, see the next section.

printFloatingArea { *properties* **}**
Defines the floating area used for printable output. For an overview of floating areas, see *Form part*. For property details, see *Properties of a print floating area*.

Properties of a screen floating area

The set of properties after **screenFloatingArea** is delimited by braces ({ }), and each property is separated from the next by a comma. The properties are as follows:

screenSize = [*rows***,** *columns***]**
Number of rows and columns in the online presentation area, including any lines or columns used as margins. The default is as follows:

```
screenSize=[24,80]
```

topMargin= *rows*
Number of lines left blank at the top of the presentation area. The default is 0.

bottomMargin= *rows*
Number of lines left blank at the bottom of the presentation area. The default is 0.

leftMargin= *columns*
Number of columns left blank at the left of the presentation area. The default is 0.

rightMargin= *columns*
Number of columns left blank at the right of the presentation area. The default is 0.

Properties of a print floating area

The set of properties after **printFloatingArea** is delimited by braces ({ }), and each property is separated from the next by a comma. The properties are as follows:

pageSize = [*rows***,** *columns***]**
Number of rows and columns in the printable presentation area, including any lines or columns used as margins. This property is required if you specify a print floating area.

deviceType = singleByte, deviceType = doubleByte
Specifies whether the floating-area declaration is for a printer that supports single-byte output (as is the default) or double-byte output. Specify **doubleByte** if any of the forms include items of type DBCHAR or MBCHAR.

topMargin = *rows*
Number of lines left blank at the top of the presentation area. The default is 0.

bottomMargin = *rows*
Number of lines left blank at the bottom of the presentation area. The default is 0.

leftMargin = *columns*
Number of columns left blank at the left of the presentation area. The default is 0.

rightMargin = *columns*
Number of columns left blank at the right of the presentation area. The default is 0.

Related concepts
"EGL projects, packages, and files" on page 13
"FormGroup part" on page 143
"Form part" on page 144

Related reference
"EGL source format" on page 478
"Form part in EGL source format"
"Naming conventions" on page 652
"pfKeyEquate" on page 666
"Use declaration" on page 930

Form part in EGL source format

You declare a form part in an EGL file, which is described in *EGL source format*. If a form part is accessed by only one form group, it is recommended that the form part be embedded in the formGroup part. If a form part is accessed by multiple form groups, it is necessary to specify the form part at the top level of an EGL file.

An example of a text form is as follows:

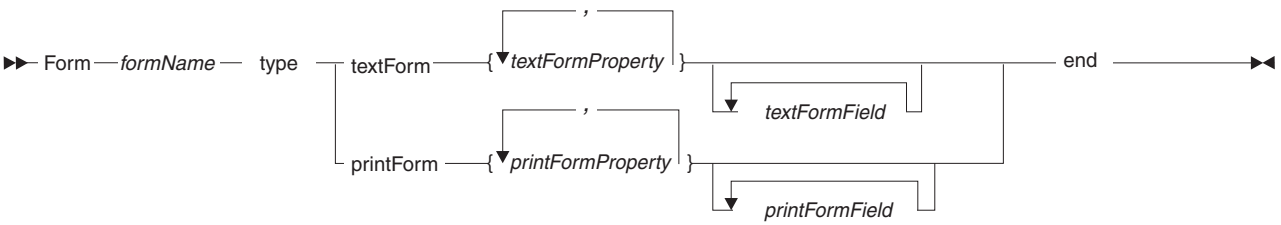
```
Form myTextForm type textForm
  {
    formsize= [24, 80],
    position= [1, 1],
    validationBypassKeys=[pf3, pf4],
    helpKey="pf1",
```

```
    helpForm="myHelpForm",
    msgField="myMsg",
    alias = "form1"
  }

   * { position=[1, 31], value="Sample Menu" } ;
   * { position=[3, 18], value="Activity:" } ;
   * { position=[3, 61], value="Command Code:" } ;

   activity char(42)[5] { position=[4,18], protect=skip } ;

   commandCode char(10)[5] { position=[4,61], protect=skip } ;

   * { position=[10, 1], value="Response:" } ;
   response char(228) { position=[10, 12], protect=skip } ;

   * { position=[13, 1], value="Command:" } ;
   myCommand char(70) { position=[13,10] } ;

   * { position=[14, 1], value="Enter=Run F3=Exit"} ;

    myMsg char(70) { position=[20,4] };

end
```

An example of a print form is as follows:

```
Form myPrintForm type printForm
  {
    formsize= [48, 80],
    position= [1, 1],
    msgField="myMsg",
    alias = "form2"
  }
  * { position=[1, 10], value="Your ID: " } ;
  ID char(70) { position=[1, 30] };
  myMsg char(70) { position=[20, 4] };
end
```

The diagram of a form part is as follows:



Form *formName* **... end**
Identifies the part as a form and specifies the part name. For the rules of naming, see Naming conventions.

textForm
Indicates that the form is a text form.

textFormProperty
A text-form property. For details, see *Text form*.

textFormField
A text-form field. For details, see *Form fields*.

printForm
Indicates that the form is a print form.

printFormProperty
A print-form property. For details, see *Print form*.

printFormField
A print-form field. For details, see *Form fields*.

Text-form properties

The text-form properties are as follows:

formSize = [*rows***, *columns***]**
Number of rows and columns in the online presentation area. This property is required.

The column value is equivalent to the number of single-byte characters that can be displayed across the presentation area.

position = [*row***, *column***]**
Row and column at which the form is displayed in the presentation area. If you omit this property, the form is a floating form and is displayed in the floating area, at the next free line where the entire form can fit in the floating area.

validationBypassKeys = [*bypassKeyValue***]**
Identifies one or more user keystrokes that causes the EGL run time to skip input-field validations. This property is useful for reserving a keystroke that ends the program quickly. The *bypassKeyValue* option is as follows:

pf*n*
The name of an F or PF key, including a number between 1 and 24, inclusive

Note: Function keys on a PC keyboard are often *F* keys such as F1, but EGL uses the IBM *PF* terminology so that (for example) F1 is called PF1.

If you wish to specify more than one key value, delimit the set of values with parentheses and separate each value from the next with a comma, as in the following example:

```
validationBypassKeys = [pf3, pf4]
```

helpKey = *"helpKeyValue"*
Identifies a user keystroke that causes the EGL run time to present a help form to the user. The *helpKeyValue* option is as follows:

pf*n*
The name of an F or PF key, including a number between 1 and 24, inclusive

Note: Function keys on a PC keyboard are often *F* keys such as F1, but EGL uses the IBM *PF* terminology so that (for example) F1 is called PF1.

helpForm = *"formName"*
Name of the help form that is specific to the text form.

msgField = *"fieldName"*
Name of the text-form field that displays a message in response to a validation error or in response to the running of ConverseLib.displayMsgNum.

alias = *"alias"*
An alias of 8 characters or less, for use by the EGL run time. An alias is necessary in a COBOL environment if the form name is longer than 8 characters.

Print-form properties

The print-form properties are as follows:

formsize = [*rows***, *columns***]**
Number of rows and columns in the online presentation area. This property is required.

The column value is equivalent to the number of single-byte characters that can be displayed across the presentation area.

position = [*row***. *column***]**
Row and column at which the form is displayed in the presentation area. If you omit this property, the form is a floating form and is displayed in the floating area, at the next free line where the entire form can fit in the floating area.

addSpaceForSOSI = yes, addSpaceForSOSI = no)
Directs report production in COBOL environments. If you set the property to *no*, the EGL run time prints the line as is, including any shift-out/shift-in (SO/SI) characters that are in fields of type MBCHAR. If you set the property to *yes* (the default), the EGL run time puts a space in place of each SO/SI character.

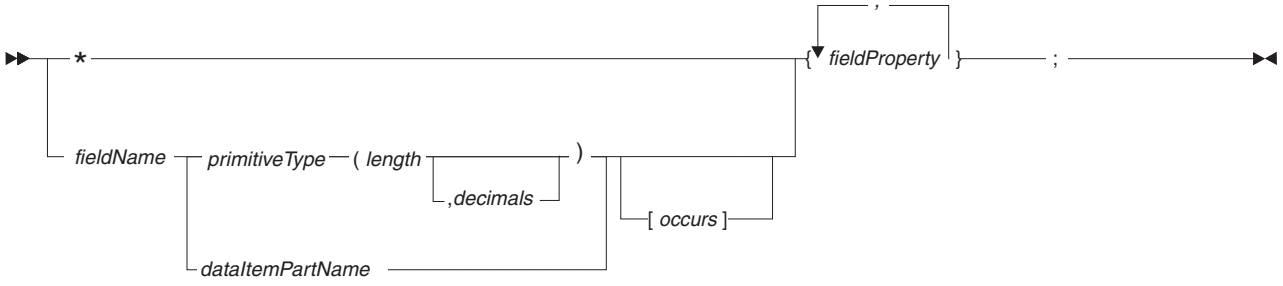
To cause the printed form to use the same columns as in the form definition, specify *no* for printers that print a space for each SO or SI character and specify *yes* for printers that strip SO or SI characters from the print line.

msgField = *"fieldName"*
Name of the text-form field that displays a message in response to the running of ConverseLib.displayMsgNum.

alias = *"alias"*
An alias of 8 characters or less, for use by the EGL run time. An alias is necessary in a COBOL environment if the form name is longer than 8 characters.

Form fields

The diagram of a form field is as follows:



* Indicates that the field is a constant field. It has no name but has a constant value, which is specified in the field-specific **value** property. Statements in your code cannot access the value in a constant field.

fieldProperty
A text-form field property. For details, see *Text-form field properties*.

fieldName
Specifies the name of the field. For rules, see *Naming conventions*.

Your code can access the value of a named field, which is also called a *variable field*.

If a text form contains a variable field that starts on one line and ends on another, the text form can be displayed only on screens where the screen width equals the width of the form.

occurs
The number of elements in a field array. Only one-dimensional arrays are supported. For further details, see *For field arrays*.

primitiveType
The primitive type assigned to the field. This specification affects the maximum length; but any numeric field is generated as type NUM.

Forms that contain fields of type DBCHAR can only be used on systems and devices that support double-byte character sets. Similarly, forms that contain fields of type MBCHAR can only be used on systems and devices that support multiple-byte character sets.

The primitive types FLOAT, SMALLFLOAT, and UNICODE are not supported for text or print forms.

length
The field's length, which is an integer that represents the maximum number of characters or digits that can be placed in the field.

decimals
For a numeric type (BIN, DECIMAL, NUM, NUMC, or PACF), you may specify *decimals*, which is an integer that represents the number of places after the decimal point. The maximum number of decimal positions is the smaller of two numbers: 18 or the number of digits declared as *length*. The decimal point is not stored with the data.

dataItemPartName
The name of a dataItem part that is a model of format for the field, as described in*typeDef*. The dataItem part must be visible to the form part, as described in *References to parts*.

Text-form field properties

Properties that are useful only in text-form fields are described later. The following properties are used more widely and also available:

- "align" on page 670
- "currency" on page 674
- "currencySymbol" on page 674
- "dateFormat" on page 675
- "fillCharacter" on page 679
- "isBoolean" on page 682
- "lineWrap" on page 684
- "lowerCase" on page 685
- "masked" on page 685
- "numericSeparator" on page 689

- "outline" on page 689
- "sign" on page 693
- "timeFormat" on page 695
- "timeStampFormat" on page 696
- "upperCase" on page 697
- "zeroFormat" on page 703

For any field

The following properties are useful for any field on a form:

position = [*row***.** *column***]**
Row and column of the attribute byte that precedes the field. This property is required.

value = *"stringLiteral"*
A character string that is displayed in the field. Quotes are required.

This property can be specified for any item; for example, in a dataItem part declaration.

Note: If VisualAge Generator compatibility is in effect and you set the text-form property **value**, the content of that property is available in the program only after the user has returned the form. For this reason, the value that you set in the program does not need to be valid for the item in the program.

fieldLen = *lengthInBytes*
Field length; the number of single-byte characters that can be displayed in the field. This value does not include the preceding attribute byte.

The value of **fieldLen** for numeric fields must be great enough to display the largest number that can be held in the field, plus (if the number has decimal places) a decimal point. The value of **fieldLen** for a field of type CHAR, DBCHAR, MBCHAR, or UNICODE must be large enough to account for the double-byte characters, as well as any shift-in/shift-out characters.

The default **fieldLen** is the number of bytes needed to display the largest number possible for the primitive type, including all formatting characters.

For variable text fields

The following properties are useful for variable text fields:

cursor = no, cursor = yes
Indicates whether the on-screen cursor is at the beginning of the field when the form is first displayed. Only one field in the form can have the cursor property set to *yes*. The default is *no*.

detectable = no, detectable = yes
Specifies whether the field's modified data tag is set when the field is selected by a light pen or (for emulator sessions) by a cursor click.

The **detectable** property is available only for COBOL programs and only for text-form fields whose **intensity** property is other than *invisible*.

The initial character in the field content (as specified in the **value** property) must be a *designator character*, which indicates what action is taken when the user clicks on the field. The most common designator characters are as follows:

& Causes an *immediate detect*, which means that clicking the field at run time is equivalent to modifying the field and pressing the ENTER key.

? Causes a *delayed detect*, which means that clicking the field at run time is equivalent to modifying the field, but that the program receives the form information only when the user presses the ENTER key or clicks a field that is configured for an immediate detect.

To prevent the user from changing the designator character in a variable field, set the **protect** property to *yes* or *skip*.

modified = no, modified = yes
Indicates whether the program will consider the field to have been modified, regardless of whether the user changed the value. For details, see *Modified data tag and modified property*.

The default is *no*.

protect = no, protect = skip, protect = yes
Specifies whether the user can access the field. Valid values are as follows:

no (the default for variable fields)
Sets the field so that the user can overwrite the value in it.

skip (the default for constant fields)
Sets the field so that the user cannot overwrite the value in it. In addition, the cursor skips the field in either of these cases:

- The user is working on the previous field in the tab order and either presses **Tab** or fills that previous field with content; or
- The user is working on the next field in the tab order and presses **Shift Tab**.

yes
Sets the field so that the user cannot overwrite the value in it.

validationOrder = *integer*
Indicates the field's position in the validation order. The default order in which the fields are validated is the order of the fields on screen, left to right, top to bottom.

For field arrays

One-dimensional arrays are supported on text and print forms. In an array declaration, the value of the **occurs** property is greater than 1, as in this example:

```
myArray char(1)[3];
```

Array elements are positioned in relation to the placement specified for the first element in the array. The default behavior is to position the elements vertically on consecutive rows.

Use the following properties to vary the default behavior:

columns = *numberOfElements*
Number of array elements in each row. The default is 1.

linesBetweenRows = *numberOfLines*
Number of lines between each row that contains array elements. The default is 0.

spacesBetweenColumns = *numberOfSpaces*
Number of spaces between each array element. The default is 1.

indexOrientation = down, indexOrientation = across
Specifies how the program references the elements of an array:

- If you set **indexOrientation** to *down*, elements are numbered from top to bottom, then left to right, so that the elements in a given column are numbered sequentially. The value of **indexOrientation** is *down* by default.
- If you set **indexOrientation** to *across*, elements are numbered from left to right, then top to bottom, so that the elements in a given row are numbered sequentially.

You can override properties for an array element. In the following field declaration, for example, the **cursor** property is overridden in the second element of myArray:

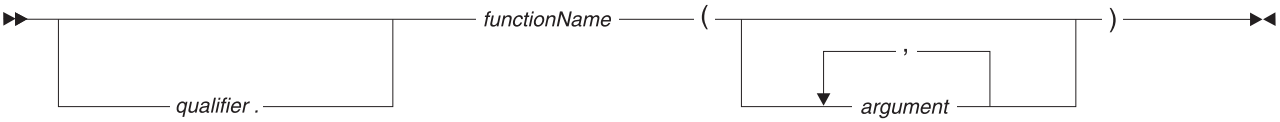
```
myArray char(10)[5]
  {position=[4,61], protect=skip, myArray[2] { cursor = yes} };
```

Related concepts
"Modified data tag and modified property" on page 150
"Overview of EGL properties" on page 60
"Print forms" on page 146
"References to parts" on page 20
"Text forms" on page 148
"Typedef" on page 25

Related reference
"Field-presentation properties" on page 62
"Formatting properties" on page 62
"Naming conventions" on page 652
"NUM" on page 48
"Primitive types" on page 31
"displayMsgNum()" on page 766
"Validation properties" on page 63

Function invocations

A function invocation runs an EGL-generated function or a system function. When the invoked function ends, processing continues either with the statement that follows the invocation or (in complex cases) with the next process required in an expression or in a list of arguments.



qualifier
One of the following symbols:
- The name of the library in which the function resides; or
- The name of the package in which the function resides, optionally followed by a period and the name of the library in which the function resides.
- *this* (identifies a function in the current program)

For details on the circumstances in which the qualifier is unnecessary, see *References to parts*.

function name
Name of the invoked function.

argument
One of the following:

- Literal
- Constant
- Variable
- A more complex numeric, text, or datetime expression, potentially including a function invocation or substring; however, the access modifier for the parameter must be IN

The effect on a variable that is passed as an argument to an EGL-generated function depends on whether the corresponding parameter is modified with IN, OUT, or INOUT. For details, see *Function parameters*.

If the invoked function returns a value, you can use the invocation in these ways:

- As a complete EGL statement (in which case the function does not return a value and is followed by a semicolon).
- As the source value in an assignment statement.
- As an operand in an expression.
- As an argument in the invocation of a function

  A function invoked as in a function invocation can cause the side effect of a variable changing values when the same variable is used in the function or even in the function invocation itself. Consider this example, which assumes that the function Sum is returning the sum of three arguments and that the function Increment is adding one to a passed argument:

  ```
  b INT = 1;
  x INT = Sum( Increment(b), b, Increment(b) );
  ```

  If the argument to Increment is related to a parameter modified with INOUT, the effect of the preceding statements is as follows:

  – b = 1
  – The first (leftmost) invocation of Increment revises the value of b, which equals 2 on the return from Increment
  – The second argument in the invocation of Sum is 2
  – The second (rightmost) invocation of Increment revises the value of b, which equals 3 on the return from Increment
  – After Sum runs, x receives the value 7 because the logic in that function used the values 2, 2, and 3

  If the second argument in the invocation of Sum is related to a parameter modified with INOUT, evaluation of that argument occurs after both invocations of Increment. The effect of the preceding code is as follows:

  – b = 1
  – The first (leftmost) invocation of Increment revises the value of b, which equals 2 on the return from Increment
  – The second (rightmost) invocation of Increment revises the value of b, which equals 3 on the return from Increment
  – The logic in Sum begins to run, and only then is the memory associated with the second argument referenced; the value in that memory is equal to 3
  – After Sum runs, x receives the value 8 because the logic in that function used the values 2, 3, and 3

  The general rule is that side effects can be identified by reference to the usual order of evaluation of expressions, which is left to right but can be overridden by parentheses. The use of INOUT is a further complication, as shown.

When the access modifier of a parameter is IN or OUT, the compatibility rules are as described in *Assignment compatibility*. When the access modifier of a parameter is INOUT (or when the parameter is in the onPageLoad function of a pageHandler), the compatibility rules are as described in *Reference compatibility*.

Other rules also apply:

literals
If the access modifier is IN or INOUT, you can code a literal as the argument. The EGL-generated code creates a temporary variable of the parameter type, initializes that variable with the value, and passes the variable to the function.

fixed record
If the argument is a fixed record, the parameter must be a fixed record.

The following rules apply to fixed records that are not of type basicRecord:

- The type of the argument and parameter must be identical
- The access modifier must be of type INOUT

In relation to fixed records that are of type basicRecord, the type of the argument and parameter can vary:

- If the access modifier is of type IN, the size of the argument must be greater than or equal to the size of the parameter.
- If the access modifier is of type OUT or INOUT, the size of the argument must be less than or equal to the size of the parameter.

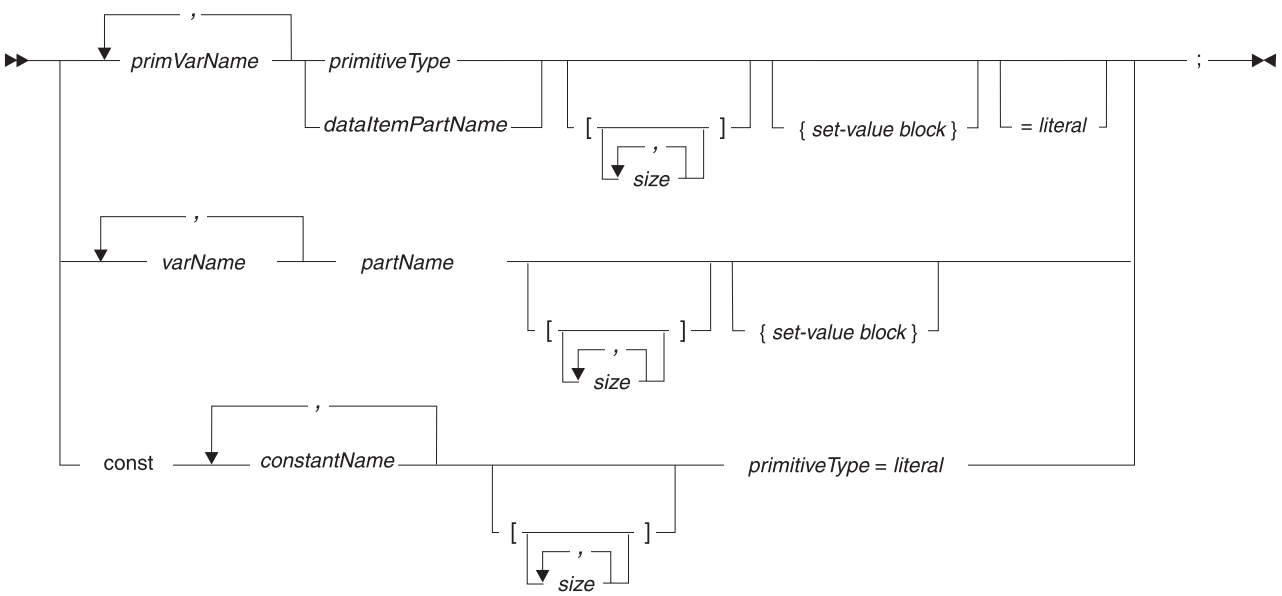
Related concepts
"Function part" on page 132
"References to parts" on page 20
"Syntax diagram for EGL statements and commands" on page 733

Related tasks
"Assignments" on page 352

Related reference
"Assignment compatibility in EGL" on page 347
"EGL statements" on page 83
"Function parameters" on page 508
"Function part in EGL source format" on page 513
"Primitive types" on page 31
"Reference compatibility in EGL" on page 718

Function variables

The syntax diagram for each variable in a function is as follows:



primVarName
Specifies the name of a local primitive variable. For details on usage in the function, see *References to variables and constants*. For other rules, see *Naming conventions*.

primitiveType
The type of a primitive field. Depending on the type, the following information may be required:

- The parameter's length, which is an integer that represents the number of characters or digits in the memory area.
- For some numeric types, you may specify an integer that represents the number of places after the decimal point. The decimal point is not stored with the data.
- For an item of type INTERVAL or TIMESTAMP, you may specify a datetime mask, which assigns a meaning (such as "year digit") to a given position in the item value.

dataItemPartName
The name of a dataItem part that is visible to the program. For details on visibility, see *References to parts*.

The part acts as a model of format, as described in *Typedef*.

size
Number of elements in the array. If you specify the number of elements, the array is initialized with that number of elements.

set-value block
For details, see *Overview of EGL properties* and *Set-value blocks*

= *literal*
Specifies the initial value of the primitive variable.

varName
Name of the variable, which can be of any type that is based on a part.

partName
Name of a part that is visible to the program or is predefined. For details on visibility, see *References to parts*.

The part acts as a model of format, as described in *Typedef*.

const *constantName primitiveType=literal*
Name, type, and value of a constant. Specify a quoted string (for a character type); a number (for a numeric type); or an array of appropriately typed values (for an array). Examples are as follows:

```
const myString String = "Great software!";
const myArray  BIN[] = [36, 49, 64];
const myArray02 BIN[][] = [[1,2,3],[5,6,7]];
```

For the rules of naming, see *Naming conventions*.

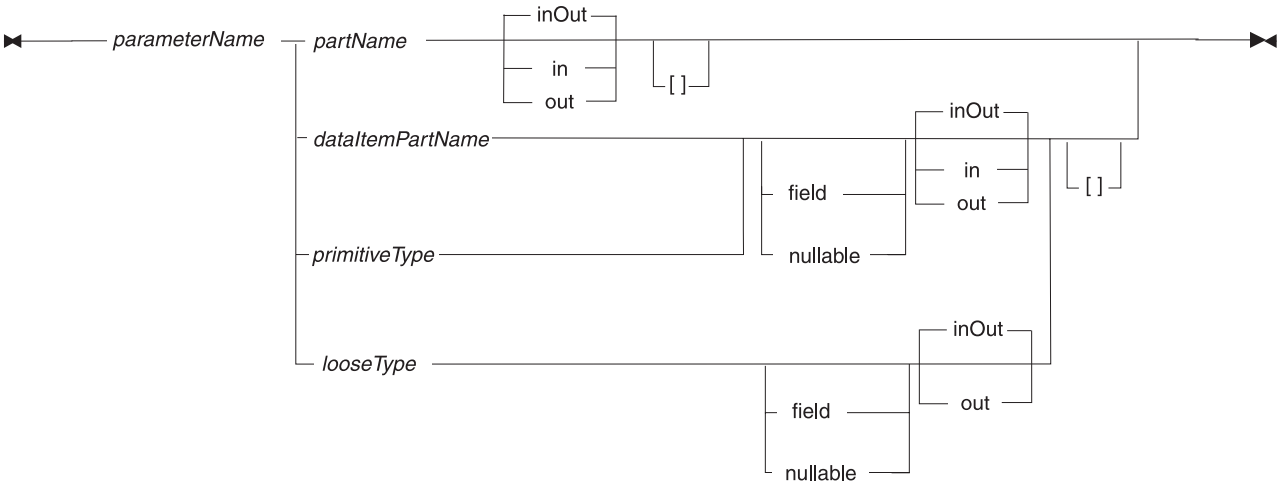
Related concepts
"Function part" on page 132
"Parts" on page 17
"References to parts" on page 20
"References to variables in EGL" on page 55
"Syntax diagram for EGL statements and commands" on page 733
"Typedef" on page 25

Related tasks
"Function part in EGL source format" on page 513

Related reference
"Arrays" on page 69
"INTERVAL" on page 39
"Naming conventions" on page 652
"TIMESTAMP" on page 41

Function parameters

The syntax diagram for a function parameter is as follows:



parameterName
Specifies the name of a parameter, which may be a record or data item; or an array of records or data items. For rules, see *Naming conventions*.

If you specify the modifier **inOut** or **out**, any changes that are made to the parameter value are available in the invoking function. Those modifiers are described later and in the section "Implications of inOut and the related modifiers" on page 511.

A parameter is not visible to functions that are invoked by the function containing the parameter; however, a parameter can be passed as an argument to those other functions.

A parameter that ends with brackets ([ ]) is a dynamic array, and the other specifications declare aspects of each element of that array.

inOut
The function receives the argument value as an input, and the invoker receives any changes to the parameter when the function ends. If the argument is a literal or constant, however, the argument is treated as if the modifier **in** were in effect.

The **inOut** modifier is necessary if the parameter is an item and you specify the modifier **field**, which indicates that the parameter has testable, form-field attributes such as *blanks* or *numeric*.

If the parameter is a record (not a fixed record), the **inOut** modifier is the only one that is valid.

If the parameter is a fixed record, the following rules apply:

- If you intend to use that record to access a file or database in the current function (or in a function invoked by the current function), you must specify the **inOut** modifier or accept that modifier by default
- If the type of record is the same for argument and parameter (for example, if both are serial records), the record-specific state information such as end-of-file status is available in the function and is returned to the invoker, but only if the **inOut** modifier is in effect

If the **inOut** modifier is in effect, the related argument must be reference-compatible with the parameter, as described in *Reference Compatibility in EGL*.

in The function receives the argument value as an input, but the invoker is not affected by changes made to the parameter.

You cannot use the **in** modifier for an item that has the modifier **field**. Also, you cannot specify the **in** modifier for a record (other than a fixed record); or for a fixed record that is used to access a file or database either in the current function or in a function invoked by the current function.

out
The function does not receive the argument value as an input; rather, the input value is initialized according to the rules described in *Data Initialization*. The value of the parameter is assigned to the argument when the function returns.

If the argument is a literal or constant, the argument is treated as if the modifier **in** were in effect.

You cannot use the **out** modifier for a parameter that has the modifier **field**. Also, you cannot specify the **out** modifier for a record; or for a fixed record that is used to access a file or database either in the current function or in a function invoked by the current function.

partName
A record part that is visible to the function and that is acting as a typedef (a model of format) for a parameter. For details on what parts are visible, see *References to parts*.

The following statements apply to input or output (I/O) against a fixed record:

- A fixed record passed from another function in the same program includes record state such as the I/O error value *endOfFile*, but only if the record is of the same record type as the parameter. Similarly, any change in the record state is returned to the caller, so if you perform I/O against a record parameter, any tests on that record can occur in the current function, in the caller, or in a function that is called by the current function.

  Library functions do not receive record state.
- Any I/O operation performed against the fixed record uses the record properties specified for the parameter, not the record properties specified for the argument.
- For fixed records of type indexedRecord, mqRecord, relativeRecord, or serialRecord, the file or message queue associated with the record declaration is treated as a run-unit resource rather than a program resource. Local record declarations share the same file (or queue) whenever the record property **fileName** (or **queueName**) has the same value. Only one physical file at a time can be associated with a file or queue name no matter how many records are associated with the file or queue in the run unit, and EGL enforces this rule by closing and reopening files as appropriate.

dataItemPartName
A dataItem part that is visible to the function and that is acting as a typedef (a model of format) for a parameter.

primitiveType
The type of a primitive field. Depending on the type, the following information may be required:

- The parameter's length, which is an integer that represents the number of characters or digits in the memory area.
- For some numeric types, you may specify an integer that represents the number of places after the decimal point. The decimal point is not stored with the data.
- For an item of type INTERVAL or TIMESTAMP, you may specify a datetime mask, which assigns a meaning (such as ″year digit″) to a given position in the item value.

looseType
A loose type is a special kind of primitive type that is used only for function parameters. You use this type if you wish the parameter to accept a range of argument lengths. The benefit is that you can invoke the function repeatedly and can pass an argument of a different length each time.

Valid values are as follows:

- CHAR
- DBCHAR
- HEX
- MBCHAR
- NUMBER
- UNICODE

If you wish the parameter to accept a number of any primitive type and length, specify NUMBER as the loose type. In this case, the number passed to the parameter must not have any decimal places.

If you wish the parameter to accept a string of a particular primitive type but any length, specify CHAR, DBCHAR, MBCHAR, HEX, or UNICODE as the loose type and make sure that the argument is of the corresponding primitive type.

The definition of the argument determines what occurs when a statement in the function operates on a parameter of a loose type.

Loose types are not available in functions that are declared in *libraries*.

For details on primitive types, see *Primitive types*.

field
Indicates that the parameter has form-field attributes such as *blanks* or *numeric*. Those attributes can be tested in a logical expression.

The **field** modifier is available only if you specify the **inOut** modifier or accept the **inOut** modifier by default.

The **field** modifier is not available for function parameters in a library of type nativeLibrary.

nullable
Indicates the following characteristics of the parameter:

- The parameter can be set to null
- The parameter has access to the state information necessary to test for truncation or null in a logical expression

The **nullable** modifier is meaningful only if the argument passed to the parameter is a structure item in an SQL record. The following rules apply:

- The parameter can be set to null and tested for null only if the item property **isNullable** is set to *yes*.
- In a Java program, the ability to test for truncation is available regardless of the value of **isNullable**. In a COBOL program, however, the ability to test for truncation is available only if **isNullable** is set to *yes*.
- You can specify **nullable** regardless of whether the modifier **inOut**, **in**, or **out** is in effect.

Implications of inOut and the related modifiers

To better understand the modifiers **inOut**, **out**, and **in**, review the following example, which shows (in comments) the values of different variables at different points of execution.

```
program inoutpgm
 a int;
 b int;
 c int;

 function main()
  a = 1;
  b = 1;
  c = 1;

  func1(a,b,c);

  // a = 1
```

```
 // b = 3
 // c = 3
end

function func1(x int in, y int out, z int inout)
 // a = 1          x = 1
 // b = 1          y = 0
 // c = 1          z = 1

 x = 2;
 y = 2;
 z = 2;

 // a = 1          x = 2
 // b = 1          y = 2
 // c = 2          z = 2

 func2();
 func3(x, y, z);
 // a = 1          x = 2
 // b = 1          y = 3
 // c = 3          z = 3

end

function func2()
 // a = 1
 // b = 1
 // c = 2

end

function func3(q int in, r int out, s int inout)
 // a = 1          x = unresolved   q = 2
 // b = 1          y = unresolved   r = 2
 // c = 2          z = unresolved   s = 2

 q = 3;
 r = 3;
 s = 3;

 // a = 1          x = unresolved   q = 3
 // b = 1          y = unresolved   r = 3
 // c = 3          z = unresolved   s = 3


end
```

Related concepts
“Function part” on page 132
“Library part of type basicLibrary” on page 133
“Library part of type basicLibrary” on page 133
“Parts” on page 17
“References to parts” on page 20
“References to variables in EGL” on page 55
“Typedef” on page 25

Related reference
“Basic record part in EGL source format” on page 357
“Data initialization” on page 459
“EGL source format” on page 478
“Function part in EGL source format” on page 513
“Indexed record part in EGL source format” on page 520
“INTERVAL” on page 39

"Logical expressions" on page 484
"MQ record part in EGL source format" on page 642
"Naming conventions" on page 652
"Primitive types" on page 31
"Reference compatibility in EGL" on page 718
"Relative record part in EGL source format" on page 719
"Serial record part in EGL source format" on page 722
"SQL record part in EGL source format" on page 726
"TIMESTAMP" on page 41

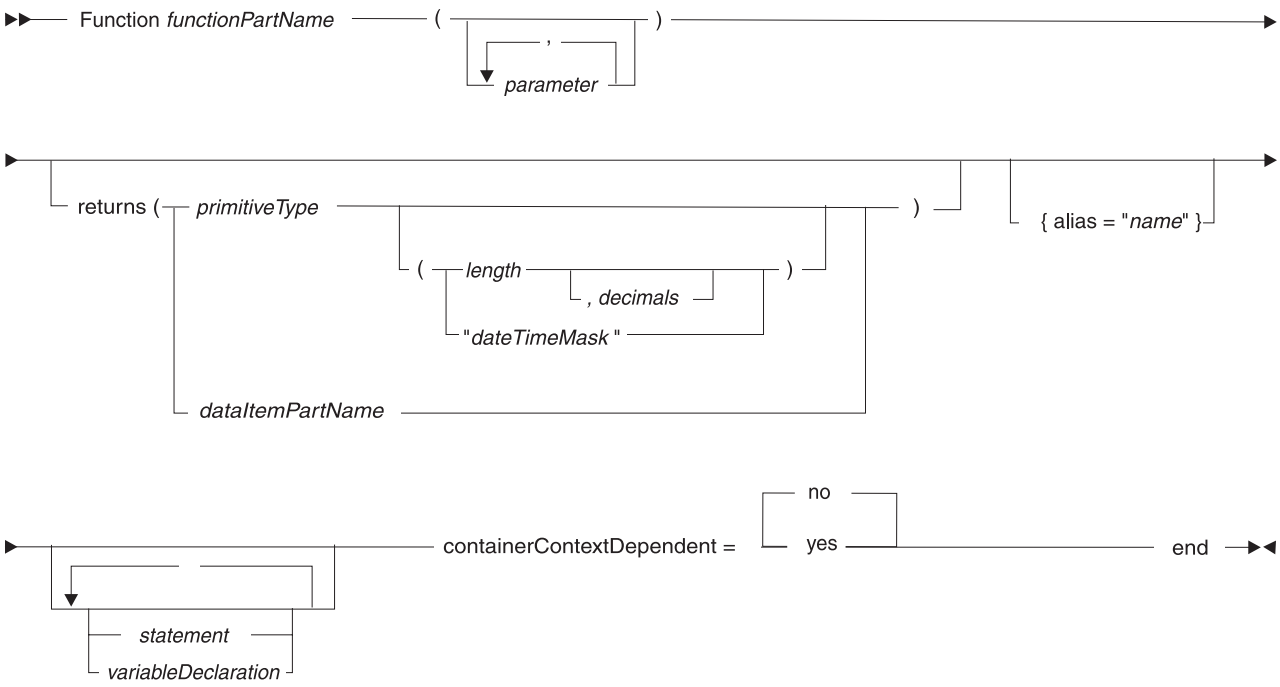
Function part in EGL source format

You can declare functions in an EGL file, as described in *EGL source format*.

The following example shows a program part with two embedded functions, along with a standalone function and a standalone record part:

```
Program myProgram(employeeNum INT)
  {includeReferencedFunctions = yes}

  // program-global variable
    employees record_ws;
    employeeName char(20);

  // a required embedded function
  Function main()

    // initialize employee names
    recd_init();

    // get the correct employee name
    // based on the employeeNum passed
    employeeName = getEmployeeName(employeeNum);
  end

  // another embedded function
  Function recd_init()
    employees.name[1] = "Employee 1";
    employees.name[2] = "Employee 2";
  end
end

// standalone function
Function getEmployeeName(employeeNum INT) returns (CHAR(20))

  // local variable
  index BIN(4);
  index = syslib.size(employees.name);
  if (employeeNum > index)
    return("Error");
  else
    return(employees.name[employeeNum]);
  end

end

// record part that acts as a typeDef for employees
Record record_ws type basicRecord
  10 name CHAR(20)[2];
end
```

The syntax diagram for a function part is as follows:



Function *functionPartName* **... end**
Identifies the part as a function and specifies the part name. For the rules of naming, see *Naming conventions*.

parameter
A parameter, which is an area of memory that is available throughout the function and that may receive a value from the invoking function. For details on the syntax used to declare a parameter, see *Function parameters*.

returns (*returnType***)**
Describes the data that the function returns to the invoker. The characteristics of the return type must match the characteristics of the variable that receives the value in the invoking function.

{alias = *name***}**
Is valid only if the function is in a library of type nativeLibrary. In that context, *name* is the name of the DLL-based function and defaults to the EGL function name. Set the **alias**property explicitly if a validation error occurs when you name the EGL function with the name of the DLL-based function.

dataItemPartName
A dataItem part that is visible to the function and that is acting as a typedef (a model of format) for the return value.

primitiveType
The primitive type of the data returned to the invoker.

length
The length of the data returned to the invoker. The length is an integer that represents the number of characters or digits in the returned value.

decimals
For some numeric types, you may specify *decimals*, which is an integer that represents the number of places after the decimal point. The maximum number of decimal positions is the smaller of two numbers: 18 or the number of digits declared as *length*. The decimal point is not stored with the data.

"dateTimeMask"
For TIMESTAMP and INTERVAL types, you may specify *"dateTimeMask"*, which assigns a meaning (such as "year digit") to a given position in the datetime value. The mask is not stored with the data.

statement
An EGL statement, as described in *EGL statements*. Most end with a semicolon.

variableDeclaration
A variable declaration, as described in *Function variables*.

containerContextDependent
An indication of whether to extend the namespace used to resolve the functions that are invoked by the function being declared. The default is *no*.

This indicator is for use in code that was migrated from VisualAge Generator. For details, see *containerContextDependent*.

Related concepts
"EGL projects, packages, and files" on page 13
"Function part" on page 132
"Import" on page 30
"Library part of type basicLibrary" on page 133
"Library part of type basicLibrary" on page 133
"Parts" on page 17
"References to parts" on page 20
"References to variables in EGL" on page 55
"Syntax diagram for EGL statements and commands" on page 733
"Typedef" on page 25

Related reference
"Arrays" on page 69
"containerContextDependent" on page 453
"EGL statements" on page 83
"Function invocations" on page 504
"Function parameters" on page 508
"Function variables" on page 506
"INTERVAL" on page 39
"I/O error values" on page 522
"Naming conventions" on page 652
"Primitive types" on page 31
"TIMESTAMP" on page 41

Generated output

The next table lists the generated output. For details on the names given to each kind of output file, see *Generated output (reference)*.

Output type	Purpose	Generation type
Build plan	Lists the code-preparation steps that will occur on the target platform	All
COBOL program	Runs as a COBOL program on iSeries	COBOL
Enterprise JavaBean (EJB) session bean	Runs in an EJB container	Java wrapper

Output type	Purpose	Generation type
Java program and related classes	Runs either outside of J2EE or in the context of a J2EE client application, web application, or EJB container	Java
Java wrapper	Invokes an EGL-generated program from non-EGL-generated Java code	Java wrapper
J2EE environment file	Provides entries for insertion into the Java deployment descriptor	Java
Library (generated output)	Provides functions and values for use by other generated output	Java
Linkage properties file	Guides how calls are made from generated Java code, but only if decisions are final at deployment time rather than generation time	Java or Java wrapper
PageHandler part	Creates output that controls a user's run-time interaction with a Web page	Java
Program properties file	Contains Java run-time properties in a format that is accessible only when you are debugging a Java program in a non-J2EE Java project	Java
Results file	Gives status information on the code-preparation steps that occurred on the target platform	All

Related concepts
"Introduction to EGL" on page 1
"Java program, PageHandler, and library" on page 306
"Java runtime properties" on page 327
"Run-time configurations" on page 9

Related tasks
"Building EGL output" on page 305

Related reference
"Generated output (reference)"

Generated output (reference)

The output of EGL generation largely depends on whether you are generating COBOL, Java, or a Java wrapper. The next table shows the file names of generated output that do not come from a specific EGL part.

Output type	File name
"Build plan" on page 305	*alias*BuildPlan.xml
"Enterprise JavaBean (EJB) session bean" on page 295	*alias*EJBHome.java for the home interface, *alias*EJB.java for the remote bean interface, and *alias*EJBBean.java for the bean implementation
"J2EE environment file" on page 336	*alias*-env.txt
"Program properties file" on page 329	*alias*.properties
"Results file" on page 309	*alias*_Results_*timeStamp*.xml

alias
The alias, if any, that is specified in the program part. If the alias is not specified, the name of the program part is used but is truncated (if necessary) to the maximum number of characters allowed in the run-time environment.

Other characteristics of *alias* are determined by the kind of output:

- If you are generating a COBOL program and related output, all letters of *alias* are uppercase
- If you are generating a Java program, the case of each letter in *alias* is taken without change from the source code
- If you are generating a Java wrapper, the rules for naming the wrapper and EJB session bean are as follows:
  - The first letter in *alias* is uppercase
  - Every subsequent letter is lower case, with this exception: any underscore or hyphen is eliminated, and the subsequent letter is uppercase

timeStamp
The date and time when the file was created. The format reflects the settings on the development operating system.

For details on file names, see the appropriate reference topic:

- "Output of COBOL generation" on page 655
- "Output of Java program generation" on page 655
- "Output of Java wrapper generation" on page 656

Related concepts
"Build plan" on page 305

"Enterprise JavaBean (EJB) session bean" on page 295
"Generated output" on page 515
"Generation" on page 301
"J2EE environment file" on page 336
"Program properties file" on page 329
"Results file" on page 309

Related reference
"Output of COBOL generation" on page 655
"Output of Java program generation" on page 655
"Output of Java wrapper generation" on page 656

Generation Results view

The Generation Results view shows you code-preparation messages that are the result of generation performed in the workbench. These messages may be errors, warnings, or informational messages. This view is available only when generating from the Workbench. The format is as follows:

```
msgid message
```

msgid
Is the message identifier. For example, IWN.VAL.4610.e is the message ID for Enterprise Developer validation error number 4610.

message
Is the text of the message.

Generation results are displayed in the view by primary part (program, PageHandler, form group, data table, library), with a different tab for each part. The results can be a combination of validation results and generation results.

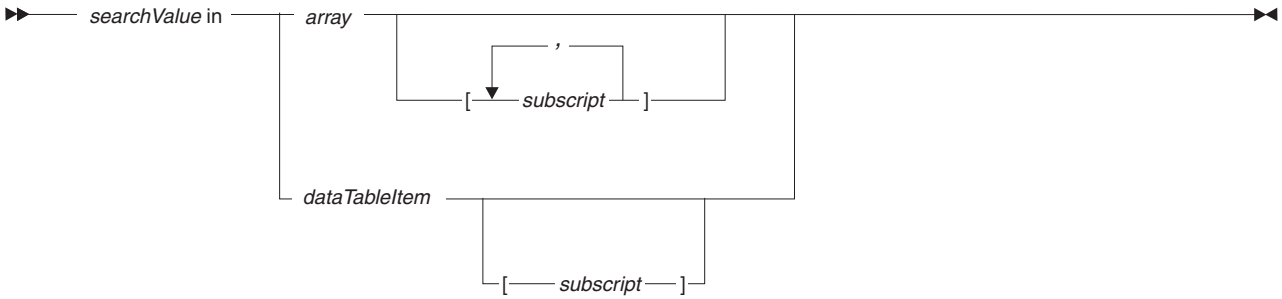
You can open this view at any time, but it displays data only after you generate output.

Related concepts
"Development process" on page 8
"Generated output" on page 515
"Generation" on page 301

Related reference
"Generated output (reference)" on page 516

in operator

The operator **in** is a binary operator used in an elementary logical expression that has the following format:



searchValue
A literal or item, but not a system variable.

array A one-dimensional or multidimensional array. The operator **in** operates on a one-dimensional array, which may be an element of a multidimensional array.

subscript
An integer, or an item (or system variable) that resolves to an integer. The value of a subscript is an index that refers to a specific element in an array.

An item used as a subscript of an array can not itself be an array element. In each of the following examples, myItemB[1] is both a subscript and an array element; as a result, the following syntax is *not* valid:

```
/* the next syntax is not valid */
myItemA[myItemB[1]]

// this next syntax is not valid; but only
// because myItemB is myItemB[1], the
// first element of a one-dimensional array
myItemA[myItemB]
```

dataTableItem
The name of a dataTable item. The item represents a column in the data table. The **in** operator interacts with that column as if the column were a one-dimensional array.

The logical expression resolves to true if the generated program finds the search value. The search begins at the element identified by the last array subscript. If *array* is a one-dimensional array, the last subscript is optional and defaults to 1. If *array* is a multidimensional array, the following statements are true:

- A subscript must be present for each dimension
- The generated program searches the one-dimensional array that is identified by the sequence of subscripts other than the last subscript
- The search begins at the element identified by the last subscript

In relation to both one-dimensional and multidimensional arrays, the search ends at the last element of the one-dimensional array under review.

The logical expression that includes **in** resolves to false in either of these cases:

- The search value is not found
- The value of the last subscript is greater than the number of entries in the one-dimensional array being searched

If the elementary logical expression resolves to true, the operation **in** sets the system variable **sysVar.arrayIndex** to the subscript value of the element that contains the search value. If the expression resolves to false, the operation sets **sysVar.arrayIndex** to zero.

Examples with a one-dimensional array

Let's assume that the structure item myString is substructured to an array of three characters:

```
structureItem name="myString" length=3
  structureItem name="myArray" occurs=3 length=1
```

The next table shows the effect of the operator **in** if myString is ″ABC″.

Logical expression	Value of expression	Value of sysVar. ArrayIndex	Comment
″A″ in myArray	true	1	The subscript of a single-dimension array defaults to 1
″C″ in myArray[2]	true	3	Search begins at second element
″A″ in myArray[2]	false	0	The search ends at the last element

Examples with a multidimension array

Let's assume that the array myArray01D is substructured to an array of three characters:

```
structureItem name="myArray01D" occurs=3 length=3
  structureItem name="myArray02D" occurs=3 length=1
```

In this example, myArray01D is a one-dimensional array, with each element containing a string that is substructured to an array of three characters. myArray02D is a two-dimensional array, with each element (such as myArray02D[1,1]) containing a single character.

If the content of myArray01D is ″ABC″, ″DEF″, and ″GHI″, the content of myArray02D is as follows:

```
"A"   "B"   "C"
"D"   "E"   "F"
"G"   "H"   "I"
```

The next table shows the effect of the operator **in**.

Logical expression	Value of expression	Value of sysVar. ArrayIndex	Comment
″DEF″ in myArray01D	true	2	A reference to a one-dimensional array does not require a subscript; by default, the search begins at the first element
″C″ in myArray02D[1]	—	—	The expression is invalid because a reference to a multidimensional array must include a subscript for each dimension
″I″ in myArray02D[3,2]	true	3	Search begins at the third row, second element
″G″ in myArray02D[3,2]	false	0	Search ends at the last element of the row being reviewed
″G″ in myArray02D[2,4]	false	0	The second subscript is greater than the number of columns available to search

Related tasks
“Syntax diagram for EGL statements and commands” on page 733

Related reference
“Arrays” on page 69
“Logical expressions” on page 484
“Operators and precedence” on page 653
“arrayIndex” on page 901

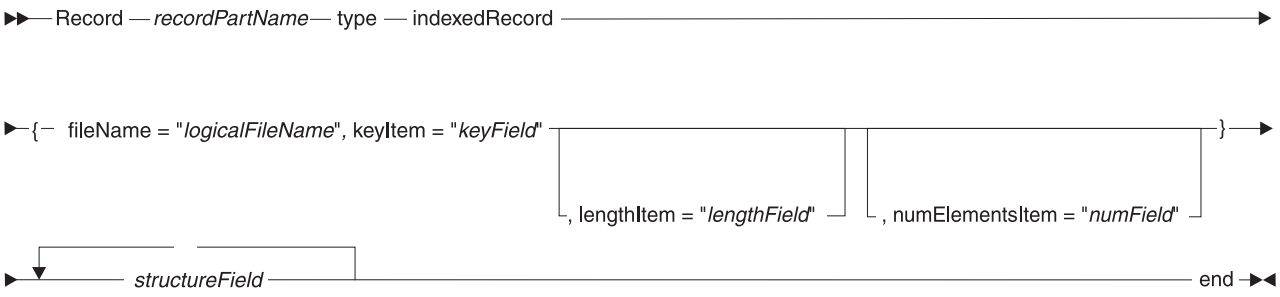
Indexed record part in EGL source format

You declare a record part of type indexedRecord in an EGL file, which is described in *EGL source format*.

An example of an indexed record part is as follows:

```
Record myIndexedRecordPart type indexedRecord
  {
    fileName = "myFile",
    keyItem  = "myKeyItem"
  }
  10 myKeyItem CHAR(2);
  10 myContent CHAR(78);
end
```

The syntax diagram for an indexed record part is as follows:



Record *recordPartName* **indexedRecord**
Identifies the part as being of type indexedRecord and specifies the name. For rules, see *Naming conventions*.

fileName = *"logicalFileName"*
The file name. For details on the meaning of your input, see *Resource associations (overview)*. For rules, see *Naming conventions*.

keyItem = *"keyItem"*
The key item, which can only be a structure item that is unique in the same record. You must use an unqualified reference for *keyItem*; for example, use *myItem* rather than *myRecord.myItem*. (In a function, however, you can reference that structure item as you would reference any structure item.)

lengthItem = *"lengthItem"*
The length item, as described in *Properties that support variable-length records*.

numElementsItem = *"numElementsItem"*
The number of elements item, as described in *Properties that support variable-length records*.

structureItem
A structure item, as described in *Structure item in EGL source format*.

Related concepts

“EGL projects, packages, and files” on page 13
“References to parts” on page 20
“Parts” on page 17
“Record parts” on page 124
“References to variables in EGL” on page 55
“Resource associations and file types” on page 286
“Typedef” on page 25

Related tasks

“Syntax diagram for EGL statements and commands” on page 733

Related reference

“Arrays” on page 69
“DataItem part in EGL source format” on page 461
“EGL source format” on page 478
“Function part in EGL source format” on page 513
“MQ record part in EGL source format” on page 642
“Naming conventions” on page 652
“Primitive types” on page 31
“Program part in EGL source format” on page 707
“Properties that support variable-length records” on page 716
“Relative record part in EGL source format” on page 719

"Serial record part in EGL source format" on page 722
"SQL record part in EGL source format" on page 726
"Structure field in EGL source format" on page 730

I/O error values

The next table describes the EGL error values for input/output (I/O) operations that affect databases, files, and MQSeries message queues. The values associated with hard errors are available to your code only if the system variable VGVar.handleHardIOErrors is set to 1, as described in *Exception handling*.

Error value	Type of error	Type of Record	Meaning of error value
deadLock	Hard	SQL	Two program instances are trying to change a record, but neither can do so without system intervention. If you are accessing an SQL table in DB2, deadlock indicates that the value of sqlcode is -911.
duplicate	Soft	Indexed or Relative	Your code tried to access a record having a key that already exists, and the attempt succeeded. For details, see *duplicate*.
endOfFile	Soft	Indexed, Relative, Serial	For details, see *endOfFile*.
ioError	Hard or Soft	Any	EGL received a non-zero return code from the I/O operation.
format	Hard	Any	The accessed file is incompatible with the record definition. For details, see *format*.
fileNotAvailable	Hard	Any	fileNotAvailable is possible for any I/O operation and could indicate, for example, that another program is using the file or that resources needed to access the file are scarce.
fileNotFound	Hard	Indexed, Message queue, Relative, Serial	A file was not found.
full	Hard	Indexed, Relative, Serial	full is set in these cases: • An indexed or serial file is full
hardIOError	Hard	Any	A hard error occurred, which is any error except endOfFile, noRecordFound, or duplicate.
noRecordFound	Soft	Any	For details, see *noRecordFound*.
unique	Hard	Indexed, Relative, or SQL	UNQ indicates *unique*: your code tried to add or replace a record having a key that already exists, and the attempt failed. For details, see *unique*.

duplicate

For an indexed or relative record, **duplicate** is set in these cases:

- An **add** statement tries to insert a record whose key or record ID already exists in the file or in an alternate index, and the insertion succeeds.

- A **replace** statement overwrites a record successfully, and the replacement values include a key that is the same as the alternate-index key of another record.
- A **get**, **get next**, or **get previous** statement reads a record successfully (or a **set** statement of the form *set record position* runs successfully), and a second record has the same key.

The **duplicate** setting is returned only if the access method returns the information, as is true on some operating systems but not on others. The option is not available during SQL database access.

If you are accessing an emulated VSAM file from an EGL-generated COBOL program on iSeries, see *Association elements* for a description of the **duplicates** property in the resource associations part that is used at generation time.

endOfFile

endOfFile is set in these conditions:

- Your code issues a **get next** statement for a serial or relative record when the related file pointer is at the end of the file. The pointer is at the end when the last record in the file was accessed by a previous **get** or **get next** statement.
- Your code issues a **get next** statement for an indexed record when the related file pointer is at the end of file, as occurs in these situations:
  - The last record in the file was accessed by a previous **get** or **get next** statement; or
  - The last record in the file was accessed by a previous **set** statement of type *set record position* when either of these conditions applies:
    - The key value matched the key of the last record in the file; or
    - Every byte in the key value was set to hexadecimal FF. (If a **set** statement of type *set record position* runs with a key value set to all hexadecimal FF, the statement sets the position pointer to the end of the file.)
- Your code issues a **get previous** statement for an indexed record when the related file pointer is at the beginning of file, as occurs in these situations:
  - The first record in the file was accessed by a previous **get** or **get previous** statement;
  - Your code did not previously access the same file; or
  - A **set** statement of type *set record position* ran with a key when no keys in the file were previous to that key.
- A **get next** statement attempts to retrieve data from an empty or uninitialized file into an indexed record.

  (An empty file is one from which all records have been deleted. An uninitialized file is one that has never had any records added to it.)
- A **get previous** statement attempts to retrieve data from an empty file into an indexed record.
- In relation to COBOL generation, a **get previous** statement attempts to retrieve data from an uninitialized file into an indexed record.

format

format can result from any kind of I/O operation and could be set for these reasons, among others:

- **Record format**

  The file format (fixed or variable length) is different from the EGL record format.

- **Record length**

  In relation to fixed-length records, the length of a record in the file is different from the length of the EGL record. In relation to variable-length records, the length of a record in the file is larger than the length of the EGL record.
- **File type**

  The file type specified for the record does not match the file type at run time.
- **Key length**

  The key length in the file is different from the key length in the EGL indexed record.
- **Key offset**

  The key position in the file is different from the key position in the EGL indexed record.

noRecordFound

noRecordFound is set in these conditions:

- For an indexed record, no record is found that matches the key specified in a **get** statement.
- For EGL-generated Java, your code issues a **get next** or **get previous** statement for an indexed record when the VSAM file is empty or uninitialized.
- For a relative record, no record is found that matches the record ID specified in a **get** statement. Alternatively, a **get next** statement tries to access a record that is beyond the end of the file.
- For a SQL record, no row is found that matches the specified SELECT statement; or a **get next** statement occurs when no selected rows are left to review.

unique

For an indexed or relative record, **unique** is set in these cases:

- An **add** statement tries to insert a record whose key or record ID already exists in the file or in an alternate index, and the insertion fails because of the duplication.
- A **replace** statement fails to overwrite a record because the replacement values include a key that is the same as the alternate-index key of another record.

The **unique** setting is returned only if the access method returns the information, as is true on some operating systems but not on others.

During SQL database access, **unique** is set when a SQL row being added or replaced has a key that already exists in a unique index. The corresponding sqlcode is -803.

Related reference
"add" on page 544
"Association elements" on page 352
"close" on page 551

"delete" on page 554
"Exception handling" on page 89
"execute" on page 557
"get" on page 567
"get next" on page 579
"get previous" on page 584
"Logical expressions" on page 484

"open" on page 598
"prepare" on page 611
"replace" on page 613

isa operator

The operator **isa** is a binary operator that tests whether a given expression is of a particular type. The main purpose is to test the type of the data that is held in a field of type ANY.

The operator is used in an elementary logical expression that has the following format:

```
testExpression isa typeSpecification
```

testExpression
A numeric, text, or datetime expression, which may be composed of a single field or literal.

typeSpecification
A type specification, which may be any of these:

- A part name.
- A primitive-type specification such as STRING; however, if the primitive type can be associated with a length, the length must be specified, as in these examples:
  - BIN(9)
  - CHAR(5)

  Do not include a datetime mask.
- A type specification (as described previously) followed by paired brackets. In this case, the complete specification indicates a dynamic array of a particular type, length (where appropriate), and number of dimensions.

The logical expression resolves to true if *testExpression* matches the type identified in *typeSpecification*; and otherwise resolves to false.

Related reference
"Arrays" on page 69
"Logical expressions" on page 484
"Operators and precedence" on page 653

Java runtime properties (details)

The next table describes the properties that can be included in the deployment descriptor or program properties file, as well as the source of the value generated into the J2EE environment file, if any. The Java type for each property is java.lang.String unless the description column says otherwise.

Runtime property	Description	Source of the generated value
cso.cicsj2c.timeout	Specifies the number of milliseconds before a timeout occurs during a call that uses protocol CICSJ2C. The default value is 30000, which represents 30 seconds. If the value is set to 0, no timeout occurs. The value must be greater than or equal to 0. The Java type in this case is Java.lang.Integer. The property has no effect on calls when the code is running in WebSphere 390; for details, see *Setting up the J2EE server for CICSJ2C calls*.	Build descriptor option **cicsj2cTimeout**
cso.linkageOptions.*LO*	Specifies the name of a linkage properties file that guides how the generated program or wrapper calls other programs. *LO* is the name of the linkage options part used at generation. For details, see *Deploying a linkage properties file*.	*LO* is from the build descriptor option **linkage**; and the default value is the name of the linkage options part followed by the extension *.properties*
tcpiplistener.port	Specifies the number of the port on which an EGL TCP/IP listener (of class CSOTcpipListener or CSOTcpipListenerJ2EE) listens. No default exists. For details, see the topics that concern *Setting up the TCP/IP listener*. The Java type in this case is Java.lang.Integer.	Not generated
tcpiplistener.trace.file	Specifies the name of the file in which to record the activity of one or more EGL TCP/IP listeners (each is of class CSOTcpipListener or CSOTcpipListenerJ2EE). The default file is tcpiplistener.out.	Not generated; tracing is only for use by IBM

Runtime property	Description	Source of the generated value
tcpiplistener.trace.flag	Specifies whether to trace the activity of one or more EGL TCP/IP listeners (each of class CSOTcpipListener or CSOTcpipListenerJ2EE). Select one of these: • 1 for recording the activity into the file identified in property **tcpiplistener.trace.flag** • 0 (the default value) for not recording the activity The Java type in this case is Java.lang.Integer.For details, see the topics that concern *Setting up the TCP/IP listener*.	Not generated; tracing is only for use by IBM
vgj.datemask.gregorian.long.*locale*	Contains the date mask used in either of two cases: • The Java code generated for the system variable *VGVar.currentFormattedGregorianDate* is invoked; or • EGL validates a page item or text-form field that has a length of 10 or more, if the item property **dateFormat** is set to *systemGregorianDateFormat*. *locale* is the code specified in property **vgj.nls.code**. In Web applications, you may change the date-mask property in use by assigning a different value to sysLib.setLocale.	Build descriptor value for the long Gregorian date mask; the default value is specific to the locale
vgj.datemask.gregorian.short.*locale*	Contains the date mask used when EGL validates a page item or text-form field that has a length of less than 10, if the item property **dateFormat** is set to *systemGregorianDateFormat*. *locale* is the code specified in property **vgj.nls.code**. In Web applications, you may change the date-mask property in use by assigning a different value to sysLib.setLocale.	Build descriptor value for the short Gregorian date mask; the default value is specific to the locale

Runtime property	Description	Source of the generated value
vgj.datemask.julian.long.*locale*	Contains the date mask used in either of two cases: • The Java code generated for the system variable VGVar.currentFormattedJulianDate is invoked; or • EGL validates a page item or text-form field that has a length of 10 or more, if the item property **dateFormat** is set to *systemJulianDateFormat*. *locale* is the code specified in property **vgj.nls.code**. In Web applications, you may change the date-mask property in use by assigning a different value to sysLib.setLocale.	Build descriptor value for the long Julian date mask; the default value is specific to the locale
vgj.datemask.julian.short.*locale*	Contains the date mask used when EGL validates a page item or text-form field that has a length of less than 10, if the item property **dateFormat** is set to *systemJulianDateFormat*. *locale* is the code specified in property **vgj.nls.code**. In Web applications, you may change the date-mask property in use by assigning a different value to sysLib.setLocale.	Build descriptor value for the short Julian date mask; the default value is specific to the locale
vgj.default.databaseDelimiter	Specifies the symbol used to separate one value from the next in the system functions **SysLib.loadTable** and **SysLib.unLoadTable**. The default value is a pipe (|).	
vgj.default.dateFormat	Sets the initial value of system variable **StrLib.defaultDateFormat**; for details on valid values, see *Date, time, and timestamp specifiers*	
vgj.defaultI4GLNativeLibrary	Specifies the DLL name accessed by a library of type nativeLibrary. The property is required if you did not specify the library property **dllName**	
vgj.default.moneyFormat	Sets the initial value of system variable **StrLib.defaultMoneyFormat**; for details on valid values, see *formatNumber()*	

Runtime property	Description	Source of the generated value
vgj.default.numericFormat	Sets the initial value of system variable **StrLib.defaultNumericFormat**; for details on valid values, see *formatNumber()*	
vgj.default.timeFormat	Sets the initial value of system variable **StrLib.defaultTimeFormat**; for details on valid values, see *Date, time, and timestamp specifiers*	
vgj.default.timestampFormat	Sets the initial value of system variable **StrLib.defaultTimestampFormat**; for details on valid values, see *Date, time, and timestamp specifiers*	
vgj.jdbc.database.*SN*	Specifies the JDBC database name that is used when a database connection is made by way of the system function sysLib.connect or VGLib.connectionService. The meaning of the value is different for J2EE connections as compared with standard (non-J2EE) connections: • In relation to J2EE connections (as is needed in a production environment), the value is the name to which the datasource is bound in the JNDI registry; for example, jdbc/MyDB • In relation to a standard JDBC connection (as may be used for debugging), the value is the connection URL; for example, jdbc:db2:MyDB You must customize the name of the property itself when you specify a substitution value for *SN*, at deployment time. The substitution value in turn must match either the server name that is included in the invocation of VGLib.connectionService or the database name that is included in the invocation of sysLib.connect.	Build descriptor value for the database name that you want to associate with the specified ″server name″
vgj.jdbc.default.database.autoCommit	Specifies whether a commit occurs after every change to the default database. Valid values are true and false, as described in *sqlCommitControl*.	Build descriptor option **sqlCommitControl**

Runtime property	Description	Source of the generated value
vgj.jdbc.default.database.*programName*	Specifies the default database name that is used for an SQL I/O operation if no prior database connection exists. EGL includes the program name (or program alias, if any) as a substitution value for *programName* so that each program has its own default database. The program name is optional, however, and a property named vgj.jdbc.default.database is used as a default for any program that is not referenced in a program-specific property of this kind. The meaning of the value in the property itself is different for J2EE connections as compared with non-J2EE connections: • In relation to J2EE connections, the value is the name to which the datasource is bound in the JNDI registry; for example, jdbc/MyDB • In relation to a standard JDBC connection, the value is the connection URL; for example, jdbc:db2:MyDB	Depends on the connection type: • For J2EE connections, build descriptor option **sqlJNDIName** • For non-J2EE connections, build descriptor option **sqlDB**
vgj.jdbc.default.password	Specifies the password for accessing the database connection identified in **vgj.jdbc.default.database**. To avoid exposing passwords in the J2EE environment file, do one of the following tasks: • Specify a password in program and function scripts by using the system function sysLib.connect or VGLib.connectionService; or • Include a user ID and password in the datasource specification in the web application server, as described in *Setting up a J2EE JDBC connection*.	Build descriptor option **sqlPassword**
vgj.jdbc.default.userid	Specifies the userid for accessing the database connection identified in **vgj.jdbc.default.database**.	Build descriptor option **sqlID**
vgj.jdbc.drivers	Specifices the driver class for accessing the database connection identified in **vgj.jdbc.default.database**. This property is not present in the deployment descriptor or J2EE environment file and is used only for a standard (non-J2EE) JDBC connection.	Build descriptor option **sqlJDBCDriverClass**

Runtime property	Description	Source of the generated value
vgj.messages.file	Specifies a properties file that includes messages you create or customize. The file is searched in these cases: • When EGL run time responds to the invocation of function SysLib.getMessage, which returns a message that you created; for details, see *SysLib.getMessage* • When EGL runtime is handling a consoleUI application and attempts to present help or comment text from a file identified in the system variable **ConsoleLib.messageResource**, but that variable has no value. • When EGL attempts to display a Java runtime message, as explained in *Message customization for EGL runtime messages*	
vgj.nls.code	Specifies the three-letter NLS code of the program. For a list of valid values, see targetNLS. If the property is not set, these rules apply: • The value defaults to the NLS code that corresponds to the default Java locale • The value is ENU if the default Java locale does not correspond to any of the NLS codes supported by EGL	Build descriptor option **targetNLS**
vgj.nls.currency	Specifies the character used as a currency symbol. The default is determined by the locale associated with **vgj.nls.code**.	Build descriptor option **currencySymbol**
vgj.nls.number.decimal	Specifies the character used as a decimal symbol. The default is determined by the locale associated with **vgj.nls.code**.	Build descriptor option **decimalSymbol**

Runtime property	Description	Source of the generated value
vgj.properties.file	Used only if the first program in a non-J2EE run unit was generated with VisualAge Generator or with an EGL version that preceded 6.0. **vgj.properties.file** specifies an alternate properties file. The file is used throughout a non-J2EE run unit in place of any non-global program properties file. Use of the global file is unaffected. (In run units whose first program was generated with the older EGL or with VisualAge Generator, the global file is called **vgj.properties**.) The file referenced by the property **vgj.properties.file** is used only if you include that property in a command-line directive, as in this example: `java -Dvgj.properties.file= c:\new.properties` The value of **vgj.properties.file** includes the fully qualified path to the properties file. Specifying the property **vgj.properties.file** in a properties file has no effect.	
vgj.ra.*QN*.conversionTable	Specifies the name of the conversion table used by a generated Java program during access of the MQSeries message queue identified by *QN*. Valid values are programControlled, NONE, or a conversion table name. The default is NONE.	Resource associations property **conversionTable**
vgj.ra.*FN*.fileType	Specifies the type of file associated with *FN*, which is a file or queue name identified in the record part. The property value is seqws or mq, as described in *Record and file type cross-reference*. You must specify this deployment descriptor property for each logical file that your program uses.	Resource associations property **fileType**

Runtime property	Description	Source of the generated value
vgj.ra.*FN*.replace	Specifies the effect of an add statement on a record associated with *FN*, which is a file name identified in a record. Select one of two values: • 1 if the statement replaces the file record • 0 (the default) if the statement appends a record to the file The Java type in this case is java.lang.Integer.	Resource associations property **replace**
vgj.ra.*FN*.systemName	Specifies the name of the physical file or message queue associated with *FN*, which is a file or queue name identified in the record part. You must specify this deployment descriptor property for each logical file that your program uses.	Resource associations property **systemName**
vgj.ra.*FN*.text	Specifies whether to cause a generated Java program to do the following when accessing a file by way of a serial record: • Append end-of-line characters during the **add** operation. On non-UNIX platforms, those characters are a carriage return and linefeed; on UNIX platforms, the only character is a linefeed. • Remove end-of-line characters during the **get next** operation. *FN* is the file name associated with the serial record. Select one of these values: • 1 for make the changes • 0 (the default) for refrain from making the changes The Java type in this case is java.lang.Integer.	Resource associations property **text**
vgj.trace.device.option	Destination of trace data, if any. Select one of these values: • 0 for write to System.out • 1 for write to System.err • 2 (the default) for write to the file specified in **vgj.trace.device.spec** with this exception: for VSAM I/O traces, write to vsam.out The Java type in this case is java.lang.Integer.	The generated value, if any, is 2

Runtime property	Description	Source of the generated value
vgj.trace.device.spec	Specifies the name of the output file if **vgj.trace.device.option** is set to 2. An exception is that VSAM I/O traces are written to vsam.out.	The generated value, if any, is vgjtrace.out
vgj.trace.type	Specifies the runtime trace setting. Sum the values of interest to get the tracing you want: • -1 for trace all • 0 for no trace (the default) • 1 for general trace, including function invocations and call statements • 2 for system functions that handle math • 4 for system functions that handle strings • 16 for data passed on a call statement • 32 for the linkage options used on a call • 128 for jdbc I/O • 256 for file I/O • 512 for all the properties except vgj.jdbc.default.password The Java type in this case is java.lang.Integer.	The generated value, if any, is 0

Related concepts
"Java runtime properties" on page 327
"Library part of type basicLibrary" on page 133
"Linkage properties file" on page 343

Related tasks
"Deploying a linkage properties file" on page 342
"Setting up a J2EE JDBC connection" on page 341
"Setting up the J2EE run-time environment for EGL-generated code" on page 333

"Setting up the TCP/IP listener for a called appl in a J2EE appl client module" on page 338
"Setting up the TCP/IP listener for a called non-J2EE application" on page 332
"Understanding how a standard JDBC connection is made" on page 245

Related reference
"callLink element" on page 395
"cicsj2cTimeout" on page 366
"connect()" on page 867
"connectionService()" on page 888
"currentFormattedGregorianDate" on page 916
"currentFormattedJulianDate" on page 917
"currentShortGregorianDate" on page 919
"currentShortJulianDate" on page 920
"Date, time, and timestamp format specifiers" on page 42

"decimalSymbol" on page 368
"defaultDateFormat" on page 848
"defaultMoneyFormat" on page 848
"defaultNumericFormat" on page 849
"defaultTimeFormat" on page 849
"defaultTimestampFormat" on page 849
"formatNumber()" on page 851
"getMessage()" on page 875
"linkage" on page 378
"Linkage properties file (details)" on page 637
"loadTable()" on page 876
"Message customization for EGL Java run time" on page 641
"Record and file type cross-reference" on page 716
"setLocale()" on page 880
"sqlCommitControl" on page 384
"sqlDB" on page 384
"sqlID" on page 385
"sqlJDBCDriverClass" on page 386
"sqlJNDIName" on page 387
"sqlPassword" on page 387
"targetNLS" on page 389
"unloadTable()" on page 884

Java wrapper classes

When you request that a program part be generated as a Java wrapper, EGL produces a wrapper class for each of the following:

- The generated program
- Each fixed record or form that is declared as a parameter in that program
- Each dynamic array that is declared as a parameter; and if the array is array of fixed records, the class for the dynamic array class is in addition to the class for the fixed record itself
- Each structure item that has these characteristics:
  - Is in one of the fixed records or forms for which a wrapper class is generated
  - Has at least one subordinate structure item; in other words, is substructured
  - Is an array; in this case, a substructured array

  An example of a fixed record part with a substructured array is as follows:

```
Record myPart type basicRecord
  10 MyTopStructure CHAR(20)[5];
    20 MyStructureItem01 CHAR(10);
    20 MyStructureItem02 CHAR(10);
end
```

Later descriptions refer to the wrapper classes for a given program as the *program wrapper class*, the *parameter wrapper classes*, the *dynamic array wrapper classes*, and the *substructured-item-array wrapper classes*.

EGL generates a BeanInfo class for each parameter wrapper class, dynamic array wrapper class, or substructured-item-array wrapper class. The BeanInfo class allows the related wrapper class to be used as a Java-compliant Java bean. You probably will not interact with the BeanInfo class.

When you generate a wrapper, the parameter list of the called program cannot include parameters of type BLOB, CLOB, STRING, Dictionary, ArrayDictionary, or non-fixed record.

Overview of how to use the wrapper classes

To use the wrapper classes to communicate with a program generated with VisualAge Generator or EGL, do as follows in the native Java program:

- Instantiate a class (a subclass of CSOPowerServer) to provide middleware services such as converting data between native Java code and a generated program:

```
import com.ibm.javart.v6.cso.*;

public class MyNativeClass
{
  /* declare a variable for middleware */
  CSOPowerServer powerServer = null;

  try
  {
    powerServer = new CSOLocalPowerServerProxy();
  }
  catch (CSOException exception)
  {
    System.out.println("Error initializing middleware"
      + exception.getMessage());
    System.exit(8);
  }
}
```

- Instantiate a program wrapper class to do as follows:
  - Allocate data structures, including dynamic arrays, if any
  - Provide access to methods that in turn access the generated program

  The call to the constructor includes the middleware object:

```
myProgram = new MyprogramWrapper(powerServer);
```

- Declare variables that are based on the parameter wrapper classes:

```
Mypart  myParm = myProgram.getMyParm();
Mypart2 myParm2 = myProgram.getMyParm2();
```

  If your program has parameters that are dynamic arrays, declare additional variables that are each based on a dynamic array wrapper class:

```
myRecArrayVar myParm3 = myProgram.getMyParm3();
```

  For details on interacting with dynamic arrays, see "Dynamic array wrapper classes" on page 540.
- In most cases (as in the previous step), use the parameter variables to reference and change memory that was allocated in the program wrapper object
- Set a userid and password, but only in these cases:
  - The Java wrapper accesses an iSeries-based program by way of the iSeries Toolbox for Java; or
  - The generated program runs on a CICS for z/OS region that authenticates remote access.

  The userid and password are not used for database access.

  You set and review the userid and password by using the callOptions variable of the program object, as in this example:

```
myProgram.callOptions.setUserID("myID");
myProgram.callOptions.setPassword("myWord");
myUserID = myProgram.callOptions.getUserID();
myPassword = myProgram.callOptions.getPassword();
```

- Access the generated program, in most cases by invoking the execute method of the program wrapper object:

  ```
  myProgram.execute();
  ```

- Use the middleware object to establish database transaction control, but only in the following situation:
  - The program wrapper object either is accessing a generated program on CICS for z/OS or is accessing an iSeries-based COBOL program by way of the IBM Toolbox for Java. In the latter case, the value of **remoteComType** for the call is JAVA400.
  - In the linkage options part used to generate the wrapper classes, you specified that the database unit of work is under client (in this case, wrapper) control; for details, see the reference to **luwControl** in *callLink element*.

  If the database unit of work is under client control, processing includes use of commit and rollback methods of the middleware object:

  ```
  powerServer.commit();
  powerServer.rollback();
  ```

- Close the middleware object and allow for garbage collection:

  ```
  if (powerServer != null)
  {
    try
    {
      powerServer.close();
      powerServer = null;
    }

    catch(CSOException error)
    {
      System.out.println("Error closing middleware"
        + error.getMessage());
      System.exit(8);
    }
  }
  ```

The program wrapper class

The program wrapper class includes a private instance variable for each parameter in the generated program. If the parameter is a record or form, the variable refers to an instance of the related parameter wrapper class. If the parameter is a data item, the variable has a primitive Java type.

A table at the end of this help page describes the conversions between EGL and Java types.

A program wrapper object includes the following public methods:

- **get** and **set** methods for each parameter, where the format of the name is as follows:

  purposeParmname()

  purpose
  The word **get** or **set**

  Parmname
  Name of the data item, record, or form; the first letter is upper case, and aspects of the other letters are determined by the naming convention described in "Naming conventions for Java wrapper classes" on page 542

- An **execute** method for calling the program; you use this method if the data that will be passed as arguments on the call is in the memory allocated for the program wrapper object

Instead of assigning values to the instance variables, you can do as follows:

- Allocate memory for parameter wrapper objects, dynamic array wrapper objects, and primitive types
- Assign values to the memory you allocated
- Pass those values to the program by invoking the **call** method of the program wrapper object rather than the **execute** method

The program wrapper object also includes the callOptions variable, which has the following purposes:

- If you generated the Java wrapper so that linkage options for the call are set at generation time, the callOptions variable contains the linkage information. For details on when the linkage options are set, see remoteBind in *callLink element*.
- If you generated the Java wrapper so that linkage options for the call are set at run time, the callOptions variable contains the name of the linkage properties file. The file name is *LO*.**properties**, where *LO* is the name of the linkage options part used for generation.
- In either case, the callOptions variable provides the following methods for setting or getting a userid and password:

  ```
  setPassword(passWord)
  setUserid(userid)
  getPassword()
  getUserid()
  ```

  The userid and password are used when you set the **remoteComType** property of the callLink element to one of the following values:

  – CICSECI
  – CICSJ2C
  – JAVA400

Finally, consider the following situation: your native Java code requires notification when a change is made to a parameter of primitive type. To make such a notification possible, the native code registers as a listener by invoking the **addPropertyChangeListener** method of the program wrapper object. In this case, either of the following situations triggers the PropertyChange event that causes the native code to receive notification at run time:

- Your native code invokes a **set** method on a parameter of primitive type
- The generated program returns changed data to a parameter of primitive type

The PropertyChange event is described in the JavaBean specification of Sun Microsystems, Inc.

The set of parameter wrapper classes

A parameter wrapper class is produced for each record that is declared as a parameter in the generated program. In the usual case, you use a parameter wrapper class only to declare a variable that references the parameter, as in the following example:

```
Mypart myRecWrapperObject = myProgram.getMyrecord();
```

In this case, you are using the memory allocated by the program wrapper object.

You also can use the parameter wrapper class to declare memory, as is necessary if you invoke the call method (rather than the execute method) of the program object.

The parameter wrapper class includes a set of private instance variables, as follows:

- One variable of a Java primitive type for each of the parameter's low-level structure items, but only for a structure item that is neither an array nor within a substructured array
- One array of a Java primitive type for each EGL structure item that is an array and is not substructured
- An object of an inner, array class for each substructured array that is not itself within a substructured array; the inner class can have nested inner classes to represent subordinate substructured arrays

The parameter wrapper class includes several public methods:

- A set of **get** and **set** methods allows you to get and set each instance variable. The format of each method name is as follows:

  purposesiName()

  purpose
  The word **get** or **set**.

  siName
  Name of the structure item. The first letter is upper case, and aspects of the other letters are determined by the naming convention described in "Naming conventions for Java wrapper classes" on page 542.

  Note: Structure items that you declared as fillers are included in the program call; but the array wrapper classes do not include public get or set methods for those structure items.
- The method **equals** allow you to determine whether the values stored in another object of the same class are identical to the values stored in the parameter wrapper object. The method returns **true** only if the classes and values are identical.
- The method **addPropertyChangeListener** is invoked if your program requires notification of a change in a variable of a Java primitive type.
- A second set of **get** and **set** methods allow you to get and set the null indicators for each structure item in an SQL record parameter. The format of each of these method names is as follows:

  *purposesiName***NullIndicator**()

  purpose
  The word **get** or **set**.

  siName
  Name of the structure item. The first letter is upper case, and aspects of the other letters are determined by the naming convention described in "Naming conventions for Java wrapper classes" on page 542.

The set of substructured-item-array wrapper classes

A substructured-item-array wrapper class is an inner class of a parameter class and represents a substructured array in the related parameter. The substructured-item-array wrapper class includes a set of private instance variables that refer to the structure items at and below the array itself:

- One variable of a Java primitive type for each of the array's low-level structure items, but only for a structure item that is neither an array nor within a substructured array

- One array of a Java primitive type for each EGL structure item that is an array and is not substructured
- An object of an inner, substructured-item-array wrapper class for each substructured array that is not itself within a substructured array; the inner class can have nested inner classes to represent subordinate substructured arrays

The substructured-item-array wrapper class includes the following methods:

- A set of **get** and **set** methods for each instance variable

  Note: Structure items that you declared as nameless fillers are used in the program call; but the substructured-item-array wrapper classes do not include public get or set methods for those structure items.
- The method **equals** allows you to determine whether the values stored in another object of the same class are identical to the values stored in the substructured-item-array wrapper object. The method returns **true** only if the classes and values are identical.
- The method **addPropertyChangeListener**, for use if your program requires notification of a change in a variable of a Java primitive type

In most cases, the name of the top-most substructured-item-array wrapper class in a parameter wrapper class is of the following form:

```
ParameterClassname.ArrayClassName
```

Consider the following record, for example:

```
Record CompanyPart type basicRecord
10 Departments CHAR(20)[5];
   20 CountryCode CHAR(10);
   20 FunctionCode CHAR(10)[3];
      30 FunctionCategory CHAR(4);
      30 FunctionDetail CHAR(6);
end
```

If the parameter `Company` is based on `CompanyPart`, you use the string `CompanyPart.Departments` as the name of the inner class.

An inner class of an inner class extends use of a dotted syntax. In this example, you use the symbol `CompanyPart.Departments.Functioncode` as the name of the inner class of `Departments`.

For additional details on how the substructured-item-array wrapper classes are named, see *Output of Java wrapper generation*.

Dynamic array wrapper classes

A dynamic array wrapper class is produced for each dynamic array that is declared as a parameter in the generated program. Consider the following EGL program signature:

```
Program myProgram(intParms int[], recParms MyRec[])
```

The name of the dynamic array wrapper classes are IntParmsArray and MyRecArray.

You use a dynamic array wrapper class to declare a variable that references the dynamic array, as in the following examples:

```
IntParmsArray myIntArrayVar = myProgram.getIntParms();
MyRecArray    myRecArrayVar = myProgram.getRecParms();
```

After declaring the variables for each dynamic array, you might add elements:

```
// adding to an array of Java primitives
// is a one-step process
myIntArrayVar.add(new Integer(5));

// adding to an array of records or forms
// requires multiple steps; in this case,
// begin by allocating a new record object
MyRec myLocalRec = (MyRec)myRecArrayVar.makeNewElement();

// the steps to assign values are not shown
// in this example; but after you assign values,
// add the record to the array
myRecArrayVar.add(myLocalRec);

// next, run the program
myProgram.execute();

// when the program returns, you can determine
// the number of elements in the array
int myIntArrayVarSize = myIntArrayVar.size();

// get the first element of the integer array
// and cast it to an Integer object
Integer firstIntElement = (Integer)myIntArrayVar.get(0);

// get the second element of the record array
// and cast it to a MyRec object
MyRec secondRecElement = (MyRec)myRecArrayVar.get(1);
```

As suggested by that example, EGL provides several methods for manipulating the variables that you declared.

Method of the dynamic array class	Purpose
add(*int*, *Object*)	To insert an object at the position specified by *int* and to shift the current and subsequent elements to the right.
add(*Object*)	To append an object to the end of the dynamic array.
addAll(*ArrayList*)	To append an ArrayList to the end of the dynamic array.
get()	To retrieve the ArrayList object that contains all elements in the array
get(*int*)	To retrieve the element that is in the position specified by *int*
makeNewElement()	To allocate a new element of the array-specific type and to retrieve that element, without adding that element to the dynamic array.
maxSize()	To retrieve an integer that indicates the maximum (but not actual) number of elements in the dynamic array
remove(*int*)	To remove the element that is in the position specified by *int*
set(*ArrayList*)	To use the specified ArrayList as a replacement for the dynamic array
set(*int*, *Object*)	To use the specified object as a replacement for the element that is in the position specified by *int*
size()	To retrieve the number of elements that are in the dynamic array

Exceptions occur in the following cases, among others:

- If you specify an invalid index in the **get** or **set** method
- If you try to add (or set) an element that is of a class incompatible with the class of each element in the array
- If you try to add elements to a dynamic array when the maximum size of the array cannot support the increase; and if the method **addAll** fails for this reason, the method adds no elements

Naming conventions for Java wrapper classes

EGL creates a name in accordance with these rules:

- If the name is all upper case, lower case all letters.
- If the name is a keyword, precede it with an underline
- If a hyphen or underline is in the name, remove that character and upper case the next letter
- If a dollar sign ($), at sign (@), or pound sign (#) is in the name, replace each of those characters with a double underscore (__) and precede the name with an underscore (_).
- If the name is used as a class name, upper case the first letter.

The following rules apply to dynamic array wrapper classes:

- In most cases, the name of a class is based on the name of the part declaration (data item, form, or record) that is the basis of each element in the array. For example, if a record part is called `MyRec` and the array declaration is `recParms myRec[]`, the related dynamic array wrapper class is called `MyRecArray`.
- If the array is based on an item declaration that has no related part declaration, the name of the dynamic array class is based on the array name. For example, if the array declaration is `intParms int[]`, the related dynamic array wrapper class is called `IntParmsArray`.

Data type cross-reference

The next table indicates the relationship of EGL primitive types in the generated program and the Java data types in the generated wrapper.

EGL primitive type	Length in chars or digits	Length in bytes	Decimals	Java data type	Maximum precision in Java
CHAR	1-32767	2-32766	NA	String	NA
DBCHAR	1-16383	1-32767	NA	String	NA
MBCHAR	1-32767	1-32767	NA	String	NA
UNICODE	1-16383	2-32766	NA	String	NA
HEX	2-75534	1-32767	NA	byte[]	NA
BIN, SMALLINT	4	2	0	short	4
BIN, INT	9	4	0	int	9
BIN, BIGINT	18	8	0	long	18
BIN	4	2	>0	float	4
BIN	9	4	>0	double	15
BIN	18	8	>0	double	15
DECIMAL, PACF	1-3	1-2	0	short	4

EGL primitive type	Length in chars or digits	Length in bytes	Decimals	Java data type	Maximum precision in Java
DECIMAL, PACF	4-9	3-5	0	int	9
DECIMAL, PACF	10-18	6-10	0	long	18
DECIMAL, PACF	1-5	1-3	>0	float	6
DECIMAL, PACF	7-18	4-10	>0	double	15
NUM, NUMC	1-4	1-4	0	short	4
NUM, NUMC	5-9	5-9	0	int	9
NUM, NUMC	10-18	10-18	0	long	18
NUM, NUMC	1-6	1-6	>0	float	6
NUM, NUMC	7-18	7-18	>0	double	15

Related concepts
"Java wrapper" on page 282
"Run-time configurations" on page 9

Related tasks
"Generating Java wrappers" on page 282

Related reference
"callLink element" on page 395
"How Java wrapper names are aliased" on page 650
"Linkage properties file (details)" on page 637
"Output of Java wrapper generation" on page 656
"remoteBind in callLink element" on page 407

JDBC driver requirements in EGL

The JDBC driver requirements vary by database management system, whether for EGL debug time or run time:

DB2 UDB

The DB2 Universal driver is compatible with EGL, but the related App driver is not compatible; specifically, the App driver cannot process an EGL **open** or **get** statement that includes the option forUpdate.

IBM recommends that you not use the Net driver at all.

If you are running J2EE applications in WebSphere Application Server v6.x, you need DB2 version 8.1.6 or higher. If you are running those applications in WebSphere v5.x Test Environment, you need DB2 version 8.1.3 or higher.

Informix

The minimum acceptable Informix JDBC driver is 2.21.JC6. This driver level does not comply with JDBC 3.0 and therefore does not support the hold option in the EGL **open** statement. An Informix JDBC 3.0-compliant driver may now be available and should support the hold option.

Oracle

The JDBC driver that is packaged with Oracle 10i is acceptable.

The following rules apply to any JDBC driver used with EGL:

- The driver must support JDBC 2.0 or higher

- The value java.sql.ResultSet.CONCUR_UPDATABLE must be allowed in these contexts:
  - As the second argument to java.sql.Connection.createStatement(int,int)
  - As the third argument to java.sql.Connection.prepareStatement(String,int,int) and java.sql.Connection.prepareCall(String,int,int)
- If you wish to support the hold option in the EGL **open** statement, the driver must support JDBC 3.0, and the value java.sql.ResultSet.HOLD_CURSORS_OVER_COMMIT must be allowed in these contexts:
  - As the third argument to java.sql.Connection.createStatement(int,int,int)
  - As the fourth argument to java.sql.Connection.prepareStatement(String,int,int,int) and java.sql.Connection.prepareCall(String,int,int,int)

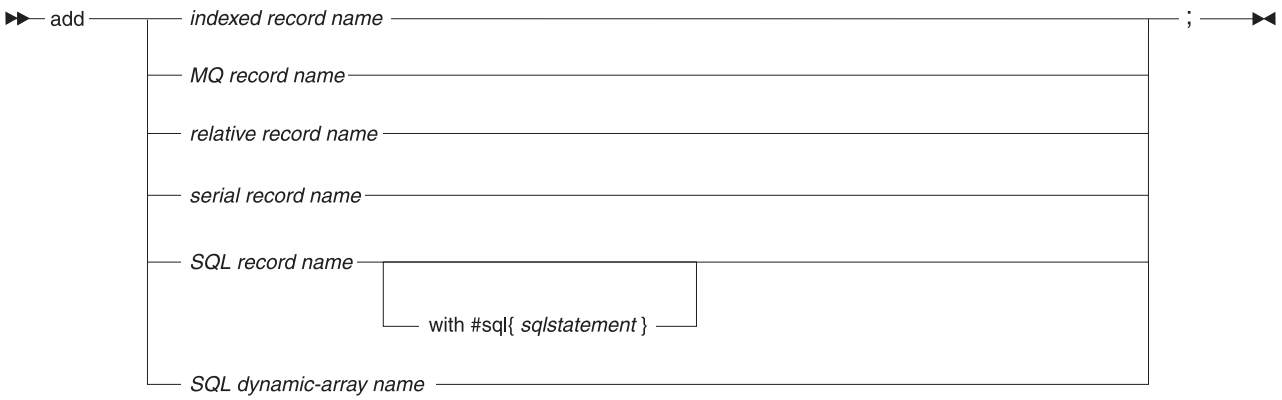
For any database management system, JDBC drivers from third- or fourth-party vendors are acceptable.

Related tasks
"Setting up a J2EE JDBC connection" on page 341
"Understanding how a standard JDBC connection is made" on page 245

Keywords

add

The EGL **add** statement places a record in a file, message queue, or database; or places a set of records in a database.



record name
Name of the I/O object to add: an indexed, MQ, relative, serial, or SQL record

with #sql{ *sqlStatement* **}**
An explicit SQL INSERT statement. Leave no space after #sql.

SQL dynamic-array name
The name of a dynamic array of SQL records. The elements are inserted into the database, each at the position specified by the element-specific key values. The operation stops at the first error or when all elements are inserted.

An example is as follows:

```
if (userRequest == "A")
  try
    add record1;
  onException
    myErrorHandler(12);
  end
end
```

The behavior of the **add** statement depends on the record type. For details on SQL processing, see *SQL record*.

Indexed record

When you add an indexed record, the key in the record determines the logical position of the record in the file. Adding a record to a file position that is already in use results in the hard I/O error UNIQUE or (if duplicates are allowed) in the soft I/O error DUPLICATE.

MQ record

When you add a MQ record, the record is placed at the end of the queue. This placement occurs because the add invokes one or more MQSeries calls:

- MQCONN connects the generated code to the default queue manager and is invoked when no connection is active
- MQOPEN establishes a connection to the queue and is invoked when a connection is active but the queue is not open
- MQPUT puts the record in the queue and is always invoked unless an error occurred in an earlier MQSeries call

Relative record

When you add a relative record, the key item specifies the position of the record in the file. Adding a record to a file position that is already in use, however, results in the hard I/O error UNIQUE.

The record key item must be available to any function that uses the record and can be any of these:

- An item in the same record
- An item in a record that is global to the program or is local to the function that is running the **add** statement
- A data item that is global to the program or is local to the function that is running the **add** statement

Serial record

When you add a serial record, the record is placed at the end of the file.

If the generated program adds a serial record and then issues a **get next** statement for the same file, the program closes and reopens the file before executing the **get next** statement. A **get next** statement that follows an **add** statement therefore reads the first record in the file. This behavior also occurs when the **get next** and **add** statements are in different programs, and one program calls another.

It is recommended that you avoid having the same file open in more than one program at the same time.

SQL record

Some error conditions are as follows:

- You specify an SQL statement of a type other than INSERT

- You specify some but not all clauses of an SQL INSERT statement
- You specify an SQL INSERT statement (or accept an implicit SQL statement) that has any of these characteristics--
  - Is related to more than one SQL table
  - Includes only host variables that you declared as read only
  - Is associated with a column that either does not exist or is incompatible with the related host variable

The result is as follows when you add an SQL record without specifying an explicit SQL statement:

- The format of the generated SQL INSERT statement is like this--

```
INSERT INTO tableName
  (column01, ... columnNN)
  values (:recordItem01, ... :recordItemNN)
```

- The key value in the record determines the logical position of the data in the table. A record that does not have a key is handled in accordance with the SQL table definition and the rules of the database.
- As a result of the association of record items and SQL table columns in the record part, the generated code places the data from each record item into the related SQL table column.
- If you declared a record item to be read only, the generated SQL INSERT statement does not include that record item, and the database management system sets the value of the related SQL table column to the default value that was specified when the column was defined.

An example that uses a dynamic array of SQL records is as follows:

```
try
  add employees;
  onException
    sysLib.rollback();
end
```

Related concepts

"References to parts" on page 20
"Record types and properties" on page 126
"SQL support" on page 213

Related tasks

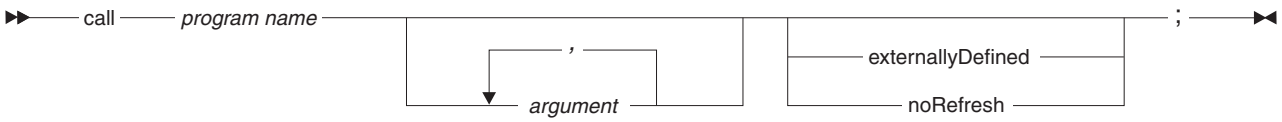
"Syntax diagram for EGL statements and commands" on page 733

Related reference

"close" on page 551
"delete" on page 554
"get" on page 567
"get next" on page 579
"get previous" on page 584
"Exception handling" on page 89
"execute" on page 557
"I/O error values" on page 522
"open" on page 598
"prepare" on page 611
"EGL statements" on page 83
"replace" on page 613
"SQL item properties" on page 63

call

The EGL call statement transfers control to another program and optionally passes a series of values. Control returns to the caller when the called program ends; and if the called program changes any data that was passed by way of a variable, the storage area available to the caller is changed, too.



program name

Name of the called program. The program is either generated by EGL or is considered *externally defined*.

The specified name cannot be a reserved word. If the caller must call a non-EGL program that has the same name as an EGL reserved word, use a different program name in the call statement, then use a linkage options part, **callLink** element to specify an alias, which is the name used at run time.

If the called program is a Java program, the called program name is case-sensitive; *calledProgram* is different from *CALLEDPROGRAM*. Otherwise, the determination of whether the program name is case-sensitive depends on the system on which the called program resides: case-sensitive for UNIX, case-insensitive otherwise.

In the EGL debugger, the called program name is case-insensitive.

argument

One of a series of value references, each separated from the next by a comma. An argument may be a primitive variable; a form; a record; a fixed record; a non-numeric literal; a non-numeric constant; or (if EGL has access to the called program at generation time) a more complex datetime, numeric, or text expression. You may not pass a field of type ANY, ArrayDictionary, Blob, Clob, DataTable, or Dictionary. Also, you may not pass arrays of those types or records that include any of those types.

externallyDefined

An indicator that the program is externally defined. This indicator is available only if you set the project property for VisualAge Generator compatibility.

It is recommended that a non-EGL-generated program be identified as externally defined not in the **call** statement, but in the linkage options part that is used at generation time. (The related property is in the linkage options part, callLink element, and is also called **externallyDefined**.)

noRefresh

An indicator that a screen refresh is to be avoided when the called program returns control.

The indicator is supported (at development time) if the program property **VAGCompatibility** is selected or (at generation time) if the build descriptor option **VAGCompatibility** is set to *yes*.

This indicator is appropriate if the caller is in a run unit that presents text forms to a screen and either of these situations is in effect:

- The called program does not present a text form; or
- The caller writes a full-screen text form after the call.

It is recommended that you indicate your preference for screen refresh not in the **call** statement, but in the linkage options part that is used at generation time. (The related property is in the linkage options part, callLink element, and is called **refreshScreen**.)

An example is as follows:

```
if (userRequest == "C")
  try
    call programA;
  onException
    myErrorHandler(12);
  end
end
```

The number, type, and sequence of arguments in a call statement must correspond to the number, type, and sequence of values expected by the called program.

It is strongly recommended that the number of bytes passed in each argument be the same as the number of bytes expected.In the case of an EGL-generated Java program, a length mismatch causes an error only if the run-time correction of that mismatch causes a type mismatch:

- If the called Java program receives too few bytes, the end of the passed data is padded with blanks.
- If the called Java program receives too many bytes, the end of the passed data is truncated.

In the case of Java, an error occurs if blanks are added to a data item of type NUM, for example, but not if blanks are added to a data item of type CHAR.

The following rules apply to literals and constants:

- The size of a passed literal or constant must equal the size of the receiving parameter
- A numeric literal or constant cannot be passed as an argument
- A literal or constant that includes only single-byte characters may be passed to a parameter of type CHAR or MBCHAR
- A literal or constant that includes only double-byte characters may be passed only to a parameter of type DBCHAR
- A literal or constant that includes a combination of single- and double-byte characters may be passed to a parameter of type MBCHAR

The behavior of call depends partly on the target system, as shown in the next table.

Target system	Platform-specific details
AIX	Recursive calls are supported.
iSeries COBOL	Recursive calls are not supported.
iSeries Java	Recursive calls are supported.
Linux	Recursive calls are supported.
z/OS batch	Recursive calls are not supported.

Target system	Platform-specific details
Windows 2000, Windows NT	Recursive calls are supported.
z/OS UNIX System Services	Recursive calls are supported.

The call is affected by the linkage options part, if any, that is used at generation time. (You include a linkage options part by setting the build descriptor option **linkage**.)

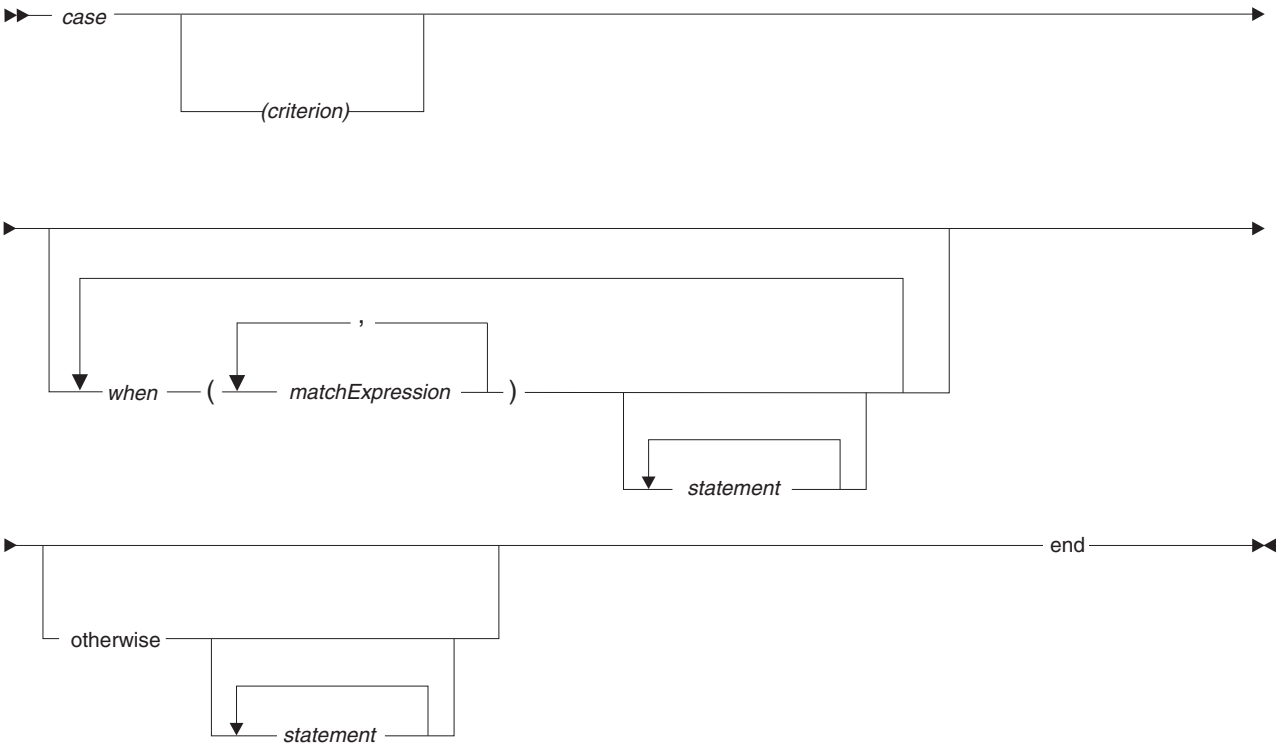
For details on linkage, see *Linkage options part*.

Related concepts
"Linkage options part" on page 291
"Syntax diagram for EGL statements and commands" on page 733

Related reference
"EGL statements" on page 83
"Exception handling" on page 89
"linkage" on page 378
"Primitive types" on page 31

case

The EGL **case** statement marks the start of multiple sets of statements, where at most only one of those sets is run. The **case** statement is equivalent to a C or Java **switch** statement that has a break at the end of each case clause.



criterion
An item, constant, expression, literal, or system variable, including ConverseVar.eventKey or sysVar.systemType.

If you specify *criterion*, each of the subsequent when clauses (if any) must contain one or more instances of *matchExpression*. If you do not specify *criterion*, each of the subsequent when clauses (if any) must contain a *logical expression*.

when
The beginning of a clause that is invoked only in these cases:

- You specified a *criterion*, and the when clause is the first to contain a *matchExpression* that is equal to the *criterion*; or
- You did not specify a *criterion*, and the when clause is the first to contain a *logical expression* that evaluates to true.

If you wish the when clause to have no effect when invoked, code the clause without EGL statements.

A **case** statement may have any number of when clauses.

matchExpression
One of the following values:

- A numeric or string expression
- A symbol for comparison to ConverseVar.eventKey or sysVar.systemType

The primitive type of *matchExpression* value must be compatible with the primitive type of the criterion value. For details on compatibility, see *Logical expressions*.

logicalExpression
A *logical expression*.

statement
An EGL statement.

otherwise
The beginning of a clause that is invoked if no when clause runs.

After the statements run in a when or otherwise clause, control passes to the EGL statement that immediately follows the end of the **case** statement. Control does not fall through to the next when clause under any circumstance. If no when clause is invoked and no default clause is in use, control also passes to the next statement immediately folowing the end of the **case** statement, and no error situation is in effect.

An example of a **case** statement is as follows:

```
case (myRecord.requestID)
  when (1)
    myFirstFunction();
  when (2, 3, 4)
    try
      call myProgram;
    onException
      myCallFunction(12);
    end
  otherwise
    myDefaultFunction();
end
```

If a single clause includes multiple instances of *matchExpression* (2, 3, 4 in the previous example), evaluation of those instances is from left to right, and the evaluation stops as soon as one *matchExpression* is found that corresponds to the criterion value.

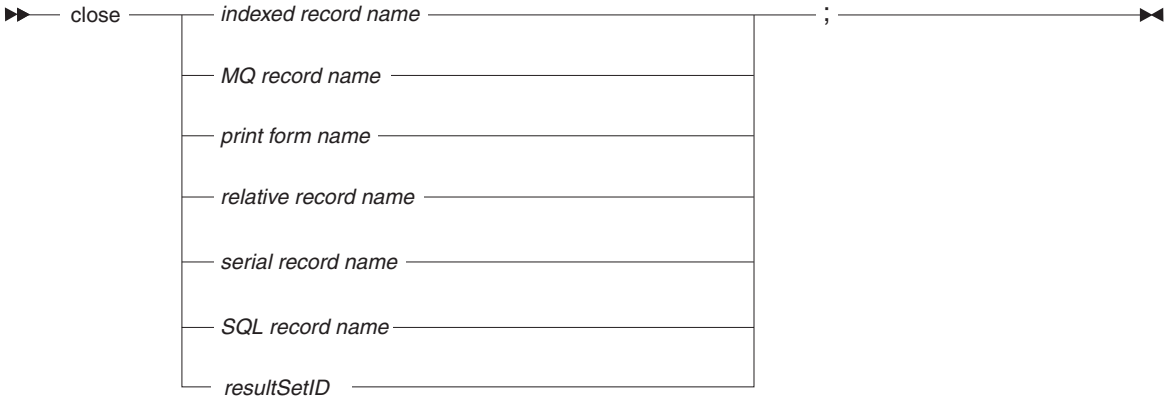
Related tasks
"Syntax diagram for EGL statements and commands" on page 733

Related reference

"EGL statements" on page 83
"Logical expressions" on page 484
"eventKey" on page 895
"systemType" on page 911

close

The EGL **close** statement disconnects a printer; or closes the file or message queue associated with a given record; or, in the case of an SQL record, closes the cursor that was open by an EGL **open** or **get** statement.



name
Name of the I/O object that is associated with the resource being closed; that object is a print form or an indexed, MQ, relative, serial, or SQL record

resultSetIdentifier
For SQL processing only, an ID that ties the **close** statement to a **get** or **open** statement run earlier in the same program. For details, see *resultSetID*.

Example:

```
if (userRequest == "C")
   try
     close fileA;
   onException
      myErrorHandler(12);
   end
end
```

The behavior of a **close** statement depends on the type of I/O object.

Indexed, serial, or relative record

When you use the name of an indexed, serial, or relative record in a **close** statement, EGL closes the file associated with that record.

If a file is open and you use the fileAssociation item to change the resource name associated with that file, EGL closes the file automatically before executing the next statement that affects the file. For details, see *resourceAssociation*.

EGL also closes any file that is open when the program ends.

MQ record

When you use the name of a MQ record in a **close** statement, EGL ensures that the MQSeries command MQCLOSE is executed for the message queue associated with that record.

Print form

If the I/O object is a print form, the close statement issues a form feed and either disconnects from the printer or (if the print form is spooled to a file) closes the file.

Before you use ConverseVar.printerAssociation to change the print destination, close the printer or file specified by the current value of ConverseVar.printerAssociation. Issue a close statement option for each print destination, as multiple printer or print files can be open at the same time.

EGL run time ensures that all printers are closed when the program ends.

SQL record

When you use the name of an SQL record in a **close** statement, EGL closes the SQL cursor that is open for that record.

EGL automatically closes a cursor in these cases:

- A cursor-processing loop follows an **open** statement and continues until a No Record Found (NRF) condition indicates that all rows in the set were processed
- EGL runs a **get** statement for an SQL record when a single row is read and neither forUpdate nor singleRow was specified as an option
- EGL runs a **replace** or **delete** statement that uses the cursor opened by a **get** statement; in this case, forUpdate was specified as an option in the **get** statement
- EGL begins to process an **open** or **get** statement for a record that is associated with an open cursor; the close precedes the other processing
- The program runs either **sysLib.commit** or **sysLib.rollback**; but the close does not occur if the option withHold is in effect, as explained in relation to open

EGL closes all open cursors in this case:

- The program is of type textUI, does an automatic commit before conversing a form, and is unaffected by the option withHold when the converse occurs; for details on textUI programs and the **converse** statement, see *Segmentation*

Related concepts
"Record types and properties" on page 126
"resultSetID" on page 722
"Segmentation in text applications" on page 149
"SQL support" on page 213

Related tasks
"Syntax diagram for EGL statements and commands" on page 733

Related reference
"add" on page 544
"delete" on page 554

"EGL statements" on page 83
"Exception handling" on page 89
"execute" on page 557
"get" on page 567
"get next" on page 579
"get previous" on page 584
"I/O error values" on page 522
"open" on page 598
"prepare" on page 611
"replace" on page 613
"recordName.resourceAssociation" on page 832
"SQL item properties" on page 63
"commit()" on page 866
"rollback()" on page 878
"printerAssociation" on page 896
"terminalID" on page 913

continue

The EGL **continue** statement transfers control to the end of a **for**, **forEach**, or **while** statement that itself contains the **continue** statement. Execution of the containing statement continues or ends depending on the logical test that is conducted as usual at the start of the containing statement.

The **continue** statement must be in the same function as the containing statement.

for, forEach, or while
Identifies the innermost containing statement of the specified type. If you specify one of those statement types, the **continue** statement must be contained in a statement of that type. If you do not specify a statement type, the result is as follows:

- The **continue** statement transfers control to the end of the innermost containing **for**, **forEach**, or **while** statement; and
- The **continue** statement must be contained in a statement of one of those types.

Related tasks
"Syntax diagram for EGL statements and commands" on page 733

Related reference
"EGL statements" on page 83

converse

The EGL **converse** statement presents a text form in a text application.

The program waits for a user response, receives the text form from the user, and continues processing with the statement that follows the **converse** statement.

For an overview of text-form processing, see these pages in order:

1. Text forms
2. Segmentation

textFormName
Name of a text form that is visible to the program. For details on visibility, see *References to parts*.

An example is as follows:

```
converse myTextForm;
```

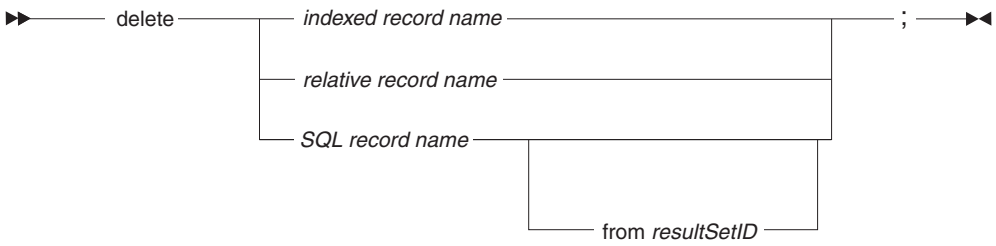
These considerations apply:

- In relation to text forms, a **converse** statement is always valid in a called program; but if you are running a main program that is segmented, the **converse** statement is not valid in these kinds of code--
  - A function that has parameters, local storage, or return values
  - A function that is invoked (directly or indirectly) by a function that has parameters, local storage, or return values.

Related concepts
"References to parts" on page 20
"Segmentation in text applications" on page 149

delete

The EGL **delete** statement removes either a record from a file or a row from a database.



record name
Name of the I/O object: an indexed, relative, or SQL record associated with the file record or SQL row being deleted

from *resultSetID*
ID that ties the **delete** statement to a **get** or **open** statement run earlier in the same program. For details, see *resultSetID*.

An example is as follows:

```
if (userRequest == "D")
  try
    get myRecord forUpdate;
    onException
      myErrorHandler(12);   // exits the program
  end

  try
    delete myRecord;
    onException
      myErrorHandler(16);
  end
end
```

The behavior of the **delete** statement depends on the record type. For details on SQL processing, see *SQL record*.

Indexed or relative record

If you want to delete an indexed or relative record, do as follows:

- Issue a **get** statement for the record and specify the forUpdate option
- Issue the **delete** statement, with no intervening I/O operation against the same file

After you issue the **get** statement, the effect of the next I/O operation on the same file is as follows:

- If the next I/O operation is a **replace** statement on the same EGL record, the record is changed in the file
- If the next I/O operation is a **delete** statement on the same EGL record, the record in the file is marked for deletion
- If the next I/O operation is a **get** on the same file (with the forUpdate option), a subsequent replace or delete is valid on the newly read file record
- If the next I/O operation is a **get** on the same EGL record (with no forUpdate option) or is a close on the same file, the file record is released without change

For details on the forUpdate option, see *get*.

SQL record

In the case of SQL processing, you must use the forUpdate option on an EGL **get** or **open** statement to retrieve a row for subsequent deletion:

- You can issue a **get** statement to retrieve the row; or
- You can issue an **open** statement to select a set of rows and then invoke a **get next** statement to retrieve the row of interest.

In either case, the EGL **delete** statement is represented in the generated code by an SQL DELETE statement that references the current row in a cursor. You cannot modify that SQL statement, which is formatted as follows:

```
DELETE FROM tableName
  WHERE CURRENT OF cursor
```

If you wish to write your own SQL DELETE statement, use the EGL **execute** statement.

You cannot use a single EGL **delete** statement to remove rows from multiple SQL tables.

Related concepts
"Record types and properties" on page 126
"resultSetID" on page 722
"Run unit" on page 721
"SQL support" on page 213

Related tasks
"Syntax diagram for EGL statements and commands" on page 733

Related reference
"add" on page 544

"close" on page 551
"EGL statements" on page 83
"Exception handling" on page 89
"execute" on page 557
"get" on page 567
"get next" on page 579
"get previous" on page 584
"I/O error values" on page 522
"prepare" on page 611
"open" on page 598
"replace" on page 613
"SQL item properties" on page 63

display

The EGL **display** statement adds a text form to a run-time buffer but does not present data to the screen. For details on the run-time behavior, see *Text forms*.

Note: If you are working in VisualAge Generator compatibility mode, you can issue a statement of the following form:

```
display printForm;
```

printForm
Name of a print form that is visible to the program.

In that case, **display** is equivalent to print.

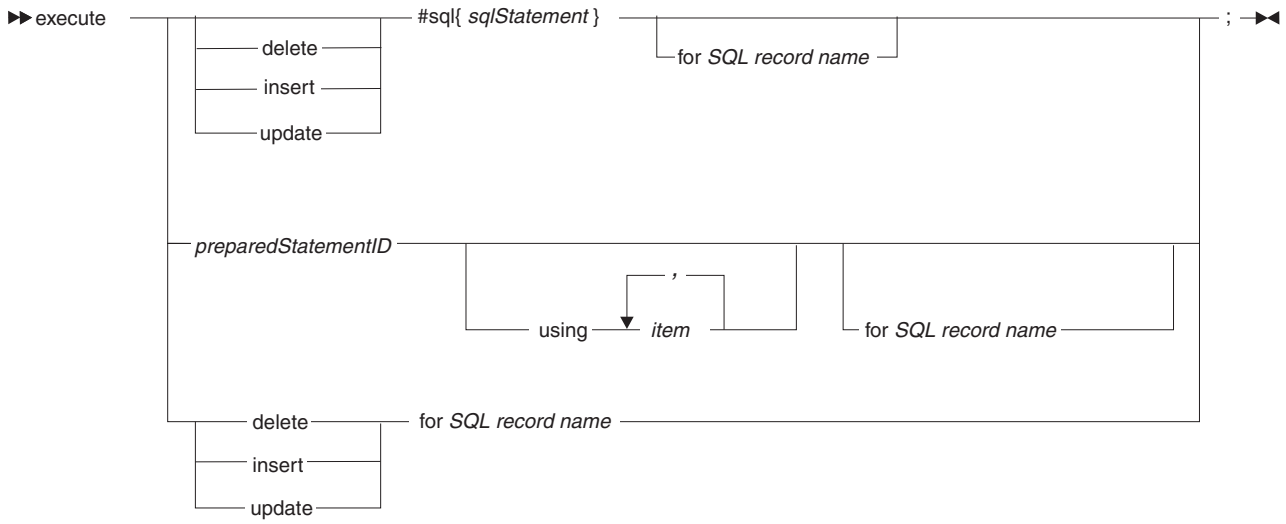
textFormName
Name of a text form that is visible to the program. For details on visibility, see *References to parts*.

Related concepts
"References to parts" on page 20
"Text forms" on page 148

Related reference
"print" on page 613

execute

The EGL **execute** statement lets you write one or more SQL statements; in particular, SQL data-definition statements (of type CREATE TABLE, for example) and data-manipulation statements (of type INSERT or UPDATE, for example)



#sql{ *sqlStatement* **}**
An explicit SQL statement. If you want the SQL statement to update or delete a row in a result set, code an SQL UPDATE or DELETE statement that includes the following clause:

```
WHERE CURRENT OF resultSetID
```

resultSetID
The resultSetID specified in the EGL open statement that made the result set available.

Leave no space between #sql and the left brace.

for *SQL record name*
Name of an SQL record.

If you specify a statement type (delete, insert, or update), EGL uses the SQL record to build an implicit SQL statement, as described later. In any case, you can use the SQL record to test the outcome of the operation.

preparedStatementID
Refers to an EGL prepare statement that has the specified ID. If you do not reference a prepare statement, you must specify either an explicit SQL statement or a combination of an SQL record and a statement type (delete, insert, or update).

delete, insert, update
Indicates that EGL is to provide an implicit SQL statement of the specified type. A declaration-time error occurs if you specify a statement type but not an SQL record name.

If you do not set a statement type, you must specify either an explicit SQL statement or a reference to a prepare statement.

For an overview of implicit SQL statements, see *SQL support*.

Several example statements are as follows (assuming that employeeRecord is an SQL record):

```
execute
  #sql{
    create table employee (
      empnum decimal(6,0) not null,
      empname char(40) not null,
     empphone char(10) not null)
  };

execute update for employeeRecord;

execute
  #sql{
    call aStoredProcedure( :argumentItem)
  };
```

You can use an **execute** statement to issue SQL statements of the following types:

- ALTER
- CALL
- CREATE ALIAS
- CREATE INDEX
- CREATE SYNONYM
- CREATE TABLE
- CREATE VIEW
- DECLARE global temporary table
- DELETE
- DROP INDEX
- DROP SYNONYM
- DROP TABLE
- DROP VIEW
- GRANT
- INSERT
- LOCK
- RENAME
- REVOKE
- SAVEPOINT
- SET
- SIGNAL
- UPDATE
- VALUES

You cannot use an **execute** statement to issue SQL statements of the following types:

- CLOSE
- COMMIT
- CONNECT
- CREATE FUNCTION
- CREATE PROCEDURE
- DECLARE CURSOR
- DESCRIBE
- DISCONNECT
- EXECUTE
- EXECUTE IMMEDIATE
- FETCH
- OPEN
- PREPARE
- ROLLBACK WORK

- SELECT
- INCLUDE SQLCA
- INCLUDE SQLDA
- WHENEVER

Implicit SQL DELETE

The effect of requesting an implicit SQL DELETE statement is that an SQL record property (**defaultSelectCondition**) determines what table rows are deleted, so long as the value in each SQL table key column is equal to the value in the corresponding key item of the SQL record. If you specify neither a record key nor a default selection condition, all table rows are deleted.

The implicit SQL DELETE statement for a particular record is similar to the following statement:

```
DELETE FROM tableName
WHERE  keyColumn01 = :keyItem01
```

You cannot use a single EGL statement to delete rows from more than one database table.

Implicit SQL INSERT

The effect of requesting an implicit SQL INSERT statement is as follows by default:

- The key value in the record determines the logical position of the data in the table. A record that does not have a key is handled in accordance with the SQL table definition and the rules of the database.
- As a result of the association of record items and SQL table columns in the record part, the generated code places the data from each record item into the related SQL table column.
- If you declared a record item to be read only, the generated SQL INSERT statement does not include that record item, and the database management system sets the value of the related SQL table column to the default value that was specified when the column was defined.

The format of the implicit SQL INSERT statement is like this:

```
INSERT INTO tableName
  (column01, ... columnNN)
  values (:recordItem01, ... :recordItemNN)
```

Some error conditions are as follows:

- You specify an SQL statement of a type other than INSERT
- You specify some but not all clauses of an SQL INSERT statement
- You specify an SQL INSERT statement (or accept an implicit SQL statement) that has any of these characteristics--
  - Is related to more than one SQL table
  - Includes only host variables that you declared as read only
  - Is associated with a column that either does not exist or is incompatible with the related host variable

Implicit SQL UPDATE

The effect of requesting an implicit SQL UPDATE statement is as follows by default:

- An SQL record property (**defaultSelectCondition**) determines what table rows are selected, so long as the value in each SQL table key column is equal to the

value in the corresponding key item of the SQL record. If you specify neither a record key nor a default selection condition, all table rows are updated.

- As a result of the association of record items and SQL table columns in the SQL record declaration, a given SQL table column receives the content of the related record item. If an SQL table column is associated with a record item that is read only, however, that column is not updated.

The format of the implicit SQL UPDATE statement for a particular record is similar to the following statement:

```
UPDATE tableName
SET    column01 = :recordItem01,
       column02 = :recordItem01, ...
       columnNN = :recordItemNN
WHERE  keyColumn01 = :keyItem01
```

An error occurs in any of the following cases:

- All the items are identified as read only
- The statement attempts to update more than one SQL table
- An item whose value is being written to the database is associated with a column that either does not exist at run time or is incompatible with that item

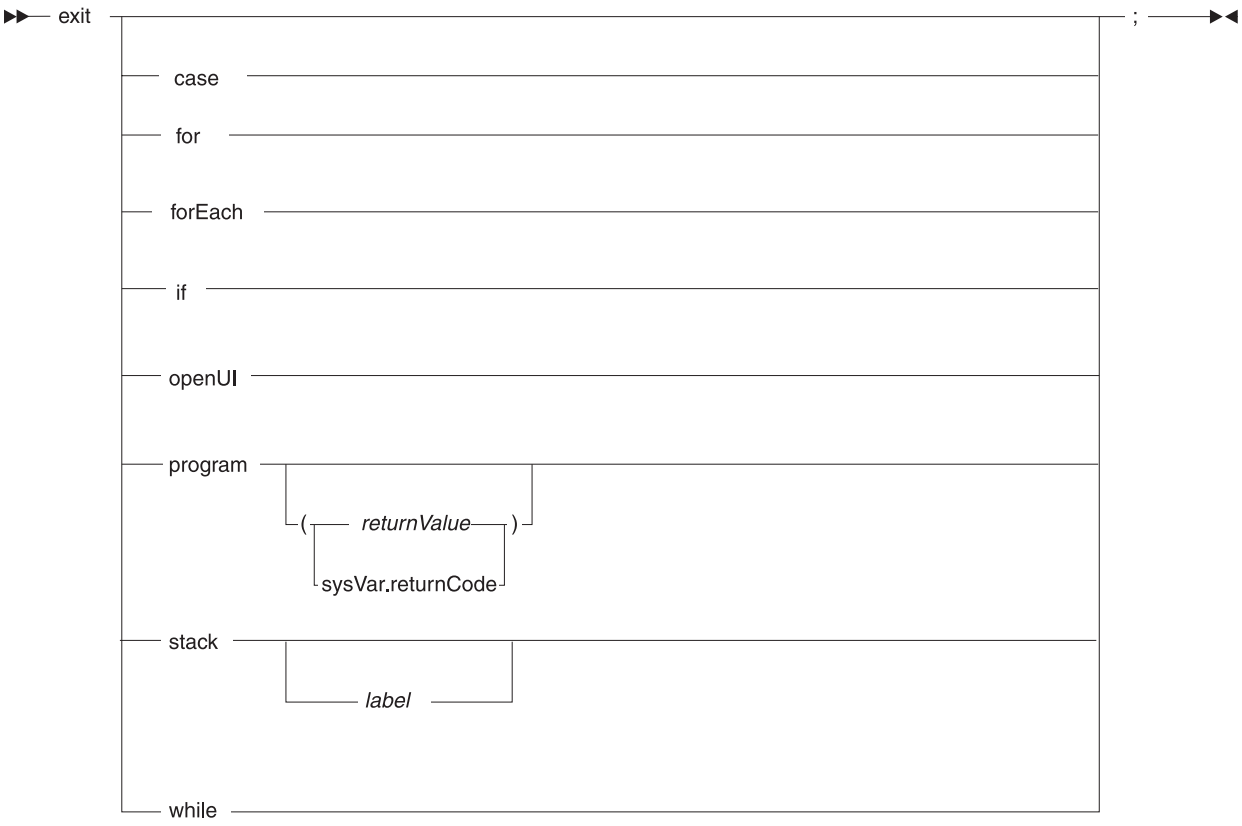
Related concepts
"Record types and properties" on page 126
"SQL support" on page 213
"References to parts" on page 20

Related tasks
"Syntax diagram for EGL statements and commands" on page 733

Related reference
"add" on page 544
"close" on page 551
"delete" on page 554
"EGL statements" on page 83
"Exception handling" on page 89
"get" on page 567
"get next" on page 579
"get previous" on page 584
"I/O error values" on page 522
"open" on page 598
"prepare" on page 611
"replace" on page 613
"SQL item properties" on page 63
"terminalID" on page 913

exit

The EGL **exit** statement leaves the specified block, which by default is the block that immediately contains the **exit** statement.



case
Leaves the most recently entered **case** statement in which the **exit** statement resides. Continues processing after the **case** statement.

An error occurs if the **exit** statement is not inside a **case** statement that begins in the same function.

for
Leaves the most recently entered **for** statement in which the **exit** statement resides. Continues processing after the **for** statement.

An error occurs if the **exit** statement is not inside a **for** statement that begins in the same function.

forEach
Leaves the most recently entered **forEach** statement in which the **exit** statement resides. Continues processing after the **forEach** statement.

An error occurs if the **exit** statement is not inside a **forEach** statement that begins in the same function.

if Leaves the most recently entered **if** statement in which the **exit** statement resides. Continues processing after the **if** statement.

An error occurs if the **exit** statement is not inside an **if** statement that begins in the same function.

program
Leaves the program.

The value in the system variable **sysVar.returnCode** is returned to the operating system in any of the following cases:

- The program ends with an **exit** statement that does not include a return code
- The program ends with an exit statement that returns **sysVar.returnCode**
- The program ends without a terminating **exit** statement

If the program ends with a terminating exit statement that includes a return code other than **sysVar.returnCode**, the specified value is used in place of any value that may be in **sysVar.returnCode**.

returnValue
A literal integer or an item, constant, or numeric expression that resolves to an integer. The return value is made available to the operating system.

For Java output, the value must be in the range of -2147483648 to 2147483647, inclusive. For COBOL output, the value must be in the range of 0 to 512, inclusive.

For other details on return values, see *sysVar.returnCode*.

sysVar.returnCode
The system variable that includes the value returned to the operating system.

For details, see *sysVar.returnCode*.

stack
Returns control to the main function without setting a return value for the current function.

A statement of the form *exit stack* removes all references to the intermediate functions in the run-time *stack*, which is a list of functions; specifically, the current function plus the series of functions whose running made possible the running of the current function.

The main function may have invoked a function (now in the stack), and the invocation may have included a parameter that had the modifier **out** or **inOut**. In those cases, the **exit** statement of the form *exit stack* makes the value of the parameters available to the main function.

If you do not specify a *label* (as described later), processing continues at the statement after the most recently run function invocation in the main function. If you specify a label, processing continues at the statement that follows the label in the main function. The label may precede or follow the most recently run function invocation in the main function.

If you specify an exit statement of the form *exit stack* in the main function, the next statement is processed, even if you specify a label. For details on how to go to a specified label in the current function, see *goTo*.

label
A series of characters that are displayed in the main function and outside of any blocks, including these:

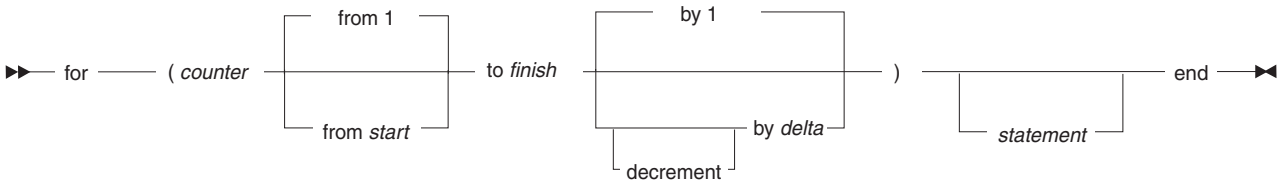
- if
- else
- inside a **case** statement
- while
- try

When displayed at the location where processing continues, the label is followed by colon. For details on valid characters for the label, see *Naming conventions*.

Related reference
"goTo" on page 590
"Naming conventions" on page 652
"returnCode" on page 908

for

The EGL keyword **for** begins a statement block that runs in a loop for as many times as a test evaluates to true. The test is conducted at the beginning of the loop and indicates whether the value of a counter is within a specified range. The keyword **end** marks the close of the **for** statement.



counter
A numeric variable without decimal places. EGL statements in the **for** statement can change the value of *counter*.

from *start*
The initial value of *counter*. The initial value is 1 if you do not specify a clause that begins with **from**.

start can be any of these:

- An integer literal
- A numeric variable without decimal places
- A numeric expression, which must resolve to an integer

to *finish*
If you do not specify **decrement**, *finish* is the upper limit of *counter*; and if the value of *counter* exceeds that limit, the test mentioned earlier resolves to false, the statement block is no longer executed, and the **for** statement ends.

If you specify **decrement**, *finish* is the lower limit of *counter*; and if the value of *counter* is below that limit, the test resolves to false, the statement block is no longer executed, and the **for** statement ends.

finish can be any of these:

- An integer literal
- A numeric variable without decimal places
- A numeric expression, which must resolve to an integer

EGL statements in the **for** statement can change the value of *finish*.

by *delta*
If you do not specify **decrement**, *delta* is the value added to *counter* after the EGL statement block is executed and before the value of *counter* is tested.

If you specify **decrement**, *delta* is the value subtracted from *counter* after the EGL statement block is executed and before the value of *counter* is tested.

delta can be any of these:

- An integer literal

- A numeric variable without decimal places
- A numeric expression, which must resolve to an integer

EGL statements in the **for** statement can change the value of *delta*.

statement
A statement in the EGL language

An example is as follows:

```
sum = 0;

// adds 10 values to sum
for (i from 1 to 10 by 1)
  sum = inputArray[i] + sum;
end
```

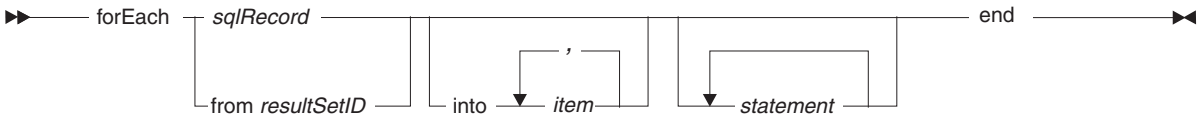
Related tasks
"Syntax diagram for EGL statements and commands" on page 733

Related reference
"EGL statements" on page 83

forEach

The EGL keyword **forEach** marks the start of a set of statements that run in a loop. The first iteration occurs only if a specified result set is available. (If the result set is not available, the statement fails with a hard error.) The loop reads each row in the result set, continuing until one of these events occurs:

- All rows are retrieved;
- An **exit** statement runs; or
- A hard or soft error occurs.



sqlRecord
Name of an SQL record that is used in a previously run **open** statement. You must specify either an SQL record or a result set ID, and it is recommended that you specify the result set ID.

from *resultSetID*
The result-set identifier that is used in a previously run **open** statement. For details, see *resultSetID*.

into ... *item*
An INTO clause, which identifies the EGL host variables that receive values from the cursor or stored procedure. In a clause like this one (which is outside of a **#sql{ }** block), do not include a semicolon before the name of a host variable.

A specification of an INTO clause in this context overrides any INTO clause identified in the related **open** statement.

statement
A statement in the EGL language

In most cases, the EGL run time issues an implicit **close** statement occurs after the last iteration of the **forEach** statement. That implicit statement modifies the SQL system variables, for which reason you may want to save the values of SQL-related system variables in the body of the **forEach** statement.

The EGL run time does not issue an implicit **close** statement if the **forEach** statement ends because of an error other than noRecordFound.

An example is as follows, as further described in *SQL examples*:

```
VGVar.handleHardIOErrors  = 1;

try
  open selectEmp
    with #sql{
      select empnum, empname
      from employee
      where empnum >= :empnum
      for update of empname
    }
    into empnum, empname;
onException
  myErrorHandler(6);    // exits program
end

try
  forEach (from selectEmp)
    empname = empname + " " + "III";

    try
      execute
        #sql{
          update employee
          set empname = :empname
        where current of selectEmp
        };
    onException
      myErrorHandler(10); // exits program
    end
  end  // end forEach; cursor is closed automatically
       // when the last row in the result set is read

onException
  // the exception block related to forEach is not run if the condition
  // is "sqlcode = 100", so avoid the test "if (sqlcode != 100)"
  myErrorHandler(8);  // exits program
end

sysLib.commit();
```

Related concepts
"resultSetID" on page 722
"SQL support" on page 213

Related tasks
"Syntax diagram for EGL statements and commands" on page 733

Related reference
"EGL statements" on page 83
"exit" on page 560
"open" on page 598
"SQL examples" on page 224

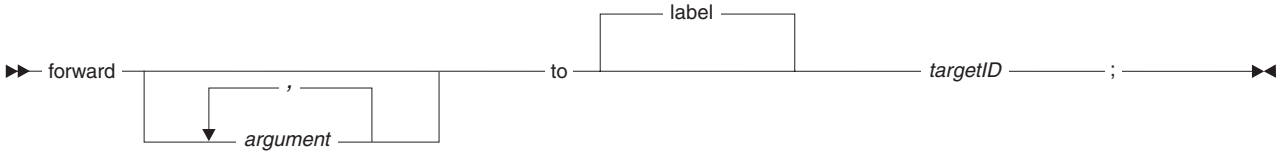
forward

The EGL **forward** statement is invoked from a PageHandler. The primary purpose is to display a Web page with variable information; but the statement also can invoke a servlet or Java program that runs in the Web application server.

The statement acts as follows:

1. Commits recoverable resources, closes files, and releases locks
2. Forwards control
3. Ends the code that runs the **forward** statement

The syntax diagram is as follows:



argument
An item or record that is passed to the code being invoked. The names of an argument and its corresponding parameter must be the same in all cases. You may not pass literals.

If you are invoking a PageHandler, the arguments must be compatible with the parameters specified for the **onPageLoad** function of the PageHandler. The function (if any) may have any valid name and is referenced by the PageHandler property **OnPageLoadFunction**. If you are invoking a program, the arguments must be compatible with the program parameters.

The following details may be of interest, depending on how you are using the technology:

- The argument must be named the same as the corresponding parameter because the name is used as a key in storing and retrieving the argument value on the Web application server.
- Instead of passing an argument, the invoker can do as follows before invoking the **forward** statement:
  - Place a value in the request block by invoking the system function J2EELib.setRequestAttr; or
  - Place a value in the session block by invoking the system function J2EELib.setSessionAttr.

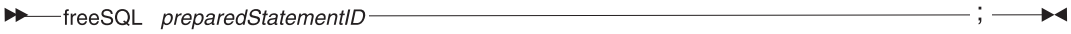
  In this case, the receiver does not receive the value as an argument, but by invoking the appropriate system function:
  - J2EELib.getRequestAttr (to access data from the request block); or
  - J2EELib.getSessionAttr (to access data from the session block).
- A character item is passed as an object of type Java String.
- A record is passed as a Java Bean.

to label *targetID*
Specifies a Java Server Faces (JSF) label, which identifies a mapping in a run-time JSF-based configuration file. The mapping in turn identifies the object to invoke, whether a JSP (usually one associated with an EGL PageHandler), an EGL program, a non-EGL program, or a servlet. The word **label** is optional, and *targetID* is a quoted string.

Related refere nce
"Function invocations" on page 504
"getRequestAttr()" on page 779
"getSessionAttr()" on page 780
"transferName" on page 914

freeSQL

The EGL **freeSQL** statement frees any resources associated with a dynamically prepared SQL statement, closing any open cursor associated with that SQL statement.



preparedStatementID
An identifier that identifies a **prepare** statement. No error occurs if that statement did not run previously.

After you issue a **freeSQL** statement, you cannot run the **execute**, **open**, or **get** statement for the prepared SQL statement without reissuing the **prepare** statement.

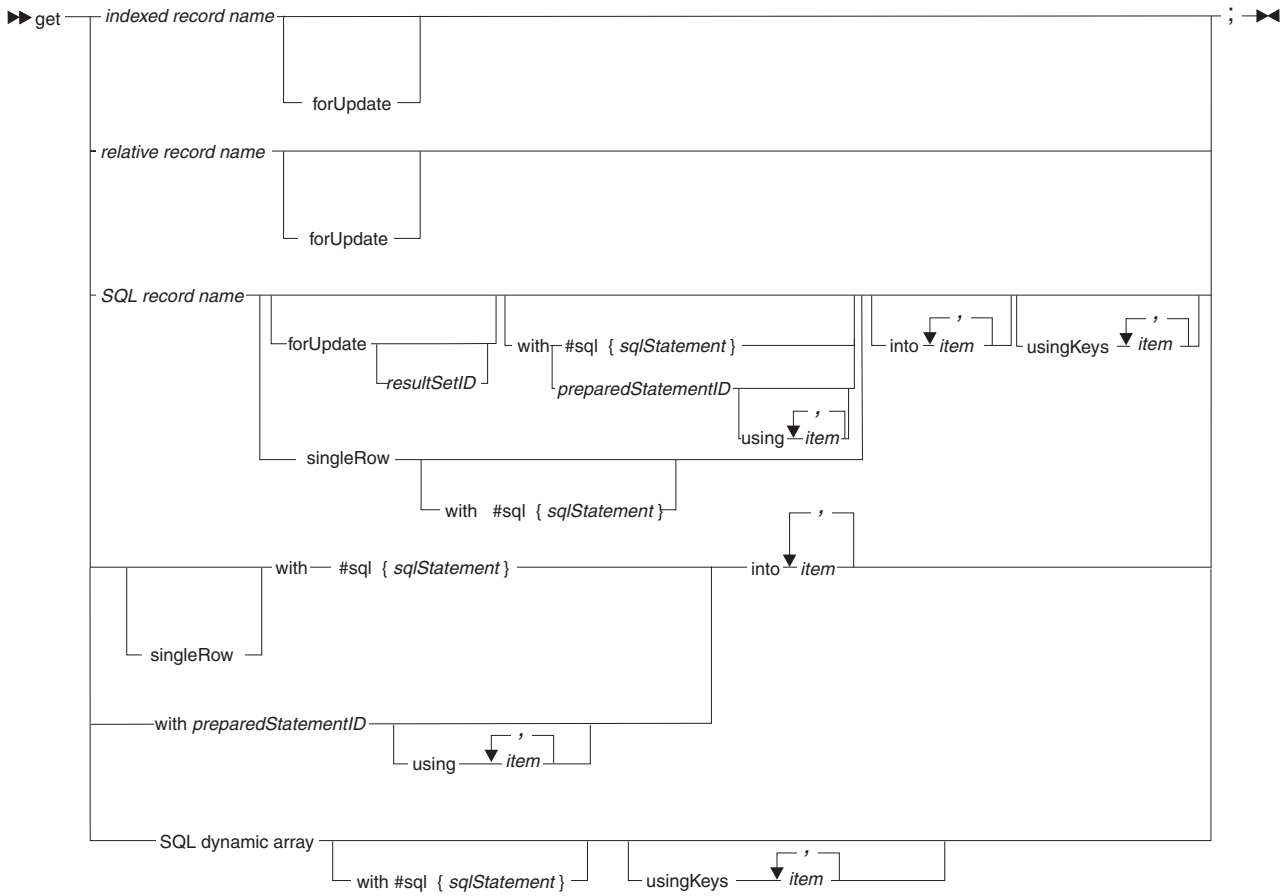
Related concepts
"SQL support" on page 213

Related reference
"execute" on page 557
"get"
"open" on page 598
"prepare" on page 611
"Syntax diagram for EGL statements and commands" on page 733

get

The EGL **get** statement retrieves a single file record or database row and provides an option that lets you replace or delete the stored data later in your code. In addition, this statement allows you to retrieve a set of database rows and place each succeeding row into the next SQL record in a dynamic array.

The **get** statement is sometimes identified as **get by key value** and is distinct from other statements that begin with the word *get*.



record name
Name of an I/O object: an indexed, relative, or SQL record. For SQL processing, the record name is required if the EGL INTO clause (described later) is not specified.

forUpdate
Option that lets you use a later EGL statement to replace or delete the data that was retrieved from the file or database.

If the resource is recoverable (as in the case of a VSAM file or SQL database), the **forUpdate** option locks the record so that it cannot be changed by other programs until a commit occurs. For details on commit processing, see *Logical unit of work*.

resultSetID
A result-set identifier for use in an EGL **replace**, **delete**, or **execute** statement, as well as in an EGL **close** statement. For details, see *resultSetID*.

singleRow
Option that causes generation of more efficient SQL, as is appropriate when you are sure that the search criterion in the **get** statement applies to only one row and when you do not intend to update or delete the row. A run-time I/O error results if you specify this option when the search criterion applies to multiple rows. For additional details, see *SQL record*.

#sql{ *sqlStatement* **}**
An explicit SQL SELECT statement, as described in *SQL support*. Leave no space between #sql and the left brace.

into ... *item*
An EGL INTO clause, which identifies the EGL host variables that receive values from a relational database. This clause is required when you are processing SQL, in either of these cases:

- An SQL record is not specified; or
- Both an SQL record and an explicit SQL SELECT statement are specified, but a column in the SQL SELECT clause is not associated with a record item. (The association is in the SQL record part, as noted in *SQL item properties*.)

In a clause like this one (which is outside of an **#sql{ }** block), do not include a semicolon before the name of a host variable.

preparedStatementID
The identifier of an EGL **prepare** statement that prepares an SQL SELECT statement at run time. The **get** statement runs the SQL SELECT statement dynamically. For details, see *prepare*.

using ... *item*
A USING clause, which identifies the EGL host variables that are made available to the prepared SQL SELECT statement at run time. In a clause like this one (which is outside of an **sql**-and-**end** block), do not include a semicolon before the name of a host variable.

usingKeys ... *item*
Identifies a list of key items that are used to build the key-value component of the WHERE clause in an implicit SQL statement. The implicit SQL statement is used at run time if you do not specify an explicit SQL statement.

If you do not specify a **usingKeys** clause, the key-value component of the implicit statement is based on the SQL record part that is either referenced in the **get** statement or is the basis of the dynamic array referenced in the **get** statement.

In the case of a dynamic array, the items in the **usingKeys** clause (or the host variables in the SQL record) must *not* be in the SQL record that is the basis of the dynamic array.

The **usingKeys** information is ignored if you specify an explicit SQL statement.

SQL dynamic array
Name of a dynamic array that is composed of SQL records.

The following example shows how to read and replace a file record:

```
emp.empnum = 1;         // sets the key in record emp

try
  get emp forUpdate;
onException
  myErrorHandler(8);  // exits the program
end

emp.empname = emp.empname + " Smith";

try
  replace emp;
onException
  myErrorHandler(12);
end
```

The next **get** statement uses the SQL record emp when retrieving a database row, with no subsequent update or deletion possible:

```
try
  get emp singleRow into empname with
    #sql{
      select empname
      from Employee
      where empnum = :empnum
    };
onException
  myErrorHandler(8);
end
```

The next example uses the same SQL record to replace an SQL row:

```
try
  get emp forUpdate into empname with
    #sql{
      select empname
      from Employee
      where empnum = :empnum
    };

onException
  myErrorHandler(8);  // exits the program
end

emp.empname = emp.empname + " Smith";

try
  replace emp;
onException
  myErrorHandler(12);
end
```

Details on the **get** statement depend on the record type. For details on SQL processing, see *SQL record*.

Indexed record

When you issue a **get** statement against an indexed record, the key value in the record determines what record is retrieved from the file.

If you want to replace or delete an indexed (or relative) record, you must issue a **get** statement for the record and then issue the file-changing statement (**replace** or **delete**), with no intervening I/O operation against the same file. After you issue the **get** statement, the effect of the next I/O operation on the same file is as follows:

- If the next I/O operation is a **replace** statement on the same EGL record, the record is changed in the file
- If the next I/O operation is a **delete** statement on the same EGL record, the record in the file is marked for deletion
- If the next I/O operation is a **get** statement on a record in the same file and includes the forUpdate option, a subsequent **replace** or **delete** statement is valid on the newly read file record
- If the next I/O operation is a **get** statement on the same EGL record (with no forUpdate opton) or is a **close** statement on the same file, the file record is released without change

If the file is a VSAM file, the EGL **get** statement (with the forUpdate option) prevents the record from being changed by other programs. In iSeries COBOL programs, the lock remains until a commit occurs, which may not happen until the end of the run unit, as described in *Run unit*.

Relative record

When you issue a **get** statement against a relative record, the key item associated with the record determines what record is retrieved from the file. The key item must be available to any function that uses the record and can be any of these:

- An item in the same record
- An item in a record that is global to the program or is local to the function that is running the **get** statement
- A data item that is global to the program or is local to the function that is running the **get** statement

If you want to replace or delete an indexed (or relative) record, you must issue a **get** statement for the record and then issue the file-changing statement (**replace** or **delete**), with no intervening I/O operation against the same file. After you issue the **get** statement, the effect of the next I/O operation on the same file is as follows:

- If the next I/O operation is a **replace** statement on the same EGL record, the record is changed in the file
- If the next I/O operation is a **delete** statement on the same EGL record, the record in the file is marked for deletion
- If the next I/O operation is a **get** on the same file (with the forUpdate option), a subsequent replace or delete is valid on the newly read file record
- If the next I/O operation is a **get** on the same EGL record (with no forUpdate option) or is a close on the same file, the file record is released without change

SQL record

The EGL **get** statement results in an SQL SELECT statement in the generated code. If you specify the singleRow option, the SQL SELECT statement is a stand-alone statement. Alternatively, the SQL SELECT statement is a clause in a cursor, as described in *SQL support*.

Error conditions: The following conditions are among those that are not valid when you use a **get** statement to read data from a relational database:

- You specify an SQL statement of a type other than SELECT
- You specify an SQL INTO clause directly in an SQL SELECT statement
- Aside from an SQL INTO clause, you specify some but not all of the clauses of an SQL SELECT statement
- You specify (or accept) an SQL SELECT statement that is associated with a column that either does not exist or is incompatible with the related host variable

The following error conditions are among those that can occur when you use the forUpdate option:

- You specify (or accept) an SQL statement that shows an intent to update multiple tables; or
- You use an SQL record as an I/O object, and all the record items are read only.

Also, the following situation causes an error:

- You customize an EGL **get** statement with the forUpdate option, but fail to indicate that a particular SQL table column is available for update; and
- The replace statement that is related to that **get** statement tries to revise the column.

You can solve the previous mismatch in any of these ways:

- When you customize the EGL **get** statement, include the column name in the SQL SELECT statement, FOR UPDATE OF clause; or
- When you customize the EGL replace statement, eliminate reference to the column in the SQL UPDATE statement, SET clause; or
- Accept the defaults for both the **get** and **replace** statements.

Implicit SQL SELECT statement: When you specify an SQL record as an I/O object for the **get** statement but do not specify an explict SQL statement, the implicit SQL SELECT has the following characteristics:

- The record-specific property called **defaultSelectCondition** determines what table row is selected, so long as the value in each SQL table key column is equal to the value in the corresponding key item of the SQL record. If you specify neither a record key nor a default selection condition, all table rows are selected. If multiple table rows are selected for any reason, the first retrieved row is placed in the record.
- As a result of the association of record items and SQL table columns in the record definition, a given item receives the content of the related SQL table column.
- If you specify the forUpdate option, the SQL SELECT FOR UPDATE statement does not include record items that are read only.
- The SQL SELECT statement for a particular record is similar to the following statement, except that the FOR UPDATE OF clause is present only if the **get** statement includes the forUpdate option :

```
SELECT column01,
       column02, ...
       columnNN
FROM   tableName
WHERE  keyColumn01 = :keyItem01
FOR UPDATE OF
       column01,
       column02, ...
       columnNN
```

  The SQL INTO clause on the standalone SQL SELECT or on the cursor-related FETCH statement is similar to this clause:

```
INTO   :recordItem01,
       :recordItem02, ...
       :recordItemNN
```

  EGL derives the SQL INTO clause if the SQL record is accompanied by an explicit SQL SELECT statement when you have not specified an INTO clause. The items in the derived INTO clause are those that are associated with the columns listed in the SELECT clause of the SQL statement. (The item-and-column association is in the SQL record part, as noted in *SQL item properties*.) An EGL INTO clause is required if a column is not associated with an item.

When you specify a dynamic array of SQL records as an I/O object for the **get** statement but do not specify an explict SQL statement, the implicit SQL SELECT is similar to that described for a single SQL record, with these differences:

- The key-value component of the query is a set of relationships that is based on a greater-than-or-equal-to condition:

```
keyColumn01 >= :keyItem01 &
keyColumn02 >= :keyItem02 &
        .
        .
        .
keyColumnN  >= :keyItemN
```

- The items in the **usingKeys** clause (or the host variables in the SQL record) must *not* be in the SQL record that is the basis of the dynamic array.

Related concepts
"Logical unit of work" on page 288
"Record types and properties" on page 126
"References to parts" on page 20
"resultSetID" on page 722
"SQL support" on page 213

Related tasks
"Syntax diagram for EGL statements and commands" on page 733

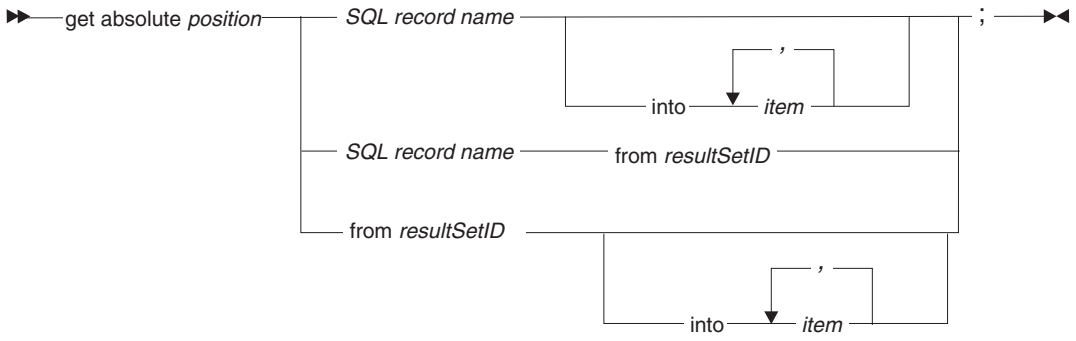
Related reference
"add" on page 544
"close" on page 551

"delete" on page 554
"EGL statements" on page 83
"Exception handling" on page 89
"execute" on page 557
"get next" on page 579
"get previous" on page 584
"I/O error values" on page 522
"open" on page 598
"prepare" on page 611
"replace" on page 613
"SQL item properties" on page 63
"terminalID" on page 913

get absolute

The EGL **get absolute** statement reads a numerically specified row in a relational-database result set. The row is identified in relation either to the beginning of the result set (if you specify a positive value) or to the end of the result set (if you specify a negative value).

You can use this statement only if you specified the scroll option in the related **open** statement. The scroll option is available only if you are generating output in Java.



position
An integer item or literal.

If the value of *position* is positive, the row is identified in relation to the beginning of the result set. Specifying **get absolute 1**, for example, retrieves the first row and is equivalent to specifying **get first**. Specifying **get absolute 2** retrieves the second row.

If the value of *position* is negative, the row is identified in relation to the end of the result set. Specifying **get absolute -1**, for example, retrieves the last row and is equivalent to specifying **get last**. Specifying **get absolute -2** retrieves the second to last row.

A value of zero for *position* causes a hard error, as described in *Exception handling*.

record name
Name of an SQL record.

from *resultSetID*
An ID that ties the **get absolute** statement to an **open** statement run earlier in the same program. For details, see *resultSetID*.

into
Begins an EGL into clause, which lists the items that receive values from a relational-database table.

item
An item that receives the value of a particular column. Do *not* precede the item name with a colon (:).

If you issue a **get absolute** statement to retrieve a row that was selected by an **open** statement that has the forUpdate option, you can do any of these:

- Change the row with an EGL **replace** statement
- Remove the row with an EGL **delete** statement
- Change or remove the row with an EGL **execute** statement

An SQL FETCH statement represents the EGL **get absolute** statement in the generated code. The format of the generated SQL statement cannot be changed, except to set the INTO clause.

If you issue a **get absolute** statement that attempts to access a row that is not in the result set, the EGL run time acts as follows:

- Does not copy data from the result set

- Leaves the cursor open, with the cursor position unchanged
- Sets the SQL record (if any) to **noRecordFound**

In general, if an error occurs and processing continues, the cursor remains open, with the cursor position unchanged.

Finally, when you specify SQL COMMIT or sysLib.commit, your code retains position in the cursor that was declared in the **open** statement, but only if you use the hold option in the **open** statement.

Related concepts
"resultSetID" on page 722
"SQL support" on page 213

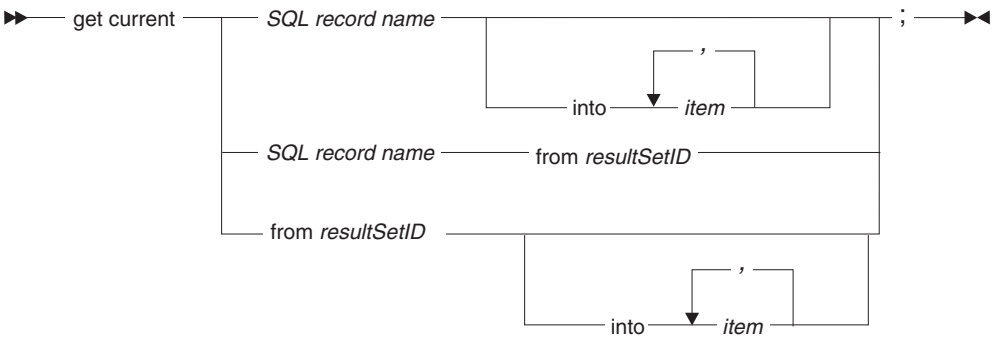
Related tasks
"Syntax diagram for EGL statements and commands" on page 733

Related reference
"delete" on page 554
"Exception handling" on page 89
"execute" on page 557
"get" on page 567
"get current"
"get first" on page 576
"get last" on page 578
"get next" on page 579
"get previous" on page 584
"get relative" on page 588
"EGL statements" on page 83
"open" on page 598
"replace" on page 613

get current

The EGL **get current** statement reads the row at which the cursor is already positioned in a relational-database result set.

You can use this statement only if you specified the scroll option in the related **open** statement. The scroll option is available only if you are generating output in Java.



record name
Name of an SQL record.

from *resultSetID*
An ID that ties the **get current** statement to an **open** statement run earlier in the same program. For details, see *resultSetID*.

into
Begins an EGL into clause, which lists the items that receive values from a relational-database table.

item
An item that receives the value of a particular column. Do *not* precede the item name with a colon (:).

If you issue a **get current** statement to retrieve a row that was selected by an **open** statement that has the forUpdate option, you can do any of these:

- Change the row with an EGL **replace** statement
- Remove the row with an EGL **delete** statement
- Change or remove the row with an EGL **execute** statement

An SQL FETCH statement represents the EGL **get current** statement in the generated code. The format of the generated SQL statement cannot be changed, except to set the INTO clause.

If an error occurs and processing continues, the cursor remains open.

Finally, when you specify SQL COMMIT or sysLib.commit, your code retains position in the cursor that was declared in the **open** statement, but only if you use the hold option in the **open** statement.

Related concepts
"resultSetID" on page 722
"SQL support" on page 213

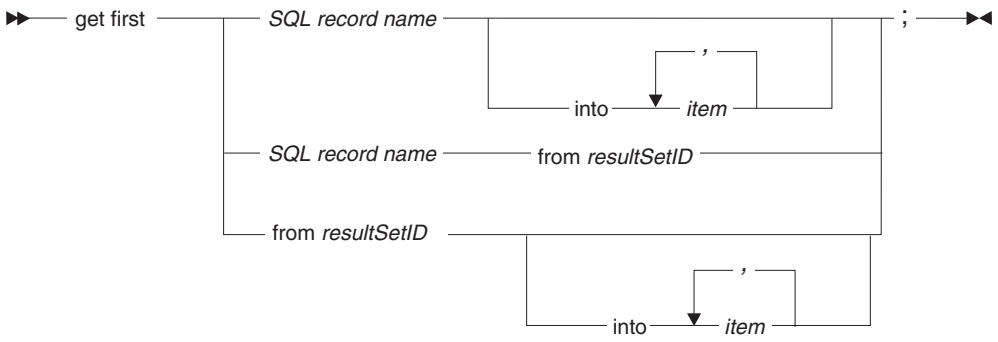
Related tasks
"Syntax diagram for EGL statements and commands" on page 733

Related reference
"delete" on page 554
"execute" on page 557
"get" on page 567
"get absolute" on page 573
"get first"
"get last" on page 578
"get next" on page 579
"get previous" on page 584
"get relative" on page 588
"EGL statements" on page 83
"open" on page 598
"replace" on page 613

get first

The EGL **get first** statement reads the first row in a relational-database result set.

You can use this statement only if you specified the scroll option in the related **open** statement. The scroll option is available only if you are generating output in Java.



record name
Name of an SQL record.

from *resultSetID*
An ID that ties the **get first** statement to an **open** statement run earlier in the same program. For details, see *resultSetID*.

into
Begins an EGL into clause, which lists the items that receive values from a relational-database table.

item
An item that receives the value of a particular column. Do *not* precede the item name with a colon (:).

If you issue a **get first** statement to retrieve a row that was selected by an **open** statement that has the forUpdate option, you can do any of these:

- Change the row with an EGL **replace** statement
- Remove the row with an EGL **delete** statement
- Change or remove the row with an EGL **execute** statement

An SQL FETCH statement represents the EGL **get first** statement in the generated code. The format of the generated SQL statement cannot be changed, except to set the INTO clause.

If an error occurs and processing continues, the cursor remains open.

Finally, when you specify SQL COMMIT or sysLib.commit, your code retains position in the cursor that was declared in the **open** statement, but only if you use the hold option in the **open** statement.

Related concepts
"resultSetID" on page 722
"SQL support" on page 213

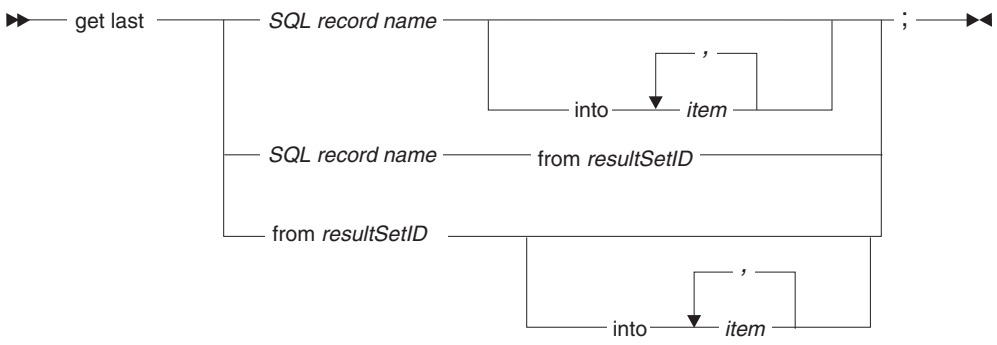
Related tasks
"Syntax diagram for EGL statements and commands" on page 733

Related reference
"delete" on page 554
"execute" on page 557
"get" on page 567
"get absolute" on page 573
"get current" on page 575
"get last"
"get next" on page 579
"get previous" on page 584
"get relative" on page 588
"EGL statements" on page 83
"open" on page 598
"replace" on page 613

get last

The EGL **get last** statement reads the last row in a relational-database result set.

You can use this statement only if you specified the scroll option in the related **open** statement. The scroll option is available only if you are generating output in Java.



record name
Name of an SQL record.

from *resultSetID*
An ID that ties the **get last** statement to an **open** statement run earlier in the same program. For details, see *resultSetID*.

into
Begins an EGL into clause, which lists the items that receive values from a relational-database table.

item
An item that receives the value of a particular column. Do *not* precede the item name with a colon (:).

If you issue a **get last** statement to retrieve a row that was selected by an **open** statement that has the forUpdate option, you can do any of these:

- Change the row with an EGL **replace** statement
- Remove the row with an EGL **delete** statement
- Change or remove the row with an EGL **execute** statement

An SQL FETCH statement represents the EGL **get last** statement in the generated code. The format of the generated SQL statement cannot be changed, except to set the INTO clause.

If an error occurs and processing continues, the cursor remains open.

Finally, when you specify SQL COMMIT or sysLib.commit, your code retains position in the cursor that was declared in the **open** statement, but only if you use the hold option in the **open** statement.

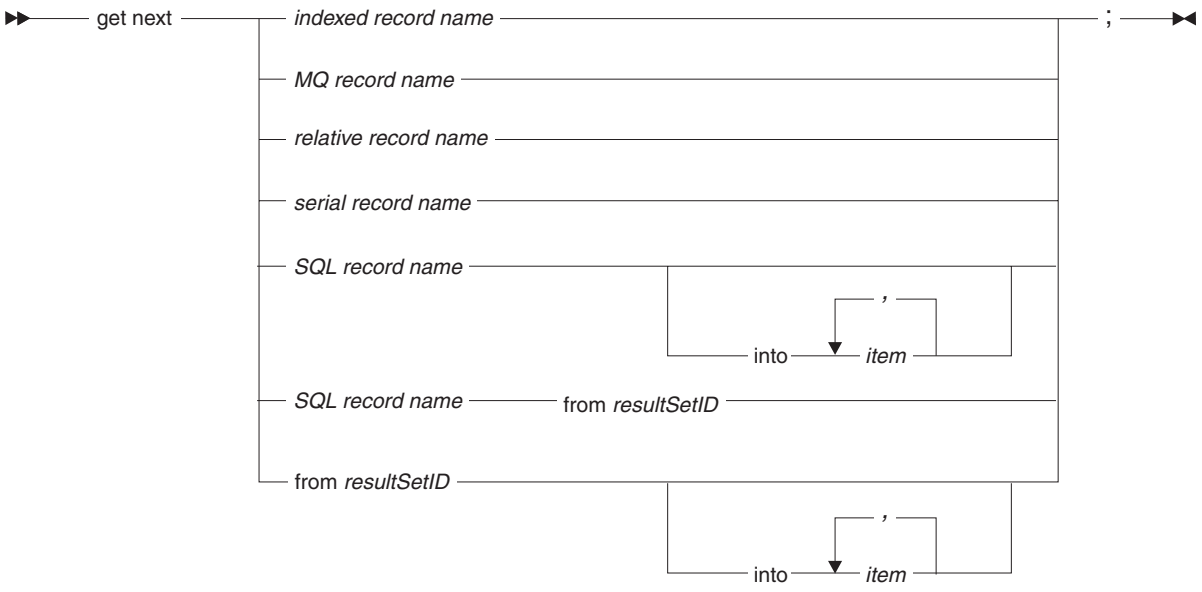
Related concepts
"resultSetID" on page 722
"SQL support" on page 213

Related tasks
"Syntax diagram for EGL statements and commands" on page 733

Related reference
"delete" on page 554
"execute" on page 557
"get" on page 567
"get absolute" on page 573
"get current" on page 575
"get first" on page 576
"get next"
"get previous" on page 584
"get relative" on page 588
"EGL statements" on page 83
"open" on page 598
"replace" on page 613

get next

The EGL **get next** statement reads the next record from a file or message queue, or the next row from a database.



record name
Name of the I/O object: an indexed, MQ, relative, serial, or SQL record.

from *resultSetID*
For SQL processing only, an ID that ties the **get next** statement to an **open** statement run earlier in the same program. For details, see *resultSetID*.

into
Begins an EGL into clause, which lists the items that receive values from a relational-database table.

item
An item that receives the value of a particular column. Do *not* precede the item name with a colon (:).

An example of file access is as follows:

```
try
  open record1 forUpdate;
  onException
    myErrorHandler(8);
    return;
end
try
  get next record1;
  onException
    myErrorHandler(12);
    return;
end

while (record1 not endOfFile)
  makeChanges(record1); // process the record

  try
    replace record1;
  onException
    myErrorHandler(16);
    return;
  end

  try
    get next record1;
  onException
    myErrorHandler(12);
    return;
  end
end // end while

sysLib.commit();
```

Details on the **get next** statement depends on the record type. For details on SQL processing, see "SQL processing" on page 583.

Indexed record

When a **get next** statement operates on an indexed record, the effect is based on the current file position, which is set by either of these operations:

- A successful input or output (I/O) operation such as a **get** statement or another **get next** statement; or
- A **set** statement of the form *set record position*.

Rules are as follows:

- When the file is not open, the **get next** statement reads a record with the lowest key value in the file.

- Each subsequent **get next** statement reads a record that has the next highest key value in relation to the current file position. An exception for duplicate keys is described later.
- After a **get next** statement reads the record with the highest key value in the file, the next **get next** statement results in the I/O error value **endOfFile**.
- The current file position is affected by any of these operations:
  - An EGL **set** statement of the form *set record position* establishes a file position based on the *set value*, which is the key value in the indexed record that is referenced by the **set** statement. The subsequent **get next** statement against the same indexed record reads the file record that has a key value equal to or greater than the set value. If no such record exists, the result of the **get next** is endOfFile.
  - A successful I/O statement other than a **get next** statement establishes a new file position, and the subsequent **get next** statement issued against the same EGL record reads the *next* file record. After a **get previous** statement reads a file record, for example, the **get next** statement either reads the file record with the next-highest key or returns endOfFile.
  - If a **get previous** statement returns **endOfFile**, the subsequent **get next** statement retrieves the first record in the file.
  - After an unsuccessful **get**, **get next**, or **get previous** statement, the file position is undefined and must be re-established by a **set** statement of the form *set record position* or by an I/O operation other than a **get next** or **get previous** statement.
- When you are using an alternate index and duplicate keys are in the file, the following rules apply:
  - Retrieval of a record with a higher-valued key occurs only after **get next** statements have read all the records that have the same key as the most recently retrieved record. The order in which duplicate-keyed records are retrieved is the order in which VSAM returns the records.
  - If a **get next** follows a successful I/O operation other than a **get next**, the **get next** skips over any duplicate-keyed records and retrieves the record with the next higher key.
  - The EGL error value **duplicate** is not set when your program retrieves the last record in a group of records containing the same key.

Consider a file in which the keys are as follows:

```
1, 2, 2, 2, 3, 4
```

Each of the following tables illustrates the effect of running a sequence of EGL statements on the same indexed record.

The next two tables apply to EGL-generated COBOL code.

EGL statement (in order)	Key in the indexed record	Key in the file record retrieved by the statement	EGL error value for COBOL
get	2	2 (the first of three)	duplicate
get next	any	3	—

EGL statement (in order)	Key in the indexed record	Key in the file record retrieved by the statement	EGL error value for COBOL
set (of the form *set record position*)	2	no retrieval	duplicate
get next	any	2 (the first of three)	duplicate
get next	any	2 (the second)	duplicate
get next	any	2 (the third)	—
get next	any	3	—

The next two tables apply to EGL-generated Java code.

EGL statement (in order)	Key in the indexed record	Key in the file record retrieved by the statement	EGL error value for Java
get	2	2 (the first of three)	duplicate
get next	any	2 (the second)	duplicate
get next	any	2 (the third)	—
get next	any	3	—

EGL statement (in order)	Key in the indexed record	Key in the file record retrieved by the statement	EGL error value for Java
set (of the form *set record position*)	2	no retrieval	duplicate
get next	any	2 (the first of three)	—
get next	any	2 (the second)	duplicate
get next	any	2 (the third)	—
get next	any	3	—

Message queue

When a **get next** operates on a MQ record, the first record in the queue is read into the MQ record. This placement occurs because the **get next** invokes one or more MQSeries calls:

- MQCONN connects the generated code to the default queue manager and is invoked when no connection is active
- MQOPEN establishes a connection to the queue and is invoked when a connection is active but the queue is not open
- MQGET removes the record from the queue and is always invoked unless an error occurred in an earlier MQSeries call

Relative record

When a **get next** statement operates on a relative record, the effect is based on the current file position, which is set by a successful input or output (I/O) operation such as a **get** statement or another **get next** statement. Rules are as follows:

- When the file is not open, the **get next** statement reads the first record in the file.
- Each subsequent **get next** reads a record that has the next highest key value in relation to the current file position.
- A **get next** does not return **noRecordFound** if the next record is deleted. Instead, the **get next** skips deleted records and retrieves the next record in the file.

- After a **get next** statement reads the record with the highest key value in the file, the next **get next** statement results in the EGL error value **endOfFile**.
- The current file position is affected by any of these operations:
  - A successful I/O statement other than a **get next** establishes a new file position, and the subsequent **get next** against the same EGL record reads the *next* file record.
  - After an unsuccessful **get**, **get next**, or **get previous** statement, the file position is undefined and must be re-established by a **set** statement of the form *set record position* or by an I/O operation other than a **get next** statement.
- After a **get next** statement reads the last record in the file, the next **get next** statement results in the EGL error values **endOfFile** and **noRecordFound**.

Serial record

When a **get next** statement operates on a serial record, the effect is based on the current file position, which is set by another **get next** statement. Rules are as follows:

- When the file is not open, the **get next** statement reads the first record in the file.
- Each subsequent **get next** statement reads the next record.
- After a **get next** statement reads the last record, the subsequent **get next** statement results in the EGL error value **endOfFile**.
- If the generated code adds a serial record and then issues the equivalent of a **get next** statement on the same file, EGL closes and reopens the file before executing the **get next** statement. A **get next** statement that follows an **add** statement therefore reads the first record in the file. This behavior also occurs when the **get next** and **add** statements are in different programs, and one program calls another.

It is recommended that you avoid having the same file open in more than one program at the same time.

SQL processing

When a **get next** statement operates on an SQL record, your code reads the next row from those selected by an **open** statement. If you issue a **get next** statement to retrieve a row that was selected by an **open** statement that has the forUpdate option, you can do any of these:

- Change the row with an EGL **replace** statement
- Remove the row with an EGL **delete** statement
- Change or remove the row with an EGL **execute** statement

An SQL FETCH statement represents the EGL **get next** statement in the generated code. The format of the generated SQL statement cannot be changed, except to set the INTO clause.

If you issue a **get next** statement that attempts to access a row that is beyond the last selected row, the following statements apply:

- No data is copied from the result set
- EGL sets the SQL record (if any) to **noRecordFound**
- If the related **open** statement included the scroll option, the cursor remains open with the cursor position unchanged. The scroll option is valid only if you are generating output in Java.
- If you have not set the scroll option, the cursor is closed.

Finally, when you specify SQL COMMIT or sysLib.commit, your code retains position in the cursor that was declared in the **open** statement, but only if you use the hold option in the **open** statement.

Related concepts
"Record types and properties" on page 126
"resultSetID" on page 722
"SQL support" on page 213

Related tasks
"Syntax diagram for EGL statements and commands" on page 733

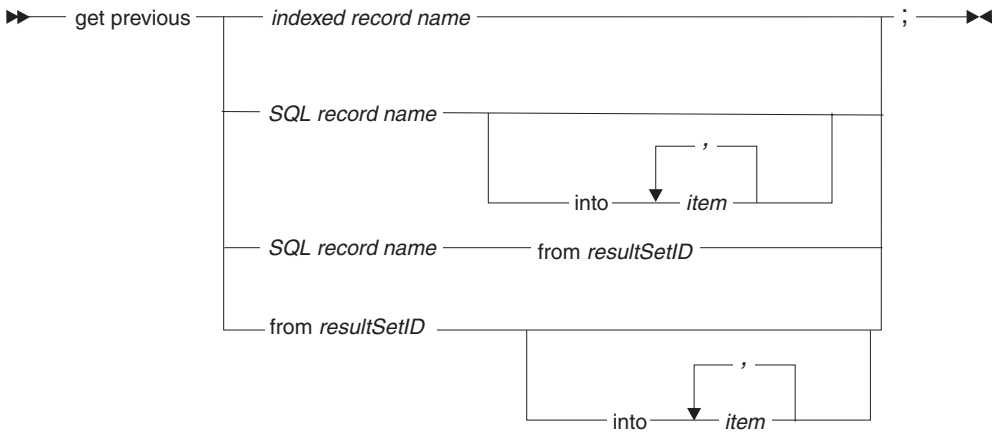
Related reference
"add" on page 544

"close" on page 551
"delete" on page 554
"Exception handling" on page 89
"execute" on page 557
"get" on page 567
"get previous"
"I/O error values" on page 522
"EGL statements" on page 83
"open" on page 598
"prepare" on page 611
"replace" on page 613
"set" on page 617

get previous

The EGL **get previous** statement either reads the previous row from a relational-database result set or reads the previous record in the file that is associated with a specified EGL indexed record.

You can use this statement for a relational-database result set only if you specified the scroll option in the related **open** statement. The scroll option is available only if you are generating output in Java.



record name
Name of the I/O object: an indexed or SQL record.

from *resultSetID*
For SQL processing only, an ID that ties the **get previous** statement to an **open** statement run earlier in the same program. For details, see *resultSetID*.

into
Begins an EGL into clause, which lists the items that receive values from a relational-database table.

item
An item that receives the value of a particular column. Do *not* precede the item name with a colon (:).

An example for an indexed record is as follows:

```
record1.hexKey = "FF";
set record1 position;

try
  get previous record1;
onException
  myErrorHandler(8);
  return;
end

while (record1 not endOfFile)
  processRecord(record1); // handle the data

  try
    get previous record1;
  onException
    myErrorHandler(8);
    return;
  end
end
```

Details on the **get previous** statement depend on whether you are using an indexed record or are concerned with "SQL processing" on page 587.

Indexed record

When a **get previous** statement operates on an indexed record, the effect is based on the current file position, which is set by either of these operations:

- A successful input or output (I/O) operation such as a **get** statement or another **get previous** statement; or
- A **set statement** of the form *set record position*.

Rules for an indexed record are as follows:

- When the file is not open, the **get previous** statement reads a record with the highest key value in the file.
- Each subsequent **get previous** reads a record that has the next lowest key value in relation to the current file position. An exception for duplicate keys is described later.
- After a **get previous** statement reads the record with the lowest key value in the file, the next **get previous** statement results in the EGL error value **endOfFile**.
- The current file position is affected by any of these operations:
  - An EGL **set statement** of the form *set record position* establishes a file position based on the *set value*, which is the key value in the indexed record that is referenced by the **set statement**. The subsequent **get previous** statement

against the same indexed record reads the file record that has a key value equal to or less than the set value. If no such record exists, the result of the **get previous** statement is **endOfFile**.

If the set value is filled with hexadecimal FF, the result of a **set** statement of the form *set record position* is as follows:

- The **set** statement establishes a file position after the last record in the file
- If a **get previous** statement is the next I/O operation, the generated code retrieves the last record in the file

– A successful I/O statement other than a **get previous** statement establishes a new file position, and the subsequent **get previous** statement against the same EGL record reads the *previous* file record. After a **get next** statement reads a file record, for example, the **get previous** statement either reads the file record with the next-lowest key or returns **endOfFile**.

– If a **get next** statement returns **endOfFile**, the subsequent **get previous** statement retrieves the last record in the file.

– After an unsuccessful **get**, **get next**, or **get previous** statement, the file position is undefined and must be re-established by a **set** statement of the form *set record position* or by an I/O operation other than a **get next** or **get previous** statement.

• When you are using an alternate index and duplicate keys are in the file, the following rules apply:

– Retrieval of a record with a lower-valued key occurs only after **get previous** statements have read all the records that have the same key as the most recently retrieved record. The order in which duplicate-keyed records are retrieved is the order in which VSAM returns the records.

– If a **get previous** statement follows a successful I/O operation other than a **get previous**, the **get previous** statement skips over any duplicate-keyed records and retrieves the record with the next lower key.

– The EGL error value **duplicate** is not set when your program retrieves the last record in a group of records containing the same key.

Consider a file in which the keys in an alternate index are as follows:

```
1, 2, 2, 2, 3, 4
```

Each of the following tables illustrates the effect of running a sequence of EGL statements on the same indexed record.

The next three tables apply to EGL-generated COBOL code.

EGL statement (in order)	Key in the indexed record	Key in the file record retrieved by the statement	EGL error value for COBOL
get	3	3	—
get previous	any	2 (the first of three)	duplicate
get previous	any	2 (the second)	duplicate
get previous	any	2 (the third)	—
get previous	any	1	—

EGL statement (in order)	Key in the indexed record	Key in the file record retrieved by the statement	EGL error value for COBOL
set (of the form *set record position*)	2	—	—

EGL statement (in order)	Key in the indexed record	Key in the file record retrieved by the statement	EGL error value for COBOL
get next	any	2 (the first)	duplicate
get next	any	2 (the second)	—
get previous	any	1	—
get previous	any	--	endOfFile

EGL statement (in order)	Key in the indexed record	Key in the file record retrieved by the statement	EGL error value for COBOL
set (of the form *set record position*)	1	--	--
get previous	any	1	--

The next three tables apply to EGL-generated Java code.

EGL statement (in order)	Key in the indexed record	Key in the file record retrieved by the statement	EGL error value for Java
get	3	3	—
get previous	any	2 (the first of three)	duplicate
get previous	any	2 (the second)	duplicate
get previous	any	2 (the third)	—
get previous	any	1	—

EGL statement (in order)	Key in the indexed record	Key in the file record retrieved by the statement	EGL error value for Java
set (of the form *set record position*)	2	—	duplicate
get next	any	2 (the first)	—
get next	any	2 (the second)	duplicate
get previous	any	1	—
get previous	any	--	endOfFile

EGL statement (in order)	Key in the indexed record	Key in the file record retrieved by the statement	EGL error value for Java
set (of the form *set record position*)	1	--	--
get previous	any	1	--

SQL processing

When a **get previous** statement operates on an SQL record, your code reads the previous row from those selected by an **open** statement, but only if you have specified the scroll option. If you issue a **get previous** statement to retrieve a row that was selected by an **open** statement that also has the forUpdate option, you can do any of these:

- Change the row with an EGL **replace** statement
- Remove the row with an EGL **delete** statement

- Change or remove the row with an EGL **execute** statement

An SQL FETCH statement represents the EGL **get previous** statement in the generated code. The format of the generated SQL statement cannot be changed, except to set the INTO clause.

If you issue a **get previous** statement that attempts to access a row that is previous to the first selected row, the EGL run time acts as follows:

- Does not copy data from the result set
- Leaves the cursor open, with the cursor position unchanged
- Sets the SQL record (if any) to **noRecordFound**

In general, if an error occurs and processing continues, the cursor remains open, with the cursor position unchanged.

Finally, when you specify SQL COMMIT or sysLib.commit, your code retains position in the cursor that was declared in the **open** statement, but only if you use the hold option in the **open** statement.

Related concepts
"Record types and properties" on page 126

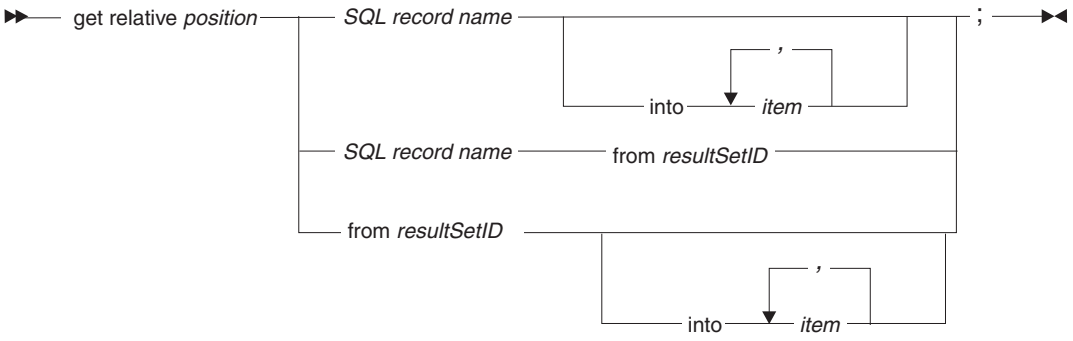
Related tasks
"Syntax diagram for EGL statements and commands" on page 733

Related reference
"add" on page 544
"close" on page 551
"delete" on page 554
"Exception handling" on page 89
"execute" on page 557
"get" on page 567
"get next" on page 579
"I/O error values" on page 522
"open" on page 598
"prepare" on page 611
"EGL statements" on page 83
"replace" on page 613
"set" on page 617

get relative

The EGL **get relative** statement reads a numerically specified row in a relational-database result set. The row is identified in relation to the cursor position in the result set.

You can use this statement only if you specified the scroll option in the related **open** statement. The scroll option is available only if you are generating output in Java.



position
An integer item or literal.

If the value of *position* is positive, the position is an increment to the current numeric position in the result set. Specifying **get relative 2** when the cursor is on the first row, for example, retrieves the third row; and specifying **get relative 1** is equivalent to specifying **get next**.

If the value of *position* is negative, the position is a decrement to the current numeric position in the result set. Specifying **get relative -2** when the cursor is on the third row, for example, retrieves the first row; and specifying **get relative -1** is equivalent to specifying **get previous**.

A value of zero for *position* retrieves the row at the cursor position already in effect and is equivalent to specifying **get current**.

record name
Name of an SQL record.

from *resultSetID*
An ID that ties the **get relative** statement to an **open** statement run earlier in the same program. For details, see *resultSetID*.

into
Begins an EGL into clause, which lists the items that receive values from a relational-database table.

item
An item that receives the value of a particular column. Do *not* precede the item name with a colon (:).

If you issue a **get relative** statement to retrieve a row that was selected by an **open** statement that has the forUpdate option, you can do any of these:

- Change the row with an EGL **replace** statement
- Remove the row with an EGL **delete** statement
- Change or remove the row with an EGL **execute** statement

An SQL FETCH statement represents the EGL **get relative** statement in the generated code. The format of the generated SQL statement cannot be changed, except to set the INTO clause.

If you issue a **get relative** statement that attempts to access a row that is not in the result set, the EGL run time acts as follows:

- Does not copy data from the result set

- Leaves the cursor open with the cursor position unchanged
- Sets the SQL record (if any) to **noRecordFound**

In general, if an error occurs and processing continues, the cursor remains open with the cursor position unchanged.

Finally, when you specify SQL COMMIT or sysLib.commit, your code retains position in the cursor that was declared in the **open** statement, but only if you use the hold option in the **open** statement.

Related concepts
"resultSetID" on page 722
"SQL support" on page 213

Related tasks
"Syntax diagram for EGL statements and commands" on page 733

Related reference
"delete" on page 554
"Exception handling" on page 89
"execute" on page 557
"get" on page 567
"get absolute" on page 573
"get current" on page 575
"get first" on page 576
"get last" on page 578
"get next" on page 579
"get previous" on page 584
"EGL statements" on page 83
"open" on page 598
"replace" on page 613

goTo

The EGL **goTo** statement causes processing to continue at a specified label, which must be in the same function as the statement and outside of a block.

label
A series of characters that are displayed elsewhere in the function, outside of any blocks, including these:

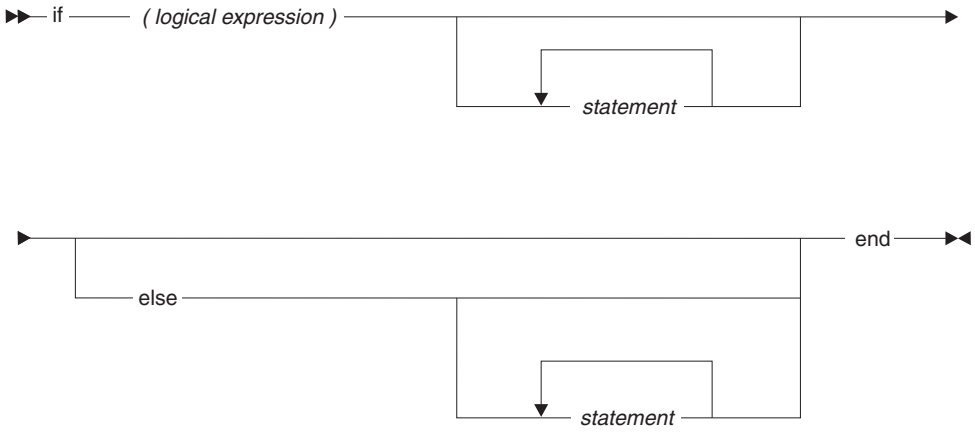
- if
- else
- when (in a **case** statement)
- while
- try

When displayed at the location where processing continues, the label is followed by colon. For details on valid characters for the label, see *Naming conventions*.

Related reference
"Naming conventions" on page 652

if, else

The EGL keyword **if** marks the start of a set of statements (if any) that run only if a logical expression resolves to true. The optional keyword **else** marks the start of an alternative set of statements (if any) that run only if the logical expression resolves to false. The keyword *end* marks the close of the *if* statement.



logical expression
An expression (a series of operands and operators) that evaluates to true or false

statement
One or more EGL statements

You may nest **if** and other end-terminated statements to any level. Each **end** keyword refers to the most recent statement that was not ended and that begins with one of these keywords:

- **if**
- **case**
- **try**
- **while**

None of those statements is followed by a semicolon.

An example is as follows:

```
if (userRequest == "U")
  try
    update myRecord;
    onException
      myErrorHandler(12);  // ends program
  end
  try
    myRecord.myItem=25;
    replace record1;
    onException
      myErrorHandler(16);
  end
else
  try
    add record2;
```

```
    onException
      myErrorHandler(18);  // ends program
  end
  if (sysVar.systemType is WIN)
    myFunction01();
  else
    myFunction02();
  end
end
```

Related tasks
"Syntax diagram for EGL statements and commands" on page 733

Related reference
"Logical expressions" on page 484
"EGL statements" on page 83

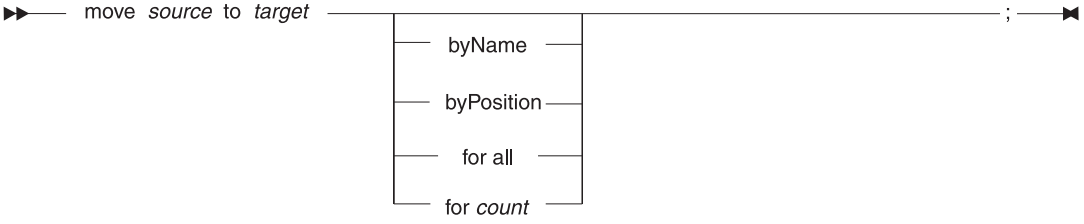
move

The EGL **move** statement copies data in any of three ways. The first option copies data byte by byte; the second (called *by name*) copies data from the named fields in one structure to the same-named fields in another; and the third (called *by position*) copies data from each field in one structure to the field at the equivalent position in another.

The following general rules apply:

- If the source value is one of these, the default is to copy data byte by byte--
  - A primitive variable
  - A field that is in a fixed structure
  - A literal
  - A constant

  Otherwise, the default is to copy data by name.
- Moves are checked for field-to-field compatibility. The rules for truncation, padding, and type conversion are the same as those detailed for the **assignment** statement; however, the overall behavior of the **move** statement is different from that of the **assignment** statement.
- When you are working with dynamic arrays, the last element is determined by the array's current size. The **move** statement never adds an element to an array; to expand a dynamic array, use the array-specific functions appendElement or appendAll, as described in *Arrays*.



The statement is best understood by reference to the following categories:

byName

When you specify **byName**, data is written from each field in the source to a same-named field in the target. The operation occurs in the order in which the fields are in the source.

Source and target can be as follows:

source

One of these:

- A dynamic array of fixed records; but the array is valid only if the target is not a record
- A record
- A fixed record
- A structure field with a substructure
- A structure-field array with a substructure; but this array is valid only if the target is not a record
- A dataTable
- A form

A fixed-structure field whose name is an asterisk (*) is not available as a source field, but any named fields in a substructure of that field are available.

target

One of these:

- A dynamic array of fixed records; but this array is valid only if the source is not a record
- A record
- A fixed record
- A structure field with a substructure
- A structure-field array with a substructure; but this array is valid only if the source is not a record
- A dataTable
- A form
-

An example statement is as follows:

```
move myRecord01 to myRecord02 byName;
```

The operation is not valid in any of these cases:

- Two or more fields in the source have the same name;
- Two or more fields in the destination have the same name;
- The source field is either a multidimensional structure-field array or a one-dimensional structure-field array whose container is an array; or
- The target field is either a multidimensional structure-field array or a one-dimensional structure-field array whose container is an array.

The operation works as follows:

- In a simple case, the source is a fixed structure but is not itself an array element, and the same is true of the target. These rules apply--
  - If no arrays are involved, the value of each subordinate field in the source structure is copied to the same-named field in the target structure.

– If an array of structure fields is being copied to an array of structure fields, the operation is treated as a *move for all*:
  - The elements of the source field are copied to successive elements of the target field
  - If the source array has fewer elements than the target array, processing stops when the last element of the source array is copied

- In another case, the source or target is a record. The fields of the source are assigned to the same-named fields in the target.
- A less simple case is best introduced by example. The source is an array of 10 fixed records, each of which includes these structure fields:

```
10 empnum  CHAR(3);
10 empname CHAR(20);
```

  The target is a fixed structure that includes these structure fields:

```
10 empnum CHAR(3)[10];
10 empname CHAR(20)[10];
```

  The operation copies the value of field empnum in the first fixed record to the first element of the structure-field array empnum; copies the value of field empname in the first fixed record to the first element of the structure-field array empname; and does a similar operation for each fixed record in the source array.

  The equivalent operation occurs if the source is a single fixed record that has a substructure like this:

```
10 mySubStructure[10]
  15 empnum  CHAR(3);
  15 empname CHAR(20);
```

- Finally, consider the case in which the source is a fixed record that includes these structure fields:

```
10 empnum  CHAR(3);
10 empname CHAR(20)[10];
```

  The target is a form, fixed record, or structure field that has the following substructure:

```
10 empnum char(3)[10];
10 empname char(20);
```

  The value of field empnum is copied from the source to the first element of empnum in the target; and the value of the first element of empname is copied from the source to the field empname in the target.

byPosition

The purpose of **byPosition** is to copy data from each field in one structure to the field at the equivalent position in another.

Source and target can be as follows:

source

One of these:

- A dynamic array of fixed records; but the array is valid only if the target is not a record
- A record
- A fixed record
- A structure field with a substructure
- A structure-field array with a substructure; but this array is valid only if the target is not a record
- A dataTable

target

One of these:

- A dynamic array of fixed records; but this array is valid only if the source is not a record
- A record
- A fixed record
- A structure field with a substructure
- A structure-field array with a substructure; but this array is valid only if the source is not a record
- A dataTable

When you move data between a record and a fixed structure, only the top-level fields of the fixed structure are considered. When you move data between two fixed structures, only the lowest-level (leaf) fields of either structure are considered.

The operation is not valid if the source or target field is a multidimensional structure-field array, or a one-dimensional structure field array whose container is an array.

The operation works as follows:

- In a simple case, the source is a fixed structure but is not itself an array element, and the same is true of the target. These rules apply--
  - If no arrays are involved, the value of each leaf field in the source structure is copied to the leaf field in the target structure at the corresponding position.
  - If an array of structure fields is being copied to an array of structure fields, the operation is treated as a *move for all*:
    - The elements of the source field are copied to successive elements of the target field
    - If the source array has fewer elements than the target array, processing stops when the last element of the source array is copied
- In another case, the source or target is a record. The top-level or leaf fields of the source (depending on the source type) are assigned to the top-level or leaf fields in the target (depending on the target type).
- A less simple case is best introduced by example. The source is an array of 10 fixed records, each of which includes these structure fields:

  ```
  10 empnum  CHAR(3);
  10 empname CHAR(20);
  ```

  The target is a fixed structure that includes these structure fields:

  ```
  10 empnum CHAR(3)[10];
  10 empname CHAR(20)[10];
  ```

  The operation copies the value of field empnum in the first fixed record to the first element of the structure-field array empnum; copies the value of field empname in the first fixed record to the first element of the structure-field array empname; and does a similar operation for each fixed record in the source array.

  The equivalent operation occurs if the source is a single fixed record that has a substructure like this:

  ```
  10 mySubStructure[10]
    15 empnum  CHAR(3);
    15 empname CHAR(20);
  ```

- Finally, consider the case in which the source is a fixed record that includes these structure fields:

  ```
  10 empnum  CHAR(3);
  10 empname CHAR(20)[10];
  ```

  The target is a form, fixed record, or structure field that has the following substructure:

  ```
  10 empnum char(3)[10];
  10 empname char(20);
  ```

  The value of field empnum is copied from the source to the first element of empnum in the target; and the value of the first element of empname is copied from the source to the field empname in the target.

for all

The purpose of **for all** is to assign values to all elements in a target array.

Source and target can be as follows:

source

One of these:

- A dynamic array of records, fixed records, or primitive variables
- A record
- A fixed record
- A structure field with or without a substructure
- A structure-field array with or without a substructure
- A primitive variable
- A literal or constant

target

One of these:

- A dynamic array of records, fixed records, or primitive variables
- A structure-field array with or without a substructure
- An element of a dynamic or structure-field array

The **move** statement in this case is equivalent to multiple **assignment** statements, one per target array element, and an error occurs if an attempted assignment is not valid. For details on validity, see *Assignments*.

If a source or target element has a fixed structure, the **move** statement treats that structure as a field of type CHAR unless the top level of the structure specifies a different primitive type. When **for all** is in use, the **move** statement gives no consideration to substructure.

If the source is an element of an array, the source is treated as an array in which the specified element is the first element, and previous elements are ignored.

If the source is an array or an element of an array, each successive element of the source array is copied to the sequentially next element of the target array. Either the target array or the source array can be longer, and the operation ends when data is copied from the last element having a matching element in the other array.

If the source is neither an array nor an element of an array, the operation uses the source value to initialize every element of the target array.

for *count*

The purpose of **for** *count* is to assign values to a sequential subset of elements in a target array. Examples are as follows:

- The next statement moves ″abc″ to elements 7, 8, and 9 in target:

```
move "abc" to target[7] for 3
```

- The next statement moves elements 2, 3, and 4 from source into elements 7, 8, and 9 in target:

```
move source[2] to target[7] for 3
```

The operation works as follows:

- If the source is neither an array nor an element of an array, the operation uses the source value to initialize elements of the target array.
- If the source an array, the first element of that array is the first in a set of elements to be copied. If the source is an element of an array, that element is the first in a set of elements to be copied.
- If the target is an array, the first element of that array is the first in a set of elements to receive data. If the target is an element of an array, that element is the first in a set of elements that receives data.

The *count* value indicates how many target elements are to receive data. The value can be any of these:

- An integer literal
- A variable that resolves to an integer
- A numeric expression, but not a function invocation

The **move** statement is equivalent to multiple **assignment** statements, one per target array element, and an error occurs if an attempted assignment is not valid. For details on validity, see *Assignments*.

If a source or target element has an internal structure, the **move** statement treats that structure as a field of type CHAR unless the top level of that structure specifies a different primitive type. When **for** *count* is in use, the **move** statement gives no consideration to substructure.

When the source and target are both arrays, either the target array or the source array can be longer, and the operation ends after the first of two events occurs:

- Data is copied between the last elements for which the operation is requested; or
- Data is copied from the last element having a matching element in the other array.

When the source is not an array, the operation ends after the first of two events occurs:

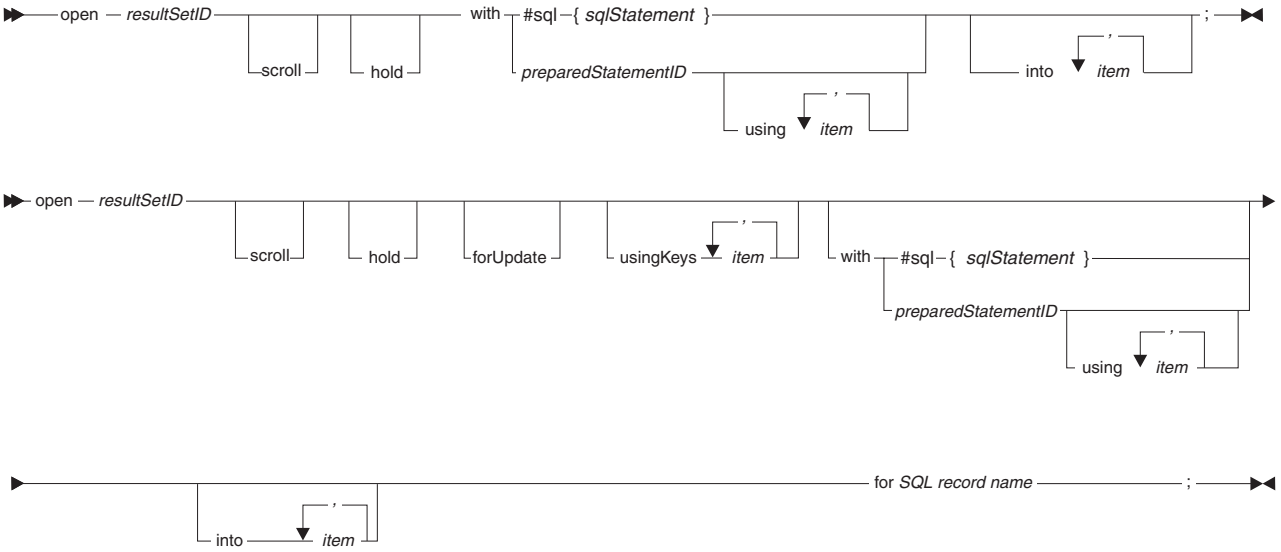
- Data is copied to the last element for which the operation is requested; or
- Data is copied to the last element in the array.

If the source is a record array (or an element of one), the target must be a record array. If the source is a primitive-variable array (or an element of one), the target must be either a primitive-variable array or a structure-field array. If the source is a structure-field array (or an element of one), the target must be either a primitive-variable array or a structure-field array.

Related reference
"Arrays" on page 69
"Assignments" on page 352

open

The EGL **open** statement selects a set of rows from a relational database for later retrieval with **get next** statements. The **open** statement may operate on a cursor or on a called procedure.



resultSetID
ID that ties the open statement to later **get next**, **replace**, **delete**, and **close** statements. For details, see*resultSetID*.

scroll

Option that lets you move through a result set in various ways. The statement **get next** is always available to you, but use of **scroll** allows you to use the following statements too:

- **get absolute**
- **get current**
- **get first**
- **get last**
- **get previous**
- **get relative**

The **scroll** option is available only if you are generating output in Java.

hold
Maintains position in a result set when a commit occurs.

Note: The **hold** option is available for Java programs only if the JDBC driver supports JDBC 3.0 or higher. The option is available for COBOL programs.

The **hold** option is appropriate in the following case:

- You are using the EGL **open** statement to open a cursor rather than a stored procedure; and

- You want to commit changes periodically without losing your position in the result set; and
- Your database management system supports use of the WITH HOLD option in the SQL cursor declaration.

You code might do as follows, for example:

1. Declare and open a cursor by running an EGL **open** statement
2. Fetch a row by running an EGL **get next** statement
3. Do the following in a loop--
   a. Process the data in some way
   b. Update the row by running an EGL **replace** statement
   c. Commit changes by running the system function sysLib.commit
   d. Fetch another row by running an EGL **get next** statement

If you do not specify **hold**, the first run of step 3d fails because the cursor is no longer open.

Cursors for which you specify **hold** are not closed on a commit, but a rollback or database connect closes all cursors.

If you have no need to retain cursor position across a commit, do not specify **hold**.

forUpdate
Option that lets you use a later EGL statement to replace or delete the data that was retrieved from the database.

You cannot specify forUpdate if you are calling a stored procedure to retrieve a result set.

usingKeys ... *item*
Identifies a list of key items that are used to build the key-value component of the WHERE clause in an implicit SQL statement. The implicit SQL statement is used at run time if you do not specify an explicit SQL statement.

If you do not specify a **usingKeys** clause, the key-value component of the implicit statement is based on the SQL record part that is referenced in the **open** statement.

The **usingKeys** information is ignored if you specify an explicit SQL statement.

with #sql{ *sqlStatement* **}**
An explicit SQL SELECT statement, which is optional if you also specify an SQL record. Leave no space between the #sql and the left brace.

into ... *item*
An INTO clause, which identifies the EGL host variables that receive values from the cursor or stored procedure. In a clause like this one (which is outside of a **#sql{ }** block), do not include a semicolon before the name of a host variable.

with *preparedStatementID*
The identifier of an EGL **prepare** statement that prepares an SQL SELECT or CALL statement at run time. The **open** statement runs the SQL SELECT or CALL statement dynamically. For details, see *prepare*.

using ... *item*
A USING clause, which identifies the EGL host variables that are made

available to the prepared SQL SELECT or CALL statement at run time. In a clause like this one (which is outside of a **#sql{ }** block), do not include a semicolon before the name of a host variable.

SQL record name
Name of a record of type SQLRecord. Either the record name or a value for *sqlStatement* is required; if *sqlStatement* is omitted, the SQL SELECT statement is derived from the SQL record.

Examples are as follows (assuming an SQL record called *emp*):

```
open empSetId forUpdate for emp;

open x1 with
  #sql{
    select empnum, empname, empphone
    from employee
    where empnum >= :empnum
    for update of empname, empphone
  }

open x2 with
  #sql{
    select empname, empphone
    from employee
    where empnum = :empnum
  }
for emp;

open x3 with
  #sql{
    call aResultSetStoredProc(:argumentItem)
  }
```

Default processing

The effect of an open statement is as follows by default, when you specify an SQL record:

- The open statement makes a set of rows available. Each column in the selected rows is associated with a structure item, and except for the columns that are associated with a read-only structure item, all the columns are available for subsequent update by an EGL replace statement.
- If you declare only one key item for the SQL record, the open statement selects all rows that fulfill the record-specific **default select condition**, so long as the value in the SQL table key column is greater than or equal to the value in the key item of the SQL record.
- If multiple keys are declared for the SQL record, the record-specific **default select condition** is the only search criterion, and the **open** statement retrieves all rows that meet that criterion.
- If you specify neither a record key nor a default selection condition, the **open** statement selects all rows in the table.
- The selected rows are not sorted.

The EGL **open** statement is represented in the generated code by a cursor declaration that includes an SQL SELECT or an SQL SELECT FOR UPDATE statement. The following is true by default:

- The FOR UPDATE clause (if any) does not include structure items that are read only
- The SQL SELECT statement for a particular record is similar to the following statement:

```
SELECT column01,
       column02, ...
       columnNN
INTO   :recordItem01,
       :recordItem02, ...
       :recordItemNN
FROM   tableName
WHERE  keyColumn01 = :keyItem01
FOR UPDATE OF
       column01,
       column02, ...
       columnNN
```

You may override the default by specifying an SQL statement in the EGL **open** statement.

Error conditions

Various conditions are not valid, including these:

- You include an SQL statement that lacks a clause required for SELECT; the required clauses are SELECT, FROM, and (if you specify **forUpdate**) FOR UPDATE OF
- Your SQL record is associated with a column that either does not exist at run time or is incompatible with the related structure item
- You specify the option **forUpdate**, and your code tries to run an **open** statement against either of the following SQL records:
  - An SQL record whose only structure items are read only; or
  - An SQL record that is related to more than one SQL table.

A problem also arises in the following case:

1. You customize an EGL **open** statement for update, but fail to indicate that a particular SQL table column is available for update; and
2. The **replace** statement that is related to the **open** statement tries to revise the column.

You can solve this problem in any of these ways:

- When you customize the EGL **open** statement, include the column name in the SQL SELECT statement, FOR UPDATE OF clause; or
- When you customize the EGL **replace** statement, eliminate reference to the column in the SQL UPDATE statement, SET clause; or
- Accept the defaults for both the **open** and **replace** statements.

Related concepts
"Record types and properties" on page 126
"SQL support" on page 213
"resultSetID" on page 722
"References to parts" on page 20

Related tasks
"Syntax diagram for EGL statements and commands" on page 733

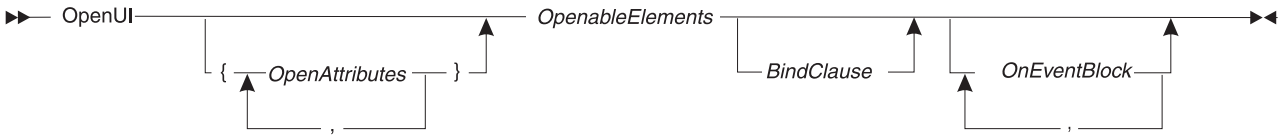
Related reference
"add" on page 544
"close" on page 551
"delete" on page 554
"EGL statements" on page 83

"Exception handling" on page 89
"execute" on page 557
"get" on page 567
"get next" on page 579
"get previous" on page 584
"I/O error values" on page 522
"prepare" on page 611
"replace" on page 613
"SQL item properties" on page 63
"terminalID" on page 913

openUI

The **OpenUI** statement allows the user to interact with a program whose interface is based on consoleUI. The statement defines user and program events and specifies how to respond to each.

The syntax of the OpenUI statement is as follows:



OpenAttributes

OpenAttributes defines a set of property-and-value pairs, each separated from the next by a comma, as in this example:

```
allowAppend = yes, allowDelete = no
```

Each of the properties affects the user interaction, and some overwrite a property of the consoleUI variable referenced in *OpenableElements*. The properties are as follows:

allowAppend

Specifies whether the user can insert data at the end of on on-screen arrayDictionary; if *yes*, the implications are as follows:

- The user inserts a row of data by moving the cursor to the arrayDictionary line which follows the last line that includes data
- The user's action appends an element to the dynamic array that is bound to that arrayDictionary

For variable type: *ArrayDictionary*

Property type: *Boolean*

Example: *allowAppend = no*

Default: *yes; but the default is no if the openUI property* ***displayOnly*** *is set to yes*

allowDelete

Specifies whether the user can delete a row from an on-screen arrayDictionary; if *yes*, the implications are as follows:

- The user deletes a row by moving the cursor to that row and pressing the key referenced in **ConsoleLib.key_delete**.
- The user's action deletes the related element in the dynamic array that is bound to that arrayDictionary.

  For variable type: *ArrayDictionary*

  Property type: *Boolean*

  Example: *allowDelete = no*

  Default: *yes; but the default is no if the openUI property* ***displayOnly*** *is set to yes*

allowInsert

Specifies whether the user can insert a row into an on-screen arrayDictionary; if *yes*, the implications are as follows:

- The user inserts the row by moving the cursor to an existing row and pressing the key referenced in **ConsoleLib.key_insert**.
- The new row precedes the row that shows the cursor
- The user's action inserts an element to the dynamic array that is bound to that arrayDictionary.

  For variable type: *ArrayDictionary*

  Property type: *Boolean*

  Example: *allowInsert = no*

  Default: *yes; but the default is no if the openUI property* ***displayOnly*** *is set to yes*

bindingByName

Indicates how to bind a series of variables to a series of ConsoleFields; specifically, whether to match each variable name with a ConsoleField name. The variable name is listed in *BindClause*, and the ConsoleField name is the value in the ConsoleField name field.

For variable type: *ConsoleForm, ConsoleField, or Dictionary; but not arrayDictionary*

Property type: *Boolean*

Example: *bindByName = yes*

Default: *no*

Values are as follows:

no (the default)
Match variables and ConsoleFields by position:

- The position of each variable in the list; and
- The position of each ConsoleField in the consoleForm.

Whether consoleFields are listed explicitly in the openUI statement or are listed in a dictionary declaration, their order defines the order of consoleFields for the purpose of binding by position. (Their order also defines the tab order for user input, as noted in *ConsoleUI parts and related variables*.)

yes
Match variables and ConsoleFields by name.

If a consoleField is listed or is in a dictionary declaration when no matching variable is in the binding list, the user's input to the consoleField is ignored. Similarly, a binding variable that does not match any field is ignored.

At least one consoleField and variable must be bound together at run time; otherwise, an error occurs.

color

Specifies the color of the text in the ConsoleFields. The value overrides the color specified in the ConsoleField declaration.

For variable type: *ConsoleForm, ConsoleField, ArrayDictionary, or Dictionary*

Property type: *ColorKind*

Example: *color = red*

Default: *white*

Values are as follows:

defaultColor or white (the default)
White

black
Black

blue
Blue

cyan
Cyan

green
Green

magenta
Magenta

red
Red

yellow
Yellow

currentArrayCount

Specifies the number of elements that are available in the dynamic array to which the on-screen arrayDictionary is bound. If you do not specify this value, all the elements are available for use in the arrayDictionary.

For variable type: *ArrayDictionary*

Property type: *INT*

Example: *currentArrayCount = 4*

Default: *none*

displayOnly

Specifies whether consoleFields are displayed for viewing only. If *yes*, the user cannot modify the data, which is protected from update.

For variable type: *ArrayDictionary, Dictionary, ConsoleField, ConsoleForm*

Property type: *Boolean*

Example: *displayOnly = yes*

Default: *no*

help
Specifies the text to display when the user presses the key identified in **ConsoleLib.key_help**.

This help text is for the **openUI** command. In some cases, the text associated with the key is more context specific. For instance, each option in a menu can have its own help message.

For variable type: *ConsoleForm, ConsoleField, ArrayDictionary, or Dictionary*

Primitive type: *String*

Example: help = *"Update the value"*

Default: *Empty string*

helpKey
Specifies an access key for searching the resource bundle that contains text for display when the following situation is in effect:

- The cursor is in a ConsoleUI variable (such as ConsoleForm) that is identified in *OpenableElements*; and
- The user presses the key identified in **ConsoleLib.key_help**.

If you specify both **help** and **helpKey**, **help** is used.

For variable type: *ConsoleForm, ConsoleField, ArrayDictionary, or Dictionary*

Property type: *String*

Example: *helpKey = "myKey"*

Default: *Empty string*

The resource bundle is identified by the system variable **ConsoleLib.messageResource**, as described in *messageResource*.

highlight
Specifies the special effects (if any) that are used when displaying the ConsoleField. The value overrides the equivalent value specified in the ConsoleField declaration.

For variable type: *ConsoleForm, ConsoleField, ArrayDictionary, or Dictionary*

Property type: *HighlightKind[]*

Example: *highlight = [reverse, underline]*

Default: *[noHighLight]*

Values are as follows:

noHighlight (the default)
Causes no special effect. Use of this value overrides any other.

blink
Has no effect.

reverse
Reverses the text and background colors so that (for example) if the display has a black background with white letters, the background becomes white and the text becomes black.

underline
Places an underline under the affected areas. The color of the underline is the color of the text, even if the value **reverse** is also specified.

intensity
Specifies the strength of the displayed font.

For variable type: *ConsoleField, ConsoleForm, ArrayDictionary, or Dictionary*

Property type: *IntensityKind[]*

Example: *intensity = [bold]*

Default: *[normalIntensity]*

Values are as follows:

normalIntensity (the default)
Causes no special effect. Use of this value overrides any other.

bold
Causes the text to appear in boldface.

dim
Causes the text to appear with a lessened intensity, as appropriate when an input field is disabled.

invisible
Removes any indication that the ConsoleField is on the form.

isConstruct

Specifies whether the purpose of the **openUI** statement is to create selection criteria for use in an SQL statement such as SELECT.

For variable type: *ConsoleField, ConsoleForm, Dictionary*

Property type: *Boolean*

Example: *isConstruct = no*

Default: *yes*

Values are as follows:

no (the default)
Each ConsoleField is bound to a variable, as usual.

yes

The **openUI** statement must be bound to a single variable of a character type. That variable does not provide initial values for the ConsoleFields, but does receive the user's input, which is formatted for use in an SQL WHERE clause.

maxArrayCount

Specifies the maximum number of rows that can be in the dynamic array bound to the on-screen arrayDictionary. After the maximum is reached, the user is unable to insert more rows.

For variable type: *ArrayDictionary*

Property type: *INT*

Example: *maxArrayCount = 20*

Default: *none*

setInitial

Specifies whether the initial value of a ConsoleField (as defined in the consoleForm declaration) is displayed until the user modifies that value. (You specify the initial value by setting the **initialValue** field of ConsoleField.)

For variable type: *ConsoleField, ConsoleForm, Dictionary, ArrayDictionary*

Property type: *Boolean*

Example: *setInitial = yes*

Default: *no*

If the value of **setInitial** is **no**, the values of the bound variables are fetched and displayed initially.

OpenableElements
The ConsoleUI variables on which the **openUI** statement can act:

- ConsoleForm
- A consoleField or one of these:
  - A list of consoleFields, each separated from the next by a comma
  - A dictionary that is declared in a consoleForm and refers to a set of consoleFields in that consoleForm
  - An arrayDictionary that is declared in a consoleForm and refers to a set of consoleField arrays in that consoleForm.
- Menu
- Prompt
- Window

BindClause
The list of primitive variables, records, or arrays that are bound to the ConsoleUI variables. Characteristics of the binding variables depend on the characteristics of the consoleUI variable on which the **openUI** statement acts:

- For a consoleField, you can specify a primitive variable.
- For an on-screen arrayDictionary, you can specify an array of records, one element per row in the arrayDictionary; and if each row in the arrayDictionary represents a single value, you can specify an array of primitive variables.
- For a dictionary or a list of consoleFields, you can specify a list of primitive variables. Alternatively, you can specify an array of records, with each element containing a series of fields that are bound to the consoleFields. This alternative is equivalent to binding a dynamic array with an on-screen arrayDictionary that has only one row; you can append, insert, or delete a record to change the dynamic array, and in any case only one record is displayed at a time.
- For a prompt, you can specify a primitive field that receives the user's response.

For details on binding, see the section on *OnEventBlock* (later), as well as *ConsoleUI parts and related variables*.

OnEventBlock
An *event block* is a programming structure that includes 0 to many *event handlers*, each of which contains the code that you've written for responding to a particular event. An event handler begins with an OnEvent header:

```
OnEvent(eventKind: eventQualifiers)
```

eventKind
One of several events. Valid values are described in "Event types" on page 608.

eventQualifier
Data that further defines the event. Such data might be the ConsoleField entered or the keystroke pressed.

The EGL statements that respond to a given event are between the OnEvent header and the next OnEvent header (if any), as shown in a later example.

The user continually interacts with the program, and the program runs an event handler when the event occurs that is associated with that event handler. If the purpose of the **openUI** statement is to display a prompt, however, the user-program interaction is less like a loop:

1. An event handler (potentially one of several) traps a user keystroke and responds
2. The **openUI** statement ends

No event block is available for a window.

Consider the following example for guiding the interaction between the user and a ConsoleForm:

```
openUI {bindingByName=yes}
  activeForm
  bind firstName, lastName, ID
  OnEvent(AFTER_FIELD:"ID")
    if (employeeID == 700)
      firstName = "Angela";
      lastName = "Smith";
    end
end
```

That code acts as follows:

- Opens the *active ConsoleForm* (which is the consoleForm that was most recently displayed in the active window);
- Binds a set of primitive variables to each of the ConsoleFields; and
- Specifies that after the user types a value in *employeeID* and leaves that ConsoleField, EGL places strings in two other variables.

Consider these details about the preceding example:

- The cursor starts in the first of the listed consoleFields; but should start in the ID consoleField so that the user's input in the other consoleFields is not wiped out by the event handler.
- The event handler updates the variables that are bound to the firstName and lastName consoleFields but does not cause those values to be displayed until the cursor enters those fields. You might want to display the values earlier.

You can end an **openUI** statement by issuing an **exit** statement of the form **exit openUI**.

Event types

ConsoleUI supports the following events:

BEFORE_OPENUI
EGL run time begins to run the **OpenUI** statement. This event is available for all ConsoleUI variables other than those based on Window.

AFTER_OPENUI
EGL run time is about to stop running the **OpenUI** statement. This event is available for all ConsoleUI variables other than those based on Window.

ON_KEY:(*ListOfStrings***)**
The user has pressed a specific key, as indicated by a string such as ″ESC″, ″F2″, or ″CONTROL_W″. You can identify multiple keys by separating one string from the next, as in this example:

```
ON_KEY:("a", "ESC")
```

This event is available for all ConsoleUI variables other than those based on a Window.

BEFORE_FIELD:(*ListOfStrings***)**
The user has moved the cursor into the specified ConsoleField, as indicated by a string that matches the value of the ConsoleField name field. You can identify multiple consoleFields in the same consoleForm by separating one string from the next, as in this example:

```
BEFORE_FIELD:("field01", "field02")
```

AFTER_FIELD:(*ListOfStrings***)**
The user has moved the cursor out of the specified ConsoleField, as indicated by a string that matches the value of the ConsoleField name field. You can identify multiple consoleFields in the same consoleForm by separating one string from the next, as in this example:

```
AFTER_FIELD:("field01", "field02")
```

BEFORE_DELETE
In relation to an on-screen arrayDictionary, the user has pressed the key specified in **ConsoleLib.key_deleteLine**, but EGL run time has not yet deleted the row. The program can invoke **consoleLib.cancelDelete** to avoid deleting the row.

BEFORE_INSERT
In relation to an on-screen arrayDictionary, the user has pressed the key specified in **ConsoleLib.key_insertLine**, but EGL run time has not yet inserted a row. The program can invoke **consoleLib.cancelInsert** to avoid inserting the row.

BEFORE_ROW
In relation to an on-screen arrayDictionary, the user has moved the cursor into a row.

AFTER_DELETE
In relation to an on-screen arrayDictionary, the user has pressed the key specified in **ConsoleLib.key_deleteLine**, and EGL run time has deleted a row.

AFTER_INSERT
In relation to an on-screen arrayDictionary, the user has pressed the key specified in **ConsoleLib.key_insertLine**; EGL run time has inserted a row; and the cursor leaves the row that was inserted.

The user can edit the row before committing changes to a database, as typically happens in the AFTER_INSERT handler.

AFTER_ROW
The user has moved the cursor from a row in an on-screen arrayDictionary.

MENU_ACTION:(*ListOfStrings***)**
The user has selected a menuItem, as indicated by a string that matches the value of the menuItem name field. You can identify multiple menuItems by separating one string from the next, as in this example:

```
MENU_ACTION:("item01", "item02")
```

isConstruct

When the property **isConstruct** = *yes*, the text placed in the variable bound to the **openUI** command is specially formatted, as shown in this example:

1. An openUI statement acts on a ConsoleForm of three ConsoleFields (*employee*, *age*, and *city*), and each field is associated with an SQL table column of the same name.

You associate a consoleField with an SQL table column by setting the consoleField property **SQLColumnName**; and you must set the consoleField property **dataType**, as noted in *ConsoleField Properties and Fields*.

2. The user acts as follows:
   - Leaves the *employee* field blank
   - Types this in the *age* field:

     ```
     > 25
     ```

   - Types this in the *city* field:

     ```
     = 'Sarasota'
     ```

3. When the user leaves the on-screen variable on which the openUI statement acts, the bound variable receives the following content:

   ```
   AGE > 28 AND CITY = 'Sarasota'
   ```

As shown, EGL places the operator AND between each clause that the user provides.

The next table shows valid user input and the resulting clause. The phrase *simple SQL types* refers to SQL types that are neither structured nor LOB-like types.

Symbol	**Definition**	**Supported data types**	**Example**	**Resulting clause (for a character column named *C*)**	**Resulting clause (for a numeric column named *C*)**
=	Equal to	Simple SQL Types	=x, ==x	C = 'x'	C = x
>	Greater than	Simple SQL Types	>x	C > 'x'	C > x
<	Less than	Simple SQL Types	<x	C < 'x'	C < x
>=	Not less than	Simple SQL Types	>=x	C >= 'x'	C >= x
<=	Not greater than	Simple SQL Types	<=x	C <= 'x'	C <= x
<> or !=	Not equal to	Simple SQL Types	<>x or !=x	C != 'x'	C != x
..	Range	Simple SQL Types	x.y or x..y	C BETWEEN 'x' AND 'y'	C BETWEEN x AND y
*	Wildcard for String (as described in next table)	CHAR	*x or x* or *x*	C MATCHES '*x'	not applicable
?	Single character wildcard (as described in next table)	CHAR	?x, x?, ?x?, x??	C MATCHES '?x'	not applicable
|	Logical Or	Simple SQL Types	x|y	C IN ('x', 'y')	C IN (x, y)

The equal sign (=) can mean IS NULL; and the not-equal sign (!= or <>) can mean IS NOT NULL.

A MATCHES clause results from the user's specifying one of the wildcard characters described in the next table.

Symbol	Effect
*	Matches zero or more characters.
?	Matches any single character.
[ ]	Matches any enclosed character.
- (hyphen)	When used between characters inside brackets, a hyphen matches any character in the range between and including the two characters. For example, [a-z] matches any lowercase letter or special character in the lower case range.
^	When used in brackets, an initial caret matches any character not included within the brackets. For example, [^abc] matches any character except a, b, or c.
\	Is the default escape character; the next character is a literal. Allows any of the wildcard characters to be included in the string without having the wildcard effect.
Any other character outside of brackets	Must match exactly.

Related concepts
"Console user interface" on page 165

Related reference
"EGL library ConsoleLib" on page 735
"ConsoleUI parts and related variables" on page 167
"ConsoleUI screen options for UNIX" on page 171
"Text UI program in EGL source format" on page 710

Related tasks
"Creating an interface with consoleUI" on page 166

prepare

The EGL **prepare** statement specifies an SQL PREPARE statement, which optionally includes details that are known only at run time. You run the prepared SQL statement by running an EGL **execute** statement or (if the SQL statement returns a result set) by running an EGL **open** or **get** statement.

preparedStatementID
An identifier that relates the **prepare** statement to the **execute**, **open**, or **get** statement.

stringExpression
A *string expression* that contains a valid SQL statement.

SQL record name
Name of an SQL record. Specifying this name has two benefits:

- The EGL editor provides a dialog to derive an SQL statement based on your specifications
- After the **prepare** statement runs, you can test the record name against an I/O error value to determine whether the statement succeeded, as in the following example:

```
try
  prepare prep01 from
   "insert into " + aTableName +
   "(empnum, empname) " +
   "value ?, ?"
  for empRecord;

onException
  if empRecord is unique
    myErrorHandler(8);
  else
    myErrorHandler(12);
  end
end
```

Another example is as follows:

```
myString =
  "insert into myTable " +  "(empnum, empname) " +
  "value ?, ?";

try
  prepare myStatement
    from myString;
onException
  myErrorHandler(12);      // exits the program
end

try
  execute myStatement
  using :myRecord.empnum,
        :myRecord.empname;
onException
  myErrorHandler(15);
end
```

As shown in the previous examples, the developer can use a question mark (?) where a host variable should appear. Then, the name of the host variable used at run time is placed in the using clause of the **execute**, **open**, or **get** statement that runs the prepared statement.

A **prepare** statement that acts on a row of a result set may include a phrase of the format *where current of resultSetIdentifier*. This technique is valid only in the following situation:

- The phrase is coded in a literal; and
- The result set is open when the **prepare** statement runs.

An example is as follows:

```
prepare prep02 from
  "update myTable " +
  "set empname = ?,  empphone = ?  where current of x1" ;

execute prep02 using empname, empphone ;
freeSQL prep02;
```

For a example of how parentheses affect the use of a plus (+) sign, see *Expressions*.

Related concepts
"References to parts" on page 20
"Record types and properties" on page 126
"SQL support" on page 213

Related reference
"add" on page 544
"close" on page 551
"delete" on page 554
"Exception handling" on page 89
"execute" on page 557
"Expressions" on page 482
"freeSQL" on page 567
"get" on page 567
"get next" on page 579
"get previous" on page 584
"I/O error values" on page 522
"EGL statements" on page 83
"open" on page 598
"replace"
"SQL item properties" on page 63
"Text expressions" on page 492
"Syntax diagram for EGL statements and commands" on page 733

print

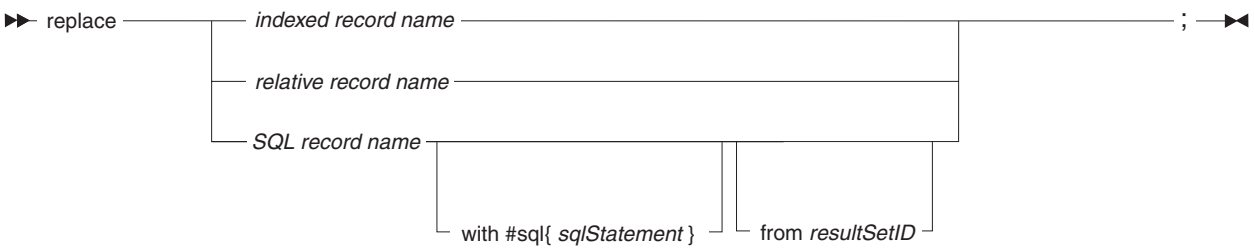
The EGL **print** statement adds a print form to a run-time buffer, as described in *Print forms*.

printFormName
Name of a print form that is visible to the program. For details on visibility, see *References to parts*.

Related concepts
"Print forms" on page 146
"References to parts" on page 20

replace

The EGL **replace** statement puts a changed record into a file or database.



record name
Name of the I/O object: an indexed, relative, or SQL record.

with #sql{ *sqlStatement* **}**
An explicit SQL UPDATE statement. Leave no space between #sql and the left brace.

from *resultSetID*
ID that ties the **replace** statement to a **get** or **open** statement run earlier in the same program. For details, see *resultSetID*.

The following example shows how to read and replace a file record:

```
emp.empnum = 1;         // sets the key in record emp

try
  get emp forUpdate;
onException
  myErrorHandler(8);  // exits the program
end

emp.empname = emp.empname + " Smith";

try
  replace emp;
onException
  myErrorHandler(12);
end
```

Details on the **replace** statement depend on the record type. For details on SQL processing, see *SQL record*.

Indexed or relative record

If you want to replace an indexed or relative record, you must issue a **get** statement for the record with the forUpdate option, then issue the **replace** statement with no intervening I/O operation against the same file. After you invoke the **replace** statement, the effect of the next I/O operation on the same file is as follows:

- If the next I/O operation is a **replace** statement on the same EGL record, the record is changed in the file
- If the next I/O operation is a **delete** statement on the same EGL record, the record in the file is marked for deletion
- If the next I/O operation is a **get** statement on a record in the same file and includes the forUpdate option, a subsequent **replace** or **delete** statement is valid on the newly read file record
- If the next I/O operation is a **get** statement on the same EGL record (with no forUpdate option) or is a **close** statement on the same file, the file record is released without change

For details on the forUpdate option, see *get*.

SQL record

In the case of SQL processing, the EGL **replace** statement results in an SQL UPDATE statement in the generated code.

You must retrieve a row for subsequent replacement, in either of two ways:

- Issue a **get** statement (with the forUpdate option) to retrieve the row; or
- Issue an **open** statement to select a set of rows, then invoke a **get next** statement to retrieve the row of interest.

Error conditions: The following conditions are among those that are not valid when you use a **replace** statement:

- You specify an SQL statement of a type other than UPDATE
- You specify some but not all clauses of an SQL UPDATE statement
- You do not specify a *resultSetID* value when one is necessary; for details, see *resultSetID*
- You specify (or accept) an UPDATE statement that has one of these characteristics--
  - Updates multiple tables
  - Is associated with a column that either does not exist or is incompatible with the related host variable
- You use an SQL record as an I/O object, and all the record items are read only

The following situation also causes an error:

- You customize an EGL **get** statement with the forUpdate option, but fail to indicate that a particular SQL table column is available for update; and
- The **replace** statement that is related to the **get** statement tries to revise the column.

You can solve the previous mismatch in any of these ways:

- When you customize the EGL **get** statement, include the column name in the SQL SELECT statement, FOR UPDATE OF clause; or
- When you customize the EGL **replace** statement, eliminate reference to the column in the SQL UPDATE statement, SET clause; or
- Accept the defaults for both the **get** and **replace** statements.

Implicit SQL statement: By default, the effect of a **replace** statement that writes an SQL record is as follows:

- As a result of the association of record items and SQL table columns in the record declaration, the generated code copies the data from each record item into the related SQL table column
- If you defined a record item to be read only, the value in the column that corresponds to that record item is unaffected

The SQL statement has these characteristics by default:

- The SQL UPDATE statement does not include record items that are read only
- The SQL UPDATE statement for a particular record is similar to the following statement:

```
UPDATE tableName
SET column01 = :recordItem01,
    column02 = :recordItem02,
               .
               .
               .
    columnNN = :recordItemNN    WHERE CURRENT OF cursor
```

Related concepts

"Record types and properties" on page 126
"References to parts" on page 20
"resultSetID" on page 722
"Run unit" on page 721
"SQL support" on page 213

Related tasks
“Syntax diagram for EGL statements and commands” on page 733

Related reference
“add” on page 544
“close” on page 551
“delete” on page 554
“EGL statements” on page 83
“Exception handling” on page 89
“execute” on page 557
“get” on page 567
“get next” on page 579
“get previous” on page 584
“I/O error values” on page 522
“open” on page 598
“prepare” on page 611
“SQL item properties” on page 63
“terminalID” on page 913

return

The EGL **return** statement exits from a function and optionally returns a value to the invoking function.

returnValue
An item, literal, or constant that is compatible with the **returns** specification in the EGL function declaration.

Although an item must correspond in all ways to the **returns** specification, the rules for literals and constants are as follows:

- A numeric literal or constant can be returned only if the primitive type in the **returns** specification is a numeric type
- A literal or constant that includes only single-byte characters can be returned only if the primitive type in the **returns** specification is CHAR or MBCHAR
- A literal or constant that includes only double-byte characters can be returned only if the primitive type in the **returns** specification is DBCHAR
- A literal or constant that includes a combination of single- and double-byte characters can be returned only if the primitive type in the **returns** specification is MBCHAR
- A literal or constant cannot be returned if the primitive type in the **returns** specification is HEX

A function that includes a **returns** specification must terminate with a **return** statement that includes a value. A function that lacks a **returns** specification may terminate with a **return** statement, which must not include a value.

The **return** statement gives control to the first statement that follows invocation of the function, even if the statement is in an **OnException** clause in a try block.

set

The following sections describe the effect of an EGL **set** statement:

- "Effect on a record (or fixed record) as a whole"
- "Effect on a form as a whole" on page 618
- "Effect on a field in any context" on page 620
- "Effect on a field in a text form" on page 620

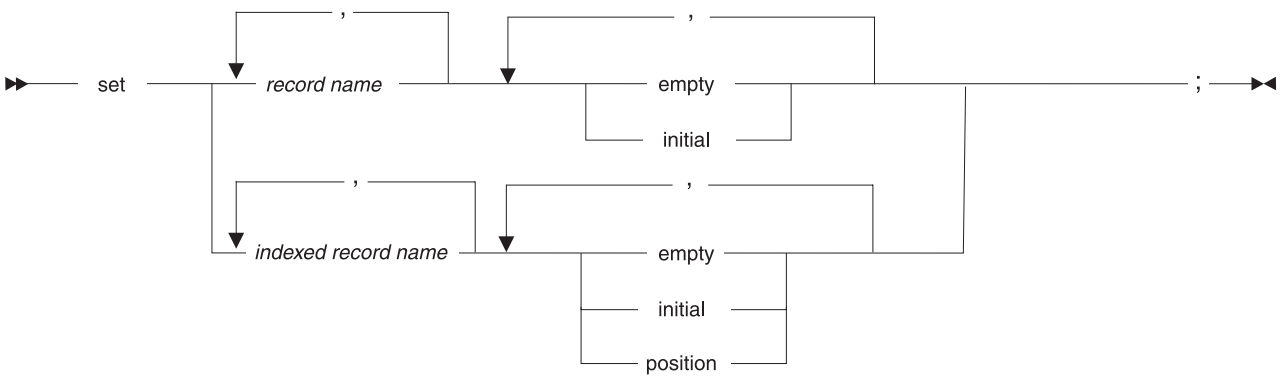
Effect on a record (or fixed record) as a whole

The next table describes the **set** statements that affect a record as a whole, a fixed record as a whole, or an array of either.

Format of set statement	Effect
set record empty	Empties each of the elementary fields. For a record, each subordinate record is emptied, as is each subordinate of those subordinates, and so on. For a fixed record (which may itself be in a record), the elementary fields are at the lowest level of the fixed structure. The effect on each elementary field depends on the primitive type of that field: • For a field of type ANY, the **set** statement initializes the field according to the field's current type; and if the field is of type ANY and has no other type, the **set** statement has no effect • For details on fields of other types, see *Data initialization*
set record initial	Resets the field values to those specified by the **value** property at development time, as is possible for a record or fixed record that is declared in a pageHandler or form. A value set by assignment is never reinstated. If the **value** property has no value or if the record is not in a pageHandler or form, the effect of *set record initial* is the same as the effect of *set record empty*, with one exception: for a field that is of type ANY, the **set** statement removes any type specification other than ANY.
set record position	Establishes position in the VSAM file associated with a fixed record of type indexedRecord, as described later. This set-statement format is not available for an array.

You can combine statement formats by inserting a comma to separate the options. For a given record, the options take effect in the order in which they appear in the **set** statement. Also, you can specify multiple records by inserting a comma to separate one from the next.

The syntax diagram is as follows:



record name
Name of a record or fixed record of any type. You can specify an array.

indexed record name
Name of a fixed record of type indexedRecord. You can specify an array only if you do not include *set record position*.

empty
As described in the previous table.

initial
As described in the previous table.

position
Establishes a file position based on the *set value*, which is the key value in an indexed record. The overall effect depends on the next input or output operation that your code performs against the same indexed record:

- If the next operation is an EGL **get next** statement, that statement reads the first file record that has a key value equal to or greater than the set value. If no such record exists, the result of the **get next** statement is **endOfFile**.
- If the next operation after *set record position* is an EGL **get previous** statement, that statement reads the first file record that has a key value equal to or less than the set value. If no such record exists, the result of **get previous** is **endOfFile**.
- Any other operation after *set record position* resets the file position, and the *set record position* has no effect.

If the set value is filled with hexadecimal FF values, the following is true:

- The *set record position* establishes a file position after the last record in the file
- If the next operation is a **get previous** statement, the last record in the file is retrieved

Effect on a form as a whole

The next table describes the **set** statements that affect a form as a whole.

Format of set statement	Effect
set form alarm	For text forms only; sounds an alarm the next time that a **converse** statement presents the form.

Format of set statement	Effect
set form empty	Empties the value of each field in the form, clearing any content. The effect on a given field depends on the primitive type: • For a field of type ANY, the **set** statement initializes the field according to the field's current type; and if the field is of type ANY and has no other type, the **set** statement has no effect • For details on fields of other types, see *Data initialization*
set form initial	Resets each form field to its originally defined state, as expressed in the form declaration. Changes that were made by the program are canceled. for a field that is of type ANY, the **set** statement removes any type specification other than ANY.
set form initialAttributes	Resets each form field to its originally defined state, as expressed in the form declaration. The content of the field is not affected, neither (in the case of a field of type ANY) is the type affected.

You can combine statement formats by inserting a comma to separate options such as **empty** and **alarm**. Also, you can specify multiple forms by inserting a comma to separate one form from the next.

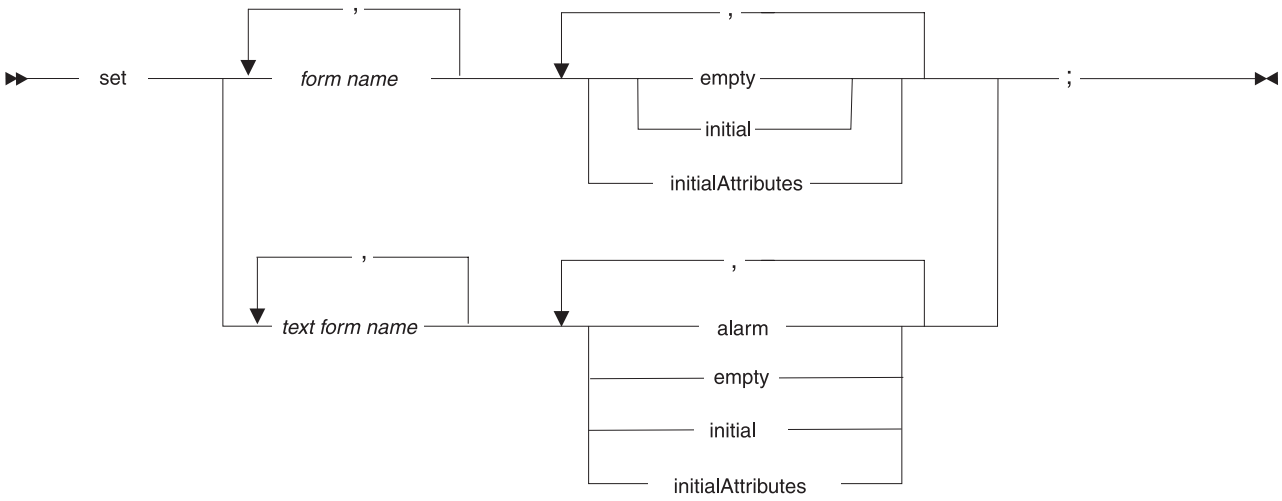
Of the following formats, you can choose one or none:

- *set form empty*
- *set form initial*

Of the following formats, you can choose one, both, or none:

- *set form alarm* (available only for text forms)
- *set form initialAttributes*

The syntax diagram is as follows:



form name
Name of a form of type *text* or *print*, as described in *Form part*.

text form name
Name of a form of type *text*, as described in *Form part*.

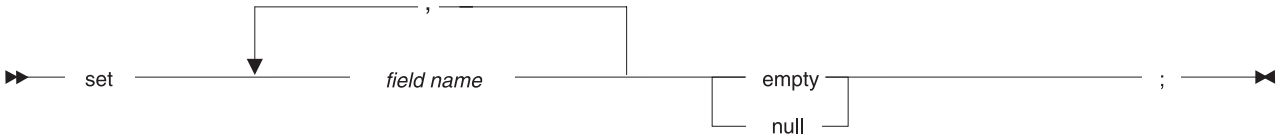
The options are as described in the previous table.

Effect on a field in any context

The next table describes the format of the **set** statement that affects a field in any context.

Format of set statement	Effect
set field empty	Empties the field or (for a fixed field that has a substructure) empties every subordinate, elementary field. The effect depends on the primitive type of a field: • For a field of type ANY, the **set** statement initializes the field according to the field's current type; and if the field is of type ANY and has no other type, the **set** statement has no effect • For details on fields of other types, see *Data initialization*
set field null	Nulls the field, if doing so is valid. For details on when the operation is valid, see *itemsNullable*. For details on null processing in SQL records, see *SQL item properties*.

The syntax diagram is as follows:



field name
Name of the field.

You may select one or the other option, and each is described in the previous table.

Effect on a field in a text form

The next table describes the **set** statements that affect a field or an array of fields in a text form. A given **set** statement can combine options only in a particular set of ways, as described later.

Note: Many of the actions described are dependent on the device where the text form is displayed. It is recommended that you test your output on each of the devices that you are supporting.

Format of set statement	Effect
set field blink	Causes the text to blink repeatedly. This option is available only in COBOL programs.
set field bold	Cause the text to appear in boldface.
set field cursor	Positions the cursor in the specified field. If the field identifies an array and has no occurs value, the cursor is positioned at the first array element by default. If your program runs multiple statements of the format *set field cursor*, the last is in effect when the **converse** statement runs.
set field defaultColor	Sets the field-specific **color** property to *defaultColor*, which means that other conditions determine the displayed color. For details, see *Field-presentation properties*.
set field dim	Causes the field to be appear in lower intensity than normal. Use this effect to deemphasize field contents.
set field empty	Initializes the value of the field, clearing any content. The effect on a given field depends on the primitive type, as described in *Data initialization*.
set field full	Sets an empty, blank, or null field to a series of identical characters before the form is presented: • The character is an asterisk (*) if the field property **fillCharacter** is the following value (which is also the default value for **fillCharacter**): – 0 for fields of type HEX – space for fields of a numeric type – empty string for other fields • If **fillCharacter** is not set as described, the character is identical to the value of **fillCharacter**. The on-form characters are returned to the program only if the modified data tag for the field is set, as described in *Modified data tag and property*. A user who changes the field must remove all the on-field characters to prevent their return to the program. Use of *set field full* has an effect only if the form group is generated with the build descriptor option *setFormItemFull*. A field of type MBCHAR is considered to be empty if it contains all single-byte spaces. In relation to such fields, *set field full* assigns a series of single-byte characters.

Format of set statement	Effect
set field initial	Resets the field to its originally defined state, independent of any changes made by the program
set field initialAttributes	Resets the field to its originally defined state, without using the **value** property (which specifies the current content of the field)
set field invisible	Makes the field text invisible
set field masked	Appropriate for password fields. If the text form is presented by a Java program, an asterisk is displayed instead of any non-blank character that the user types into an input field. If the text form is presented by a COBOL program, this option makes the field text invisible.
set field modified	Sets the modified data tag, as described in *Modified data tag and property*.
set field noHighlight	Eliminates the special effects of blink, reverse, and underline.
set field normal	Resets the fields as described in relation to the following formats: • Set field normalIntensity • Set field unmodified • Set field unprotected For details, see the next table.
set field normalIntensity	Sets the field to be visible, without boldface.
set field protect	Sets the field so that the user cannot overwrite the value in it. See also *set field skip*.
set field reverse	Reverses the text and background colors, so that (for example) if the display has a dark background with light letters, the background becomes light and the text becomes dark.
set field *selectedColor*	Sets the field-specific **color** property to the value you specify. The valid values for *selectedColor* are as follows: • black • blue • green • pink • red • torquoise • white • yellow

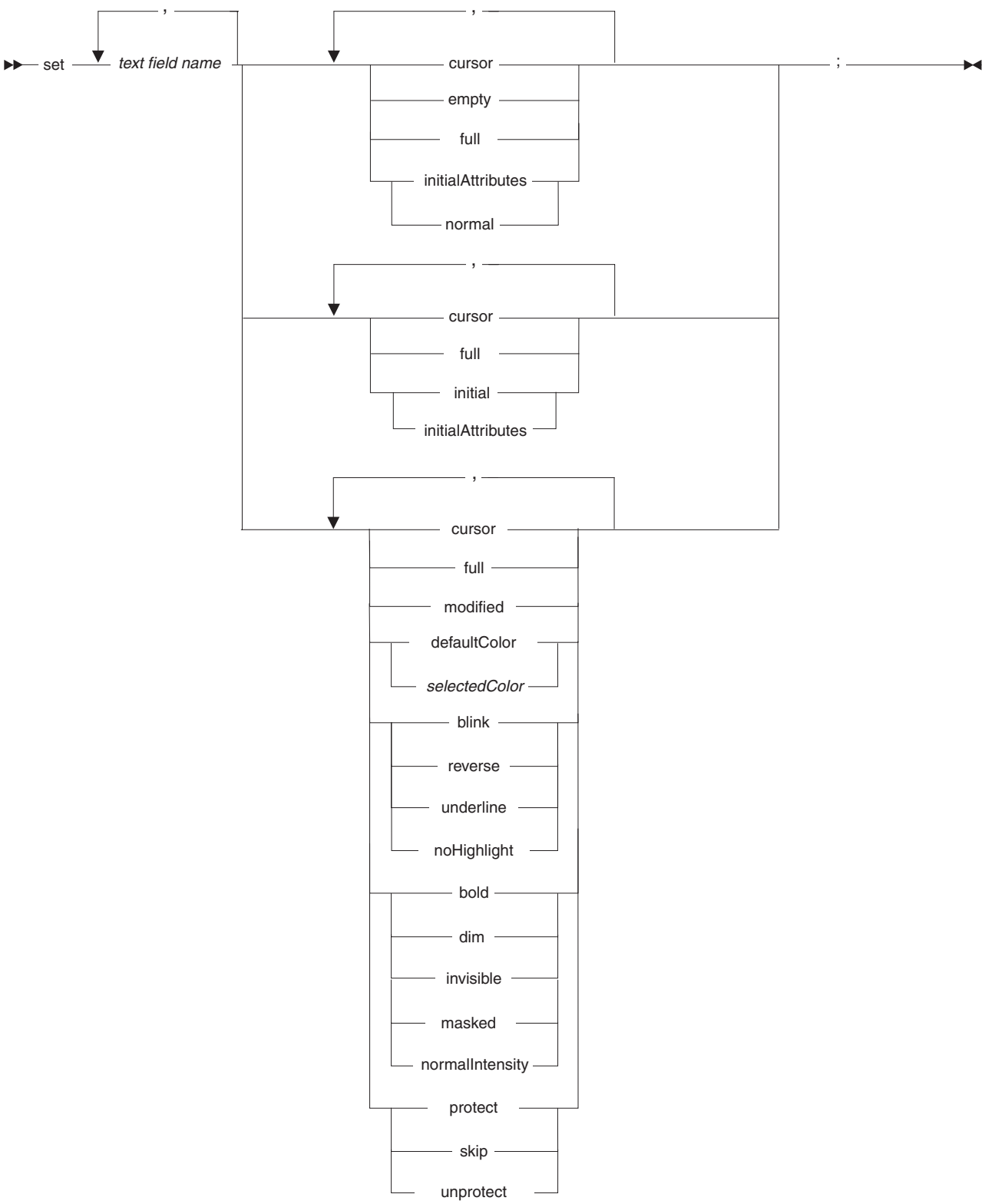
Format of set statement	Effect
set field skip	Sets the field so that the user cannot overwrite the value in it. In addition, the cursor skips the field in either of these cases: • The user is working on the previous field in the tab order and either presses **Tab** or fills that previous field with content; or • The user is working on the next field in the tab order and presses **Shift Tab**.
set field underline	Places an underline at the bottom of the field.
set field unprotect	Sets the field so that the user can overwrite the value in it.

You can combine statement formats, inserting a comma to separate options such as **cursor** and **full**, in any of three ways:

1. You can construct a **set** statement as follows--
   - Choose one or none of these field-attribute formats:
     - *set field initialAttributes*
     - *set field normal*
   - Choose any number of the next formats:
     - *set field cursor*
     - *set field empty*
     - *set field full*
2. Second, you can construct a **set** statement from any number of the next formats:
   - *set field cursor*
   - *set field full*
   - *set field initial* or *set field initialAttributes*
3. Last, you can construct a **set** statement as follows--
   - Choose any number of the next formats:
     - *set field cursor*
     - *set field full*
     - *set field modified*
   - Choose one or none of the color formats:
     - *set field defaultColor*
     - *set field selectedColor*
   - Choose one or none of the highlight formats:
     - *set field blink*
     - *set field reverse*
     - *set field underline*
     - *set field noHighlight*
   - Choose one or none of the intensity formats:
     - *set field bold*
     - *set field dim*
     - *set field invisible*
     - *set field masked*

  - *set field normalIntensity*
- Choose one or none of the protection formats:
  - *set field protect*
  - *set field skip*
  - *set field unprotect*

The syntax diagram is as follows:



field name
Name of the field in a text form. The name may refer to an array of fields.

The options are as described in the previous table.

Related concepts
"Form part" on page 144
"Modified data tag and modified property" on page 150
"Syntax diagram for EGL statements and commands" on page 733

Related reference
"Data initialization" on page 459
"EGL statements" on page 83
"Field-presentation properties" on page 62
"get next" on page 579
"get previous" on page 584
"itemsNullable" on page 377
"setFormItemFull" on page 383
"SQL item properties" on page 63

show

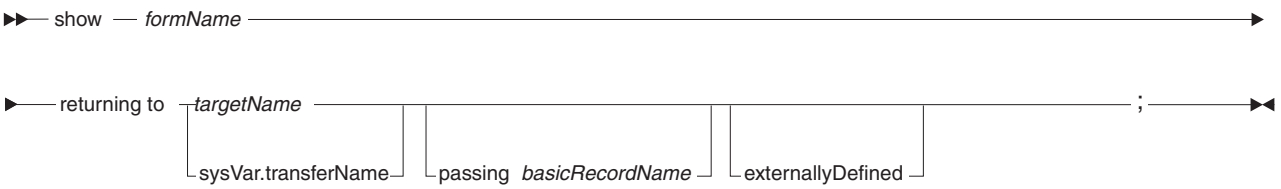
The **show** statement presents a text form from a main program:

1. Commits recoverable resources, closes files, and releases locks
2. Optionally, passes a basic record for use by the program that is specified in the **show** statement's returning clause (if any)
3. Ends the first program
4. Presents the text form

The **show** statement is not available in a called program.

If you include a returning clause in the **show** statement, the EGL run time invokes the specified program when the user presses an event key. The form data is assigned to the receiving program's *input form*. The passed record (unchanged by user input) is assigned to the receiving program's *input record*.

If you do not include a returning clause, the operation ends when the text form is presented.



formPartName
Name of a text form that is visible to the program. For details on visibility, see *References to parts*. If you include a returning clause in the statement, the text form must be equivalent to the text form specified in the **inputForm** property of the program being invoked.

targetName
Name of the program that is invoked after the user submits the text form.

sysVar.transferName
A system variable that contains the identifier of the program to be invoked. Use this variable to set the identifier at run time.

basicRecordName
Name of a record of type basicRecord. The content is assigned to the receiving program's *input record*.

externallyDefined
An indicator that the program is externally defined. This indicator is available only if you set the project property for VisualAge Generator compatibility and is appropriate only if you are generating a COBOL program.

It is recommended that a non-EGL-generated program be identified as externally defined not in the **show** statement, but in the linkage options part that is used at generation time. (The related property is in the linkage options part, transferLink element, and is also called **externallyDefined**.) You can make the identification, however, in either way.

Related concepts
"References to parts" on page 20

Related reference
"transferName" on page 914

transfer

The EGL **transfer** statement gives control from one main program to another, ends the transferring program, and optionally passes a record whose data is accepted into the receiving program's *input record*. You cannot use a **transfer** statement in a called program.

Your program can transfer control by a statement of the form *transfer to a transaction* or by a statement of the form *transfer to a program*:

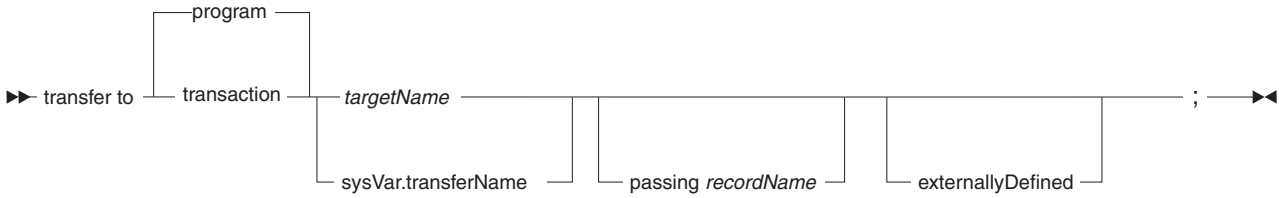
- A transfer to a transaction acts as follows--
  - In a program that runs as a Java main text or main batch program, the behavior depends on the setting of build descriptor option **synchOnTrxTransfer**--
    - If the value of **synchOnTrxTransfer** is YES, the transfer statement commits recoverable resources, closes files, closes cursors, and starts a program in the same run unit.
    - If the value of **synchOnTrxTransfer** is NO (the default), the transfer statement also starts a program in the same run unit, but does not close or commit resources, which are available to the invoked program.
  - In a PageHandler, a transfer to a transaction is not valid; use the **forward** statement instead.
- A *transfer to a program* does not commit or rollback recoverable resources, but closes files, closes cursors, and starts a program in the same run unit.

The linkage options part, **transferLink** element has no effect when you are transferring control from Java code to Java code, but is meaningful otherwise.

If you are transferring control code to code that was *not* written with EGL or VisualAge Generator, it is recommended that you set the linkage options part, **transferLink** element. Set the **linkType** property to *externallyDefined*.

If you are running in VisualAge Generator compatibility mode, you can specify the option **externallyDefined** in the transfer statement, as occurs for programs migrated from VisualAge Generator; but it is recommended that you set the

equivalent value in the linkage options part instead. For details on VisualAge Generator compatibility mode, see *Compatibility with VisualAge Generator*.



program *targetName* **(the default)**
The program that receives control. If you are generating for COBOL and specify a program name of more than 8 characters, the program name is truncated to 8 characters with character substitutions (if needed), as described in *Name aliasing*.

transaction *targetName*
The program that receives control, as described earlier.

sysVar.transferName
A system function that contains a target name that can be set at run time. For details, see *sysVar.transferName*.

passing *recordName*
A record that is received as the input record in the target program. The passed record may be of any type, but the length and primitive types must be compatible with the record that receives the data. The input record in the target program must be of type basicRecord.

externallyDefined
Not recommended for new development, as described earlier.

Related concepts
"Compatibility with VisualAge Generator" on page 428
"Name aliasing" on page 646

Related reference
"transferName" on page 914

try

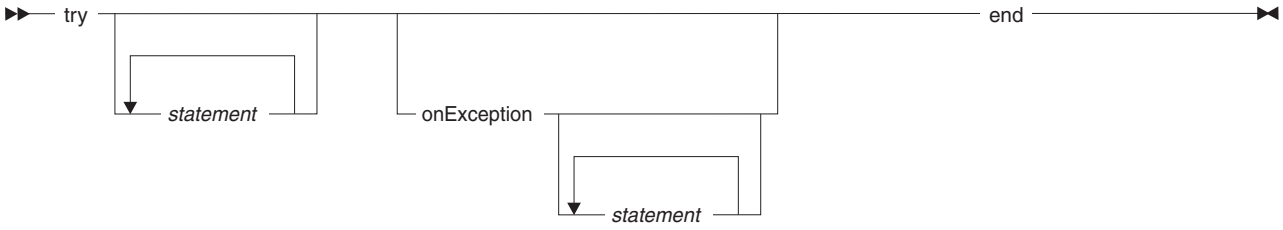
The EGL **try** statement indicates that the program continues running if a statement of any of the following kinds results in an error and is within the **try** statement:

- An input/output (I/O) statement
- A system-function invocation
- A **call** statement

If an exception occurs, processing resumes at the first statement in the **onException** block (if any), or at the first statement following the end of the **try** statement. A hard I/O error, however, is handled only if the system variable **VGVar.handleHardIOErrors** is set to 1; otherwise, the program displays a message (if possible) and ends.

A **try** statement has no effect on run-time behavior when an exception occurs in a function or program that is invoked from within the **try** statement.

For other details, see *Exception handling*.

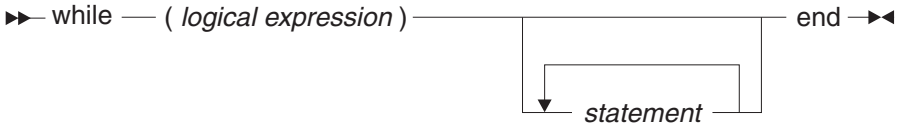


statement
Any EGL statement.

OnException
A block of statements that run if an exception condition occurs.

while

The EGL keyword **while** marks the start of a set of statements that run in a loop. The first run occurs only if a logical expression resolves to true, and each subsequent iteration depends on the same test. The keyword **end** marks the close of the **while** statement.



logical expression
An expression (a series of operands and operators) that evaluates to true or false

statement
A statement in the EGL language

An example is as follows:

```
sum = 0;
i = 1;
while (i < 4)
  sum = inputArray[i] + sum;
  i = i + 1;
end
```

Related tasks
"Syntax diagram for EGL statements and commands" on page 733

Related reference
"Logical expressions" on page 484
"EGL statements" on page 83

Library (generated output)

A library part for Java output is generated as a Java class. The name of the class is the part alias (or is the part name, if no alias is specified), but EGL makes character substitutions as described in *How Java names are aliased*.

Related concepts
"Library part of type basicLibrary" on page 133
"Library part of type basicLibrary" on page 133
"Run unit" on page 721

Related tasks
"How COBOL names are aliased" on page 648
"How Java names are aliased" on page 649

Related reference
"Library part in EGL source format"

Library part in EGL source format

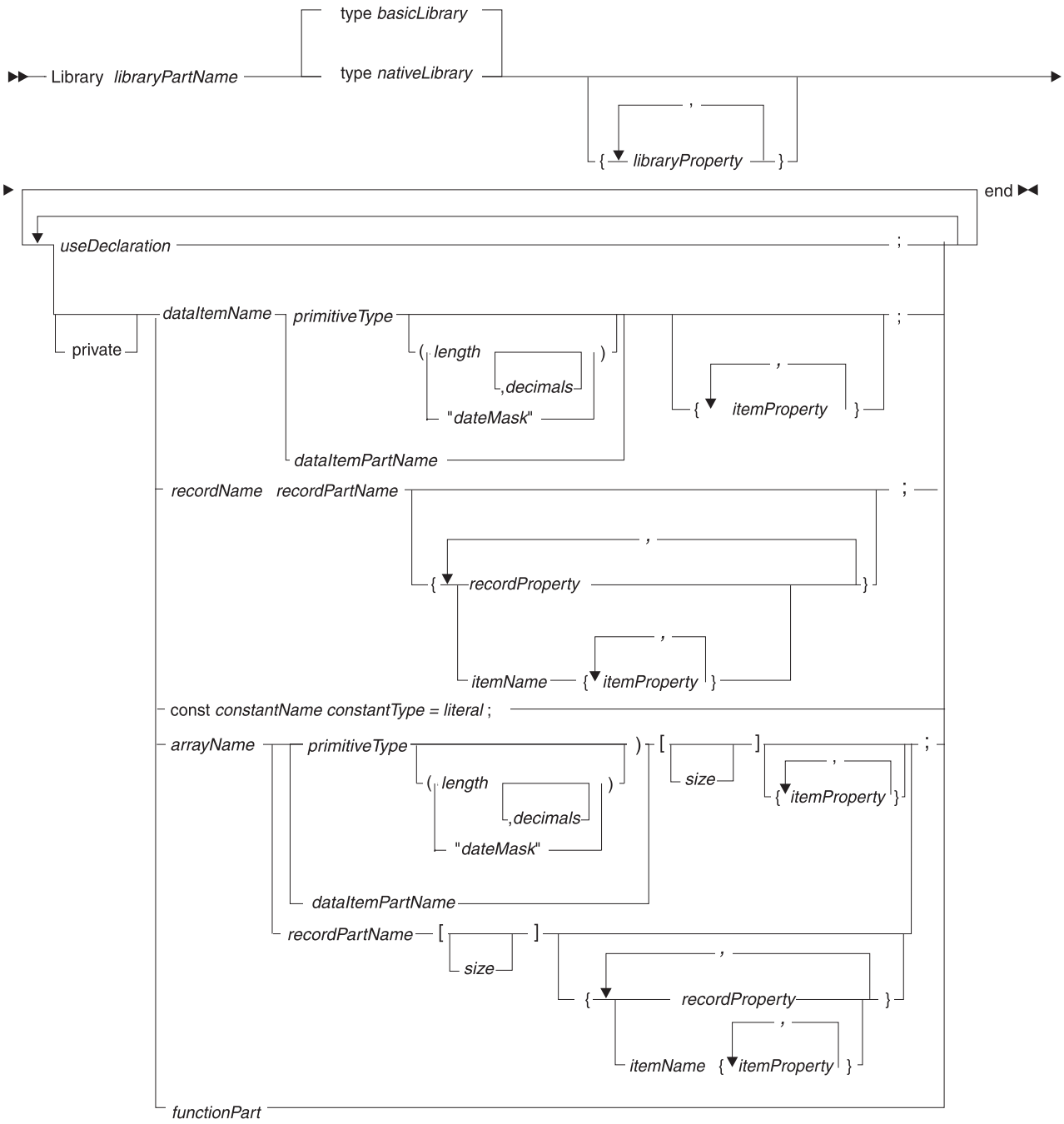
You declare a library part in an EGL file, which is described in *EGL source format*.

An example of a library part is as follows:

```
Library CustomerLib3

 // Use declarations
 Use StatusLib;

 // Data Declarations
 exceptionId ExceptionId ;

 // Retrieve one customer for an email
 //   In:  customer, with emailAddress set
 //   Out: customer, status
 Function getCustomerByEmail ( customer CustomerForEmail, status int )
  status = StatusLib.success;
  try
   get customer ;
  onException
   exceptionId = "getCustomerByEmail" ;
   status = sqlCode ;
  end
  commit();
 end

 // Retrieve one customer for a customer ID
 //   In: customer, with customer ID set
 //   Out: customer, status
 Function getCustomerByCustomerId ( customer Customer, status int )
  status = StatusLib.success;
  try
   get customer ;
  onException
   exceptionId = "getCustomerByCusomerId" ;
   status = sqlCode ;
  end
  commit();
 end

 // Retrieve multiple customers for an email
 //   In: startId
 //   Out: customers, status
 Function getCustomersByCustomerId
( startId CustomerId, customers Customer[], status int )
  status = StatusLib.success;
  try
   get customers usingKeys startId ;
  onException
   exceptionId = "getCustomerForEmail" ;
```

```
    status = sqlCode ;
  end
  commit();
 end

end
```

The diagram of a library part is as follows:



Library *libraryPartName* **... end**
Identifies the part as a library part and specifies the name. If you do not set

the **alias** property (as described later), the name of the generated library is either *libraryPartName* or, if you are generating COBOL, the first eight characters of *libraryPartName*.

For other rules, see *Naming conventions*.

type *basicLibrary*, **type** *nativeLibrary*
Indicates the library type:

- A basic library (type basicLibrary) contains EGL-written functions and values for runtime use in other EGL logic; for details, see *Library part of type basicLibrary*.
- A native library (type nativeLibrary) acts as an interface for an external DLL; for details, see *Library part of type nativeLibrary*

The library is of type basicLibrary by default.

libraryProperties
The library properties are as follows:

- **alias**
- **allowUnqualifiedItemReferences**
- **callingConvention** (which is available only in libraries of type nativeLibrary)
- **dllName** (which is available only in libraries of type nativeLibrary)
- **handleHardIOErrors**
- **includeReferencedFunctions**
- **localSQLScope**
- **messageTablePrefix**
- **throwNrfEofExceptions**

All are optional:

- **alias =** *"alias"* identifies a string that is incorporated into the names of generated output. If you do not set the **alias** property, the program-part name (or a truncated version)is used instead.
- **allowUnqualifiedItemReferences = no**, **allowUnqualifiedItemReferences = yes** specifies whether to allow your code to reference structure items but to exclude the name of the *container*, which is the data table, record, or form that holds the structure item. Consider the following record part, for example:

```
Record aRecordPart type basicRecord
  10 myItem01 CHAR(5);
  10 myItem02 CHAR(5);
end
```

The following variable is based on that part:

```
myRecord aRecordPart;
```

If you accept the default value of **allowUnqualifiedItemReferences** (*no*), you must specify the record name when referring to myItem01, as in this assignment:

```
myValue = myRecord.myItem01;
```

If you set the property **allowUnqualifiedItemReferences** to *yes*, however, you can avoid specifying the record name:

```
myValue = myItem01;
```

It is recommended that you accept the default value, which promotes a best practice. By specifying the container name, you reduces ambiguity for people who read your code and for EGL.

EGL uses a set of rules to determine the area of memory to which a variable name or item name refers. For details, see *References to variables and constants*.

- As used in a library of type nativeLibrary, **callingConvention = I4GL** specifies how the EGL run time passes data between two kinds of code:
  - The EGL code that invokes the library function; and
  - The function in the DLL being accessed.

  The only value now available for **callingConvention** is *I4GL*. For additional details, see *Library part of type nativeLibrary*.
- As used in a library of type nativeLibrary, **dllName** specifies the DLL name, which is final; it cannot be overridden at deployment time. If you do not specify a value for the library property **dllName**, you must specify the DLL name in the Java runtime property vgj.defaultI4GLNativeLibrary.

  For additional details, see *Library part of type nativeLibrary*.
- **handleHardIOErrors = yes**, **handleHardIOErrors = no** sets the default value for the system variable **VGVar.handleHardIOErrors**. The variable controls whether a program continues to run after a hard error has occurred on an I/O operation in a try block. The default value for the property is *yes*, which sets the variable to 1.

  For other details, see *VGVar.handleHardIOErrors* and *Exception handling*.
- **includeReferencedFunctions = no**, **includeReferencedFunctions = yes** indicates whether the library contains a copy of each function that is neither inside the library nor in a library accessed by the current library. The default value is *no*, which means that you can ignore this property if all functions that are to be part of this library are inside the library.

  If the library is using shared functions that are not in the library, generation is possible only if you set the property **includeReferencedFunctions** to *yes*.
- **localSQLScope = yes**, **localSQLScope = no** indicates whether identifiers for SQL result sets and prepared statements are local to the library code during invocation by a program or pageHandler, as is the default. If you accept the value *yes*, different programs can use the same identifiers independently, and the program or pageHandler that uses the library can independently use the same identifiers as are used in the library.

  If you specify *no*, the identifiers are shared throughout the run unit. The identifiers created when the SQL statements in the library is invoked are available in other code that invokes the library, although the other code can use **localSQLScope = yes** to block access to those identifiers. Also, the library may reference identifiers created in the invoking program or pageHandler, but only if the SQL-related statements were already run in the other code and if the other code did not block access.

  The effects of sharing SQL identifiers are as follows:
  - You can open a result set in one code and get rows from that set in another
  - You can prepare an SQL statement in one code and run that statement in another

  In any case, the identifiers available when the program or pageHandler accesses the library are available when the same program or pageHandler accesses the same or another function in the same library.
- **msgTablePrefix =** *"prefix"* specifies the first one to the four characters in the name of a data table that is used as a message table. (The message table is available to forms that are output by library functions.) The other characters in the name correspond to one of the national language codes listed in *DataTable part in EGL source format*.

- **throwNrfEofExceptions = no**, **throwNrfEofExceptions = yes** specifies whether a soft error causes an exception to be thrown. The default is *no*. For background information, see *Exception handling*.

useDeclaration
Provides easier access to a data table or library, and is needed to access forms in a form group. For details, see *Use declaration*.

private
Indicates that the variable, constant, or function is unavailable outside the library. If you omit the term **private**, the variable, constant, or function is available.

You cannot specify **private** for a function in a library of type nativeLibrary.

dataItemName
Name of a data item. For the rules of naming, see *Naming conventions*.

primitiveType
The primitive type of a data item or (in relation to an array) the primitive type of an array element.

length
The parameter's length or (in relation to an array) the length of an array element. The length is an integer that represents the number of characters or digits in the memory area referenced either by *dataItemName* or (in the case of an array) *dynamicArrayName*.

decimals
For a numeric type, you may specify *decimals*, which is an integer that represents the number of places after the decimal point. The maximum number of decimal positions is the smaller of two numbers: 18 or the number of digits declared as *length*. The decimal point is not stored with the data.

"dateTimeMask"
For TIMESTAMP and INTERVAL types, you may specify *"dateTimeMask"*, which assigns a meaning (such as "year digit") to a given position in the datetime value. The mask is not stored with the data.

dataItemPartName
The name of a dataItem part that is visible to the program. For details on visibility, see *References to parts*.

The part acts as a model of format, as described in *Typedef*.

recordName
Name of a record. For the rules of naming, see *Naming conventions*.

recordPartName
Name of a record part that is visible to the program. For details on visibility, see *References to parts*.

The part acts as a model of format, as described in *Typedef*.

constantName literal
Name and value of a constant. The value is either a quoted string or a number. For the rules of naming, see *Naming conventions*.

itemProperty
An item-specific property-and-value pair, as described in*Overview of EGL properties and overrides*.

recordProperty
A record-specific property-and-value pair. For details on the available properties, see the reference topic for the record type of interest.

A basic record has no properties.

itemName
Name of a record item whose properties you wish to override. See *Overview of EGL properties and overrides*.

arrayName
Name of a dynamic or static array of records or data items. If you use this option, the other symbols to the right (*dataItemPartName*, *primitiveType*, and so on) refer to each element of the array.

size
Number of elements in the array. If you specify the number of elements, the array is static; otherwise, the array is dynamic.

functionPart
A function. No parameter in the function can be of a loose type. For details, see *Function part in EGL source format*.

Related concepts
"EGL projects, packages, and files" on page 13
"Library part of type basicLibrary" on page 133
"Library part of type basicLibrary" on page 133
"Overview of EGL properties" on page 60
"References to parts" on page 20
"References to variables in EGL" on page 55
"Typedef" on page 25

Related reference
"Basic record part in EGL source format" on page 357
"DataTable part in EGL source format" on page 462
"EGL source format" on page 478
"Exception handling" on page 89
"Function part in EGL source format" on page 513
"Indexed record part in EGL source format" on page 520
"Input form" on page 715
"Input record" on page 715
"I/O error values" on page 522
"Java runtime properties (details)" on page 525
"MQ record part in EGL source format" on page 642
"Primitive types" on page 31
"Relative record part in EGL source format" on page 719
"Serial record part in EGL source format" on page 722
"SQL record part in EGL source format" on page 726

"Use declaration" on page 930
"handleHardIOErrors" on page 920

like operator

In a logical expression, you can compare a text expression against another string (called a *like criterion*), character position by character position from left to right. Use of this feature is similar to use of the SQL keyword **like** in SQL queries.

An example is as follows:

```
// variable myVar01 is the string expression
// whose contents will be compared to a like criterion
myVar01 = "abcdef";

// the next logical expression evaluates to "true"
if (myVar01 like "a_c%")
  ;
end
```

The like criterion can be either a literal or item of type CHAR or MBCHAR; or an item of type UNICODE. You can include any of these characters in the like criterion:

% Acts as a wild card, matching zero or more of any characters in the string expression

_ (underscore)
Acts as a wild card, matching a single character in the string expression

\ Indicates that the next character is to be compared to a single character in the string expression. The backward virgule (\) is called an *escape character* because it causes an escape from the usual processing; the escape character is not compared to any character in the string expression.

The escape character usually precedes a percent sign (%), an underscore (_), or another backward virgule.

When you use the backward virgule as an escape character (as is the default behavior), a problem arises because EGL uses the same escape character to allow inclusion of a quote mark in any text expression. In the context of a like criterion, you must specify two backward virgules because the text available at run time is the text that lacks the initial virgule.

It is recommended that you avoid this problem. Specify another character as the escape character by using the escape clause, as shown in a later example. However, you cannot use a double quote mark (") as an escape character.

Any other character in *likeCriterion* is a literal that is compared to a single character in *string expression*.

The following example shows use of an escape clause:

```
// variable myVar01 is the string expression
// whose contents will be compared to a like criterion
myVar01 = "ab%def";

// the next logical expression evaluates to "true"
if (myVar01 like "ab\\%def")
  ;
end

// the next logical expression evaluates to "true"
if (myVar01 like "ab+%def" escape "+")
  ;
end
```

Related reference
"EGL statements" on page 83
"Logical expressions" on page 484
"Text expressions" on page 492

Linkage properties file (details)

When you generate a calling Java program or wrapper, you can specify that linkage information is required at run time. You make that specification by setting the linkage-option values for the called program as follows:

- The value of the callLink element property **type** is remoteCall or ejbCall; and
- The value of the callLink element property **remoteBind** is RUNTIME.

A linkage properties file may be handwritten, but EGL generates a file if (in addition to the settings described earlier) you generate a Java program or wrapper with the build descriptor option **genProperties** set to GLOBAL or PROGRAM.

How the linkage properties file is identified at run time

If the callLink element property **remoteBind** for a called program was set to RUNTIME in the linkage options part, the linkage properties file is sought at run time; but the source of the file name is different for Java programs and Java wrappers:

- A Java program checks the Java run-time property **cso.linkageOptions.*LO***, where *LO* is the name of the linkage options part used for generation. If the property is not present, the EGL run-time code seeks a linkage properties file named ***LO*.properties**. Again, *LO* is the name of the linkage options part used for generation.

  In this case, if the EGL run-time code seeks a linkage properties file but is unable to find that file, an error occurs on the first call statement that requires use of that file. For details on the result, see Exception handling.
- The Java wrapper stores the name of the linkage properties file in the program object variable *callOptions*, which is of type CSOCallOptions. The generated name of the file is ***LO*.properties**, where *LO* is the name of the linkage options part used for generation.

  In this case, if the Java Virtual Machine seeks a linkage properties file but is unable to find that file, the program object throws an exception of type CSOException.

Format of the linkage properties file

As used during run time, the linkage properties file includes a series of entries to handle each call from the generated Java program or wrapper that you are deploying.

The primary entry is of type cso.serverLinkage and can include any property-and-value pair that you can set in a callLink element of the linkage options part, with the following exceptions:

- Property **remoteBind** is necessarily RUNTIME and should not appear
- Property **type** cannot be localCall, because linkage for local calls must be established at generation time

cso.serverLinkage entries

In the most elementary case, each entry in the linkage properties file is of type cso.serverLinkage. The format of the entry is as follows:

```
cso.serverLinkage.programName.property=value
```

programName
The name of the called program. If the called program is generated by EGL, the name you specify is that of a program part.

property
Any of the properties appropriate for a Java program, except for properties **remoteBind** and **pgmName**. For details, see *callLink element*.

value
A value that is valid for the specified property.

An example for called program Xyz is as follows, where *xxx* refers to a case-sensitive string:

```
cso.serverLinkage.Xyz.type=ejbCall
cso.serverLinkage.Xyz.remoteComType=TCPIP
cso.serverLinkage.Xyz.remotePgmType=EGL
cso.serverLinkage.Xyz.externalName=xxx
cso.serverLinkage.Xyz.package=xxx
cso.serverLinkage.Xyz.conversionTable=xxx
cso.serverLinkage.Xyz.location=xxx
cso.serverLinkage.Xyz.serverID=xxx
cso.serverLinkage.Xyz.parmForm=COMMDATA
cso.serverLinkage.Xyz.providerURL=xxx
cso.serverLinkage.Xyz.luwControl=CLIENT
```

The literal values TCPIP, EGL, and so on are not case sensitive and are examples of valid data.

cso.application entries

If you wish to create a series of cso.serverLinkage entries that refer to any of several called programs, precede those entries with one or more entries of type cso.application. Your purpose in this case is to equate a single application name to multiple program names. In the subsequent cso.serverLinkage entries, you use the application name instead of *programName*; then, at Java run time, those cso.serverLinkage entries handle calls to any of several programs.

The format of a cso.application entry is as follows:

```
cso.application.wildProgramName.appName
```

wildProgramName
A valid program name, an asterisk, or the beginning of a valid program name followed by an asterisk. The asterisk is the wild-card equivalent of one or more characters and provides a way to identify a set of names.

If *wildProgramName* refers to a program that is generated by EGL, any program name included in *wildProgramName* is the name of a program part.

appName
A series of characters that conforms to the EGL naming conventions. The value of *appName* is used in subsequent cso.serverLinkage entries.

The following example show use of an asterisk as a wild-card character. The cso.serverLinkage entries in this example handle any call to a program whose name begins with Xyz:

```
cso.application.Xyz*=myApp
cso.serverLinkage.myApp.type=remoteCall
cso.serverLinkage.myApp.remoteComType=TCPIP
cso.serverLinkage.myApp.remotePgmType=EGL
cso.serverLinkage.myApp.externalName=xxx
```

```
cso.serverLinkage.myApp.package=xxx
cso.serverLinkage.myApp.conversionTable=xxx
cso.serverLinkage.myApp.location=xxx
cso.serverLinkage.myApp.serverID=xxx
cso.serverLinkage.myApp.parmForm=COMMDATA
cso.serverLinkage.myApp.luwControl=CLIENT
```

The following example shows use of the same cso.serverLinkage entries to handle calls to any of several programs, even though the names of those programs do not begin with the same characters:

```
cso.application.Abc=myApp
cso.application.Def=myApp
cso.application.Xyz=myApp
cso.serverLinkage.myApp.type=remoteCall
cso.serverLinkage.myApp.remoteComType=TCPIP
cso.serverLinkage.myApp.remotePgmType=EGL
cso.serverLinkage.myApp.externalName=xxx
cso.serverLinkage.myApp.package=xxx
cso.serverLinkage.myApp.conversionTable=xxx
cso.serverLinkage.myApp.location=xxx
cso.serverLinkage.myApp.serverID=xxx
cso.serverLinkage.myApp.parmForm=COMMDATA
cso.serverLinkage.myApp.luwControl=CLIENT
```

If multiple cso.application entries are valid for a program, EGL uses the first entry that applies.

Related concepts
"Linkage options part" on page 291
"Linkage properties file" on page 343

Related tasks
"Editing the callLink element of a linkage options part" on page 294

"Setting up the J2EE run-time environment for EGL-generated code" on page 333

Related reference
"callLink element" on page 395
"Exception handling" on page 89
"Java runtime properties (details)" on page 525
"Naming conventions" on page 652

matches operator

In a logical expression, you can compare a string expression against another string (called a *match criterion*), character position by character position from left to right. Use of this feature is similar to use of *regular expressions* in UNIX or Perl.

An example is as follows:

```
// variable myVar01 is the string expression
// whose contents will be compared to a match criterion
myVar01 = "abcdef";

// the next logical expression evaluates to "true"
if (myVar01 matches "a?c*")
  ;
end
```

The match criterion can be either a literal or item of type CHAR or MBCHAR; or an item of type UNICODE. You can include any of these characters in the match criterion:

***** Acts as a wild card, matching zero or more of any characters in the string expression

? Acts as a wild card, matching a single character in the string expression

[ ] Acts as a delimiter such that any one of the characters between the two brackets is valid as a match for the next character in the string expression. The following component of a match criterion, for example, indicates that a, b, or c is valid as a match:

```
[abc]
```

- Creates a range within the bracket delimiters, such that any character within the range is valid as a match for the next character in the string expression. The following component of a match criterion, for example, indicates that a, b, or c is valid as a match:

```
[a-c]
```

The hyphen (-) has no special meaning outside of bracket delimiters.

^ Creates a wild-card rule such that, if the caret (^) is the first character inside bracket delimiters, any character other than the delimited characters is valid as a match for the next character in the string expression. The following component of a match criterion, for example, indicates that any character other than a, b, or c is valid as a match:

```
[^abc]
```

The caret has no special meaning in these cases:

- It is outside of bracket delimiters
- It is inside of bracket delimiters, but not in the first position

**** Indicates that the next character is to be compared to a single character in the string expression. The backward virgule (\) is called an *escape character* because it causes an escape from the usual processing; the escape character is not compared to any character in the string expression.

The escape character usually precedes a character that is otherwise meaningful in the match criterion; for example, an asterisk (*) or a question mark (?).

When you use the backward virgule as an escape character (as is the default behavior), a problem arises because EGL uses the same escape character to allow inclusion of a quote mark in any text expression. In the context of a match criterion, you must specify two backward virgules because the text available at run time is the text that lacks the initial virgule.

It is recommended that you avoid this problem. Specify another character as the escape character by using the escape clause, as shown in a later example. However, you cannot use a double quote mark (") as an escape character.

Any other character in *matchCriterion* is a literal that is compared to a single character in *string expression*.

The following example shows use of an escape clause:

```
// variable myVar01 is the string expression
// whose contents will be compared to a match criterion
myVar01 = "ab*def";

// the next logical expression evaluates to "true"
if (myVar01 matches "ab\\*[abcd][abcde][^a-e]")
  ;
end

// the next logical expression evaluates to "true"
if (myVar01 matches "ab+*def" escape "+")
  ;
end
```

Related reference
"EGL statements" on page 83
"Logical expressions" on page 484
"Text expressions" on page 492

Message customization for EGL Java run time

When an error occurs at Java run time, an EGL system message is displayed by default; but you can specify a customized message for each of those system messages or for a subset.

When a message is required, EGL first searches a properties file that you identify in the Java runtime property vgj.messages.file. The format of the referenced file is the same as for any Java properties file, as described in *Program properties file* and as shown in the current topic.

In many cases, a system message includes placeholders for the message inserts that EGL retrieves at run time. If your code submits an invalid date mask to a system function, for example, the message has two placeholders; one (placeholder 0) for the date mask itself, the other (placeholder 1) for the name of the system function. In properties-file format, the entry for the default message is as follows:

```
VGJ0216E = {0} is not a valid date mask for {1}.
```

You can change the wording of the message to include all or some of the placeholders in any order, but you cannot add placeholders. Valid examples are as follows:

```
VGJ0216E = Function {1} was given invalid date mask {0}.

VGJ0216E = Function {1} was given an invalid date mask.
```

A fatal error occurs if the file identified in property vgj.messages.file cannot be opened.

For details on the message numbers and their meaning, see *EGL Java runtime error codes*.

Other details are available in Java language documentation:

- For details on how messages are processed and on what content is valid, see the documentation for Java class java.text.MessageFormat.
- For details on handing characters that cannot be directly represented in the ISO 8859-1 character encoding (which is always used in properties files), see the documentation for Java class java.util.Properties.

Related concepts
"Program properties file" on page 329

Related reference
"EGL Java runtime error codes" on page 935
"Java runtime properties (details)" on page 525

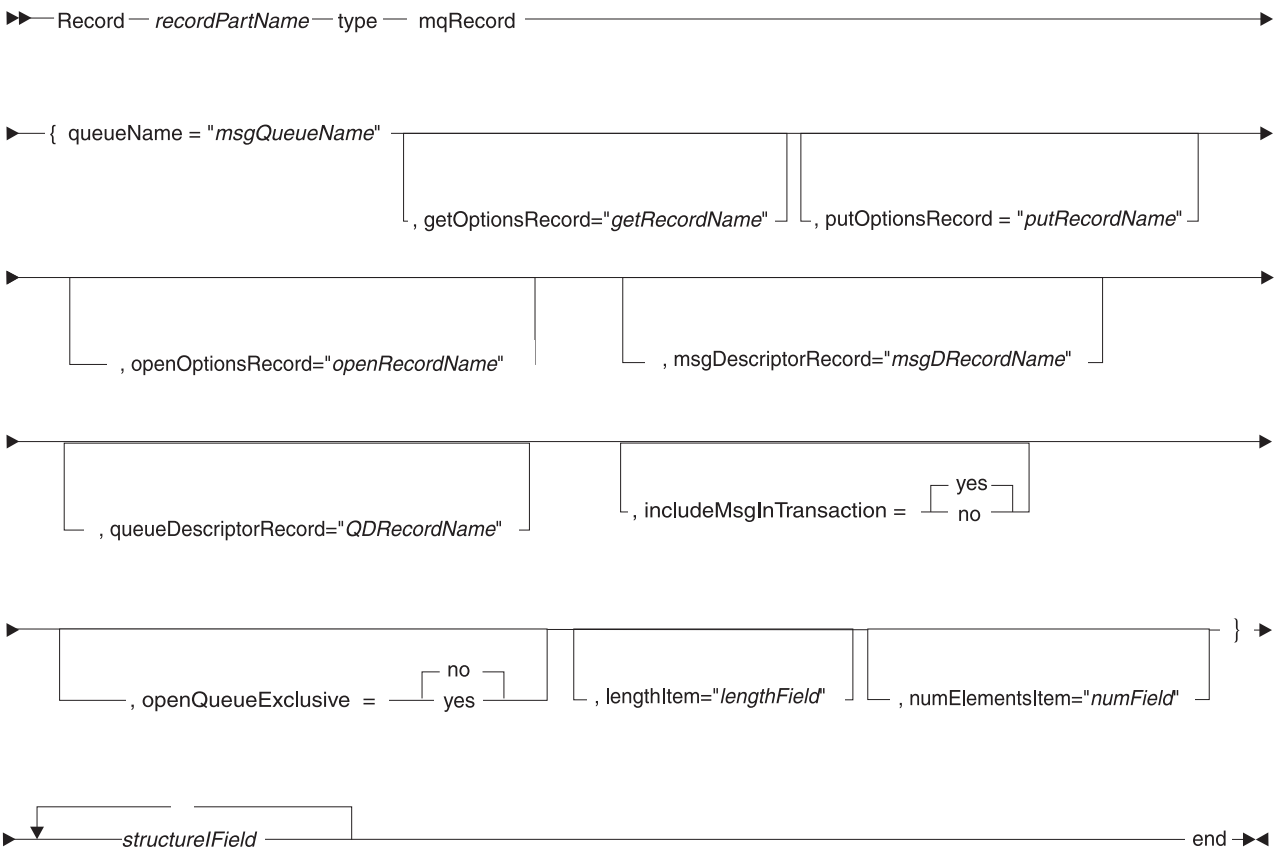
MQ record part in EGL source format

You can declare MQ record parts in an EGL source file. For an overview of that file, see *EGL source format*. For an overview of how EGL interacts with MQSeries, see *MQSeries support*.

An example of an MQ record part is as follows:

```
Record myMQRecordPart type mqRecord
  {
    queueName = "myQueue"
  }
  10 myField01 CHAR(2);
  10 myField02 CHAR(78);
end
```

The syntax diagram for an MQ record part is as follows:



Record *recordPartName* **mqRecord**
Identifies the part as being of type mqRecord and specifies the name. For rules, see *Naming conventions*.

queueName = *"msgQueueName"*
The message queue name, which is the logical queue name and usually not the name of the physical queue. For details on the format of your input, see *MQ record properties*.

getOptionsRecord = *"getRecordName"*
Identifies a program variable (a basic record) that is used as a get options record. For details, see *Options records for MQ records*. This property was formerly the **getOptions** property.

putOptionsRecord = *"putRecordName"*
Identifies a program variable (a basic record) that is used as a put options record. For details, see *Options records for MQ records*. This property was formerly the **putOptions** property.

openOptionsRecord = *"openRecordName"*
Identifies a program variable (a basic record) that is used as an open options record. For details, see *Options records for MQ records*. This property was formerly the **openOptions** property.

msgDescriptorRecord = *"msgDRecordName"*
Identifies a program variable (a basic record) that is used as a message descriptor. For details, see *Options records for MQ records*. This property was formerly the **msgDescriptor** property.

queueDescriptorRecord = *"QDRecordName"*
Identifies a program variable (a basic record) that is used as a queue descriptor. For details, see *Options records for MQ records*. This property was formerly the **queueDescriptor** property.

includeMsgInTransaction = yes, includeMsgInTransaction = no
If this property is set to *yes* (the default), each of the record-specific messages is embedded in a transaction, and your code can commit or roll back that transaction. For details on the implications of your choice, see *MQSeries support*.

openQueueExclusive = no, openQueueExclusive = yes
If this property is set to *yes*, your code has the exclusive ability to read from the message queue; otherwise, other programs can read from the queue. The default is *no*. This property is equivalent to the MQSeries option MQOO_INPUT_EXCLUSIVE.

lengthItem = *"lengthField"*
The length field, as described in *MQ record properties*.

numElementsItem = *"numElementsField"*
The number of elements field, as described in *MQ record properties*.

structureField
A structure field, as described in *Structure item in EGL source format*.

Related concepts
"EGL projects, packages, and files" on page 13
"References to parts" on page 20
"MQSeries support" on page 247
"Parts" on page 17
"Record parts" on page 124
"References to variables in EGL" on page 55
"Typedef" on page 25

Related tasks
"Syntax diagram for EGL statements and commands" on page 733

Related reference
"Arrays" on page 69
"DataItem part in EGL source format" on page 461
"EGL source format" on page 478
"Function part in EGL source format" on page 513
"Indexed record part in EGL source format" on page 520
"MQ record properties"
"Naming conventions" on page 652
"Options records for MQ records" on page 645
"Primitive types" on page 31
"Program part in EGL source format" on page 707
"Relative record part in EGL source format" on page 719
"Serial record part in EGL source format" on page 722
"SQL record part in EGL source format" on page 726
"Structure field in EGL source format" on page 730

MQ record properties

This page describes these MQ record properties:

- Queue name
- Include message in transaction
- Open input queue for exclusive use

For details on the other properties, see these pages:

- Options records for MQ records
- Properties that support variable-length records

Queue name

Queue name is required and refers to the logical queue name, which can be no more than 8 characters. For details on the meaning of your input, see *MQSeries-related EGL keywords*.

Include message in transaction

Include message in transaction, if set, embeds each of the record-specific messages in a transaction, and your code can commit or roll back that transaction.

For details on the implications of your choice, see *MQSeries support*.

Open input queue for exclusive use

If you set *Open input queue for exclusive use*, your code has the exclusive ability to read from the message queue; otherwise, other programs can read from the queue. This property is equivalent to the MQSeries option MQOO_INPUT_EXCLUSIVE.

Related concepts
"MQSeries-related EGL keywords" on page 250
"MQSeries support" on page 247
"Record types and properties" on page 126

Related reference
"Options records for MQ records"
"Properties that support variable-length records" on page 716

Options records for MQ records

Each MQ record is associated with five *options records*, which EGL uses as arguments in the hidden calls to MQSeries:

- Get options record (MQGMO)
- Put options record (MQPMO)
- Open options record (MQOO: a record with one structure item)
- Message descriptor record (MQMD)
- Queue descriptor record (MQOD)

When you specify an options record as a property of an MQ record, you are referring to a variable that uses a working storage record part (like MQOD) as a typeDef. The part resides in an EGL file that is provided with the product, as described in *MQSeries support*. Instead of using the record part as is, you can copy it into your own EGL file and customize the part.

If you do not indicate that a given options record is in use, EGL builds a default record and assigns values, as described in the following sections. The default options records are not available, however, when you access MQSeries without using MQ records.

Get options record

You can create a get options record based on the MQSeries Get Message Options (MQGMO), which is an argument on MQSeries MQGET calls. If you do not declare a get options record, EGL automatically builds a default named MQGMO, and your generated program does the following:

- Initializes the get options record with the values listed at the beginning of *Data initialization*
- Sets OPTIONS to either MQGMO_SYNCPOINT or MQGMO_NO_SYNCPOINT, depending on whether you set the MQ record property *Include message in transaction*

Put options record

You can create a put options record based on the MQSeries Put Message Options (MQPMO), which is an argument on MQSeries MQPUT calls. If you do not declare a put options record, EGL automatically builds a default named MQPMO, and your generated program does the following:

- Initializes the put options record with the values listed at the beginning of *Data initialization*
- Sets OPTIONS to either MQPMO_SYNCPOINT or MQPMO_NO_SYNCPOINT, depending on whether you set the MQ record property *Include message in transaction*

Open options record

The content of the open options record determines the value of the Options parameter that is used in calls to the MQSeries command MQOPEN or MQCLOSE. The open options record part (MQOO) is available, but if you do not declare a record based on that part, EGL automatically builds a default named MQOO as follows:

- On an MQOPEN that is invoked because of an EGL add statement, the generated program sets MQOO.OPTIONS to this:

```
MQOO_OUTPUT + MQOO_FAIL_IF_QUIESCING
```

- On an MQOPEN that is invoked because of an EGL scan statement, the generated program sets MQOO.OPTIONS to the following when the message queue record property option *Open input queue for exclusive use* is in effect:

  ```
  MQOO_INPUT_EXCLUSIVE + MQOO_FAIL_IF_QUIESCING
  ```

- On an MQOPEN that is invoked because of an EGL scan statement, the generated program sets MQOO.OPTIONS to the following when the message queue record property option *Open input queue for exclusive use* is *not* in effect:

  ```
  MQOO_INPUT_SHARED + MQOO_FAIL_IF_QUIESCING
  ```

- On an MQCLOSE that is invoked because of an EGL close statement, the generated program sets MQOO.OPTIONS to the following:

  ```
  MQCO_NONE
  ```

Message descriptor record

You can create a message descriptor record based on the MQSeries Message Descriptor (MQMD), which is a parameter on MQGET and MQPUT calls. If you do not declare a message descriptor record, EGL automatically builds a default named MQMD and initializes that record with the values listed in *Data initialization*.

Queue descriptor record

You can create a queue descriptor record based on the MQSeries Object Descriptor (MQOD), which is an argument on MQSeries MQOPEN and MQCLOSE calls. If you do not declare a queue descriptor record, EGL automatically builds a default named MQOD, and your generated program does the following:

- Initializes the queue descriptor record with the values listed at the beginning of *Data initialization*
- Sets OBJECTTYPE in that record to MQOT_Q
- Sets OBJECTMGRNAME to the queue manager name specified in the system word ***record*.resourceAssociation**; but if ***record*.resourceAssociation** does not reference the queue manager name, OBJECTQMGRNAME has no value
- Sets OBJECTNAME to the queue name in ***record*.resourceAssociation**

Related concepts
"Direct MQSeries calls" on page 252
"MQSeries-related EGL keywords" on page 250
"MQSeries support" on page 247

Related reference
"Data initialization" on page 459
"recordName.resourceAssociation" on page 832
"MQ record properties" on page 644

Name aliasing

If you use a name that is not valid in Java output, the generator creates and uses an alias for the name in the generated code, for any of these reasons:

- Differences in identifier characters allowed
- Differences in length limitations
- Differences in support for uppercase and lowercase characters
- Using a word that is a reserved word in the generated language
- Using a word that clashes with the name alias syntax (for example, **class$** is aliased because **class$** is the alias for **class** in Java generation)

An alias may be generated by substituting a valid set of characters for an invalid character, by truncating names that are too long, by adding a prefix or suffix to a name, or by producing a completely different name such as **EZE00123**.

Related concepts
"COBOL reserved-word file" on page 426

Related tasks
"Creating an EGL program part" on page 129

Related reference
"How COBOL names are aliased" on page 648
"How Java names are aliased" on page 649
"How Java wrapper names are aliased" on page 650
"Naming conventions" on page 652

Changes to EGL identifiers in JSP files and generated Java beans

You assign names to PageHandler functions, records, and items in accordance with the rules detailed in *Naming conventions*. However, EGL uses a variation of those names when creating Java identifiers in JSP files and in the Java bean that is derived from a PageHandler. You need to be aware of those variations if you use the source tab to edit a JSP file, if you use the Properties view, or if you work outside of the EGL-enabled tooling altogether.

The variations are as follows:

- The letters *EGL* precede the names of the PageHandler records, items, and functions. The purpose of this variation is to protect you from errors that could result in the Java run-time environment as a result of differences between the Java bean specification and the naming conventions in EGL.
- In several situations, a suffix is added to the name of a variable that is bound to a particular kind of output control:
  - If you bind an item to a Boolean check box, the Java identifier includes the suffix *AsBoolean*
  - If you bind an item to a selection control (a list box, combo box, radio button group, or check box group) and reference the item in the JavaServer Faces selectItems tag, the Java identifier includes the suffix *AsSelectItemsList*
  - If you bind an item to a check box in a JavaServer Faces data table (specifically, if the item is referenced in an inputRowSelect tag), the Java identifier includes the suffix *AsIntegerArray*

Aside from the variations listed earlier, EGL attempts to create an identifier that exactly matches the name in the PageHandler.

Consider the PageHandler *myJSP*, which includes variable *myIteM*. If you bind that variable to a Boolean check box, the JSP file references the Java-bean property *myJSP.EGLmyIteMAsBoolean*, and the Java-bean getter and setter functions are named as follows:

- *getEGLmyIteMAsBoolean*
- *setEGLmyIteMAsBoolean*

The source for the Boolean check-box tag in the JSP file is as follows:

```
<h:selectBooleanCheckbox styleClass="selectBooleanCheckbox"
   id="checkbox1" value="#{myJSP.EGLmyIteMAsBoolean}">
</h:selectBooleanCheckbox>
```

EGL avoids generating a name that would not be valid in Java; for details, see *How Java names are aliased*.

Related concepts
"PageHandler" on page 180

Related tasks
"Creating an EGL field and associating it with a Faces JSP" on page 184
"Associating an EGL record with a Faces JSP" on page 185

"Using the Quick Edit view for PageHandler code" on page 187

Related reference
"How Java names are aliased" on page 649
"Naming conventions" on page 652
"Page Designer support for EGL" on page 178

How names are aliased

When you are naming EGL parts, the EGL language let you use names that are not permitted in the programming language being generated. During generation, such names are replaced with names that are permitted in the language being generated.

For example, COBOL does not permit names that contain underscore characters, names that contain lowercase letters, or names longer than 30 characters. If you use such a name in EGL code, the name is replaced with a valid name in the output.

Related concepts
"Name aliasing" on page 646

Related tasks
"Creating an EGL source file" on page 120

Related reference
"How COBOL names are aliased"
"How Java names are aliased" on page 649

How COBOL names are aliased

A COBOL name begins with a letter and comprises from one to 30 characters from the following set: letters A-Z, digits 0-9, and the hyphen or minus sign (-).

An EGL part name may be aliased for any of the following reasons:

- The part name contains invalid COBOL characters
- The part name contains lowercase letters
- The part name is longer than a maximum length
- The part name is not unique in the program
- The part name is a COBOL reserved word

In all cases, all characters are made upper case.

For a subset of parts (specifically, a program, data table, form, form group, or library), you can specify an alias by assigning a value to the **alias** property; and if that value is too long or has characters that are not valid in COBOL, an error occurs. If you did not specify a value for the property and if the value of the part name is too long, the part name is truncated to the maximum, which varies by part type:

- For data tables, 7
- For forms, 8
- For form groups, 6
- For libraries, 8
- For programs, 8

For the other parts (data items, functions, and records), EGL aliases names as follows:

1. Each character that is not valid in COBOL is replaced with an *X*, except that each underscore is replaced with a hyphen (-); for example, TEMP_ITEM becomes TEMP-ITEM
2. Part names that are longer than a maximum length are changed as follows:
   - The name is prefixed with the letters EZE, a hyphen, and a one-to-five-digit number that is unique to the program
   - The new name is truncated to the maximum length

   The maximum length varies by part type:
   - For data items, 27
   - For functions, 18
   - For records, 18
3. If after the previous steps the part name is a duplicate name in the program, the prefix described earlier is added to the beginning of the second and any subsequent occurrences of the part name. The resulting alias is truncated to the maximum length as stated above.
4. If after steps 1-3 the part name matches a COBOL reserved word, the prefix described earlier is added to the beginning of the part name and the resulting alias is truncated to the maximum length as stated above.
5. If after steps 1-4 the part name begins or ends with a hyphen, the beginning or ending hyphen is changed to *X*.

Related concepts
"COBOL reserved-word file" on page 426

Related reference
"Format of COBOL reserved-word file" on page 427

How Java names are aliased

When you give a part a name, that name must be a valid Java identifier, except that you can use a hyphen or minus sign (-) in a part name. However, a hyphen cannot be the first character in a part name.

If you choose a name that is a Java keyword or a name that contains a dollar sign ($) or a hyphen or minus, the part name will not match the name in the generated output. An aliasing mechanism automatically appends a dollar sign to each part

name that is a Java keyword. If you specify a name that contains one or more dollar signs or hyphens, the aliasing mechanism replaces each symbol with a Unicode value as follows:

```
$ $0024
- $002d
```

For example, an item named **class** is aliased to **class$**, and an item named **class$** is aliased to **class$0024**.

The case you use to declare a part name is preserved. Programs `XYZ` and `xyz` are generated in XYZ.java and xyz.java respectively. On Windows 2000/NT/XP, if you generate into the same directory parts with names that differ only in case, the older files are overwritten.

EGL package names are always converted to lower case Java package names.

Finally, if the name of a program, PageHandler, or library matches the name of a class from the Java system package java.lang, a dollar sign is appended to the class name: Object becomes Object$, Error becomes Error$, and so on.

For details on how EGL creates Java identifiers in JSP files and in the Java bean that is derived from a PageHandler, see *Changes to EGL identifiers in JSP files and generated Java beans*.

Related concepts
"Name aliasing" on page 646

Related reference
"Changes to EGL identifiers in JSP files and generated Java beans" on page 647
"How COBOL names are aliased" on page 648

How Java wrapper names are aliased

The EGL generator applies the following rules to alias Java wrapper names:

1. If the EGL name is all uppercase, convert it to lowercase.
2. If the name is a class name or a method name, make the first character uppercase. (For example, the getter method for **x** is **getX()** not **getx()**.)
3. Delete every underscore (_) and hyphen (-). (Hyphens are valid in EGL names if you use VisualAge Generator compatibility mode.) If a letter follows the underscore or hyphen, change that character to uppercase.
4. If the name is a qualified name that uses a period (.) as a separator, replace every period with a low line, and add a low line at the beginning of the name.
5. If the name contains a dollar sign ($), replace the dollar sign with two low lines and add a low line at the beginning of the name.
6. If a name is a Java keyword, add a low line at the beginning of the name.
7. If the name is * (an asterisk, which represents a filler item), rename the first asterisk **Filler1**, the second asterisk **Filler2**, and so forth.

In addition, special rules apply to Java wrapper class names for program wrappers, record wrappers, and substructured array items. The remaining sections discuss these rules and give an example. In general, if naming conflicts exist between fields within a generated wrapper class, the qualified name is used to determine the class and variable names. If the conflict is still not resolved, an exception is thrown at generation time.

Program wrapper class

Record parameter wrappers are named by using the above rules applied to the type definition name. If the record wrapper class name conflicts with the program class name or the program wrapper class name, **Record** is added at the end of the record wrapper class name.

The rules for variable names are as follows:

1. The record parameter variable is named using above rules applied to the parameter name. Therefore, the **get()** and **set()** methods contain these names rather than the class name.
2. The **get** and **set** methods are named **get** or **set** followed by the parameter name with the above rules applied.

Record wrapper class

The rules for substructured array items class names are as follows:

1. The substructured array item becomes an inner class of the record wrapper class, and the class name is derived by applying the above rules to the item name. If this class name conflicts with the containing record class name, **Structure** is appended to the item class name.
2. If any item class names conflict with each other, the qualified item names are used.

The rules for **get** and **set** method names are as follows:

1. The methods are named **get** or **set** followed by the item name with the above rules applied.
2. If any item names conflict with each other, the qualified item names are used.

Substructured array items class

The rules for substructured array items class names are as follows:

1. The substructured array item becomes an inner class of the wrapper class generated for the containing substructured array item , and the class name is derived by applying the above rules to the item name.
2. If this class name conflicts with the containing substructured array item class name, **Structure** is appended to the item class name.

The rules for **get** and **set** method names are as follows:

1. The methods are named **get** or **set** followed by the item name with the above rules applied.
2. If any item names conflict with each other, the qualified item names are used.

Example

The following sample program and generated output show what should be expected during wrapper generation:

Sample program:

```
Program WrapperAlias(param1 RecordA)

end

Record RecordA type basicRecord
 10 itemA CHAR(10)[1];
 10 item_b CHAR(10)[1];
 10 item$C CHAR(10)[1];
 10 static CHAR(10)[1];
 10 itemC CHAR(20)[1];
```

```
  15 item CHAR(10)[1];
  15 itemD CHAR(10)[1];
 10 arrayItem CHAR(20)[5];
  15 innerItem1 CHAR(10)[1];
  15 innerItem2 CHAR(10)[1];
end
```

Generated output:

Names of generated output

Output	Name
Program wrapper class	**WrapperaliasWrapper**, containing a field **param1**, which is an instance of the record wrapper class **RecordA**
Parameter wrapper classes	**RecordA**, accessible through the following methods: • **getItemA** (from `itemA`) • **getItemB** (from the first `item-b`) • **get_Item__C** (from `item$C`) • **get_Static** (from `static`) • **get_ItemC_itemB** (from `itemB` in `itemC`) • **getItemD** (from `itemD`) • **getArrayItem** (from `arrayItem`) **ArrayItem** is an inner class of **RecordA** that contains fields that can be accessed through **getInnerItem1** and **getInnerItem2**.

Related concepts
"Compatibility with VisualAge Generator" on page 428
"Java wrapper" on page 282
"Name aliasing" on page 646

Related tasks
"Generating Java wrappers" on page 282

Related reference
"Java wrapper classes" on page 535
"Naming conventions"
"Output of Java wrapper generation" on page 656

Naming conventions

This page describes the rules for naming parts and variables and for assigning values to properties such as **file name**. For details on how logic parts can reference areas of memory, see *References to variables and constants* and *Arrays*.

Three categories of identifier are in EGL:

- EGL part and variables names, as described later.
- External resource names that are specified as property values in part or variable declarations. These names represent special cases, and the naming conventions depend on the conventions of the run-time system.
- EGL package names such as com.mycom.mypack. In this case, each character sequence is separated from the next by a period, and each sequence follows the naming convention for an EGL part name. For details on the relationship of package names and file structure, see *EGL projects, packages, and files*.

An EGL part or variable name is a series of 1 to 128 characters. Except as noted, a name must begin with a Unicode letter or underscore and can include additional Unicode letters as well as digits and currency symbols. Other restrictions are in effect:

- The first characters cannot be EZE in any combination of uppercase and lowercase
- A name cannot contain embedded blanks or be an EGL reserved word

Special considerations apply to parts:

- In a record part, the name of a logical file or queue can be no more than 8 characters
- In various parts, the *alias* is incorporated into the names of generated output files and Java classes. If the external name is not specified, the name of the program part is used but is truncated (if necessary) to the maximum number of characters allowed in the run-time environment.

If your code is compatible with VisualAge Generator, the following rules also apply to part and variable names but have no effect on package names:

- Initial character of a name can be an "at" sign (@)
- Subsequent characters can include "at" signs (@), hyphens (-), and pound signs (#)

Related concepts
"Compatibility with VisualAge Generator" on page 428
"EGL projects, packages, and files" on page 13
"Name aliasing" on page 646
"References to variables in EGL" on page 55

Related reference
"Arrays" on page 69
"Changes to EGL identifiers in JSP files and generated Java beans" on page 647
"EGL reserved words" on page 474
"EGL system limits" on page 481

Operators and precedence

The next table lists the EGL operators in order of decreasing precedence. Except for the unary plus (+), minus (–), and not (!), each operator works with two operands.

Operators (separated by commas)	Type of operator	Meaning
+, –	Numeric, unary	Unary plus (+) or minus (-) is a sign before an operand or parenthesized expression, not an operator between two expressions.
**	Numeric	** is the *toThePowerOfInteger* operator, which accepts a number to the specified power. For example c = a**b will result in c being assigned the value of (a^b). The first operand (a in the example above) cannot have a negative value. The second operand (b in the example above) must be an integer, or a numeric field with precision 0. The second operand can be positive, negative or 0.

Operators (separated by commas)	Type of operator	Meaning
*, /, %	Numeric	Multiplication (*) and integer division (/) are of equal precedence. The division of integers retains a fractional value, if any; for example, 7/5 yields 1.4. % is the *remainder* operator, which resolves to the modulus when the first of two operands or numeric expressions is divided by the second; for example, 7%5 yields 2. If you need to ensure that arithmetic results are consistent between EGL output in Java and COBOL, avoid using the remainder operator.
+, –	Numeric	Addition (+) and subtraction (–) are of equal precedence.
=	Numeric or string	= is the *assignment* operator, which copies a numeric or character value from an expression or operand into an operand.
!	Logical, unary	! is the *not* operator, which resolves to a Boolean value (true or false) opposite to the value of a logical expression that immediately follows. That subsequent expression must be in parentheses.
==, != , <, >, <=, >=, in, is, not	Logical for comparison	The logical operators used for comparison are of equal precedence and are described in the page on logical expressions. Each operator resolves to true or false.
&&	Logical	&& is the *and* operator, which means "both must be true." The operator resolves to true if the logical expression that precedes the operator is true and if the logical expression that follows the operator is true; otherwise, && resolves to false.
||	Logical	|| is the *or* operator, which means "one or the other or both." The operator resolves to true if the logical expression that precedes the operator is true or if the logical expression that follows the operator is true or if both are true; otherwise || resolves to false.

You may override the usual precedence (also called the *order of operations*) by using parentheses to separate one expression from another. Operations that have the same precedence in an expression are evaluated in left-to-right order.

Related reference
"in operator" on page 518
"Logical expressions" on page 484
"Numeric expressions" on page 491
"Primitive types" on page 31
"Text expressions" on page 492

Output of COBOL generation

The COBOL generation outputs for iSeries include a COBOL program, a results file (which includes status information on EGL generation and preparation), and a build plan (if the build descriptor option **buildPlan** is set to *yes*). For additional details, see *the EGL Server Guide for iSeries*, which is available in the help system.

Related concepts

"Build plan" on page 305

"COBOL program" on page 306
"Development process" on page 8

"Program part" on page 130
"Results file" on page 309
"Sources of additional information on EGL" on page 12

Related tasks
"Generating for COBOL" on page 309

Related reference
"buildPlan" on page 364
"callLink element" on page 395
"Generated output (reference)" on page 516
"Output of Java program generation"
"Output of Java wrapper generation" on page 656

Output of Java program generation

The output of Java server program generation is as follows:

- A build plan, if the build descriptor option **genProject** is omitted
- Java source code (see *Java program, PageHandler, and library*)
- Related objects needed to prepare and run your program (see *Java program, PageHandler, and library*)
- J2EE environment file
- Program properties file
- A results file, if **genProject** is omitted

You can use the EGL generator to generate entire Java programs. Programs and records are generated as separate Java classes. Functions are generated as methods in the program. Data items and structure items are generated as fields of the record or program class to which they belong.

The following table shows the names of the various types of generated Java parts:

Names of generated Java parts

Part type and name	What is generated
Program named `P`	A class named P in P.java
Function named `F` in program `P`	A method of the P class called $funcF in P.java
Record named `R`	A class named EzeR in EzeR.java

Names of generated Java parts

Part type and name	What is generated
Basic record named R, parameter to Function F	A class named Eze$paramR in Eze$paramR.java
Linkage options part named L	Linkage properties file named L.properties
Library named Lib	A class named Lib in Lib.java
DataTable named DT	A class named EzeDT in EzeDT.java
Form named F	A class named EzeF in EzeF.java
FormGroup named FG	A class named FG in FG.java

1. For the indicated part types, it is possible that two or more parts may exist with the same name. In that event the name of the second one will have an additional suffix, $v2. The name of the third will have a $v3 suffix, the fourth will have $v4, etc.

If the naming format would cause two names to be identical, EGL adds a suffix to each file generated after the first. The suffix as follows:

$v*n*

where
n Is an integer assigned in sequential order, beginning with 2.

Related concepts
"Build plan" on page 305
"J2EE environment file" on page 336
"Java program, PageHandler, and library" on page 306
"Linkage properties file" on page 343
"Program properties file" on page 329
"Results file" on page 309

Related reference
"callLink element" on page 395

Output of Java wrapper generation

The output of Java wrapper generation is as follows:

- A build plan, if the build descriptor option **genProject** is omitted
- JavaBeans™ for wrapping calls to a Java server program (see *Java wrapper)*
- EJB session beans under certain circumstances; for details, see the explanation of the callLink element in *Linkage options part*
- A results file, if **genProject** is omitted

You can use the generated beans to wrap calls to server programs from non-EGL Java classes such as servlets, EJBs, or Java applications. The following types of classes are generated:

- Beans for servers
- Beans for record parameters
- Beans for record array rows

The following table shows the names of the various types of generated Java wrapper parts:

Names of generated Java wrapper parts

Part type and name	What is generated
Program named P	A class named PWrapper in PWrapper.java
Record named R used as a parameter	A class named R in R.java
Substructured area S in record R used as a parameter	A class named R.S in R.java
Linkage options part named L	Linkage properties file named L.properties

1. For the indicated part types, it is possible that two or more parts may exist with the same name. In that event the name of the second one will have an additional suffix, $v2. The name of the third will have a $v3 suffix, the fourth will have $v4, etc.

When you request that a program part be generated as a Java wrapper, EGL produces Java class for each of the following executables:

- The program part
- Each record that is declared as a program parameter
- A session bean, if you specify a linkage options part and a the **callLink** element for the generated program has a link type of **ejbCall**

In addition, the class generated for each record includes an inner class (or a class within an inner class) for each structure item that has these characteristics:

- Is in the internal structure of one of those records
- Has at least one subordinate structure item; in other words, is substructured
- Is an array; in this case, a substructured array

Each generated class is stored in a file. The EGL generator creates names used in Java wrappers as follows:

- The name is converted to lowercase.
- Each hyphen or minus (-) or underscore (_) is deleted. A character that follows a hyphen or underscore is changed to uppercase.
- When the name is used as a class name or within a method name, the first character is translated back to uppercase.

If one of the parameters to the program is a record, EGL generates a wrapper class for that variable as well. If program Prog has a record parameter with a typeDef named Rec, the wrapper class for the parameter will be called Rec. If the typeDef of a parameter has the same name as the program, the wrapper class for the parameter will have a ″Record″ suffix.

The generator also produces a wrapper if a record parameter has an array item and the item has other items under it. This substructured array wrapper becomes an inner class of the record wrapper. In most cases, a substructured array item called AItem in Rec will be wrapped by a class called Rec.AItem. The record may contain two substructured array items with the same name, in which case the item wrappers are named by using the qualified names of the items. If the qualified name of the first AItem is Top1.AItem and the qualified name of the second is Top2.Middle2.AItem, the classes will be named Rec.Top1$_aItem and Rec.Top2$_middle2$_aItem. If the name of a substructured array is the same as the name of the program, the wrapper class for substructured array will have a Structure suffix.

Methods to set and get the value of low-level items are generated into each record wrapper and substructured array wrapper. If two low-level items in the record or substructured array have the same name, the generator uses the qualified-name scheme described in the previous paragraph.

Additional methods are generated into wrappers for SQL record variables. For each item in the record variable, the generator creates methods to get and set its null indicator value and methods to get and set its SQL length indicator.

You can use the Javadoc tool to build a *classname*.html file once the the class has been compiled. The HTML file describes the public interfaces for the class. If you use HTML files created by Javadoc, be sure that it is an EGL Java wrapper. HTML files generated from a VisualAge Generator Java wrapper are different from those generated from an EGL Java wrapper.

Example

An example of a record part with a substructured array is as follows:

```
Record myRecord type basicRecord
  10 MyTopStructure[3];
    15 MyStructureItem01 CHAR(3);
    15 MyStructureItem02 CHAR(3);
end
```

In relation to the program part, the output file is named as follows:

*alias*Wrapper.java

where
alias
Is the alias name, if any, that is specified in the program part. If the external name is not specified, the name of the program part is used.

In relation to each record declared as a program parameter, the output file is named as follows:

recordName.java

where
recordName
Is the name of the record part

In relation to a substructured array, the name and position of the inner class depends on whether the array name is unique in the record:

- If the array name is unique in the record, the inner class is within the record class and is named as follows:

  recordName.*siName*

  where
  recordName
  Is the name of the record part
  siName
  Is the name of the array
- If the array name is not unique in the record, the name of the inner class is based on the fully qualified name of the array, with one qualifier separated from the next by a combination of dollar sign ($) and underscore (_). For example, if the array is at the third level of the record, the generated class is an inner class of the record class and is named as follows:

Topname$_Secondname$_Siname

where
Topname
Is the name of the top-level structure item
Secondname
Is the name of the second-level structure item
Siname
Is the name of the substructured-array item

If another, same-named array is immediately subordinate to the highest level of the record, the inner class is also within the record class and is named as follows:

Topname$_Siname

where
Topname
Is the name of the highest-level structure item
Siname
Is the name of the substructured-array item

Finally, consider the following case: a substructured array has a name that is not unique in the record, and the array is subordinate to another substructured array whose name is not unique in the record. The class for the subordinate array is generated as an inner class of an inner class.

When you generate a Java wrapper, you also generate a Java properties file and a linkage properties file if you request that linkage options be set at run time.

Related concepts
"Build plan" on page 305
"Enterprise JavaBean (EJB) session bean" on page 295
"Java wrapper" on page 282
"Linkage options part" on page 291
"Linkage properties file" on page 343
"Results file" on page 309

Related tasks
"Generating Java wrappers" on page 282

Related reference
"callLink element" on page 395
"Java wrapper classes" on page 535

PageHandler part in EGL source format

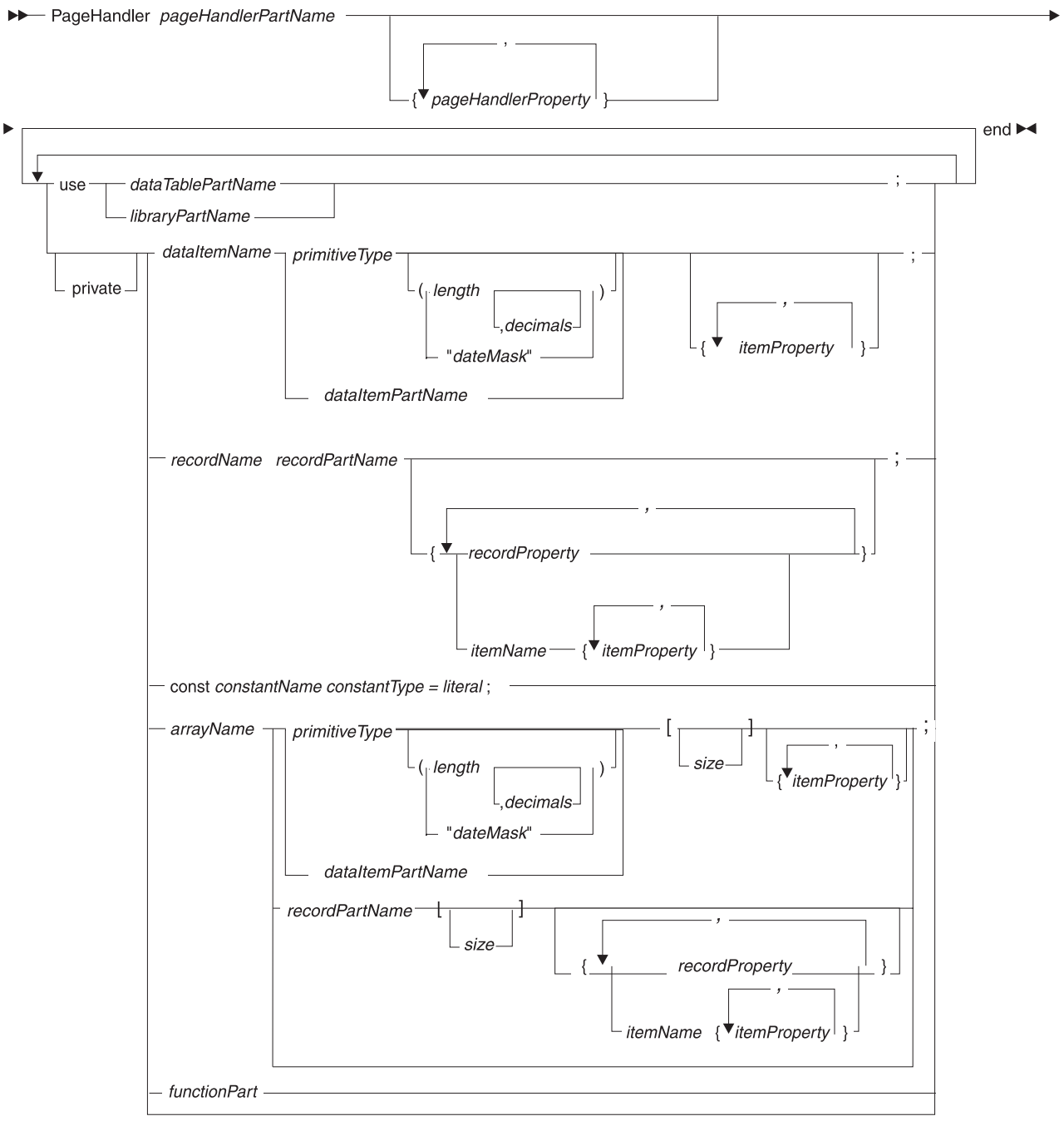
You declare a pageHandler part in an EGL file, which is described in *EGL projects, packages, and files*. This part is a generatable part, which means that it must be at the top level of the file and must have the same name as the file.

An example of a pageHandler part is as follows:

```
// Page designer requires that all pageHandlers
// be in a package named "pagehandlers".
package pagehandlers ;

PageHandler ListCustomers
  {onPageLoadFunction="onPageLoad"}

  // Library for customer table access
```

```
  use CustomerLib3;

  // List of customers
  customerList Customer[] {maxSize=100};

  Function onPageLoad()

    // Starting key to retrieve customers
    startkey CustomerId;

    // Result from library call
    status int;

    // Retrieve up to 100 customer records
    startKey = 0;
    CustomerLib3.getCustomersByCustomerId(startKey,
        customerList, status);

     if ( status != 0 && status != 100 )
      setError("Retrieval of Customers Failed.");
     end
  end

  Function returnToIntroductionClicked()
     forward to "Introduction";
  end
End
```

The diagram of a pageHandler part is as follows:



PageHandler *pageHandlerPartName* **... end**
Identifies the part as a PageHandler and specifies the part name. For the rules of naming, see *Naming conventions*.

pageHandlerProperty
A PageHandler part property, as listed in *PageHandler part properties*.

use *dataTablePartName*, **use** *libraryPartName*
A use declaration that simplifies access of a data table or library. For details, see *Use declaration*.

private
Indicates that the variable, constant, or function is unavailable to the JSP that

renders the Web page. If you omit the term **private**, you can bind the variable, constant, or function to a control on the Web page.

dataItemName
Name of a data item (a variable). For rules, see *Naming conventions*.

primitiveType
The primitive type assigned to the data item.

length
The structure item's length, which is an integer. The value of a memory area that is based on the structure item includes the specified number of characters or digits.

decimals
For a numeric type (BIN, DECIMAL, NUM, NUMC, or PACF), you may specify *decimals*, which is an integer that represents the number of places after the decimal point. The maximum number of decimal positions is the smaller of two numbers: 18 or the number of digits declared as *length*. The decimal point is not stored with the data.

dataItemPartName
The name of a dataItem part that is a model of format for the data item, as described in *typeDef*. The dataItem part must be visible to the pageHandler part, as described in *References to parts*.

itemProperty
An item property. For details, see *Page item properties*.

recordName
Name of a record (a variable). For rules, see *Naming conventions*.

recordPartName
The name of a record part that is a model of format for the record, as described in *typeDef*. The record part must be visible to the pageHandler part, as described in *References to parts*.

recordProperty
An override of a record property. For details on the record properties, see one of the following descriptions, depending on the type of record in *recordPartName*:

- Basic record part in EGL source format
- Indexed record part in EGL source format
- MQ record part in EGL source format
- Relative record part in EGL source format
- Serial record part in EGL source format
- SQL record part in EGL source format

itemName
Name of the record item whose properties you intend to override.

itemProperty
An override of an item property. For details, see *Overview of EGL properties and overrides*.

constantName literal
Name and value of a constant. For rules, see *Naming conventions*.

arrayName
Name of a dynamic or static array of records or data items. If you use this option, the other symbols to the right (*dataItemPartName*, *primitiveType*, and so on) refer to each element of the array.

functionPart
An embedded function. For details on the syntax, see *Function part in EGL source format*.

Related concepts
"EGL projects, packages, and files" on page 13
"Overview of EGL properties" on page 60
"PageHandler" on page 180
"References to parts" on page 20
"References to variables in EGL" on page 55
"Typedef" on page 25

Related reference
"Exception handling" on page 89
"Function part in EGL source format" on page 513
"Naming conventions" on page 652
"PageHandler field properties" on page 665
"PageHandler part properties"
"Primitive types" on page 31
"Reference compatibility in EGL" on page 718
"setError()" on page 879
"Use declaration" on page 930

PageHandler part properties

Excluding properties that are specific to PageHandler *fields*, the properties of the PageHandler are as follows and are optional:

alias = *"alias"*
A string that is incorporated into the names of generated output. If you do not specify an alias, the PageHandler part name is used instead.

allowUnqualifiedItemReferences = no, allowUnqualifiedItemReferences = yes
Specifies whether to allow your code to reference structure items but to exclude the name of the *container*, which is the data table, record, or form that holds the structure item. Consider the following record part, for example:

```
Record aRecordPart type basicRecord
  10 myItem01 CHAR(5);
  10 myItem02 CHAR(5);
end
```

The following variable is based on that part:

```
myRecord aRecordPart;
```

If you accept the default value of **allowUnqualifiedItemReferences** (*no*), you must specify the record name when referring to myItem01, as in this assignment:

```
myValue = myRecord.myItem01;
```

If you set the property **allowUnqualifiedItemReferences** to *yes*, however, you can avoid specifying the record name:

```
myValue = myItem01;
```

It is recommended that you accept the default value, which promotes a best practice. By specifying the container name, you reduces ambiguity for people who read your code and for EGL.

EGL uses a set of rules to determine the area of memory to which a variable name or item name refers. For details, see *References to variables and constants*.

handleHardIOErrors = yes, handleHardIOErrors = no
Sets the default value for the system variable **VGVar.handleHardIOErrors**. The variable controls whether a program continues to run after a hard error has occurred on an I/O operation in a try block. The default value for the property is *yes*, which sets the variable to 1.

For other details, see *VGVar.handleHardIOErrors* and *Exception handling*.

includeReferencedFunctions = no, includeReferencedFunctions = yes
Indicates whether the PageHandler bean contains a copy of each function that is neither inside the PageHandler nor in a library accessed by the PageHandler. The default value is *no*, which means that you can ignore this property if you are fulfilling the following practices at development time, as is recommended:

- Place shared functions in a library
- Place non-shared functions in the PageHandler

If you are using shared functions that are not in a library, generation is possible only if you set the property **includeReferencedFunctions** to *yes*.

localSQLScope = no, localSQLScope = yes
Indicates whether identifiers for SQL result sets and prepared statements are local to the pageHandler, as is the default. If you accept the value *yes*, different programs that are called from the pageHandler can use the same identifiers independently.

If you specify *no*, the identifiers are shared throughout the run unit. The identifiers created in the current code are available elsewhere, although other code can use **localSQLScope = yes** to block access to those identifiers. Also, the current code may reference identifiers created elsewhere, but only if the other code was already run and did not block access.

The effects of sharing SQL identifiers are as follows:

- You can open a result set in a pageHandler or called program and get rows from that set in the other code
- You can prepare an SQL statement in one code and run that statement in another

msgResource = *"logicalName"*
Identifies a Java resource bundle or properties file that is used in error-message presentation. The content of the resource bundle or properties file is composed of a set of keys and related values.

A particular value is displayed in response to the program's invoking the EGL system function sysLib.setError, when the invocation includes use of the key for that value.

onPageLoadFunction = *"functionName"*
The name of a PageHandler function that receives control when the related JSP initially displays a Web page. This function can be used to set up initial values of the data displayed in the page. This property was formerly the **onPageLoad** property.

Arguments passed to the function must be reference-compatible, as described in *Reference Compatibility in EGL*.

scope = session, scope = request
Specifies what occurs after pageHandler data is sent to the Web page:

- If scope is set to *session* (as is the default), the pageHandler variable values are retained throughout the user session, and the user's later access of the same pageHandler does not re-invoke the OnPageLoad function
- If scope is set to *request*, the pageHandler variable values are lost, and the user's access of the same PageHandler re-invokes the OnPageLoad function

It is recommended that you set this property explicitly to document your decision, which greatly affects the design and operation of your Web application.

throwNrfEofExceptions = no, throwNrfEofExceptions = yes

Specifies whether a soft error causes an exception to be thrown. The default is *no*. For background information, see *Exception handling*.

title = *"literal"*
The **title** property is a bind property, which means that the assigned value is used as a default when you are working in Page Designer. The property specifies the title of the page.

literal is a quoted string.

validationBypassFunctions = [*"functionNames"***]**
Identifies one or more *event handlers*, which are PageHandler functions that are associated with a button control in the JSP. Each function name is separated from the next by a comma.

If you specify an event handler in this context, the EGL run time skips input-field and page validations when the user clicks the button or hypertext link that is or related to the event handler. This property is useful for reserving a user action that ends the current PageHandler processing and that immediately transfers control to another Web resource.

validatorFunction = *"functionName"*
Identifies the PageHandler validator function, which is invoked after all the item validators are invoked, as described in *Validation in Web applications built with EGL*. This property was formerly the **validator** property.

view = *"JSPFileName"*
Identifies the name and subdirectory path to the Java Server Page (JSP) that is bound to the PageHandler. *JSPFileName* is a quoted string.

The default value is the name of the PageHandler, with the file extension **.jsp**. If you specify this property, include the file extension, if any.

When you save or generate a PageHandler, EGL adds a JSP file to your project for subsequent customization, unless a JSP file of the same name (the name specified in the **view** property) is already in the appropriate folder (the folder WebContent\WEB-INF). EGL never overwrites a JSP.

PageHandler field properties

The PageHandler field properties specify characteristics that are meaningful when a field is declared in a PageHandler part.

The properties are as follows:

- "action" on page 670

- "byPassValidation" on page 671
- "displayName" on page 677
- "displayUse" on page 678
- "help" on page 680
- "newWindow" on page 688
- "numElementsItem" on page 688
- "selectFromListItem" on page 691
- "selectType" on page 692
- "validationOrder" on page 697
- "value" on page 702

Related concepts
"Overview of EGL properties" on page 60
"PageHandler" on page 180

Related tasks
"Creating an EGL pageHandler part" on page 177
"Using the Quick Edit view for PageHandler code" on page 187

Related reference
"PageHandler part properties" on page 663
"PageHandler part in EGL source format" on page 659
"Page Designer support for EGL" on page 178

pfKeyEquate

When you declare a form group that references a text form, the property *pfKeyEquate* specifies whether the keystroke that is registered when the user presses a high-numbered function key (PF13 through PF24) is the same as the keystroke that is registered when the user presses a function key that is lower by 12.

If you accept the default value of *yes* for pfKeyEquate, your your logical expressions are able to reference only 12 of the function keys because (for example) PF2 is the same as PF14.

Note: Function keys on a PC keyboard are often *F* keys such as F1, but EGL uses the IBM *PF* terminology so that (for example) F1 is called PF1.

Related concepts
"FormGroup part" on page 143

Related reference
"FormGroup part in EGL source format" on page 494

Primitive field-level properties

The next table lists the primitive field-level properties in EGL:

property	Description
action	Identifies the code that is invoked when the user clicks the button or link.

property	Description
align	Specifies the position of data in a variable field when the length of the data is smaller than the length of the field.
byPassValidation	Indicates whether EGL-based validation is bypassed when the user clicks the button or link.
color	Specifies the color of a field in a text form.
column	Refers to the name of the database table column that is associated with the item. The default is the name of the item.
currency	Indicates whether to include a currency symbol before the value in a numeric field, with the exact position of the symbol determined by the **zeroFormat** property.
currencySymbol	Indicates which currency symbol to use when the property **currency** is in effect.
dateFormat	identifies the format for dates.
detectable	Specifies whether the field's modified data tag is set when the field is selected by a light pen or (for emulator sessions) by a cursor click.
displayName	Specifies the label that is displayed next to the field.
displayUse	associates an EGL field with a user-interface control.
fieldLen	Specifies the number of single-byte characters that can be displayed in a text-form field.
fill	Indicates whether the user is required to enter data in each field position.
fillCharacter	Indicates what character fills unused positions in a text or print form or in PageHandler data.
help	Specifies the hover-help text that is displayed when the user places the cursor over the input field.
highlight	Specifies the special effect (if any) with which to display the field.
inputRequired	Indicates whether the user is required to place data in the field.
inputRequiredMsgKey	Identifies the message that is displayed if the field property **inputRequired** is set to *yes* and the user fails to place data into the field.
intensity	Specifies the strength of the displayed font.
isBoolean	Indicates that the field represents a Boolean value.
isDecimalDigit	Determines whether to check that the input value includes only decimal digits

property	Description
isHexDigit	Determines whether to check that the input value includes only hexadecimal digits
isNullable	Indicates whether the item can be set to null, as is appropriate if the table column associated with the item can be set to NULL.
isReadOnly	Indicates whether the item and related column should be omitted from the default SQL statements that write to the database or include a FOR UPDATE OF clause.
lineWrap	Indicates whether text can be wrapped onto a new line whenever wrapping is necessary to avoid truncating text.
lowerCase	Indicates whether to set alphabetic characters to lower case in the user's single-byte character input.
masked	Indicates whether a user-entered character will or will not be displayed.
maxLen	Specifies the maximum length of field text that is written to the database column.
minimumInput	Indicates the minimum number of characters that the user is required to place in the field, if the user places any data in the field.
minimumInputMsgKey	Identifies the message that is displayed if the user acts as follows: • Places data in the field; and • Places fewer characters than the value specified in the property **minimumInputRequired**.
modified	Indicates whether the program will consider the field to have been modified, regardless of whether the user changed the value.
needsSOSI	Indicates whether EGL does a special check when the user enters data of type MBCHAR on an ASCII device.
newWindow	Indicates whether to use a new browser window when the EGL run time presents a Web page in response to the activity identified in the **action** property.
numElementsItem	Identifies a PageHandler field whose runtime value specifies the number of array elements to display.
numericSeparator	Indicates whether to place a character in a number that has an integer component of more than 3 digits.
outline	Lets you draw lines at the edges of fields on any device that supports double-byte characters.
pattern	Matches the user entered text against a specified pattern, for validation.

property	Description
persistent	Indicates whether the field is included in the implicit SQL statements generated for the SQL record.
protect	Specifies whether the user can access the field.
selectFromListItem	Identifies the array or DataTable column from which the user selects a value or values, which are then transferred to the array or primitive field being declared.
selectType	Indicates the kind of value that is retrieved into the array or primitive field being declared.
sign	Indicates the position in which a positive (+) or negative (-) sign is displayed when a number is placed in the field, whether from user input or from the program.
sqlDataCode	Identifies the SQL data type that is associated with the record item.
sqlVariableLen	Indicates whether trailing blanks and nulls in a character field are truncated before the EGL run time writes the data to an SQL database.
timeFormat	Identifies the format for times.
timeStampFormat	Identifies the format for timestamps that are displayed on a form or maintained in a PageHandler.
typeChkMsgKey	Identifies the message that is displayed if the input data is not appropriate for the field type.
upperCase	Indicates whether to set alphabetic characters to upper case in the user's single-byte character input.
validationOrder	Indicates when the field's validator function runs in relation to any other field's validator function.
validatorDataTable	Identifies a *validator table*, which is a dataTable part that acts as the basis of a comparison with user input.
validatorDataTableMsgKey	Identifies the message that is displayed if the user provides data that does not correspond to the requirements of the *validator table*, which is the table specified in the property **validatorDataTable**.
validatorFunction	Identifies a validator function, which is logic that runs after the EGL run time does the elementary validation checks, if any.
validatorFunctionMsgKey	Identifies a message that is displayed
validValues	Indicates a set of values that are valid for user input.

property	Description
validValuesMsgKey	Identifies the message that is displayed if the field property **validValues** is set and the user places out-of-range data into the field.
value	Identifies a string literal that is displayed as the field content when a Web page is displayed.
zeroFormat	Specifies how zero values are displayed in numeric fields but not in fields of type MONEY.

action

When the EGL property **displayUse** is *button* or *hyperlink*, the property **action** identifies the code that is invoked when the user clicks the button or link. The value you assign to **action** is used as a default when you place the field (or a record that includes the field) on the Web Page in Page Designer.

The value of **action** is one of these kinds of string literals:

- The name of an event-handling function in the PageHandler
- A label that maps to a Web resource (for example, to a JSP) and that corresponds to a from-outcome attribute of a navigation-rule entry in the JSF Application Configuration Resource file
- The name of a method in a Java bean, in which case these rules apply:
  - The format is the bean name followed by a period and a method name
  - The bean name must relate to one of the managed bean-name entries in the JSF Application Configuration Resource file

If you do not specify a value for **action**, the user's click of the field has the following effect:

- If the value of the property **displayUse** is *button*, validation occurs, after which JSF re-displays the same Web page.
- If the value of the property **displayUse** is *hyperlink*, no validation occurs, but JSF re-displays the same Web page.

Related concepts
"Overview of EGL properties" on page 60
"PageHandler" on page 180

Related tasks
"Binding a JavaServer Faces command component to an EGL PageHandler" on page 186
"Creating an EGL pageHandler part" on page 177
"Using the Quick Edit view for PageHandler code" on page 187

Related reference
"PageHandler field properties" on page 665
"PageHandler part properties" on page 663
"PageHandler part in EGL source format" on page 659
"Page Designer support for EGL" on page 178

align

The **align** property specifies the position of data in a variable field when the length of the data is smaller than the length of the field.

Values are of the enumeration **alignKind**:

left
Place the data at the left of the field, as is the default for character data. Initial spaces are stripped and placed at the end of the field.

none
Do not justify the data. This setting is valid only for character data.

right
Place the data at the right of the field, as is the default for numeric data. Trailing spaces are stripped and placed at the beginning of the field. This setting is required for numeric data that has a decimal position or sign.

The property is available in DataItem parts and is meaningful for fields that appear in the following contexts:

- Console forms
- Print forms
- Text forms
- Web pages

On output, character and numeric data are affected by this property. On input, character data is affected by this property, but numeric data is always right-justified.

Related concepts
"Enumerations in EGL" on page 471
"Overview of EGL properties" on page 60

Related reference
"Formatting properties" on page 62

byPassValidation

When the EGL property **displayUse** is *button* or *hyperlink*, the property **byPassValidation** indicates whether EGL-based validation is bypassed when the user clicks the button or link. You might want to bypass validation for better performance; for example, whenever the user clicks an Exit button.

The value you assign to **byPassValidation** is used as a default when you place the field (or a record that includes the field) on the Web Page in Page Designer.

The property affects only EGL-based validations, not those specified by JSF tags; for details, see *PageHandler*.

Values are of the enumeration **Boolean**:

no (the default)
The input fields are validated as usual

yes
The EGL run time does not return user data to the PageHandler

Related concepts
"Enumerations in EGL" on page 471
"Overview of EGL properties" on page 60
"PageHandler" on page 180

Related tasks
"Binding a JavaServer Faces command component to an EGL PageHandler" on page 186
"Creating an EGL pageHandler part" on page 177
"Using the Quick Edit view for PageHandler code" on page 187

Related reference
"PageHandler field properties" on page 665
"PageHandler part properties" on page 663
"PageHandler part in EGL source format" on page 659
"Page Designer support for EGL" on page 178

color

The **color** property specifies the color of a field in a text form. You can select any of these:

- black
- blue
- cyan
- defaultColor (the default)
- green
- magenta
- red
- white
- yellow

If you assign the value *defaultColor*, other conditions determine the displayed color, as shown in the next table.

Are all fields on the form assigned the value *defaultColor***?**	**Value of** *protect*	**Value of** *intensity*	**Displayed color for a field assigned the value** *defaultColor*
yes	*yes* or *skip*	not *bold*	blue
yes	*yes* or *skip*	*bold*	white
yes	*no*	not *bold*	green
yes	*no*	*bold*	red
no	any value	not *bold*	green
no	any value	*bold*	white

Related concepts
"Enumerations in EGL" on page 471
"Overview of EGL properties" on page 60

Related reference
"Field-presentation properties" on page 62

column

The property **column** refers to the name of the database table column that is associated with the item. The default is the name of the item. The column and related item affect the default SQL statements, as described in *SQL support*.

For *"columnName"*, substitute a quoted string; a variable of a character type; or a concatenation, as in this example:

```
column = "Column" + "01"
```

A special syntax applies if a column name is one of the following SQL reserved words:

- CALL
- COLUMNS
- FROM
- GROUP
- HAVING
- INSERT
- ORDER
- SELECT
- SET
- UPDATE
- VALUES
- WHERE

As shown in the following example, each of those names must be embedded in a doubled pair of quote marks, and each of the internal quote marks must be preceded with an escape character (\):

```
column = "\"SELECT\""
```

(A similar situation applies if you use of those reserved words as a table name.)

Related concepts
"Compatibility with VisualAge Generator" on page 428
"Record types and properties" on page 126
"SQL support" on page 213
"Fixed structure" on page 24
"Typedef" on page 25

Related tasks
"Retrieving SQL table data" on page 235

Related reference
"Field-presentation properties" on page 62
"add" on page 544
"close" on page 551
"Data initialization" on page 459
"delete" on page 554
"execute" on page 557
"get" on page 567
"get next" on page 579
"open" on page 598
"prepare" on page 611
"Primitive types" on page 31
"Record and file type cross-reference" on page 716
"replace" on page 613
"set" on page 617

"SQL data codes and EGL host variables" on page 723
"terminalID" on page 913
"VAGCompatibility" on page 390

currency

The **currency** property indicates whether to include a currency symbol before the value in a numeric field, with the exact position of the symbol determined by the **zeroFormat** property. The formatting of fields of type MONEY depends on the value of **strLib.defaultMoneyFormat** and is not affected by the **currency** property.

Values of the **currency** property are as follows:

No (the default)
Do not use a currency symbol.

Yes
Use the symbol specified in **currencySymbol**. If no value is specified there, use the default currency symbol.

In Java code, the default currency symbol is determined by the machine locale. In COBOL code, the default is determined by the national language option.

The property is available in DataItem parts and is meaningful for fields that appear in the following contexts:

- Print forms
- Text forms
- Web pages

The property is used at input and output.

Related concepts
"Enumerations in EGL" on page 471
"Overview of EGL properties" on page 60

Related reference
"Formatting properties" on page 62

currencySymbol

The **currencySymbol** property indicates which currency symbol to use when the property **currency** is in effect. The value is a string literal.

The property is available in DataItem parts and is meaningful for fields that appear in the following contexts:

- Print forms
- Text forms
- Web pages

The property is used at input and output.

Related concepts
"Enumerations in EGL" on page 471
"Overview of EGL properties" on page 60

Related reference
"Formatting properties" on page 62

dateFormat

The **dateFormat** property identifies the format for dates.

Valid values are as follows:

"pattern"
The value of *pattern* consists of a set of characters, as described in *Date, time, and timestamp format specifiers*.

Characters may be dropped from the beginning or end of a complete date specification, but not from the middle.

defaultDateFormat
If specified for a page field, the value of **defaultDateFormat** is the date format given in the run-time Java locale. If specified for a form field, the default pattern is equivalent to selecting **systemGregorianDateFormat**.

eurDateFormat
The pattern "dd.MM.yyyy", which is the IBM European standard date format.

isoDateFormat
The pattern "yyyy-MM-dd", which is the date format specified by the International Standards Organization (ISO).

jisDateFormat
The pattern "yyyy-MM-dd", which is the Japanese Industrial Standard date format.

usaDateFormat
The pattern "MM/dd/yyyy", which is the IBM USA standard date format.

systemGregorianDateFormat
An 8- or 10-character pattern that includes dd (for numeric day), MM (for numeric month), and yy or yyyy (for numeric year), with characters other than d, M, y, or digits used as separators.

For COBOL programs, the system administrator for EGL run-time services sets the format at installation.

For Java programs, the format is in this Java run-time property:

```
vgj.datemask.gregorian.long.NLS
```

NLS
The NLS (national language support) code that is specified in the Java run-time property **vgj.nls.code**. The code is one of those listed in targetNLS. Uppercase English (code ENP) is not supported.

For additional details on **vgj.nls.code**, see *Java run-time properties (details)*.

systemJulianDateFormat
A 6- or 8-character pattern that includes DDD (for numeric day) and yy or yyyy (for numeric year), with characters other than D, y, or digits used as separators.

For COBOL programs, the system administrator for EGL run-time services sets the format at installation.

For Java programs, the format is in this Java run-time property:

```
vgj.datemask.julian.long.NLS
```

NLS
The NLS (national language support) code that is specified in the Java

run-time property **vgj.nls.code**. The code is one of those listed in targetNLS. Uppercase English (code ENP) is not supported.

For additional details on **vgj.nls.code**, see *Java run-time properties (details)*.

The property is available in DataItem parts and is meaningful for fields that appear in the following contexts:

- Console forms
- Print forms
- Text forms
- Web pages

This property is used for both input and output, but not in the following cases:

- The field has decimal places, a currency symbol, a numeric separator, or a sign; or
- The field is of type DBCHAR, MBCHAR, or HEX; or
- The field is not long enough to contain a value that reflects the mask. For other details, see "Length considerations for dates."

Internal date formats

When the user enters valid data, the date is converted from the format specified for the field to an internal format that is used for subsequent validation.

The internal format for a character date is the same as the system default format and includes separator characters.

For a numeric date, the internal formats are as follows:

- For a Gregorian short date, 00yyMMdd
- For a Gregorian long date, 00yyyyMMdd
- For a Julian short date, 0yyDDD
- For a Julian long date, 0yyyyDDD

Length considerations for dates

In a form, the field length on the form must be greater than or equal to the length of the field mask that you specify. The length of the field must be long enough to hold the internal format of the date.

In a page field, the rules are as follows:

- The field length must be sufficient for the date mask you specify but can be longer
- In the case of a numeric field, the separator characters are excluded from the length calculation.

Examples are in the next table.

Format type	Example	Length of form field	Minimum length of page field (character type)	Valid length of page field (numeric type)
Short Gregorian	yy/MM/dd	8	8	6
Long Gregorian	yyyy/MM/dd	10	10	8
Short Julian	DDD-yy	6	6	5

Format type	Example	Length of form field	Minimum length of page field (character type)	Valid length of page field (numeric type)
Long Julian	DDD-yyyy	8	8	7

I/O considerations for dates

Data entered into a variable field is checked to ensure that the date was entered in the format specified. The user does not need to enter the leading zeros for days and months, but can specify (for example) 8/5/1996 instead of 08/05/1996. The user who omits the separator characters, however, must enter all leading zeros.

Related concepts
"Java runtime properties" on page 327
"Overview of EGL properties" on page 60

Related reference
"Date, time, and timestamp format specifiers" on page 42
"Formatting properties" on page 62
"Java runtime properties (details)" on page 525

detectable

Specifies whether the field's modified data tag is set when the field is selected by a light pen or (for emulator sessions) by a cursor click.

The **detectable** property is available only for COBOL programs and only for text-form fields whose **intensity** property is other than *invisible*.

The initial character in the field content (as specified in the **value** property) must be a *designator character*, which indicates what action is taken when the user clicks on the field. The most common designator characters are as follows:

& Causes an *immediate detect*, which means that clicking the field at run time is equivalent to modifying the field and pressing the ENTER key.

? Causes a *delayed detect*, which means that clicking the field at run time is equivalent to modifying the field, but that the program receives the form information only when the user presses the ENTER key or clicks a field that is configured for an immediate detect.

To prevent the user from changing the designator character in a variable field, set the **protect** property to *yes* or *skip*.

Related concepts
"Overview of EGL properties" on page 60

Related reference
"Form part in EGL source format" on page 497

displayName

The **displayName** property specifies the label that is displayed next to the field. The value you assign is used as a default when you place the field (or a record that includes the field) on the Web Page in Page Designer.

The value of this property is a string literal.

Related concepts
"Overview of EGL properties" on page 60
"PageHandler" on page 180

Related tasks
"Associating an EGL record with a Faces JSP" on page 185
"Creating an EGL pageHandler part" on page 177
"Creating an EGL field and associating it with a Faces JSP" on page 184
"Using the Quick Edit view for PageHandler code" on page 187

Related reference
"PageHandler field properties" on page 665
"PageHandler part properties" on page 663
"PageHandler part in EGL source format" on page 659
"Page Designer support for EGL" on page 178

displayUse

The **displayUse** property associates an EGL field with a user-interface control. The value you assign is used as a default when you place the field (or a record that includes the field) on the Web Page in Page Designer.

Values are of the enumeration **displayUseKind**:

button
The control has a button command tag

secret
The data is not visible to the user. This value is appropriate for passwords.

hyperlink
If the **action** property is the name of an event-handling function, the control has a hyperlink command tag. If the **action** property is a label, the control has a link tag. When the user clicks the link in either case, no validation occurs and no input data is returned.

input
The control accepts user input. Initially, the control may display a value provided by the PageHandler.

table
Data is within a table tag.

output
PageHandler field output, if any, is visible in the control.

Related concepts
"Enumerations in EGL" on page 471
"Overview of EGL properties" on page 60
"PageHandler" on page 180

Related tasks
"Associating an EGL record with a Faces JSP" on page 185
"Binding a JavaServer Faces command component to an EGL PageHandler" on page 186
"Creating an EGL pageHandler part" on page 177
"Creating an EGL field and associating it with a Faces JSP" on page 184
"Using the Quick Edit view for PageHandler code" on page 187

Related reference
"PageHandler field properties" on page 665
"PageHandler part properties" on page 663
"PageHandler part in EGL source format" on page 659
"Page Designer support for EGL" on page 178

fieldLen

The property **fieldLen** specifies the number of single-byte characters that can be displayed in a text-form field. This value does not include the preceding attribute byte.

The value of **fieldLen** for numeric fields must be great enough to display the largest number that can be held in the field, plus (if the number has decimal places) a decimal point. The value of **fieldLen** for a field of type CHAR, DBCHAR, MBCHAR, or UNICODE must be large enough to account for the double-byte characters, as well as any shift-in/shift-out characters.

The default **fieldLen** is the number of bytes needed to display the largest number possible for the primitive type, including all formatting characters.

Related concepts
"Overview of EGL properties" on page 60

Related reference
"Form part in EGL source format" on page 497

fill

The **fill** property indicates whether the user is required to enter data in each field position. Valid values are *no* (the default) and *yes*.

Related concepts
"Text forms" on page 148

Related reference
"Validation properties" on page 63
"validationFailed()" on page 767
"DataTable part in EGL source format" on page 462
"verifyChkDigitMod10()" on page 885
"verifyChkDigitMod11()" on page 886

fillCharacter

The **fillCharacter** property indicates what character fills unused positions in a text or print form or in PageHandler data. In addition, the property changes the effect of *set field full*, as described in *set*. The effect of this property is only at output.

The default is a space for numbers and a 0 for hex items. The default for character types depends on the medium:

- In text or print forms, the default is an empty string
- For PageHandler data, the default is blank for data of type CHAR or MBCHAR

In PageHandlers, the value of **fillCharacter** must be a space (as is the default) for items of type DBCHAR or UNICODE.

help

The **help** property specifies the hover-help text that is displayed when the user places the cursor over the input field. The value you assign is used as a default when you place the EGL field (or a record that includes the EGL field) on the Web Page in Page Designer.

The value of this property is a string literal.

Related concepts
"Overview of EGL properties" on page 60
"PageHandler" on page 180

Related tasks
"Associating an EGL record with a Faces JSP" on page 185
"Creating an EGL pageHandler part" on page 177
"Creating an EGL field and associating it with a Faces JSP" on page 184
"Using the Quick Edit view for PageHandler code" on page 187

Related reference
"PageHandler field properties" on page 665
"PageHandler part properties" on page 663
"PageHandler part in EGL source format" on page 659
"Page Designer support for EGL" on page 178

highlight

The **highlight** property specifies the special effect (if any) with which to display the field. Valid values are as follows:

blink
Causes the text to blink repeatedly. This value is available only for COBOL output (but not in the EGL Debugger), and support varies by emulator.

noHighLight (the default)
Indicates that no special effect is to occur; specifically, no blink, reverse, or underline. This value and *underline* are the only ones available for Java output.

reverse
Reverses the text and background colors, so that (for example) if the display has a dark background with light letters, the background becomes light and the text becomes dark. This value is available only for COBOL output.

underline
Places an underline at the bottom of the field.This value and *noHighLight* are the only ones available for Java output.

Related concepts
"Enumerations in EGL" on page 471
"Overview of EGL properties" on page 60

Related reference
"Field-presentation properties" on page 62

inputRequired

The **inputRequired** property indicates whether the user is required to place data in the field. Valid values are *no* (the default) and *yes*.

If the user does not place data in the field when the property value is *yes*, EGL run time displays a message, as described in relation to the field property **inputRequiredMsgKey**.

Related concepts
"Text forms" on page 148

Related reference
"Validation properties" on page 63
"validationFailed()" on page 767
"DataTable part in EGL source format" on page 462
"verifyChkDigitMod10()" on page 885
"verifyChkDigitMod11()" on page 886

inputRequiredMsgKey

The property **inputRequiredMsgKey** identifies the message that is displayed if the field property **inputRequired** is set to *yes* and the user fails to place data into the field.

The *message table* (the data table that contains the message) is identified in the program property **msgTablePrefix**. For details on the data-table name, see *DataTable part in EGL source format*.

The value of **inputRequiredMsgKey** is a string or literal that matches an entry of the first column in the message table.

If a numeric key is used with a message table that expects a character key, the number is converted to a character string. If a string literal is used with a message table that expects a numeric key, the value in the string must be a signed or unsigned integer.

Related concepts
"Text forms" on page 148

Related reference
"Validation properties" on page 63
"validationFailed()" on page 767
"DataTable part in EGL source format" on page 462
"verifyChkDigitMod10()" on page 885
"verifyChkDigitMod11()" on page 886

intensity

The **intensity** property specifies the strength of the displayed font. Valid values are as follows:

normalIntensity (the default)
Sets the field to be visible, without boldface.

bold
Causes the text to appear in boldface.

dim
Causes the text to appear with a lessened intensity, as appropriate when an input field is disabled.

invisible
Removes any indication that the field is on the form.

Related concepts
"Enumerations in EGL" on page 471
"Overview of EGL properties" on page 60

Related reference
"Field-presentation properties" on page 62

isBoolean

The **isBoolean** property (formerly the **boolean** property) indicates that the field represents a Boolean value. The property restricts the valid field values and is useful in text and print forms and in PageHandlers, for input or output.

On a Web page associated with an EGL PageHandler, a boolean item is represented by a check box. On a form, the situation is as follows:

- The value of a numeric field is 0 (for false) or 1 (for true).
- The value of a character field is represented by a word or subset of a word that is national-language dependent. In English, for example, a boolean field of three or more characters has the value *yes* (for true) or *no* (for false), and a one-character boolean field value has the truncated value *y* or *n*.

  In Java programs, the specific character values for *yes* and *no* are determined by the locale.

isDecimalDigit

The **isDecimalDigit** property determines whether to check that the input value includes only decimal digits, which are as follows:

```
0123456789
```

Valid values are *no* (the default) and *yes*.

This property applies only to character fields.

Related concepts
"Text forms" on page 148

Related reference
"Validation properties" on page 63
"validationFailed()" on page 767
"DataTable part in EGL source format" on page 462
"verifyChkDigitMod10()" on page 885
"verifyChkDigitMod11()" on page 886

isHexDigit

The **isHexDigit** property determines whether to check that the input value includes only hexadecimal digits, which are as follows:

```
0123456789abcdefABCDEF
```

Valid values are *no* (the default) and *yes*.

This property applies only to character fields.

Related concepts
"Text forms" on page 148

Related reference
"Validation properties" on page 63
"validationFailed()" on page 767
"DataTable part in EGL source format" on page 462
"verifyChkDigitMod10()" on page 885
"verifyChkDigitMod11()" on page 886

isNullable

The property **isNullable** indicates whether the item can be set to null, as is appropriate if the table column associated with the item can be set to NULL. Valid values are *yes* (the default) and *no*.

For a given item in an SQL record, the following features are available only if **isNullable** is set to *yes*:

- Your program can accept a NULL value from the database into the item.
- Your program can use a **set** statement to null the item, as described in *set*. The effect is also to initialize the item, as described in *Data initialization*.
- Your program can use an **if** statement to test whether the item is set to null.
- Your COBOL program can use an **if** statement to test whether the data received from the database was truncated. This feature is available in your Java program regardless of the value of **isNullable**.

Related concepts
"Compatibility with VisualAge Generator" on page 428
"Record types and properties" on page 126
"SQL support" on page 213
"Fixed structure" on page 24
"Typedef" on page 25

Related tasks
"Retrieving SQL table data" on page 235

Related reference
"Field-presentation properties" on page 62
"add" on page 544
"close" on page 551
"Data initialization" on page 459
"delete" on page 554
"execute" on page 557
"get" on page 567
"get next" on page 579
"open" on page 598
"prepare" on page 611
"Primitive types" on page 31
"Record and file type cross-reference" on page 716
"replace" on page 613
"set" on page 617
"SQL data codes and EGL host variables" on page 723
"terminalID" on page 913
"VAGCompatibility" on page 390

isReadOnly

The property **isReadOnly** indicates whether the item and related column should be omitted from the default SQL statements that write to the database or include a FOR UPDATE OF clause. The default value is *no*; but EGL treats the structure item as ″read only″ in these situations:

- The property **key** of the SQL record indicates that the column that is associated with the structure item is a key column; or
- The SQL record part is associated with more than one table; or
- The SQL column name is an expression.

Related concepts
“Compatibility with VisualAge Generator” on page 428
“Record types and properties” on page 126
“SQL support” on page 213
“Fixed structure” on page 24
“Typedef” on page 25

Related tasks
“Retrieving SQL table data” on page 235

Related reference
“Field-presentation properties” on page 62
“add” on page 544
“close” on page 551
“Data initialization” on page 459
“delete” on page 554
“execute” on page 557
“get” on page 567
“get next” on page 579
“open” on page 598
“prepare” on page 611
“Primitive types” on page 31
“Record and file type cross-reference” on page 716
“replace” on page 613
“set” on page 617
“SQL data codes and EGL host variables” on page 723
“terminalID” on page 913
“VAGCompatibility” on page 390

lineWrap

The EGL property **lineWrap** indicates whether text can be wrapped onto a new line whenever wrapping is necessary to avoid truncating text.

Valid values are of the enumeration **lineWrapType**:

character (the default)
The text in a field will not be split at a white space.

compress
The text in a field of type ConsoleField will be split at a white space; but when the user leaves the field (either by navigating to another field or by pressing Esc), any extra spaces that are used to wrap text are removed. This value is valid only in console fields.

word
If possible, the text in a field will be split at a white space.

The property **lineWrap** is available in DataItem parts and is meaningful for fields that appear in the following contexts:

- Console forms
- Print forms
- Text forms
- Web pages

The property affects input and output.

Related concepts
"Enumerations in EGL" on page 471
"Overview of EGL properties" on page 60

Related reference
"Formatting properties" on page 62

lowerCase

The **lowerCase** property indicates whether to set alphabetic characters to lower case in the user's single-byte character input. Values are as follows:

no (the default)
Do not set the user's input to lower case.

yes
Set the user's input to lower case.

masked

The **masked** property indicates whether a user-entered character will or will not be displayed. This property is used for entering passwords. Values are as follows:

no (the default)
The user-entered character will be displayed.

yes
The user-entered character will not be displayed.

maxLen

The property **maxLen** specifies the maximum length of field text that is written to the database column. Whenever possible, the default value for this property is the length of the field; but if the field is of type STRING, no default value exists.

In relation to EGL-generated COBOL code, the following statements apply:

- The property is required for fields of type STRING.
- In addition to affecting output, the property **maxLen** specifies the length of the input buffer that is allocated for reading a column value for the database. During a database read, the column value is truncated if that value is longer than the specified maxLen.

  Specify maxlen to be equal to the length defined for the column to make sure you get the entire value from the column.

Related concepts
"Compatibility with VisualAge Generator" on page 428
"Record types and properties" on page 126

"SQL support" on page 213
"Fixed structure" on page 24
"Typedef" on page 25

Related tasks
"Retrieving SQL table data" on page 235

Related reference
"Field-presentation properties" on page 62
"add" on page 544
"close" on page 551
"Data initialization" on page 459
"delete" on page 554
"execute" on page 557
"get" on page 567
"get next" on page 579
"open" on page 598
"prepare" on page 611
"Primitive types" on page 31
"Record and file type cross-reference" on page 716
"replace" on page 613
"set" on page 617
"SQL data codes and EGL host variables" on page 723
"terminalID" on page 913
"VAGCompatibility" on page 390

minimumInput

The **minimumInput** property indicates the minimum number of characters that the user is required to place in the field, if the user places any data in the field. The default is *0*.

If the user places fewer than the minimum number of characters, EGL run time displays a message, as described in relation to the field property **minimumInputMsgKey**.

Related concepts
"Text forms" on page 148

Related reference
"Validation properties" on page 63
"validationFailed()" on page 767
"DataTable part in EGL source format" on page 462
"verifyChkDigitMod10()" on page 885
"verifyChkDigitMod11()" on page 886

minimumInputMsgKey

The property **minimumInputMsgKey** identifies the message that is displayed if the user acts as follows:

- Places data in the field; and
- Places fewer characters than the value specified in the property **minimumInputRequired**.

The *message table* (the table that contains the message) is identified in the program property **msgTablePrefix**. For details on the table name, see *DataTable part in EGL source format*.

The value of **minimumInputMsgKey** is a string or literal that matches an entry of the first column in the message table.

If a numeric key is used with a message table that expects a character key, the number is converted to a character string. If a string literal is used with a message table that expects a numeric key, the value in the string must be a signed or unsigned integer.

Related concepts
"Text forms" on page 148

Related reference
"Validation properties" on page 63
"validationFailed()" on page 767
"DataTable part in EGL source format" on page 462
"verifyChkDigitMod10()" on page 885
"verifyChkDigitMod11()" on page 886

modified

Indicates whether the program will consider the field to have been modified, regardless of whether the user changed the value. For details, see *Modified data tag and modified property*.

The default is *no*.

Related concepts
"Modified data tag and modified property" on page 150
"Overview of EGL properties" on page 60

Related reference
"Form part in EGL source format" on page 497

needsSOSI

The **needsSOSI** property is used only for a *multibyte field* (a field of type MBCHAR) and indicates whether EGL does a special check when the user enters data of type MBCHAR on an ASCII device. Valid values are *yes* (the default) and *no*. The check determines whether the input can be converted properly to the host SO/SI format.

The property is useful because, during conversion, trailing blanks are deleted from the end of a multibyte string to allow for insertion of SO/SI delimiters around each substring of double-byte characters. For a proper conversion, the form field must have at least two blanks for each double-byte string in the multibyte value.

If **needsSOSI** is set to *no*, the user can fill the input field, in which case the conversion truncates data without warning.

If **needsSOSI** is set to *yes*, however, the result is as follows when the user enters multibyte data:

- The value is accepted as is because enough blanks are provided; or

- The value is truncated, and the user receives a warning message.

Set **needsSOSI** to *yes* if the user's ASCII input of multibyte data may be used on a z/OS or iSeries system.

Related concepts
"Text forms" on page 148

Related reference
"Validation properties" on page 63
"validationFailed()" on page 767
"DataTable part in EGL source format" on page 462
"verifyChkDigitMod10()" on page 885
"verifyChkDigitMod11()" on page 886

newWindow

The property **newWindow** indicates whether to use a new browser window when the EGL run time presents a Web page in response to the activity identified in the **action** property.

Values are of the enumeration **Boolean**:

No (the default)
The current browser window is used to display the page

Yes
A new browser window is used.

If the **action** property is not specified, the current browser window is used to display the page.

Related concepts
"Enumerations in EGL" on page 471
"Overview of EGL properties" on page 60
"PageHandler" on page 180

Related tasks
"Associating an EGL record with a Faces JSP" on page 185
"Binding a JavaServer Faces command component to an EGL PageHandler" on page 186
"Creating an EGL pageHandler part" on page 177
"Creating an EGL field and associating it with a Faces JSP" on page 184
"Using the Quick Edit view for PageHandler code" on page 187

Related reference
"action" on page 670
"PageHandler field properties" on page 665
"PageHandler part properties" on page 663
"PageHandler part in EGL source format" on page 659
"Page Designer support for EGL" on page 178

numElementsItem

When set on a structure-field array, the property **numElementsItem** identifies a PageHandler field whose runtime value specifies the number of array elements to display. The property is used only for output and is meaningful only if set on a fixed-record structure field that has an occurs value greater than 1.

The value of **numElementsItem** is a string literal that identifies the name of a PageHandler field. The property is not valid for a dynamic array, which includes an indicator of how many elements are in use; for details, see *Arrays*.

Related concepts
"Fixed record parts" on page 125
"Overview of EGL properties" on page 60
"PageHandler" on page 180

Related tasks
"Associating an EGL record with a Faces JSP" on page 185
"Creating an EGL pageHandler part" on page 177
"Creating an EGL field and associating it with a Faces JSP" on page 184
"Using the Quick Edit view for PageHandler code" on page 187

Related reference
"Arrays" on page 69
"PageHandler field properties" on page 665
"PageHandler part properties" on page 663
"PageHandler part in EGL source format" on page 659
"Page Designer support for EGL" on page 178

numericSeparator

The **numericSeparator** property indicates whether to place a character in a number that has an integer component of more than 3 digits. If the numeric separator is a comma, for example, one thousand is shown as *1,000* and one million is shown as *1,000,000*. Values are as follows:

no (the default)
Do not use a numeric separator.

yes
Use a numeric separator.

In Java code, the default is determined by the machine locale. In COBOL code, the default is determined by the national language option.

outline

The **outline** property lets you draw lines at the edges of fields on any device that supports double-byte characters. Valid values are as follows:

box
Draw lines to create a box around the field content

noOutline (the default)
Draw no lines

In addition, you can specify any or all of the components of a box. In this case, place brackets around one or more values, with each value separated from the next by a comma, as in this example:

```
outline = [left, over, right, under]
```

The partial values are as follows:

left
Draw a vertical line at the left edge of the field

over
Draw a horizontal line at the top edge of the field

right
Draw a vertical line at the right edge of the field

under
Draw a horizontal line at the bottom edge of the field

The content of each form field is preceded by an attribute byte. Be aware that you cannot place an attribute byte in the last column of a form and expect an outline value to appear in the next column, which is beyond the form's edge. (The field does not wrap to the next line.) Similarly, you cannot place an attribute byte in the first column of a form and expect the outline value to appear in that column; the outline value can appear only in the next column.

Related concepts
"Enumerations in EGL" on page 471
"Overview of EGL properties" on page 60

Related reference
"Field-presentation properties" on page 62

pattern

Matches the user entered text against a specified pattern, for validation.

Related concepts
"Overview of EGL properties" on page 60

Related reference
"Form part in EGL source format" on page 497

persistent

The property **persistent** indicates whether the field is included in the implicit SQL statements generated for the SQL record. If the value is *yes*, an error occurs at run time in this case:

- Your code relies on an implicit SQL statement; and
- No column matches the value of the field-specific **column** property. (The default value is the field name.)

Set **persistent** to *no* if you want to associate a temporary program variable with an SQL row without having a corresponding column for the variable in the database. You might want a variable, for example, to indicate whether the program has modified the row.

For details on implicit SQL statements, see *SQL support*.

Related concepts
"Compatibility with VisualAge Generator" on page 428
"Record types and properties" on page 126
"SQL support" on page 213
"Fixed structure" on page 24
"Typedef" on page 25

Related tasks
"Retrieving SQL table data" on page 235

Related reference
"Field-presentation properties" on page 62
"add" on page 544
"close" on page 551
"Data initialization" on page 459
"delete" on page 554
"execute" on page 557
"get" on page 567
"get next" on page 579
"open" on page 598
"prepare" on page 611
"Primitive types" on page 31
"Record and file type cross-reference" on page 716
"replace" on page 613
"set" on page 617
"SQL data codes and EGL host variables" on page 723
"terminalID" on page 913
"VAGCompatibility" on page 390

protect

Specifies whether the user can access the field. Valid values are as follows:

no (the default for variable fields)
Sets the field so that the user can overwrite the value in it.

skip (the default for constant fields)
Sets the field so that the user cannot overwrite the value in it. In addition, the cursor skips the field in either of these cases:

- The user is working on the previous field in the tab order and either presses **Tab** or fills that previous field with content; or
- The user is working on the next field in the tab order and presses **Shift Tab**.

yes
Sets the field so that the user cannot overwrite the value in it.

Related concepts
"Overview of EGL properties" on page 60

Related reference
"Form part in EGL source format" on page 497

selectFromListItem

The property **selectFromListItem** identifies the array or DataTable column from which the user selects a value or values, which are then transferred to the array or primitive field being declared. The value you assign to **selectFromListItem** is used as a default when you place the array or primitive field on the Web Page in Page Designer.

The value of property **selectFromListItem** is a string literal that identifies the source array or DataTable column.

If you specify this property when declaring an array, the user is allowed to select multiple values. If you specify this property when declaring a primitive field, the user can select only one value.

Any value received from the user must correspond to one of these types:

- The content of the array element or DataTable column that the user selected; or
- An array or DataTable index, which is an integer that identifies which element or column was selected. The index ranges from 1 to the number of elements available.

The property **selectType** indicates the type of value to receive, whether the content selected by the user or an index into an array or column.

Related concepts
"DataTable" on page 137
"Overview of EGL properties" on page 60
"PageHandler" on page 180

Related tasks
"Associating an EGL record with a Faces JSP" on page 185
"Creating an EGL pageHandler part" on page 177
"Creating an EGL field and associating it with a Faces JSP" on page 184
"Using the Quick Edit view for PageHandler code" on page 187

Related reference
"Arrays" on page 69
"PageHandler field properties" on page 665
"PageHandler part properties" on page 663
"PageHandler part in EGL source format" on page 659
"Page Designer support for EGL" on page 178
"selectType"

selectType

The property **selectType** indicates the kind of value that is retrieved into the array or primitive field being declared. The value you assign is used as a default when you place the array or primitive field on the Web Page in Page Designer.

The value is of enumeration **selectTypeKind**:

index (the default)
The array or primitive field being declared will receive indexes in response to a user selection. In this case, the array or primitive field must be of a numeric type.

value
The array or primitive field being declared will receive the user's selection value. In this case, the item can be of any type.

For background information, see the property **selectFromListItem**.

Related concepts
"DataTable" on page 137
"Overview of EGL properties" on page 60
"PageHandler" on page 180

Related tasks
"Associating an EGL record with a Faces JSP" on page 185
"Creating an EGL pageHandler part" on page 177
"Creating an EGL field and associating it with a Faces JSP" on page 184
"Using the Quick Edit view for PageHandler code" on page 187

Related reference
"Arrays" on page 69
"PageHandler field properties" on page 665
"PageHandler part properties" on page 663
"PageHandler part in EGL source format" on page 659
"Page Designer support for EGL" on page 178
"selectFromListItem" on page 691

sign

The **sign** property indicates the position in which a positive (+) or negative (-) sign is displayed when a number is placed in the field, whether from user input or from the program. Values are as follows:

none
A sign is not displayed.

leading
The default: a sign is displayed to the left of the first digit in the number, with the exact position of the sign determined by the **zeroFormat** property (described later).

trailing
A sign is displayed immediately to the right of the last digit in the number.

sqlDataCode

The value of property **sqlDataCode** is a number that identifies the SQL data type that is associated with the record item. The data code is used by the database management system when you access that system at declaration time, validation time, or generated-program run time.

The property **sqlDataCode** is available only if you have set up your environment for VisualAge Generator compatibility. For details, see *Compatibility with VisualAge Generator*.

The default value depends on the primitive type and length of the record item, as shown in the next table. For other details, see *SQL data codes*.

EGL primitive type	Length	SQL data code
BIN	4	501
	9	497
CHAR	<=254	453
	>254 and <=4000	449
	>4000	457
DBCHAR	<=127	469
	>127 and <=2000	465
	>2000	473
DECIMAL	any	485

EGL primitive type	Length	SQL data code
HEX	any	481
UNICODE	<=127	469
	>127 and <=2000	465
	>2000	473

Related concepts
"Compatibility with VisualAge Generator" on page 428
"Record types and properties" on page 126
"SQL support" on page 213
"Fixed structure" on page 24
"Typedef" on page 25

Related tasks
"Retrieving SQL table data" on page 235

Related reference
"Field-presentation properties" on page 62
"add" on page 544
"close" on page 551
"Data initialization" on page 459
"delete" on page 554
"execute" on page 557
"get" on page 567
"get next" on page 579
"open" on page 598
"prepare" on page 611
"Primitive types" on page 31
"Record and file type cross-reference" on page 716
"replace" on page 613
"set" on page 617
"SQL data codes and EGL host variables" on page 723
"terminalID" on page 913
"VAGCompatibility" on page 390

sqlVariableLen

The value of property **sqlVariableLen** (formerly the **sqlVar** property) indicates whether trailing blanks and nulls in a character field are truncated before the EGL run time writes the data to an SQL database. This property has no effect on non-character data.

Specify *yes* if the corresponding SQL table column is a varchar or vargraphic SQL data type.

Related concepts
"Compatibility with VisualAge Generator" on page 428
"Record types and properties" on page 126
"SQL support" on page 213
"Fixed structure" on page 24
"Typedef" on page 25

Related tasks
"Retrieving SQL table data" on page 235

Related reference
"Field-presentation properties" on page 62
"add" on page 544
"close" on page 551
"Data initialization" on page 459
"delete" on page 554
"execute" on page 557
"get" on page 567
"get next" on page 579
"open" on page 598
"prepare" on page 611
"Primitive types" on page 31
"Record and file type cross-reference" on page 716
"replace" on page 613
"set" on page 617
"SQL data codes and EGL host variables" on page 723
"terminalID" on page 913
"VAGCompatibility" on page 390

timeFormat

The **timeFormat** property identifies the format for times.

Valid values are as follows:

"pattern"
The value of *pattern* consists of a set of characters as described in *Date, time, and timestamp format specifiers*.

Characters may be dropped from the beginning or end of a complete time specification, but not from the middle.

defaultTimeFormat
The default value in a Java environment is set by the Java locale.

eurTimeFormat
The pattern *HH.mm.ss*, which is the IBM European standard time format.

isoTimeFormat
The pattern *HH.mm.ss*, which is the time format specified by the International Standards Organization (ISO).

jisTimeFormat
The pattern *HH:mm:ss*, which is the Japanese Industrial Standard time format.

usaTimeFormat
The pattern *hh:mm AM*, which is the IBM USA standard time format.

The property is available in DataItem parts and is meaningful for fields that appear in the following contexts:

- Console forms
- Print forms
- Text forms
- Web pages

The property is used for both input and output, but not in the following cases:

- The field has decimal places, a currency symbol, a numeric separator, or a sign; or

- The field is of type DBCHAR, MBCHAR, or HEX; or
- The field is not long enough to contain a value that reflects the mask. For other details, see *Length considerations for times*.

Length considerations for times

In a form, the field length must match the length of the time mask you specify. In a page field, the rules are as follows:

- The item length must be sufficient for the time mask you specify but can be longer
- In the case of a numeric item, the separator characters are excluded from the length calculation.

I/O considerations for times

Data entered into a variable field is checked to ensure that the time was entered in the format specified. The user does not need to enter the leading zeros for hours, minutes, and second, but can specify (for example) 8:15 instead of 08:15. The user who omits the separator characters, however, must enter all leading zeros.

A time stored in internal format is not recognized as a time, but simply as data. If a 6-character time field is moved to a character item of length 10, for example, EGL pads the destination field with blanks. When the 6-character value is presented on a form, however, the time is converted from its internal format, as appropriate.

Related concepts
"Java runtime properties" on page 327
"Overview of EGL properties" on page 60

Related reference
"Date, time, and timestamp format specifiers" on page 42
"Formatting properties" on page 62
"Java runtime properties (details)" on page 525

timeStampFormat

The **timeStampFormat** property identifies the format for timestamps that are displayed on a form or maintained in a PageHandler.

Valid values are as follows:

"pattern"
The value of *pattern* consists of a set of characters as described in *Date, time, and timestamp format specifiers*.

Characters may be dropped from the beginning or end of a complete timestamp specification, but not from the middle.

defaultTimeStampFormat
In a Java environment, the default is set by the Java locale.

db2TimestampFormat
The pattern *yyyy-MM-dd-HH.mm.ss.ffffff*, which is the IBM DB2 default timestamp format.

odbcTimestampFormat
The pattern *yyyy-MM-dd HH:mm:ss.ffffff*, which is the ODBC timestamp format.

Related concepts
"Java runtime properties" on page 327
"Overview of EGL properties" on page 60

Related reference
"Date, time, and timestamp format specifiers" on page 42
"Java runtime properties (details)" on page 525

typeChkMsgKey

The property **typeChkMsgKey** identifies the message that is displayed if the input data is not appropriate for the field type.

The *message table* (the table that contains the message) is identified in the program property **msgTablePrefix**. For details on the table name, see *DataTable part in EGL source format*.

The value of **typeChkMsgKey** is a string or literal that matches an entry of the first column in the message table.

If a numeric key is used with a message table that expects a character key, the number is converted to a character string. If a string literal is used with a message table that expects a numeric key, the value in the string must be a signed or unsigned integer.

Related concepts
"Text forms" on page 148

Related reference
"Validation properties" on page 63
"validationFailed()" on page 767
"DataTable part in EGL source format" on page 462
"verifyChkDigitMod10()" on page 885
"verifyChkDigitMod11()" on page 886

upperCase

The **upperCase** property indicates whether to set alphabetic characters to upper case in the user's single-byte character input.

This property is useful in forms and in PageHandlers.

The values of **upperCase** are as follows:

No (the default)
Do not set the user's input to upper case.

Yes
Set the user's input to upper case.

validationOrder

The property **validationOrder** indicates when the field's validator function runs in relation to any other field's validator function. The property is important if the validation of one field depends on the previous validation of another.

The value is a literal integer.

Validation occurs first for any fields for which you specified a value for the property **validationOrder**, and the items with the lowest-numbered values are validated first. Validation then occurs for any items for which you did not specify a value for **validationOrder**, and the order of validation in this case is the order in which the fields are defined in the PageHandler.

Related concepts
"Overview of EGL properties" on page 60
"PageHandler" on page 180

Related tasks
"Creating an EGL pageHandler part" on page 177
"Creating an EGL field and associating it with a Faces JSP" on page 184
"Using the Quick Edit view for PageHandler code" on page 187

Related reference
"PageHandler field properties" on page 665
"PageHandler part properties" on page 663
"PageHandler part in EGL source format" on page 659
"Page Designer support for EGL" on page 178

validatorDataTable

The **validatorDataTable** property (formerly the **validatorTable** property) identifies a *validator table*, which is a dataTable part that acts as the basis of a comparison with user input. Use of a validator table occurs after the EGL run time does the elementary validation checks, if any. Those elementary checks are described in relation to the following properties:

- inputRequired
- isDecimalDigit
- isHexDigit
- minimumInput
- needsSOSI
- validValues

All checks precede use of the **validatorFunction** property, which specifies a validation function that does cross-value validation.

You can specify a validator table that is of any of the following types, as described in *DataTable part in EGL source format*:

matchInvalidTable
Indicates that the user's input must be different from any value in the first column of the data table.

matchValidTable
Indicates that the user's input must match a value in the first column of the data table.

rangeChkTable
Indicates that the user's input must match a value that is between the values in the first and second column of at least one data-table row. (The range is inclusive; the user's input is also valid if it matches a value in the first or second column of any row.)

If validation fails, the displayed message is based on the value of the property **validatorDataTableMsgKey**.

Related concepts
"Text forms" on page 148

Related reference
"Validation properties" on page 63
"validationFailed()" on page 767
"DataTable part in EGL source format" on page 462
"verifyChkDigitMod10()" on page 885
"verifyChkDigitMod11()" on page 886

validatorDataTableMsgKey

The property **validatorDataTableMsgKey** (formerly the **validatorTableMsgKey** property) identifies the message that is displayed if the user provides data that does not correspond to the requirements of the *validator table*, which is the table specified in the property **validatorDataTable**.

The *message table* (the table that contains the message) is identified in the program property **msgTablePrefix**. For details on the message-table name, see *DataTable part in EGL source format*.

The value of **validatorDataTableMsgKey** is a string or literal that matches an entry of the first column in the message table.

If a numeric key is used with a message table that expects a character key, the number is converted to a character string. If a string literal is used with a message table that expects a numeric key, the value in the string must be a signed or unsigned integer.

Related concepts
"Text forms" on page 148

Related reference
"Validation properties" on page 63
"validationFailed()" on page 767
"DataTable part in EGL source format" on page 462
"verifyChkDigitMod10()" on page 885
"verifyChkDigitMod11()" on page 886

validatorFunction

The **validatorFunction** property (formerly the **validator** property) identifies a validator function, which is logic that runs after the EGL run time does the elementary validation checks, if any. Those checks are described in relation to the following properties:

- inputRequired
- isDecimalDigit
- isHexDigit
- minimumInput
- needsSOSI
- validValues

The elementary checks precede use of the validator table (as described in relation to the **validatorDataTable** property), and all checks precede use of the **validatorFunction** property. This order of events is important because the validator function can do cross-field checking, and such checking often requires valid field values.

The value of **validatorFunction** is a validator function that you write. You code that function with no parameters and such that, if the function detects an error, it requests the re-display of the form by invoking ConverseLib.validationFailed.

If validation fails when you specify one of the two system functions, the displayed message is based on the value of the property **validatorFunctionMsgKey**. If validation fails when you specify a validator function of your own, however, the function does not use **validatorFunctionMsgKey**, but displays a message by invoking ConverseLib.validationFailed.

Related concepts
"Text forms" on page 148

Related reference
"Validation properties" on page 63
"validationFailed()" on page 767
"DataTable part in EGL source format" on page 462
"verifyChkDigitMod10()" on page 885
"verifyChkDigitMod11()" on page 886

validatorFunctionMsgKey

The property **validatorFunctionMsgKey** (formerly the **validatorMsgKey** property) identifies a message that is displayed in the following case:

- The **validatorFunction** property indicates use of sysLib.verifyChkDigitMod10 or sysLib.verifyChkDigitMod11; and
- The specified function indicates that the user's input is in error.

The *message table* (the table that contains the message) is identified in the program property **msgTablePrefix**. For details on the table name, see *DataTable part in EGL source format*.

The value of **validatorFunctionMsgKey** is a string or literal that matches an entry of the first column in the message table.

If a numeric key is used with a message table that expects a character key, the number is converted to a character string. If a string literal is used with a message table that expects a numeric key, the value in the string must be a signed or unsigned integer.

Related concepts
"Text forms" on page 148

Related reference
"Validation properties" on page 63
"validationFailed()" on page 767
"DataTable part in EGL source format" on page 462
"verifyChkDigitMod10()" on page 885
"verifyChkDigitMod11()" on page 886

validValues

The **validValues** property (formerly the **range** property) indicates a set of values that are valid for user input. The property is used for numeric or character fields. The format of the property is as follows:

```
validValues = arrayLiteral
```

arrayLiteral

An array literal of singular and two-value elements, as in the following examples:

```
validValues = [ [1,3], 5, 12 ]
validValues = [ "a", ["bbb", "i"]]
```

Each singular element contains a valid value. Each two-value element contains a range:

- For numbers, the leftmost value is the lowest that is valid, the rightmost is the highest. In the previous example, the values 1, 2, and 3 are valid for a field of type INT.
- For character fields, user input is compared against the range of values, for the number of characters for which a comparison is possible. For example, the range [″a″, ″c″] includes (as valid) any input whose first character is ″a″, ″b″, or ″c″. Although the string ″cat″ is greater than ″c″ in collating sequence, ″cat″ is valid input.

  The general rule is as follows: if the first value in the range is called *lowValue* and the second is called *highValue*, the user's input is valid if *any* of these tests are fulfilled:

  – User input is equal to *lowValue* or *highValue*
  – User input is greater than *lowValue* and less than *highValue*
  – The initial series of input characters matches the initial series of characters in *lowValue*, for as long as a comparison is possible
  – The initial series of input characters matches the initial series of characters in *highValue*, for as long as a comparison is possible

Additional examples are as follows:

```
// valid values are 1, 2, 3, 5, 7, 9, and 11
validValues = [[1, 3], 5, 7, 11]

// valid values are the letters "a" and "z"
validValues = ["a", "z"]

// valid values are any string beginning with "a"
validValues = [["a", "a"]]

// valid values are any string
// beginning with a lowercase letter
validValues = [["a", "z"]]
```

If the user's input is outside the specified range, EGL run time displays a message, as described in relation to the field property **validValuesMsgKey**.

Related concepts

"Text forms" on page 148

Related reference

"Validation properties" on page 63
"validationFailed()" on page 767

"DataTable part in EGL source format" on page 462
"verifyChkDigitMod10()" on page 885
"verifyChkDigitMod11()" on page 886

validValuesMsgKey

The property **validValuesMsgKey** (formerly the **rangeMsgKey** property) identifies the message that is displayed if the field property **validValues** is set and the user places out-of-range data into the field.

The *message table* (the table that contains the message) is identified in the program property **msgTablePrefix**. For details on the table name, see *DataTable part in EGL source format*.

The value of **validValuesMsgKey** is a string or literal that matches an entry of the first column in the message table.

If a numeric key is used with a message table that expects a character key, the number is converted to a character string. If a string literal is used with a message table that expects a numeric key, the value in the string must be a signed or unsigned integer.

This property applies only to numeric fields.

Related concepts
"Text forms" on page 148

Related reference
"DataTable part in EGL source format" on page 462
"validationFailed()" on page 767
"Validation properties" on page 63
"verifyChkDigitMod10()" on page 885
"verifyChkDigitMod11()" on page 886

value

The property **value** identifies a string literal that is displayed as the field content when a Web page is displayed. That literal is used as a default when you place an EGL field on the Web Page in Page Designer.

Related concepts
"Overview of EGL properties" on page 60
"PageHandler" on page 180

Related tasks
"Associating an EGL record with a Faces JSP" on page 185
"Creating an EGL field and associating it with a Faces JSP" on page 184
"Creating an EGL pageHandler part" on page 177
"Using the Quick Edit view for PageHandler code" on page 187

Related reference
"PageHandler field properties" on page 665
"PageHandler part properties" on page 663
"PageHandler part in EGL source format" on page 659
"Page Designer support for EGL" on page 178

zeroFormat

The **zeroFormat** property specifies how zero values are displayed in numeric fields but not in fields of type MONEY. This property is affected by the **numeric separator**, **currency**, and **fillCharacter** properties. The values of **zeroFormat** are as follows:

Yes

A zero value is displayed as the number zero, which can be expressed with decimal points (0.00 is an example, if the item is defined with two decimal places) and with currency symbols and character separators ($000,000.00 is an example, depending on the values of the **currency** and **numericSeparator** properties). The following rules apply when the value of the property **zeroFormat** is *yes*:

- If the *fill character* (the value of the **fillCharacter** property) is 0, the data is formatted with the character 0
- If the fill character is a null, the data is left-justified
- If the fill character is a blank, the data is right-justified
- If the fill character is an asterisk (*), asterisks are used as the left-side fillers instead of blanks

No

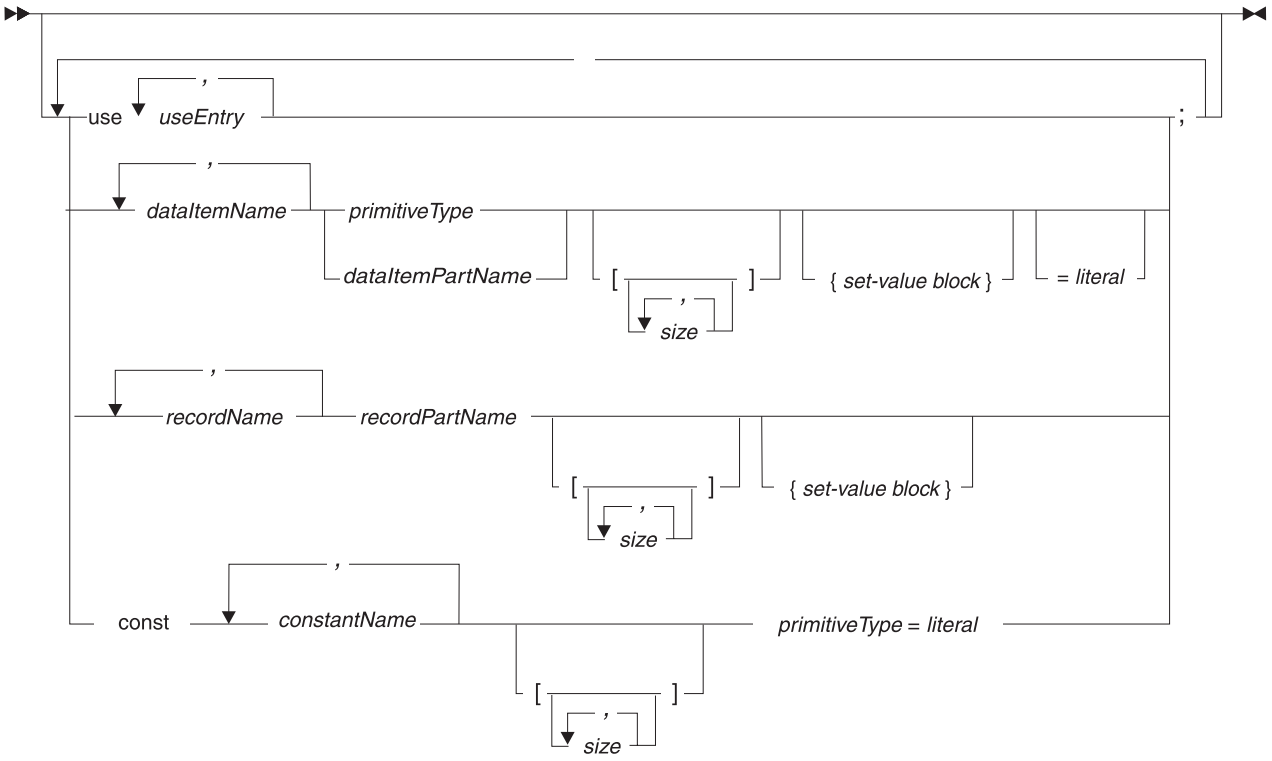
A zero value is displayed as a series of the fill character.

Related reference

"Java runtime properties (details)" on page 525
"set" on page 617
"currencySymbol" on page 367
"Date, time, and timestamp format specifiers" on page 42

Program data other than parameters

The syntax diagram for program data is as follows:



use *useEntry*
Provides easier access to a dataTable or library, and is needed to access to forms in a formGroup. For details, see *Use declaration*.

dataItemName
Name of a primitive field. For the rules of naming, see *Naming conventions*.

primitiveType
The type of a primitive field or (in relation to an array) the primitive type of an array element. Depending on the type, the following information may be required:

- The parameter's length or (in relation to an array), the length of an array element. The length is an integer that represents the number of characters or digits in the memory area.
- For some numeric types, you may specify an integer that represents the number of places after the decimal point. The decimal point is not stored with the data.
- For an item of type INTERVAL or TIMESTAMP, you may specify a datetime mask, which assigns a meaning (such as ″year digit″) to a given position in the item value.

For details, see *Primitive types* and the topic for a given type.

dataItemPartName
The name of a dataItem part that is visible to the program. For details on visibility, see *References to parts*.

The part acts as a model of format, as described in *Typedef*.

size
Number of elements in the array. If you specify the number of elements, the array is initialized with that number of elements.

set-value block
For details, see *Overview of EGL properties* and *Set-value blocks*

recordName
Name of a record. For the rules of naming, see *Naming conventions*.

recordPartName
Name of a record part that is visible to the program. For details on visibility, see *References to parts*.

The part acts as a model of format, as described in *Typedef*.

const *constantName primitiveType*=*literal*
Name, type, and value of a constant. Specify a quoted string (for a character type); a number (for a numeric type); or an array of appropriately typed values (for an array). Examples are as follows:

```
const myString String = "Great software!";
const myArray  BIN[] = [36, 49, 64];
const myArray02 BIN[][] = [[1,2,3],[5,6,7]];
```

For the rules of naming, see *Naming conventions*.

Related concepts
"EGL projects, packages, and files" on page 13
"Overview of EGL properties" on page 60
"Parts" on page 17
"Program part" on page 130
"References to variables in EGL" on page 55
"Segmentation in text applications" on page 149
"Set-value blocks" on page 63
"Syntax diagram for EGL statements and commands" on page 733
"Typedef" on page 25

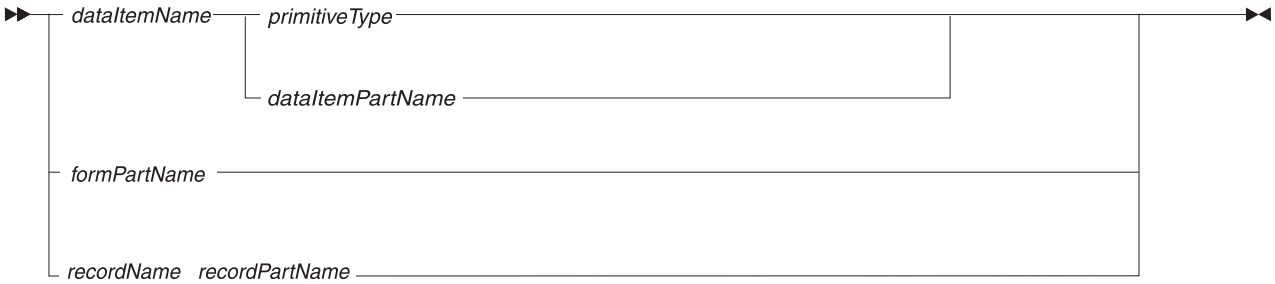
Related reference
"Arrays" on page 69
"Data initialization" on page 459
"DataItem part in EGL source format" on page 461
"DataTable part in EGL source format" on page 462
"EGL source format" on page 478
"EGL statements" on page 83
"forward" on page 566
"Function part in EGL source format" on page 513
"Indexed record part in EGL source format" on page 520
"Input form" on page 715
"Input record" on page 715
"INTERVAL" on page 39
"I/O error values" on page 522
"MQ record part in EGL source format" on page 642
"Naming conventions" on page 652
"Primitive types" on page 31
"Relative record part in EGL source format" on page 719
"Serial record part in EGL source format" on page 722
"SQL record part in EGL source format" on page 726

"TIMESTAMP" on page 41

"Use declaration" on page 930

Program parameters

The syntax diagram for a program parameter is as follows:



dataItemName
Name of a primitive field. For the rules of naming, see *Naming conventions*.

primitiveType
The type of a primitive field. Depending on the type, the following information may be required:

- The parameter's length, which is an integer that represents the number of characters or digits in the memory area.
- For some numeric types, you may specify an integer that represents the number of places after the decimal point. The decimal point is not stored with the data.
- For an item of type INTERVAL or TIMESTAMP, you may specify a datetime mask, which assigns a meaning (such as "year digit") to a given position in the item value.

dataItemPartName
The name of a dataItem part that is visible to the program. For details on visibility, see *References to parts*.

The part acts as a model of format, as described in *Typedef*.

formPartName
Name of a form.

The form must be accessible through a formGroup that is identified in one of the program's use declarations. A form accessed as a parameter cannot be displayed to the user, but can provide access to field values that are passed from another program.

For the rules of naming, see *Naming conventions*.

recordName
Name of a record or fixed record. For the rules of naming, see *Naming conventions*.

recordPartName
Name of a record part (or fixed-record part) that is visible to the program. For details on visibility, see *References to parts*.

The part acts as a model of format, as described in *Typedef*.

The following statements apply to input or output (I/O) against record parameters:

- A record passed from another program does not include record state such as the I/O error value *endOfFile*. Similarly, any change in the record state is not returned to the caller, so if you perform I/O against a record parameter, any tests on that record must occur before the program ends.
- Any I/O operation performed against the record uses the record properties specified for the parameter, not the record properties specified for the argument.
- For records of type indexedRecord, mqRecord, relativeRecord, or serialRecord, the file or message queue associated with the record declaration is treated as a run-unit resource rather than a program resource. Local record declarations share the same file (or queue) whenever the record property **fileName** (or **queueName**) has the same value. Only one physical file at a time can be associated with a file or queue name no matter how many records are associated with the file or queue in the run unit, and EGL enforces this rule by closing and reopening files as appropriate.

An arguments sent from another EGL program must be reference-compatible with the related parameter. For details, see *Reference compatibility in EGL*.

Related concepts
"Program part" on page 130
"References to parts" on page 20
"References to variables in EGL" on page 55
"Syntax diagram for EGL statements and commands" on page 733
"Typedef" on page 25

Related reference
"Arrays" on page 69
"Basic record part in EGL source format" on page 357
"DataItem part in EGL source format" on page 461
"EGL source format" on page 478
"Indexed record part in EGL source format" on page 520
"INTERVAL" on page 39
"Naming conventions" on page 652
"Primitive types" on page 31
"Reference compatibility in EGL" on page 718
"Relative record part in EGL source format" on page 719
"Serial record part in EGL source format" on page 722
"SQL record part in EGL source format" on page 726
"TIMESTAMP" on page 41

Program part in EGL source format

You declare a program part in an EGL file, which is described in *EGL source format*. When you write that file, do as follows:

- Include only those parts that are used exclusively by the program
- Do not include other primary parts (dataTable, library, program, or pageHandler)

The next example shows a called program part with two embedded functions, along with a stand-alone function and a stand-alone record part:

```
Program myProgram type basicProgram (employeeNum INT)
{
  includeReferencedFunctions = yes
}
```

```
  // program-global variables
    employees record_ws;
    employeeName char(20);


  // a required embedded function
  Function main()
    // initialize employee names
    recd_init();

    // get the correct employee name
    // based on the employeeNum passed
    employeeName = getEmployeeName(employeeNum);
  end

  // another embedded function
  Function recd_init()
    employees.name[1] = "Employee 1";
    employees.name[2] = "Employee 2";
  end
end

// stand-alone function
Function getEmployeeName(employeeNum INT) returns (CHAR(20))

  // local variable
  index BIN(4);
  index = 2;
  if (employeeNum > index)
    return("Error");
  else
    return(employees.name[employeeNum]);
  end

end

// record part that acts as a typeDef for employees
Record record_ws type basicRecord
  10 name CHAR(20)[2];
end
```

For other details, see the topic for a particular type of program.

Related concepts
"Parts" on page 17
"Program part" on page 130

Related reference

"Basic program in EGL source format"
"EGL source format" on page 478
"Function part in EGL source format" on page 513
"Text UI program in EGL source format" on page 710

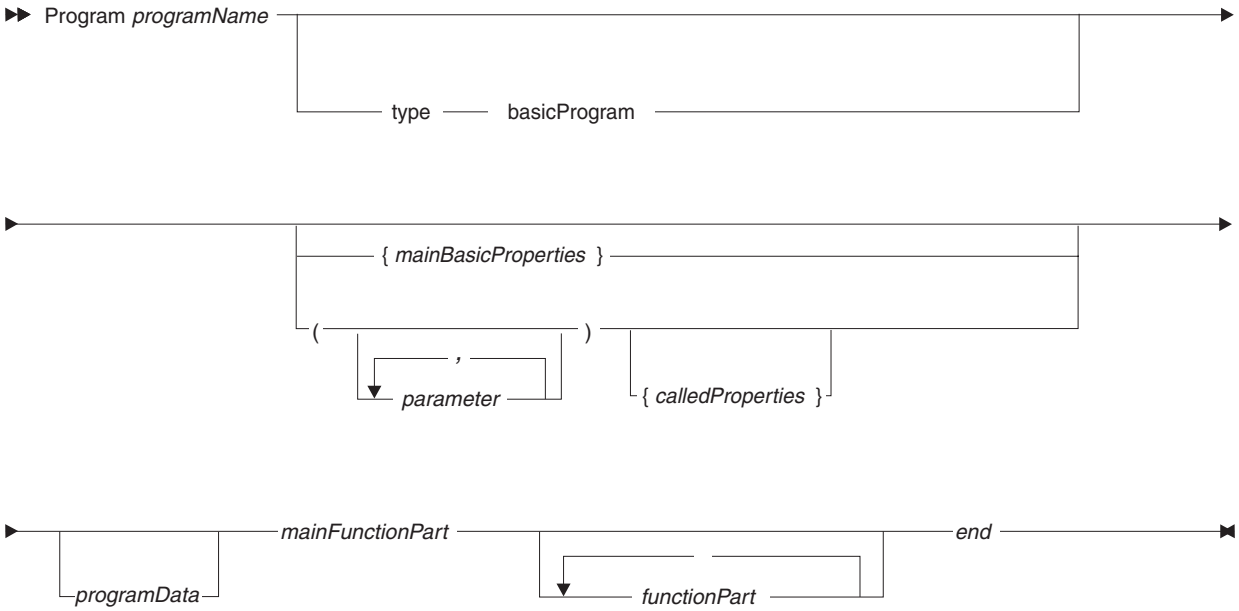
Basic program in EGL source format

An example of a basic program is as follows:

```
program myCalledProgram type basicProgram
  (buttonPressed int, returnMessage char(25))

 function main()
    returnMessage = "";
    if (buttonPressed == 1)
```

```
      returnMessage = "Message1";
    end

   if (buttonPressed == 2)
      returnMessage = "Message2";
    end
 end
end
```

The syntax diagram for a program part of type basicProgram is as follows:



Program *programPartName* **... end**
Identifies the part as a program part and specifies the name and type. If the program name is followed by a left parenthesis, the program is a called basic program.

If you do not set the **alias** property (as described later), the name of the generated program is either *programPartName* or, if you are generating COBOL, the first eight characters of *programPartName*.

For other rules, see *Naming conventions*.

mainBasicProperties
The properties for a main basic program are optional:

- **alias**
- **allowUnqualifiedItemReferences**
- **handleHardIOErrors**
- **includeReferencedFunctions**
- **inputRecord**
- **localSQLScope**
- **msgTablePrefix**
- **throwNrfEofExceptions**

For details, see *Program properties*.

parameter
Specifies the name of a parameter, which may be a data item, record, or form; or a dynamic array of records or data items. For rules, see *naming conventions*.

If the caller's argument is a variable (not a constant or literal), any changes to the parameter change the area of memory available to the caller.

Each parameter is separated from the next by a comma. For other details, see *Program parameters*.

calledProperties
The called properties are optional:

- **alias**
- **allowUnqualifiedItemReferences**
- **handleHardIOErrors**
- **includeReferencedFunctions**
- **localSQLScope**
- **msgTablePrefix**
- **throwNrfEofExceptions**

For details, see *Program properties*.

programData
Variable and use declarations, as described in *Program data other than parameters*.

mainFunctionPart
A required function named *main*, which takes no parameters. (The only program code that can take parameters is the program itself and functions other than main.)

For details on writing a function, see *Function part in EGL source format*.

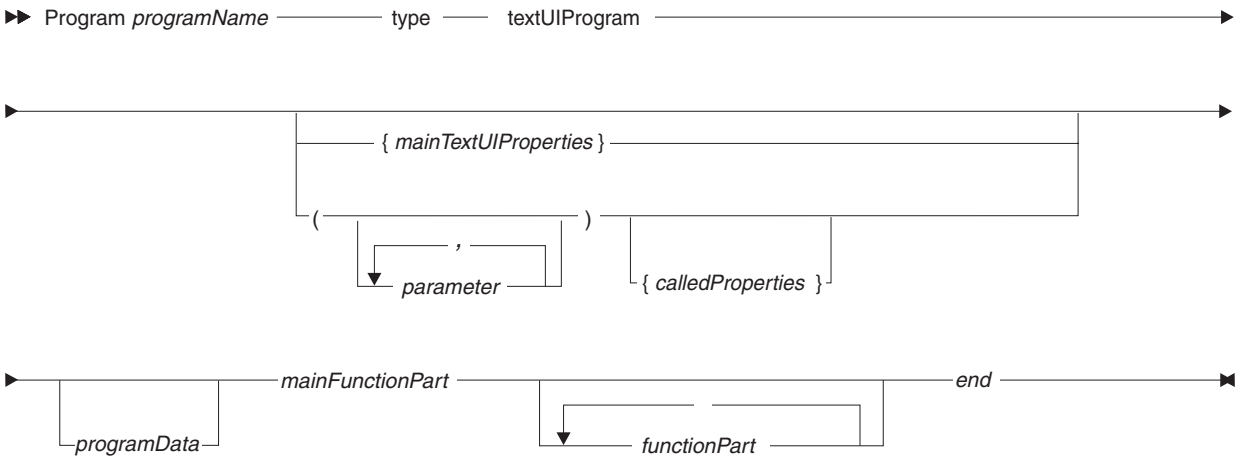
functionPart
An embedded function, which is private to this program. For details on writing a function, see *Function part in EGL source format*.

Related concepts
"EGL projects, packages, and files" on page 13
"Overview of EGL properties" on page 60
"Parts" on page 17
"Program part" on page 130
"Syntax diagram for EGL statements and commands" on page 733

Related reference
"EGL source format" on page 478
"Function part in EGL source format" on page 513
"Naming conventions" on page 652
"Program data other than parameters" on page 703
"Program parameters" on page 706
"Program part in EGL source format" on page 707
"Program part properties" on page 713
"Use declaration" on page 930

Text UI program in EGL source format

The syntax diagram for a program part of type textUIProgram is as follows:



Program *programPartName* **... end**
Identifies the part as a program part and specifies the name and type. If the program name is followed by a left parenthesis, the program is a called basic program.

If you do not set the **alias** property (as described later), the name of the generated program is either *programPartName* or, if you are generating COBOL, the first eight characters of *programPartName*.

For other rules, see *Naming conventions*.

mainTextUIProperties
The properties for a main text UI program are optional:

- **alias**
- **allowUnqualifiedItemReferences**
- **handleHardIOErrors**
- **includeReferencedFunctions**
- **inputForm**
- **inputRecord**
- **localSQLScope**
- **msgTablePrefix**
- **segmented**
- **throwNrfEofExceptions**

For details, see *Program properties*.

parameter
Specifies the name of a parameter, which may be a data item, record, or form; or a dynamic array of records or data items. For rules, see *naming conventions*.

If the caller's argument is a variable (not a constant or literal), any changes to the parameter change the area of memory available to the caller.

Each parameter is separated from the next by a comma. For other details, see *Program parameters*.

calledProperties
The called properties are optional:

- **alias**
- **allowUnqualifiedItemReferences**
- **includeReferencedFunctions**

- **msgTablePrefix**

  For details, see *Program properties*.

programData
Variable and use declarations, as described in *Program data other than parameters*.

mainFunctionPart
A required function named *main*, which takes no parameters. (The only program code that can take parameters is the program itself and functions other than main.)

For details on writing a function, see *Function part in EGL source format*.

functionPart
An embedded function, which is not available to any logic part other than the program. For details on writing a function, see *Function part in EGL source format*.

An example of a Text UI program is as follows:

```
Program HelloWorld type textUIprogram
 {}
 use myFormgroup;
 myMessage char(25);

 function main()
  while (ConverseVar.eventKey not pf3)
   myTextForm.msgField = "                ";
      myTextForm.msgField="myMessage";
      converse myTextForm;
      if (ConverseVar.eventKey is pf3)
        exit program;
      end
      if (ConverseVar.eventKey is pf1)
          myMessage = "Hello Word";
      end
  end
 end
end
```

Related concepts
"EGL projects, packages, and files" on page 13
"Overview of EGL properties" on page 60
"Parts" on page 17
"Program part" on page 130
"Segmentation in text applications" on page 149
"Syntax diagram for EGL statements and commands" on page 733

Related reference
"EGL source format" on page 478
"Function part in EGL source format" on page 513
"Naming conventions" on page 652
"Program data other than parameters" on page 703
"Program parameters" on page 706
"Program part in EGL source format" on page 707
"Program part properties" on page 713
"Use declaration" on page 930

Program part properties

Program part properties vary by whether the program is called or main and, if main, by whether the program is of type basic or text UI. The properties are as follows:

alias = *"alias"*
A string that is incorporated into the names of generated output. If you do not set the **alias** property, the program-part name (or a truncated version)is used instead.

The **alias** property is available in any program.

allowUnqualifiedItemReferences = no, allowUnqualifiedItemReferences = yes
Specifies whether to allow your code to omit container and substructure qualifiers when referencing items in structures.

The **allowUnqualifiedItemReferences** property is available in any program.

Consider the following record part, for example:

```
Record aRecordPart type basicRecord
  10 myItem01 CHAR(5);
  10 myItem02 CHAR(5);
end
```

The following variable is based on that part:

```
myRecord aRecordPart;
```

If you accept the default value of **allowUnqualifiedItemReferences** (*no*), you must specify the record name when referring to myItem01, as in this assignment:

```
myValue = myRecord.myItem01;
```

If you set the property **allowUnqualifiedItemReferences** to *yes*, however, you can avoid specifying the record name:

```
myValue = myItem01;
```

It is recommended that you accept the default value, which promotes a best practice. By specifying the container name, you reduces ambiguity for people who read your code and for EGL.

EGL uses a set of rules to determine the area of memory to which a variable name or item name refers. For details, see *References to variables and constants*.

handleHardIOErrors = yes, handleHardIOErrors = no
Sets the default value for the system variable **VGVar.handleHardIOErrors**. The variable controls whether a program continues to run after a hard error has occurred on an I/O operation in a try block. The default value for the property is *yes*, which sets the variable to 1.

Code that was migrated from VisualAge Generator may not work as before unless you set **handleHardIOErrors** to *no*, which sets the variable to 0.

This property is available in any program. For other details, see *VGVar.handleHardIOErrors* and *Exception handling*.

includeReferencedFunctions = no, includeReferencedFunctions = yes
Indicates whether the program contains a copy of each function that is neither inside the program nor in a library accessed by the program.

The **includeReferencedFunctions** property is available in any program.

The default value is *no*, which means that you can ignore this property if you are fulfilling the following practices at development time, as is recommended:

- Place shared functions in a library
- Place non-shared functions in the program

If you are using shared functions that are not in a library, generation is possible only if you set the property **includeReferencedFunctions** to *yes*.

inputForm = *"formName"*
Identifies a form that is presented to the user before the program logic runs, as described in *Input form*.

The **inputForm** property is available only in main text UI programs.

inputRecord = *"inputRecord"*
Identifies a global, basic record that a program automatically initializes and that may receive data from a program that uses a **transfer** statement to transfer control. For additional details, see *Input record*.

The **inputRecord** property is available in any main program.

localSQLScope = yes, localSQLScope = no
Indicates whether identifiers for SQL result sets and prepared statements are local to the program, as is the default. If you accept the value *yes*, different programs can use the same identifiers independently.

If you specify *no*, the identifiers are shared throughout the run unit. The identifiers created in the current code are available elsewhere, although other code can use **localSQLScope = yes** to block access to those identifiers. Also, the current code may reference identifiers created elsewhere, but only if the other code was already run and did not block access.

The effects of sharing SQL identifiers are as follows:

- You can open a result set in one program and get rows from that set in another
- You can prepare an SQL statement in one program and run that statement in another

The **localSQLScope** property is available in any program.

If you are generating COBOL output, you cannot set the property to *no*; the identifiers are always local.

msgTablePrefix = *"prefix"*
Specifies the first one to the four characters in the name of the data table that is used as the message table for the program. The other characters in the name correspond to one of the national language codes listed in*DataTable part in EGL source format*.

The **msgTablePrefix** property is available in any basic or text UI program.

Programs that run in Web applications do not use a message table, but use a JavaServer Faces message resource. For details on that resource, see the description of the **msgResource** property in:

- *PageHandler part in EGL source format*

segmented = no, segmented = yes
Indicates whether the program is segmented, as explained in *Segmentation*. The default is *no* in main text UI programs. The property is not valid in other types of programs.

throwNrfEofExceptions = no, throwNrfEofExceptions = yes

Specifies whether a soft error causes an exception to be thrown. The default is *no*. For background information, see *Exception handling*.

Related concepts
"Program part" on page 130
"References to variables in EGL" on page 55
"Segmentation in text applications" on page 149

Related reference
"DataTable part in EGL source format" on page 462
"Exception handling" on page 89
"forward" on page 566
"Input form"
"Input record"
"Naming conventions" on page 652
"PageHandler part in EGL source format" on page 659
"Syntax diagram for EGL statements and commands" on page 733

"handleHardIOErrors" on page 920

Input form

When you declare a main program that runs in a text application, you have the option to specify an *input form*, which is a form that is presented to the user before the program logic runs.

Two scenarios are possible:

- If the program is the target of a show-form-returning-to statement from an EGL-generated program, the sending program presents a form to the user, and that form must be identical to the input form of the receiving program. The receiving program is invoked only after the user submits the form. After the user submits the form, the receiving program does not present the input form a second time; instead, the initial logic (the execute function) runs.
- If the program is the target of a transfer statement from a program (EGL or non-EGL) or if the program is invoked by the user or by an operating-system command, the receiving program converses the input form. (In this case, input fields on that form are initialized before display.) After the user submits the form, the initial logic (the execute function) runs.

The input form must be in the form group that you specified in the program-part declaration.

Related reference
"Data initialization" on page 459

Input record

Any main program part can have an input record, which is a global record that the EGL-generated program automatically initializes. The record must be of type basicRecord.

If the program starts as a result of a transfer with a record, the program initializes the input record (which is internal to that program), then assigns the transferred data to the record.

If the input record is longer than the received data, the extra area in the input record retains the values assigned during record initialization. If the input record is shorter than the received data, the extra data is truncated.

If primitive types in the transferred data are incompatible with the primitive types in the equivalent positions in the input record, the receiving program may end abnormally.

Related concepts
"Overview of EGL properties" on page 60
"Parts" on page 17
"Compatibility with VisualAge Generator" on page 428

Related reference
"Data initialization" on page 459

Record and file type cross-reference

The next table shows the association of record type and file type, by target platform.

Related concepts
"Record types and properties" on page 126
"Resource associations and file types" on page 286

Related task
"Adding a resource associations part to an EGL build file" on page 289
"Editing a resource associations part in an EGL build file" on page 290
"Removing a resource associations part from an EGL build file" on page 291

Related reference
"resourceAssociations" on page 381

Properties that support variable-length records

When you declare a record part, you can include properties that support the use of variable-length records, but only as follows:

- In relation to EGL-generated COBOL programs, you can use variable-length serial or indexed records for accessing VSAM files, and you can use variable-length MQ records for accessing MQSeries message queues
- In relation to EGL-generated Java programs, you can use variable-length serial records for accessing sequential files, variable-length serial or indexed records for accessing VSAM files, and variable-length MQ records for accessing MQSeries message queues

Variable-length records with the lengthItem property

The *lengthItem* property, if present, identifies an item that is used when:

- Your code reads a record from a file or queue. The length item receives the number of bytes read into the variable-length record.
- Your code writes a record. The length item specifies the number of bytes to add to the file or queue.

The length item can be any of the following:

- A structure item in the same record

- A structure item in a record that is global to the program or is local to the function that accesses the record (the length item may be qualified with a record variable declared in the program or function)
- A data item that is global to the program or is local to the function that accesses the record

The length item has these characteristics:

- Has a primitive type of BIN, DECIMAL, INT, NUM, or SMALLINT
- Contains no decimal place
- Allows for 9 digits at most

An example of a variable-length record part with the lengthItem property is as follows:

```
Record mySerialRecordPart1 type serialRecord
  {
    fileName = "myFile",
    lengthItem = "myOtherField"
  }
  10 myField01 BIN(4);   // 2 bytes long
  10 myField02 NUM(3);   // 3 bytes long
  10 myField03 CHAR(20); // 20 bytes long
end
```

When writing a record, the value of the length item must fall between item boundaries, unless the item is a character item. For example, a record of type mySerialRecordPart1 can have the length item, myOtherField, set to 2, 5, 6, 7, ... , 24 , 25. A record with myOtherField set to 2 only contains a value for myField01; a record with myOtherField set to 5 contains values for myField01 and myField02; a record with myOtherField set to 6 through 24 also contains part of myField03.

Variable-length records with the numElementsItem property

The *NumElementsItem* property, if present, identifies an item that is used when your code adds to or updates the file or queue. The variable-length record must have an array as the last, top-level structure item. The value in the number of elements item represents the actual number of array elements that are written. The value can range from 0 to the maximum, which is the *occurs* value specified in the declaration of the last, top-level structure item in the record.

The number of bytes written is equal to the sum of the following:

- The number of bytes in the fixed-length part of the record.
- The value of the number of elements item multiplied by the number of bytes in each element of the ending array.

The number of elements item has these characteristics:

- Has a primitive type of BIN, DECIMAL, INT, NUM, SMALLINT
- Contains no decimal place
- Allows for 9 digits at most

An example of a variable-length record part with the numElementsItem property is as follows:

```
Record mySerialRecordPart2 type serialRecord
  {
    fileName = "myFile",
    numElementsItem = "myField02"
  }
```

```
    10 myField01 BIN(4);   // 2 bytes long
    10 myField02 NUM(3);   // 3 bytes long
    10 myField03 CHAR(20)[3]; // 60 bytes long
       20 mySubField01 CHAR(10);
       20 mySubField02 CHAR(10);
end
```

Writing a record of type mySerialRecordPart2 with the number of elements item myField02 set to 2 results in a variable-length record with myField01, myField02, and two occurrences of myField03 being written to the file or queue.

The number of elements item must be an item in the fixed-length part of the variable-length record. Use an unqualified reference to name the number of elements item. For example, use myField02 rather than myRecord.myField02.

The number of elements item has no effect when you are reading a record from the file.

Variable-length records with both lengthItem and numElementsItem properties

If both the lengthItem and the numElementsItem properties are specified for a variable-length record, the length of the record is calculated using the number of elements item. The calculated length is moved to the record length item before the record is written to the file.

Variable-length records passed on a call or transfer

If variable-length records are passed on a call, these statements apply:

- Space is reserved for the maximum length specified for the record
- If the value of the callLink element, property **type**, is remoteCall or ejbCall, the length item (if any) must be inside the record; for details, see*callLink element*

Similarly, if variable-length records are passed on a transfer, space is reserved for the maximum length specified for the record.

Related concepts
"MQSeries support" on page 247
"Record types and properties" on page 126

Related reference
"callLink element" on page 395
"MQ record properties" on page 644

Reference compatibility in EGL

A parameter or variable is an area of memory. In some cases, the variable contains the business data of interest; a particular name or employee ID, for example. In other cases, the variable is a *reference variable*; it contains a value (specifically, a memory address) that is used to access the business data at run time.

When you assign a non-reference variable to another non-reference variable, the result is two copies of the same business data. If the source variable in an assignment statement contains a specific employee ID, for example, the statement causes the target variable to contain that ID as well. When you assign a reference variable to another reference variable, however, the result is that the source and target each contain a value that is used to access the same area of memory.

The reference-compatibility rules (as described later) apply in these situations:

- When you assign one reference variable to another; or
- When EGL transfers data between an argument and the related parameter, but only when one of these cases is in effect:
  - The parameter in the receiving function has the modifier INOUT.
  - The parameter is in the onPageLoad function of a PageHandler.
  - The parameter is in an EGL program that is invoked by another EGL program.

  In those cases, the argument is the source, and the parameter is the target.

The rules of reference compatibility are as follows:

- You can assign or pass a reference variable only to another reference variable of the same type.
- When the source (or argument) is referring to a primitive type or an array of DataItems, these statements apply:
  - The primitive characteristics (if any) must be identical. For example, an argument of type CHAR(6) is not compatible with a parameter of type CHAR(7).
  - An argument that is nullable is compatible with a nullable or non-nullable parameter. An argument that is not nullable is compatible only with a non-nullable parameter.
- The parts in different packages are considered to be different types, with the exception of DataItem parts.
- In relation to a fixed record or structure field, the length of the argument must be greater than or equal to the length of the parameter. This rule prevents the receiving code from accessing memory that is not valid.

Related concepts
"PageHandler" on page 180

Related reference
"Function parameters" on page 508
"Function part in EGL source format" on page 513
"PageHandler part in EGL source format" on page 659
"Program parameters" on page 706
"Program part in EGL source format" on page 707

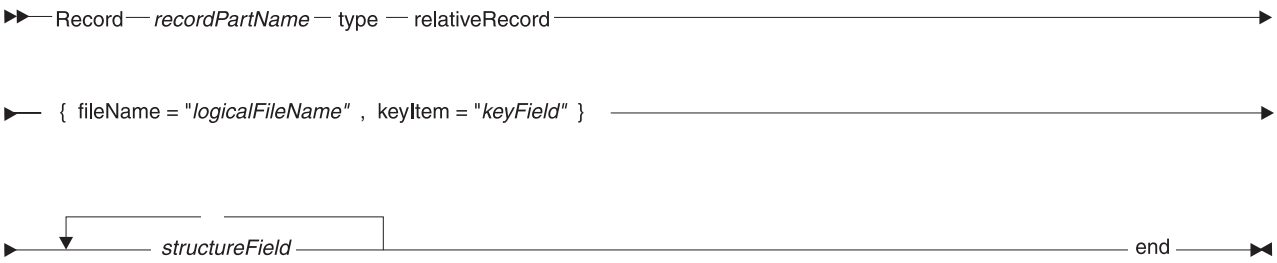
Relative record part in EGL source format

You declare a fixed record part of type relativeRecord in an EGL file, which is described in *EGL source format*.

An example of a relative record part is as follows:

```
Record myRelativeRecordPart type relativeRecord
  {
    fileName = "myFile",
    keyItem  = "myKeyItem"
  }
  10 myKeyItem NUM(4);
  10 myContent CHAR(76);
end
```

The syntax diagram of a relative record part is as follows:



Record *recordPartName* **relativeRecord**
Identifies the part being of type relativeRecord and specifies the name. For the rules of naming, see *Naming conventions*.

fileName = *"logicalFileName"*
The logical file name. For details on the meaning of your input, see *Resource associations (overview)*. For rules, see *Naming conventions*.

keyItem = *"keyField"*
The key field, which can be any of these areas of memory:

- A field in the same fixed record
- A variable or field that is global to the program or is local to the function that accesses the fixed record

You must use an unqualified reference to name the key field. For example, use *myField* rather than *myRecord.myField*. (In a function, however, you can reference the key field as you would reference any field.) The key field must be unique in the local scope of the function that accesses the record or must be absent from local scope and unique in global scope.

The key field has these characteristics:

- Has a primitive type of NUM, BIN, DECIMAL, INT, or SMALLINT
- Contains no decimal place
- Allows for 9 digits at most

Only the **get** and **add** statements use the relative record key field, but the key field must be available to any function that uses the fixed record for file access.

structureField
A structure field, as described in *Structure field in EGL source format*.

Related concepts
"EGL projects, packages, and files" on page 13
"References to parts" on page 20
"Parts" on page 17
"Record parts" on page 124
"References to variables in EGL" on page 55
"Typedef" on page 25

Related tasks
"Syntax diagram for EGL statements and commands" on page 733

Related reference
"Arrays" on page 69
"DataItem part in EGL source format" on page 461
"EGL source format" on page 478
"Function part in EGL source format" on page 513

"Indexed record part in EGL source format" on page 520
"MQ record part in EGL source format" on page 642
"Naming conventions" on page 652
"Primitive types" on page 31
"Program part in EGL source format" on page 707
"Resource associations and file types" on page 286
"Serial record part in EGL source format" on page 722
"SQL record part in EGL source format" on page 726
"Structure field in EGL source format" on page 730

Run unit

A *run unit* is a set of programs that are related by local calls or (in some cases) by transfers. Each run unit has these characteristics:

- The programs operate together as a group. When a hard error occurs but is not handled, all the programs in the run unit are removed from memory.
- The programs share the same run-time properties. The same databases and files are available throughout the run unit, for example, and when you invoke sysLib.connect or VGLib.connectionService to connect to a database dynamically, the connection is present in any program that receives control in the same run unit.

Run units are of the following types:

- The *iSeries COBOL run unit* is composed of the main program and the programs called (directly or indirectly) from that program. The run unit ends when a main program ends, as in these cases:
  - The program returns to the non-EGL program from which it was started; or
  - The program issues a **transfer** statement of the form *transfer to a transaction*.
- The *Java run unit* is composed of programs that run in a single thread.

  A new run unit can start with a main program, as when the user invokes the program. A **transfer** statement also invokes a main program but continues the same run unit.

  In the following cases, a called program is the initial program of a run unit:
  - The call is a call from an EJB session bean; or
  - The call is a remote call, except that the same run unit continues in the following case--
    - The called program is generated by EGL or VisualAge Generator; and
    - No TCP/IP listener is involved in the call.

  All programs in a Java run unit are affected by the same Java run-time properties.

Related concepts
"Java runtime properties" on page 327
"Linkage options part" on page 291

Related reference
"Default database" on page 234

"connect()" on page 867
"connectionService()" on page 888

resultSetID

The result-set identifier is in the EGL syntax and is used when you are accessing a relational database and need to relate the following kinds of statements:

- First, an **open** or **get** statement that selects a result set, or an **open** statement that calls a stored procedure that returns a result set
- Second, the statements that access the result set

If you are using an SQL record as the I/O object, the record name is sufficient to relate one kind of statement to another, unless you modify the SQL statements associated with the record to retrieve different sets of columns for update in different statements. In this case, use a result-set identifier to identify the result set associated with an EGL **replace** statement.

Related concepts
"SQL support" on page 213

Related reference
"replace" on page 613
"open" on page 598
"get" on page 567

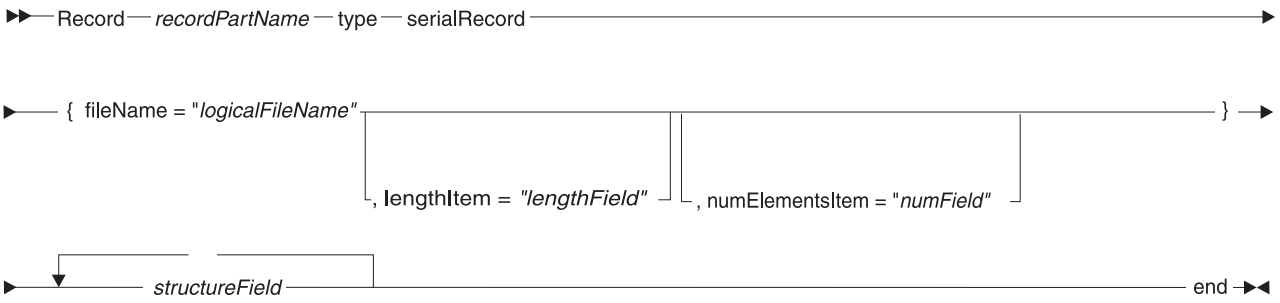
Serial record part in EGL source format

You declare a record part of type serialRecord in an EGL file, which is described in *EGL source format*.

An example of a serial record part is as follows:

```
Record mySerialRecordPart type serialRecord
  {
    fileName = "myFile"
  }
  10 myField01 CHAR(2);
  10 myField02 CHAR(78);
end
```

The syntax diagram for a serial record part is as follows:



Record *recordPartName* **serialRecord**
Identifies the part as being of type serialRecord and specifies the part name. For the rules of naming, see *Naming conventions*.

fileName = *"logicalFileName"*
The logical file name. For details on the meaning of your input, see *Resource associations (overview)*. For rules, see *Naming conventions*.

lengthItem = *"lengthField"*
The length field, as described in *Properties that support variable-length records*.

numElementsItem = *"numField"*
The number of elements field, as described in *Properties that support variable-length records*.

structureField
A structure field, as described in *Structure field in EGL source format*.

Related concepts
"EGL projects, packages, and files" on page 13
"References to parts" on page 20
"Parts" on page 17
"Record parts" on page 124
"References to variables in EGL" on page 55
"Resource associations and file types" on page 286
"Typedef" on page 25

Related tasks
"Syntax diagram for EGL statements and commands" on page 733

Related reference
"Arrays" on page 69
"DataItem part in EGL source format" on page 461
"EGL source format" on page 478
"Function part in EGL source format" on page 513
"Indexed record part in EGL source format" on page 520
"MQ record part in EGL source format" on page 642
"Naming conventions" on page 652
"Primitive types" on page 31
"Program part in EGL source format" on page 707
"Properties that support variable-length records" on page 716
"Relative record part in EGL source format" on page 719
"SQL record part in EGL source format" on page 726
"Structure field in EGL source format" on page 730

SQL data codes and EGL host variables

The property **SQL data code** identifies the SQL data type to associate with the EGL host variable. The data code is used by the database management system at declaration time, validation time, or generated-program run time.

You may want to vary the SQL data code for a host variable that is of primitive type CHAR, DBCHAR, HEX, or UNICODE. For a host variable of one of the other primitive types, however, SQL data codes are fixed.

If EGL retrieved a column definition from the database management system, do not modify the SQL data code that was retrieved, if any.

The next sections cover these topics:

Variable and fixed-length columns

To indicate that a table column is variable length or fixed length, set the SQL data code for the corresponding host variable to the appropriate value, as shown in the next table.

EGL primitive type	SQL data type	Variable or fixed	SQL data code
CHAR	CHAR (the default)	Fixed	453
	VARCHAR, length < 255	Variable	449
	VARCHAR, length > 254	Variable	457
DBCHAR, UNICODE	GRAPHIC (the default)	Fixed	469
	VARGRAPHIC, length < 128	Variable	465
	VARGRAPHIC, length > 127	Variable	473

Note: A SQL data type may require the use of null indicators, but this requirement has no effect on how you code an EGL program. For details on nulls, see *SQL support*.

Compatibility of SQL data types and EGL primitive types

An EGL host variable and the corresponding SQL table column are compatible in any of the following situations:

- The SQL column is any form of character data, and the EGL host variable is of type CHAR with a length less than or equal to the length of the SQL column.
- The SQL column is any form of DBCHAR data, and the EGL host variable is of type DBCHAR with a length less than or equal to the length of the SQL column.
- The SQL column is any form of number and the EGL host variable is of either of these types:
  - BIN, with 2 or 4 bytes and no decimal places.
  - DECIMAL, with a maximum length of 18 digits, including decimal places. The number of digits for a DECIMAL variable should be the same for the EGL host variable and for the column.
  - SMALLINT.
- The SQL column is of any data type, the EGL host variable is of type HEX, and the column and host variable contain the same number of bytes. No data conversion occurs during data transfer.

  EGL host variables of type HEX support access to any SQL column of a data type that does not correspond to an EGL primitive type.

If character data is read from an SQL table column into a shorter host variable, content is truncated on the right. To test for truncation, use the reserved word **trunc** in an EGL **if** statement.

If numeric data is read from an SQL table column into a shorter host variable, leading zeros are truncated on the left. If the number still does not fit into the host variable, fractional parts of the number (in decimal) are deleted on the right, with no indication of error. If the number still does not fit, a negative SQL code is returned to indicate an overflow condition.

The next table shows the EGL host variable characteristics that are assigned when the retrieve feature of the EGL editor extracts information from a database management system.

SQL data type	EGL host variable characteristics			SQL data code (SQLTYPE)
	Primitive type	Length	Number of bytes	
BIGINT	HEX	16	8	493
CHAR	CHAR	1–32767	1–32767	453
DATE	CHAR	10	10	453
DECIMAL	DECIMAL	1-18	1–10	485
DOUBLE	HEX	16	8	481
FLOAT	HEX	16	8	481
GRAPHIC	DBCHAR	1–16383	2–32766	469
INTEGER	BIN	9	4	497
LONG VARBINARY	HEX	65534	32767	481
LONG VARCHAR	CHAR	>4000	>4000	457
LONG VARGRAPHIC	DBCHAR	>2000	>4000	473
NUMERIC	DECIMAL	1-18	1–10	485
REAL	HEX	8	4	481
SMALLINT	BIN	4	2	501
TIME	CHAR	8	8	453
TIMESTAMP	CHAR	26	26	453
VARBINARY	HEX	2–65534	1–32767	481
VARCHAR	CHAR	≤4000	≤4000	449
VARGRAPHIC	DBCHAR	≤2000	≤4000	465

Columns with the following SQL data types cannot be accessed in a generated COBOL program because an equivalent COBOL data type does not exist:

- 460, 461: a null-terminated character string
- 476, 477: a varying-length character string, as used in Pascal

VARCHAR, VARGRAPHIC, and the related LONG data types

The definition of an SQL table column of type VARCHAR or VARGRAPHIC includes a maximum length, and the retrieve command uses that maximum to assign a length to the EGL host variable. The definition of an SQL table column of type LONG VARCHAR or VARGRAPHIC, however, does not include a maximum length, and the retrieve command uses the SQL-data-type maximum to assign a length.

DATE, TIME, and TIMESTAMP

Make sure that the format used for the EGL system default long Gregorian format is the same as the date format specified for the SQL database manager. For details on how the EGL format is set, see *VGVar.currentFormattedGregorianDate*.

You want the two formats to match so that the dates provided by the system variable *VGVar.currentFormattedGregorianDate* are in the format expected by the SQL database manager.

Related concepts
"SQL support" on page 213

Related reference
"SQL item properties" on page 63
"currentFormattedGregorianDate" on page 916

SQL record internals

You need to be aware of the internal layout of an SQL record in any of these situations:

- You use an EGL assignment statement to copy an SQL record to or from a record of a different type
- The run-time argument passed to an EGL program is an SQL record, but the program parameter is not an SQL record
- The run-time argument passed to an EGL function is an SQL record; in this case, the parameter must be a working storage record
- You receive an SQL record as a parameter in a non-EGL program

Four bytes precede each structure item in an SQL record. The first two bytes are a null indicator, and a null is interpreted as any negative value. The second two bytes are reserved for use as a length field, and you should *not* access that field.

If you are generating a COBOL program, the name of an SQL record is at the 01 level, and all structure items are at the next lowest level.

Related concepts
"Function part" on page 132
"Program part" on page 130
"SQL support" on page 213

Related reference
"Assignments" on page 352

SQL record part in EGL source format

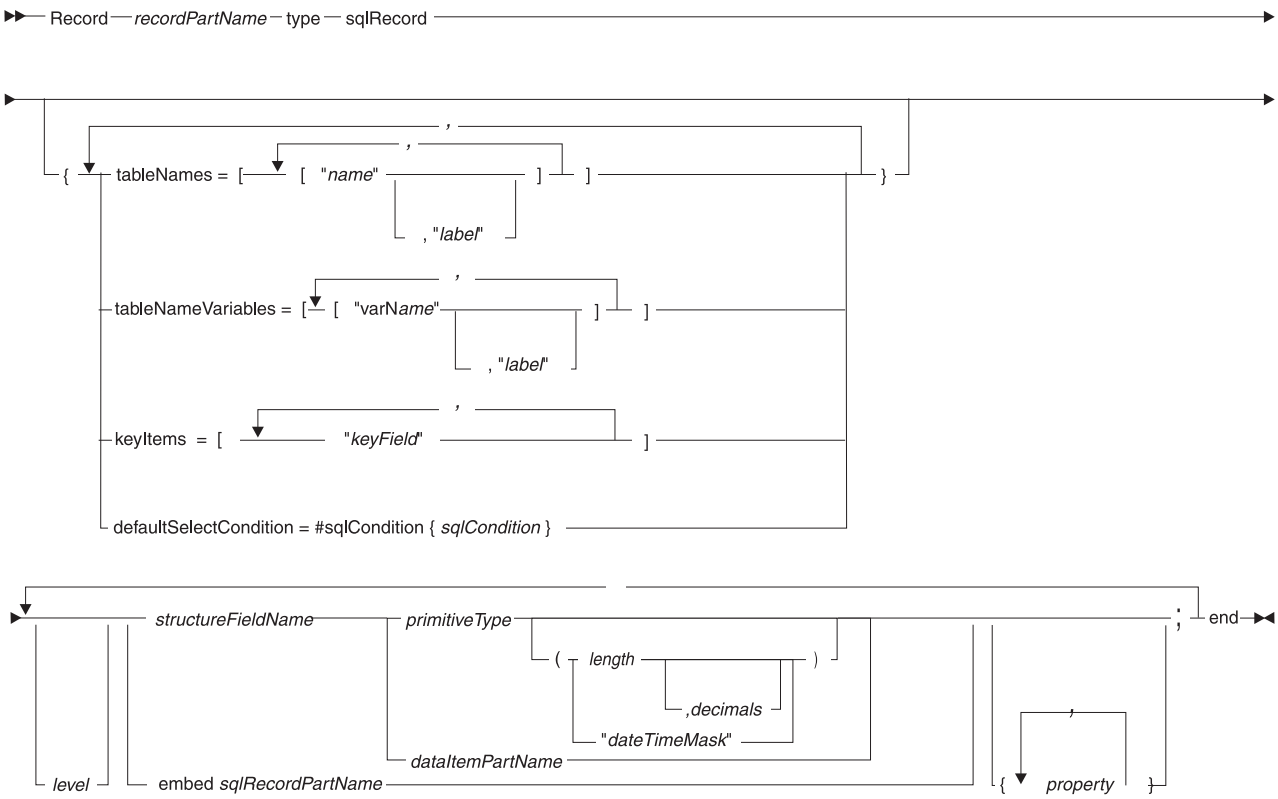
You declare a record part of type sqlRecord in an EGL file, which is described in *EGL source format*. For an overview of how EGL interacts with relational databases, see *SQL support*.

An example of a SQL record part is as follows:

```
Record mySQLRecordPart type sqlRecord
  {
    tableNames = [["mySQLTable", "T1"]],
    keyItems = ["myHostVar01"],
    defaultSelectCondition =
      #sqlCondition{ // no space between #sqlCondition and the brace
        myHostVar02 = 4 -- start each SQL comment
                        -- with a double hypen
          }
   }
```

```
    // The structure of an SQL record has no hierarchy
    10 myHostVar01 myDataItemPart01
       {
         column = "column01",
         isNullable = no,
         isReadOnly = no
       };
    10 myHostVar02 myDataItemPart02
       {
         column = "column02",
         isNullable = yes,
         isReadOnly = no
       };
end
```

The syntax diagram for an SQL record part is as follows:



Record *recordPartName* **sqlRecord**
Identifies the part as a record part of type sqlRecord and specifies the name. For rules, see *naming conventions*.

tableNames = [[*"name"***, *"label"*], ..., [***"name"***, *"label"*]]**
Lists the table or tables that are accessed by the SQL record. If you specify a label for a given table name, the label is included in the default SQL statements that are associated with the record.

You may include a double quote mark (") in a table name by preceding the quote mark with the escape character (\). That convention is necessary, for example, when a table name is one of these SQL reserved words:

- CALL

- FROM
- GROUP
- HAVING
- INSERT
- ORDER
- SELECT
- SET
- UPDATE
- UNION
- VALUES
- WHERE

Each of those names must be embedded in a doubled pair of quote marks. If the only table name is *SELECT*, for example, the tableNames clause is as follows:

```
tableNames=[["\"SELECT\""]]
```

A similar situation applies when one of those SQL reserved words is used as a column name.

tableNameVariables = [[*"varName"***, *"label"***], ..., [***"varName"***, *"label"***]]**

Lists one or more table-name variables, each of which contains the name of a table that is accessed by the SQL record. The name of a table is determined only at run time.

The variable may be qualified by a library name and may be subscripted.

If you specify a label for a given table-name variable, the label is included in the default SQL statements that are associated with the record.

You may use table-name variables alone or with table names; but the use of any table-name variable ensures that the characteristics of your SQL statement will be determined only at run time.

You may include a double quote mark (") in a table-name variable by preceding the quote mark with the escape character (\).

keyItems = = [*"item"***, ..., *"item"***]**
Indicates that the column associated with a given record item is part of the key in the database table. If the database table has a composite key, the order of the record items that are defined as keys must match the order of the columns that are keys in the database table.

defaultSelectCondition = #sqlCondition { *sqlCondition* **}**
Defines part of the search criterion in the WHERE clause of an implicit SQL statement. The value of *defaultSelectCondition* does not include the SQL keyword WHERE.

EGL provides an implicit SQL statement with a WHERE clause when you code one of these EGL statements:

- **get**
- **open**
- **execute** (only when you request an implicit SQL DELETE or UPDATE statement)

The implicit SQL statements are not stored in the EGL source code. For an overview of those statements, see *SQL support*.

level
Integer that indicates the hierarchical position of a structure field. If you exclude this value, the part is a record part; if you include this value, the part is a fixed-record part.

structureFieldName
Name of a structure field. For rules, see *Naming conventions*.

primitiveType
The primitive type assigned to the structure field.

length
The structure field's length, which is an integer. The value of a memory area that is based on the structure item includes the specified number of characters or digits.

decimals
For a numeric type (BIN, DECIMAL, NUM, NUMC, or PACF), you may specify *decimals*, which is an integer that represents the number of places after the decimal point. The maximum number of decimal positions is the smaller of two numbers: 18 or the number of digits declared as *length*. The decimal point is not stored with the data.

"dateTimeMask"
For items of type INTERVAL or TIMESTAMP, you may specify *"dateTimeMask"*, which assigns a meaning (such as "year digit") to a given position in the item value. The mask is not stored with the data.

dataItemPartName
Specifies the name of a dataItem part that acts as a model of format for the structure item being declared. For details, see *typeDef*.

embed *sqlRecordPartName*
Specifies the name of a record part of type sqlRecord and embeds the structure of that record part into the current record. The embedded structure does not add a level of hierarchy to the current record. For details, see *typeDef*.

property
An item property, as described in *Overview of EGL properties and overrides*. In an SQL record, the SQL field properties are particularly important.

Related concepts
"EGL projects, packages, and files" on page 13
"Overview of EGL properties" on page 60
"Parts" on page 17
"References to parts" on page 20
"Record parts" on page 124
"SQL support" on page 213
"Typedef" on page 25

Related tasks
"Syntax diagram for EGL statements and commands" on page 733

Related reference
"Arrays" on page 69

"DataItem part in EGL source format" on page 461

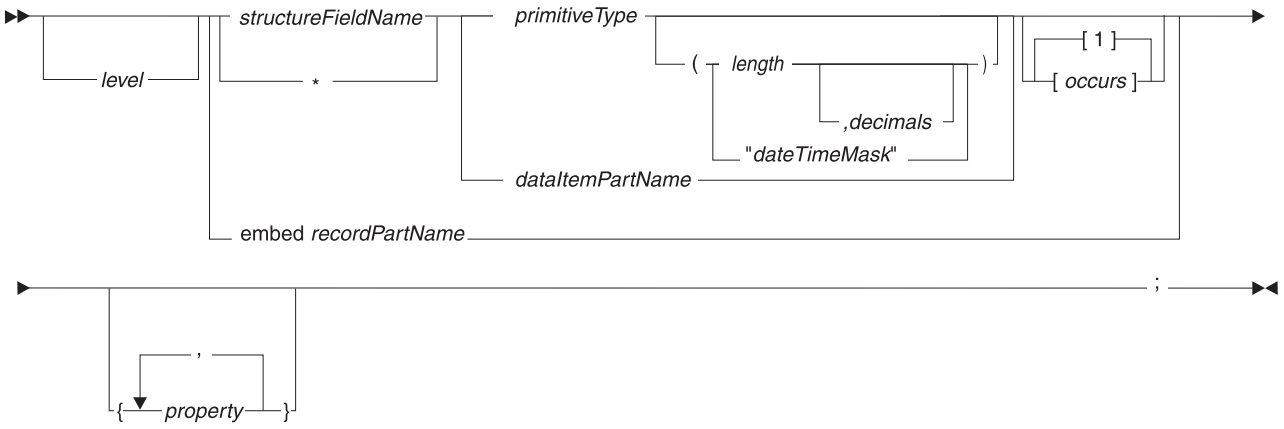
"EGL source format" on page 478
"Function part in EGL source format" on page 513
"Indexed record part in EGL source format" on page 520
"MQ record part in EGL source format" on page 642
"Naming conventions" on page 652
"Primitive types" on page 31
"Program part in EGL source format" on page 707
"References to variables in EGL" on page 55
"Relative record part in EGL source format" on page 719
"Serial record part in EGL source format" on page 722
"SQL item properties" on page 63

Structure field in EGL source format

An example of a structure field is as follows:

```
10 address;
  20 street01 CHAR(20);
  20 street02 CHAR(20);
```

The syntax diagram for a structure field is as follows:



level
Integer that indicates the hierarchical position of a structure field.

structureFieldName
Name of a structure field. For rules, see *Naming conventions*.

* Indicates that the structure field describes a *filler*, which is a memory area whose name is of no importance. An asterisk is not valid in a reference to an area of memory, as noted in *References to variables and constants*.

primitiveType
The primitive type assigned to the structure field.

length
The structure field's length, which is an integer. The value of a memory area that is based on the structure field includes the specified number of characters or digits.

decimals
For a numeric type (BIN, DECIMAL, NUM, NUMC, or PACF), you may specify *decimals*, which is an integer that represents the number of places after

the decimal point. The maximum number of decimal positions is the smaller of two numbers: 18 or the number of digits declared as *length*. The decimal point is not stored with the data.

"dateTimeMask"
For items of type INTERVAL or TIMESTAMP, you may specify *"dateTimeMask"*, which assigns a meaning (such as "year digit") to a given position in the field value. The mask is not stored with the data.

dataItemPartName
Specifies the name of a dataItem part that acts as a model of format for the structure field being declared. For details, see*typeDef*.

embed *recordPartName*
Specifies the name of a record part and embeds the structure of that record part into the current record. The embedded structure does not add a level of hierarchy to the current record. For details, see *typeDef*.

recordPartName
Specifies the name of a record part and includes the structure of that record part in the current record. In the absence of the word *embed*, the record structure is included as a substructure of the structure field being declared. For details, see *typeDef*.

occurs
The number of elements in an array of structure items. The default is 1, which means that the structure field is not an array unless you specify otherwise. For details, see Arrays.

property
An field property, as described in *Overview of EGL properties and overrides*.

Related concepts
"Syntax diagram for EGL functions" on page 732
"Overview of EGL properties" on page 60

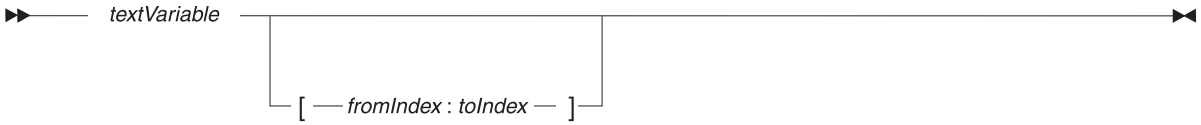
Related reference
"Arrays" on page 69
"Naming conventions" on page 652
"Primitive types" on page 31
"References to variables in EGL" on page 55
"Typedef" on page 25

Substrings

In any context in which you reference a character field, you can reference a substring, which is a sequential subset of the characters in that field. If a field value is *ABCD*, you can reference (for example) *BC*, which is the second and third character.

In addition, you can specify a substring on the left side of an assignment statement if the target field is of type CHAR, DBCHAR, or UNICODE. The substring area is filled (padded with blanks, if necessary), and the assigned text does not extend beyond the substring area (but is truncated, if necessary).

The syntax of a substring reference is as follows.



itemReference
An character or HEX field, but not a literal. The item may be a system variable or an array element.

fromIndex
The first character of interest in the item, where 1 represents the first character in the character item, 2 represents the second, and so on. You can use a numeric expression that resolves to an integer, but the expression cannot include a function invocation.

The value of *fromIndex* represents a byte position unless *itemReference* refers to an item of type DBCHAR or UNICODE, in which case the value represents a double-byte character position.

Count from the leftmost character, even if you are working with a bidirectional language such as Arabic or Hebrew.

toIndex
The last character of interest in the item, where 1 represents the first character in the character item, 2 represents the second, and so on. You can use a numeric expression that resolves to an integer, but the expression cannot include a function invocation.

The value of *toIndex* represents a byte position unless *itemReference* refers to an item of type DBCHAR or UNICODE, in which case the value represents a double-byte character position.

Count from the leftmost character, even if you are working with a bidirectional language such as Arabic or Hebrew.

Related concepts
"References to variables in EGL" on page 55

Related tasks
"Syntax diagram for EGL statements and commands" on page 733

Related reference
"Numeric expressions" on page 491

Syntax diagram for EGL functions

In the topic that describes a given EGL system function, a syntax diagram gives you details on the type of each function parameter and on the type of value returned, if any. The name of the function library is specified early in the topic.

An example diagram is as follows:

```
StrLib.clip(text STRING in)
returns (result STRING)
```

The diagram starts with the name of the function and shows a list of parameter specifications, each of which includes the following details:

- The parameter name, which you are free to specify; in this example, the name of the one parameter is *text*.

- The parameter type, which is a type in the EGL language or is a combination of types. (If the type is not in the EGL language, a further description is provided in the topic). In this example, the type is STRING.
- The modifier **in**, **out**, or **inOut**, as described in *Function parameters*.

If the parameter specification is surrounded by brackets ([ ]), the argument associated with that parameter is optional. If the specification is surrounded by braces ({ }), the argument is also optional, but in this case you can include multiple arguments that are all of the same type.

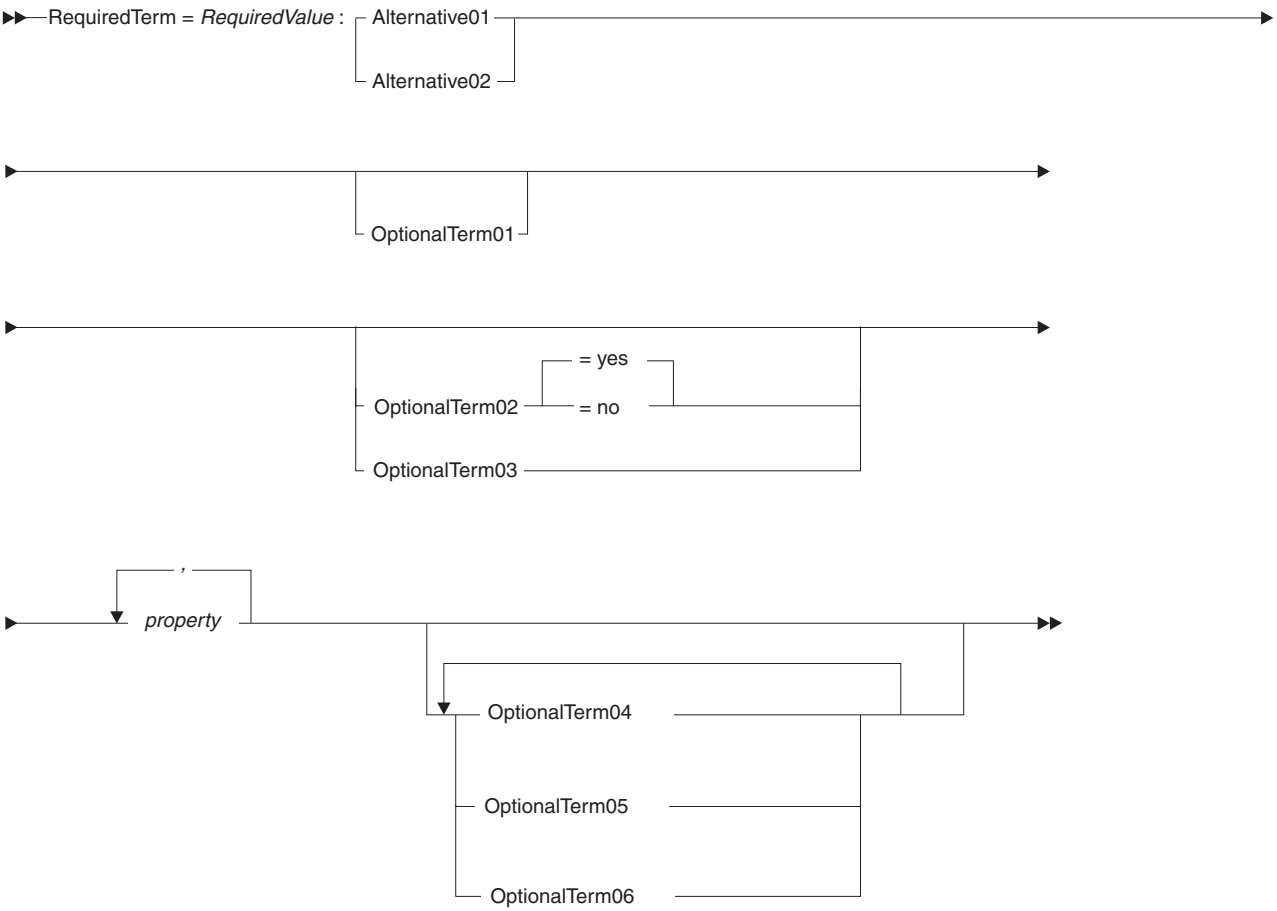
If the function returns a value, the diagram shows the word *Returns* and a parenthesized name and type. The topic refers to that name when describing the return value, but the name is otherwise meaningless.

If a returns clause is surrounded by brackets ([ ]), the return value is optional.

Related reference
"Function invocations" on page 504
"Function parameters" on page 508
"EGL library ConsoleLib" on page 735
"EGL library J2EELib" on page 778
"EGL library JavaLib" on page 781
"EGL library LobLib" on page 805
"EGL library MathLib" on page 813
"EGL library ReportLib" on page 834
"EGL library StrLib" on page 841
"EGL library SysLib" on page 860
"EGL library VGLib" on page 888

Syntax diagram for EGL statements and commands

The IBM syntax diagram lets you see quickly how to construct an EGL statement or build command. An example of such a diagram is as follows:



Read the diagram from left-to-right, top-to-bottom, following the *main path*, which is the line that begins on the left with double arrowheads (>>). As you follow the main path you may select an entry on a subordinate path, in which case you continue reading from left-to-right along the subordinate path.

In the example, the main path is composed of four line segments. It is important to see this. The second and third line segments of the main path each begins with a single arrowhead (>) and includes subordinate information. The fourth line segment of the main path line also begins with a single arrowhead (>), includes returning arrows and subordinate information, and ends with two arrowheads facing each other (><).

A term (or symbol) that is not in italics must be specified exactly as shown. In the example, you specify the term **RequiredTerm** as is. In contrast, a term in italics is a placeholder for a value that you specify. In the example, you might include any of the following symbols in place of *RequiredValue*:

```
myVariable
50
"Oh!"
```

The specific requirements for an italicized term (for example, whether a string or number is appropriate) are explained in the text that follows the syntax diagram, not in the syntax diagram itself.

If a diagram shows a non-alphanumeric character, you type that character as part of the syntax. After you specify a value for *RequiredValue*, for instance, you type a colon (:) and a blank.

If you are allowed to select from any of several terms, the terms are shown in a stack. In the example, you can specify the term **Alternative01** or **Alternative02**.

If (as in this case) you *must select* a term from those listed in a stack, one of the choices (arbitrarily specified) is on the top line of the stack. If you are not required to select a term, the terms are all below the top line of the stack, as is true of **OptionalTerm01**.

A value that is on a path but is shown in an elevated way (as is true of **= yes**) is the default value for the stack in which the value appears. The example indicates that you can specify any of the following strings, and the first two are equivalent:

```
optionalTerm01 = yes

optionalTerm01

optionalTerm01 = no

OptionalTerm02
```

An arrow returning to the left above a term indicates that you can use the term repeatedly. In the example, you specify values for *property*, each separated from the next with a comma.

An arrow returning to the left above a vertical stack means that you can choose from the list of entries in any order. In the example, each of the following strings is valid (as are other variations), but none is required:

```
  OptionalTerm04 OptionalTerm05
  OptionalTerm06
  OptionalTerm04 OptionalTerm06 OptionalTerm05
```

System Libraries

EGL library ConsoleLib

The Console library provides the consoleUI functionality to EGL programs. Using the **ConsoleLib** qualifier (for example, **ConsoleLib.activateWindow**) is optional.

Function	Description
activateWindow (*window*)	Makes the specified window the active window, and updates the **ConsoleLib** variable *activeWindow* accordingly.
activateWindowByName (*name*)	Makes the specified window the active window, and updates the **ConsoleLib** variable *activeWindow* accordingly.
cancelArrayDelete ()	Terminates the current *delete* operation in progress during the execution of a **BEFORE_DELETE OpenUI** event code block.

Function	Description
cancelArrayInsert ()	Terminates the current *insert* operation in progress during the execution of a **BEFORE_INSERT OpenUI** event code block.
clearActiveForm ()	Clears the display buffers of the all of the fields.
clearActiveWindow ()	Removes all displayed material from the active window.
clearFields ([*consoleField*{, *consoleField*}])	Clears the display buffers of the specified fields in the active form. If no fields are specified, all fields of the form are cleared.
clearFieldsByName (*fieldName*{, *fieldName*})	Clears the display buffers of the named fields in the active form. If no fields are named, all fields of the form are cleared.
clearForm (*consoleForm*)	Clears the display buffers of the all of the fields.
clearWindow (*window*)	Removes all displayed material from the specified window.
clearWindowByName (*name*)	Removes all displayed material from the specified window.
closeActiveWindow ()	Clears the window from the screen, releases the resources associated with that window, and activates the previous active window.
closeWindow (*window*)	Clears the window from the screen, releases the resources associated with that window, and activates the previous active window.
closeWindowByName (*name*)	Clears the window from the screen, releases the resources associated with that window, and activates the previous active window
result = currentArrayCount ()	Returns the number of elements in the dynamic array that is associated with the current active form
result = currentArrayDataLine ()	Returns the number of the program record within the program array that is displayed in the current line of a screen array during or immediately after the **OpenUI** statement.
result = currentArrayScreenLine ()	Returns the number of the current screen record in its screen array during an **OpenUI** statement.
displayAtLine (*text*, *line*)	Displays a string to a specified place within the active window.
displayAtPosition (*text*, *line*, *column*)	Displays a string to a specified place within the active window.
displayError (*msg*)	Causes the error window to be created and shown and displays the error message in that window.
displayFields ([*consoleField*{, *consoleField*}])	Displays form field values to the Console.

Function	Description
displayFieldsByName (*consoleFieldName*{, *consoleFieldName*)	Displays form field values to the Console.
displayForm (*consoleForm*)	Displays the form to the active window.
displayFormByName (*formName*)	Displays the form to the active window.
displayLineMode (*text*)	Displays a string in *line mode* rather than *form/window mode* .
displayMessage (*msg*)	Displays a string to a specified place within the active window and uses the *messageLine* settings of the active window to identify where to display the string.
drawBox (*row*, *column*, *depth*, *width*)	Draws a rectangle in the active window with the specified location and dimensions.
drawBoxWithColor (*row*, *column*, *depth*, *width*, *Color*)	Draws a rectangle in the active window with the specified location, dimensions, and color.
result = getKey ()	Reads a key from the input and returns the integer code for the key.
result = getKeyCode (*keyName*)	Returns the key integer code of the named key in the String.
result = getKeyName (*keyCode*)	Returns the name that represents the integer key code.
gotoField (*consoleField*)	Moves the cursor to the specified form field.
gotoFieldByName (*name*)	Moves the cursor to the specified form field.
gotoMenuItem (*item*)	Moves the menu cursor to the specified menu item.
gotoMenuItemByName (*name*)	Moves the menu cursor to the specified menu item.
hideAllMenuItems ()	Hides all menu items in the currently displayed menu.
hideErrorWindow ()	Hides the error window.
hideMenuItem (*item*)	Hides a specified menu item so that a user cannot select it.
hideMenuItemByName (*name*)	Hides a specified menu item so that a user cannot select it.
result = isCurrentField (*consoleField*)	Returns **true** if the cursor is in the specified form field; otherwise it returns **false**.
result = isCurrentFieldByName (*name*)	Returns **true** if the cursor is in the specified form field; otherwise it returns **false**.
result = isFieldModified (*consoleField*)	Returns **true** if the user changed the contents of the specified form field; a **false** return indicates that the field has not been edited.

Function	Description
result = isFieldModifiedByName (*name*)	Returns **true** if the user changed the contents of the specified form field; a **false** return indicates that the field has not been edited.
result = lastKeyTyped ()	Returns the integer code of the last physical key that was pressed on the keyboard.
nextField ()	Moves the cursor to the next form field according to the defined field travel order.
openWindow (*window*)	Makes a window visible and adds it to the top of the window stack. The form is displayed in the window.
openWindowByName (*name*)	Makes a window visible and adds it to the top of the window stack.
openWindowWithForm (*Window, form*)	Makes a window visible and adds it to the top of the window stack. The Window size will change to hold the specified form if the window size was not defined when the window was declared.
openWindowWithFormByName (*windowName, formName*)	Makes a window visible and adds it to the top of the window stack.
previousField ()	Moves the cursor to the previous form field according to the defined field travel order.
result = promptLineMode (*prompt*)	Displays a prompt message to the user in a *line mode* environment.
scrollDownLines (*numLines*)	Scrolls the data table towards the start of the data. (i.e. smaller record indices)
scrollDownPage ()	Scrolls the data table towards the start of the data. (i.e. smaller record indices)
scrollUpLines (*numLines*)	Scrolls the data table towards the end of the data. (i.e. larger record indices)
scrollUpPage ()	scrolls the data table towards the end of the data (i.e. larger record indices)
setArrayLine (*recordNumber*)	Moves the selection to the specified program record. The data table is scrolled in the display if necessary to make the selected record visible.
setCurrentArrayCount (*count*)	Sets how many records exist in the program array. Must be called prior to the **OpenUI** statement.
showAllMenuItems ()	Shows the all menu items for user selection.
showHelp (*helpkey*)	Displays the **ConsoleUI** help screen during execution of the EGL program.
showMenuItem (*item*)	Shows the specified menu item for user selection.
showMenuItemByName(*name*)	Shows the specified menu item for user selection.

Variables	Description
activeForm	The most recently displayed form in the active window.
activeWindow	The topmost window, and it is the target for window operations when no window name is specified.
commentLine	The window line where comment messages are displayed.
CurrentDisplayAttrs	Settings applied to elements displayed through the display functions.
currentRowAttrs	Highlight attributes applied to the current row.
cursorWrap	If **true**, the cursor wraps around to the first field on the form; if **false**, the statement ends when the cursor moves forwards from the last input field on the form.
defaultDisplayAttributes	Default settings of *presentation attributes* for new objects.
defaultInputAttributes	The default settings of *presentation attributes* for input operations.
deferInterrupt	If **true**, the program catches **INTR** signals and logs them in the *interruptRequested* variable, which the program is then responsible to monitor. On Windows, the signal is simulated when the logical **INTERRUPT** key is pressed, which is **CONTROL_C** by default.
deferQuit	If **true**, the program catches **QUIT** signals and logs them in the *interruptRequested* variable, which the program is then responsible to monitor. On Windows, the signal is simulated when the logical **QUIT** key is pressed, which is **CONTROL_** by default.
definedFieldOrder	If **true**, the up and down arrow keys move to the previous and next fields in the traversal order. If **false**, up and down move to the field in that direction physically on the screen.
errorLine	The window where error messages are displayed.
errorWindow	The window location where error messages are displayed in the ConsoleUI screen.
errorWindowVisible	If **true**, the error window is currently being displayed to the screen
formLine	The window line where forms are displayed.
interruptRequested	This indicates that an **INTR** signal has been received (or simulated).
key_accept	Key for successful termination of **OpenUI** statements. Default key is **ESCAPE**.

Variables	Description
key_deleteLine	Key for deleting the current row from a screen array. Default key is **F2**.
key_help	Key for showing context sensitive help during **OpenUI** statements. default key is **CTRL_W**.
key_insertLine	Key for inserting a row into a screen array. Default key is **F1**.
key_interrupt	Key for simulating an **INTR** signal. Default key is **CTRL_C**.
key_pageDown	Key for paging forwards in a screen array (data table). Default key is **F3**.
key_pageUp	Key for paging backwards in a screen array (data table). Default key is **F4**.
key_quit	Key for simulating a QUIT signal. Default key is **CTRL**_\.
menuLine	The window line where menus are displayed.
messageLine	The window line where messages are displayed.
messageResource	The file name of the resource bundle.
promptLine	The window line where error messages are displayed.
quitRequested	Indicates that a **QUIT** signal has been received (or simulated).
screen	Automatically-defined, default, borderless window; the dimensions are equal to the dimensions of the available display surface.
sqlInterrupt	If **true**, the user can interrupt SQL statements being processed. If **false**, the user can only interrupt **OpenUI** statements. Used in combination with the *deferInterrupt* and *deferQuit* variables.

activeForm

The system variable **ConsoleLib.activeForm** is the most recently displayed form in the active window.

Type: ConsoleForm

Related reference
"EGL library ConsoleLib" on page 735

activateWindow()

The system function **ConsoleLib.activateWindow** makes the specified window the active window, and updates the variable *activeWindow*.

```
ConsoleLib.activateWindow(window1 Window inOut)
```

window1
The window to activate.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

activeWindow

The system variable **ConsoleLib.activeWindow** is the topmost window or the one most recently actvated. **ConsoleLib.activeWindow** is the target for window operations when no window name is specified.

Type: Window

Related reference
"EGL library ConsoleLib" on page 735

activateWindowByName()

The system function **ConsoleLib.activateWindowByName** makes the specified window the active window, and updates the **consoleLib** variable *activeWindow* accordingly.

```
ConsoleLib.activateWindowByName(name STRING in)
```

name
The name of the window.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

cancelArrayDelete()

The system function **ConsoleLib.cancelArrayDelete** terminates the current *delete* operation in progress during the execution of a **BEFORE_DELETE OpenUI** event code block.

If at runtime, this function is executed outside the scope of an **OpenUI** statement, the effect is a null operation.

```
ConsoleLib.cancelArrayDelete( )
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

cancelArrayInsert()

The system function **ConsoleLib.cancelArrayInsert** terminates the current *insert* operation in progress during the execution of a **BEFORE_INSERT OpenUI** event code block. If at runtime, this function is executed outside the scope of an **OpenUI** statement, the effect is a null operation.

```
ConsoleLib.cancelArrayInsert( )
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

clearActiveForm()

The system function **ConsoleLib.clearActiveForm** clears the display buffers of all fields. This function has no effect on the bound data elements; data stored in the bound data elements is not cleared.

```
ConsoleLib.clearActiveForm( )
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

clearActiveWindow

The system function **ConsoleLib.clearActiveWindow** removes all displayed material from the active window. This includes erasing the constant information displayed in the current form. If the active window has a border, the border is not erased. The statement does not affect the ordering of the window stack or affect any windows that are above it in the window stack.

```
ConsoleLib.clearActiveWindow( )
```

Related reference
"EGL library ConsoleLib" on page 735

clearFields()

The system function **ConsoleLib.clearFields** clears the display buffers of the specified fields. If no fields are specified, all fields are cleared. This function has no effect on the bound data elements; any data that was stored in the bound data elements will not be cleared.

```
ConsoleLib.clearFields(
  [consoleField1 ConsoleField inOut
  {, consoleField1 ConsoleField inOut}
  ] )
```

consoleField1
The name of the variable of type ConsoleField.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

clearFieldsByName()

The system function **ConsoleLib.clearFieldsByName** clears the specified on-screen fields; and clears all fields if no fields are specified. The variables bound to the on-screen fields are not affected.

```
ConsoleLib.clearFieldsByName(
  [fieldName STRING in
  { , fieldName STRING in}])
```

fieldName
The value of a ConsoleField name field.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

clearForm()

The system function **ConsoleLib.clearForm** clears all fields in the specified form. The variables bound to those fields are not affected.

```
ConsoleLib.clearForm(consoleForm ConsoleForm inOut)
```

consoleForm
A variable of type ConsoleForm.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

clearWindow()

The system function **ConsoleLib.clearWindow** removes all displayed material from the specified window. This includes erasing the constant information displayed in the current form. If the window has a border, the border is not erased. The statement does not affect the ordering of the window stack or affect any windows that are above it in the window stack.

```
ConsoleLib.clearWindow(window1 Window inOut)
```

window1
The window to be cleared.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

clearWindowByName()

The system function **ConsoleLib.clearWindowByName** removes all displayed material from the specified window. This includes erasing the constant information displayed in the current form. If the window has a border, the border is not erased. The statement does not affect the ordering of the window stack. The **ActiveWindow** variable refers to the topmost window in the display stack.

```
ConsoleLib.cleaWindowByName(name STRING in)
```

name
The name of the window.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

closeActiveWindow()

The system function **ConsoleLib.closeActiveWindow** clears the window from the screen, releases the resources associated with the window that was cleared, and activates the previously-active window.

After **ConsoleLib.closeActiveWindow** is invoked, the window cannot be reopened by**ConsoleLib.openWindow** or **ConsoleLib.openWindowByName**. In addition, closing the SCREEN window is not allowed.

```
ConsoleLib.closeActiveWindow( )
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

closeWindow()

The Console library function **ConsoleLib.closeWindow** clears the specified window from the screen, releases the resources associated with the cleared window, and activates the previously-active window.

After **ConsoleLib.closeWindow** is invoked, the window cannot be reopened by**ConsoleLib.openWindow** or **ConsoleLib.openWindowByName**. In addition, closing the SCREEN window is not allowed.

```
ConsoleLib.closeWindow(window1 Window inOut)
```

window1
The specified window object on the screen.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

closeWindowByName()

The system function **ConsoleLib.closeWindowByName** clears the named window from the screen, releases the resources associated with the closed window, and activates the previously active window.

After **ConsoleLib.closeWindowByName** is invoked, the window cannot be reopened by**ConsoleLib.openWindow** or **ConsoleLib.openWindowByName**. The console window remains open.

```
ConsoleLib.closeWindowByName(name STRING in)
```

name
The name of the window.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

commentLine

The system variable **ConsoleLib.commentLine** is the window line where comment messages are displayed.

Type: Integer

Related reference
"EGL library ConsoleLib" on page 735

currentArrayCount()

The system function **ConsoleLib.currentArrayCount** returns the number of elements in the dynamic array that is associated with the current active form.

It is recommended that you avoid using this function, which is used to help migrate applications that were written with Informix 4GL. Instead, use the array-specific function **getSize**, as described in *Arrays*.

```
ConsoleLib.currentArrayCount( )
returns (result INT)
```

result
The number of elements.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"Arrays" on page 69
"EGL library ConsoleLib" on page 735

currentArrayDataLine()

The system function **ConsoleLib.currentArrayDataLine** returns the number of the program record within the program array that is displayed in the current line of a screen array during or immediately after the **openUI** statement.

```
ConsoleLib.currentArrayDataLine( )
returns (result INT)
```

result
Any integer.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

currentArrayScreenLine()

The system function **ConsoleLib.currentArrayScreenLine** returns the number of the current screen record in its screen array during an **openUI** statement.

```
ConsoleLib.currentArrayScreenLine( )
  returns (result INT)
```

result
Any integer.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

currentDisplayAttrs

The system variable **ConsoleLib.currentDisplayAttrs** specifies display characteristics of any text that will be shown after the variable has been set.

Variables of type PresentationAttributes include the fields color, intensity, and highlight. For details, see *ConsoleUI parts and related variables*.

Type: PresentationAttributes

Related reference
"EGL library ConsoleLib" on page 735
"ConsoleUI parts and related variables" on page 167

currentRowAttrs

The system variable **ConsoleLib.currentRowAttrs** are highlight attributes applied to the current row of a screen array.

Variables of type PresentationAttributes include the fields color, intensity, and highlight. For details, see *ConsoleUI parts and related variables*.

Type: PresentationAttributes

Related reference
currentDisplayAttrs
"EGL library ConsoleLib" on page 735

cursorWrap

The system variable **ConsoleLib.cursorWrap** indicates whether the cursor wraps around to the first field on the form after the user attempts to navigate beyond the last field. The navigation is attempted when the user presses **Tab** or **Enter** or (when autonext is set) when the user fills the field.

Valid values are **yes** (in which case the cursor wraps) and **no** (in which case the user's action causes acceptance of the form).

Type: Boolean

Related reference
"EGL library ConsoleLib" on page 735

defaultDisplayAttributes

The system variable **ConsoleLib.defaultDisplayAttributes** contains the settings used for *PresentationAttributes* in variables.

Variables of type PresentationAttributes include the fields color, intensity, and highlight. For details, see *ConsoleUI parts and related variables*.

Type: PresentationAttributes

Related reference
"EGL library ConsoleLib" on page 735

defaultInputAttributes

The system variable **ConsoleLib.defaultInputAttributes** contains the default settings of presentation attributes for input operations.

Variables of type PresentationAttributes include the fields color, intensity, and highlight. For details, see *ConsoleUI parts and related variables*.

Type: PresentationAttributes

Related reference
"EGL library ConsoleLib" on page 735

deferInterrupt

The Console UI library variable **ConsoleLib.deferInterrupt** identifies the behavior of the application when it receives the **INTERRUPT** signal. If the results are **true**, the program catches **INTR** signals and logs them in the *interruptRequested* variable, which the program is then responsible to monitor. On Windows, the signal is simulated when the logical **INTERRUPT** key is pressed, which is **CONTROL_C** by default. If the results are **false**, the program ends when the **interrupt** key is pressed.

Type: Boolean

Related reference
"EGL library ConsoleLib" on page 735

deferQuit

For the system variable **ConsoleLib.deferQuit,** if **true**, the program catches **QUIT** signals and logs them in the *quitRequested* variable, which the program is then responsible to monitor. On Windows, the signal is simulated when the logical **QUIT** key is pressed, which is **CONTROL_** by default. If **false**, receiving a quit signal will terminate the application.

Type: Boolean

Related reference
"EGL library ConsoleLib" on page 735

definedFieldOrder

The Console UI variable **ConsoleLib.definedFieldOrder** determines the behavior of the up/down arrow keys when inputting with a form. If **true,** the cursor traverses fields in the order of definition when using the up/down arrow keys. If **false,** the cursor moves up and down according to the physical arrangement of the fields on the screen.

Type: Boolean

Related reference
"EGL library ConsoleLib" on page 735

displayAtLine()

The system function **ConsoleLib.displayAtLine** displays a string to a specified place within the active window.

```
ConsoleLib.displayAtLine(
  text STRING in
  line INT in)
```

text
The string to display.

line
The number of the line on which to display the string.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

displayAtPosition()

The system function **ConsoleLib.displayAtPosition** displays a string to a specified place within the active window.

```
ConsoleLib.displayAtPosition(
  text STRING in,
  line INT in,
  column INT in)
```

text
The string to display.

line
The number of the line at which to display the string.

column
The number of the column on which to display the string.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

displayError()

The system function **ConsoleLib.displayError** causes the error window to be created and shown, and display the error message in that window. The error window floats above all other windows until it is closed by calling *hideErrorWindow()* or when a key is pressed. If applicable, the terminal bell will be activated.

```
ConsoleLib.displayError(msg STRING in)
```

msg
The error message to display.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

displayFields()

The system function **ConsoleLib.displayFields** displays form field values to the Console. If data elements are bound to the fields, the data will be retrieved from those elements and formatted according to the rules specified with the form field. For an unbound form field, data can be set directly to the fields by accessing the **ConsoleField.value** field.

```
ConsoleLib.displayFields(
  [consoleField1 ConsoleField in
   {, consoleField1 ConsoleField in}
  ])
```

consoleField1
The name of the variable of type ConsoleField.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

displayFieldsByName()

The system function **ConsoleLib.displayFieldsByName** displays form field values to the Console. If data elements are bound to the fields, the data will be retrieved from those elements and formatted according to the rules specified with the form field. For an unbound form field, data can be set directly to these fields by accessing the **ConsoleField.value** field.

```
ConsoleLib.displayFieldsByName(
  consoleFieldName1 ConsoleFieldName in
  { , consoleFieldName1 ConsoleFieldName in})
```

consoleFieldName1
The names of the fields to display.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

displayForm()

The system function **ConsoleLib.displayForm** displays the specified form to the active window.

```
ConsoleLib.displayForm(consoleForm ConsoleForm in)
```

consoleForm
The name of the variable of type ConsoleForm.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

displayFormByName()

The system function **ConsoleLib.displayFormByName** displays the named form to the active window.

```
ConsoleLib.displayFormByName(formName STRING in)
```

formName
The value of the ConsoleForm name field.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

displayLineMode()

The system function **ConsoleLib.displayLineMode** displays the designated string in **line mode** rather than **form/window mode**. The string value is sent to the standard *out* location on the running system. All display characteristics such as wrapping and scrolling become the responsibility of the standard output interface.

```
ConsoleLib.displayLine(text STRING in)
```

text
The string to display.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

displayMessage()

The system function **ConsoleLib.displayMessage** displays a string to the message line of the active window. The function uses the *MessageLine* settings of the active window to know where to display the string.

```
ConsoleLib.displayMessage(msg STRING in)
```

msg
The message to display.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

drawBox()

The system function **ConsoleLib.drawBox** draws a rectangle in the active window with the upper-left corner at *row, column* for the first two integers and *depth, width* for the next two integers. The row and column are relative to the upper-left corner of the current window.

```
ConsoleLib.drawBox(
  row INT in,
  column INT in,
  depth INT in,
  width INT in)
```

row
The row number relative to the uper left corner of the window.

column
The column number relative to the uper left corner of the window.

depth
The depth or height of the box.

width
The width of the box.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

drawBoxWithColor()

The system function **ConsoleLib.drawBoxWithColor** draws a rectangle in the active window with the upper-left corner at *row, column*n for the first two integers and *depth, width* for the next two integers. The row and column are relative to the upper-left corner of the current window. The rectangle is drawn in the specified color.

```
ConsoleLib.drawBoxWithColor(
  row INT in,
  column INT in,
  depth INT in,
  width INT in,
  color enumerationColorKind in)
```

row
The row number relative to the uper left corner of the window.

column
The column number relative to the uper left corner of the window.

depth
The depth or height of the box.

width
The width of the box.

color
The color of the box.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

errorLine

The Console UI variable **ConsoleLib.errorLine** controls the line location where error messages are displayed in the **ConsoleUI** screen.

Type: INT

Related reference
"EGL library ConsoleLib" on page 735

errorWindow

The system variable **ConsoleLib.errorWindow** is the window where an error message from **ConsoleLib.displayError()** is shown.

Type: Window

Related reference
"EGL library ConsoleLib" on page 735
"displayError()" on page 749

errorWindowVisible

The Console UI variable **ConsoleLib.errorWindowVisible** identifies the status of the error message window. If **true**, the window is visible. If **false**, the window is not visible.

Type: Boolean

Related reference
"EGL library ConsoleLib" on page 735

formLine

The system variable **ConsoleLib.formLine** is the default line location where a form is displayed in window. It affects the properties of windows when they are opened.

Type: INT

Related reference
"EGL library ConsoleLib" on page 735

getKey()

The system function **ConsoleLib.getKey** waits for a key to be pressed and returns the integer code of the physical key that was pressed. This function reads a key

from the input. Results may be interpreted in a portable way by comparing the result with the value returned by **getKeyCode(String keyname)**.

```
ConsoleLib.getKey( )
returns (result INT)
```

result
An integer that represents the key pressed.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735
"getKey()" on page 752

getKeyCode()

The system function **ConsoleLib.getKeyCode** returns the key integer code of the specified key name.

```
ConsoleLib.getKeyCode(keyName STRING in)
returns (result INT)
```

result
An integer that represents the key name.

keyName
The name of the logical or physical key for which to calculate the corresponding key code.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

getKeyName()

The system function **ConsoleLib.getKeyName** returns the name of the key that represents the integer key code.

```
ConsoleLib.getKeyName(keyCode INT in)
returns (result STRING)
```

result
The name of the key of the integer key code.

keyCode
The key integer code.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

gotoField()

The system function **ConsoleLib.gotoField** moves the cursor to the specified form field. This function is valid in an **OpenUI** statement that acts on a console form.

```
ConsoleLib.gotoField(consoleField1 ConsoleField in)
```

consoleField1
The name of the variable of type ConsoleField to which the cursor moves.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

gotoFieldByName()

The system function **ConsoleLib.gotoFieldByName** moves the cursor to the specified form field. This function is valid in an **openUI**statement that acts on a console form.

```
ConsoleLib.gotoFieldByName(name STRING in)
```

name
The name of the field to which the cursor moves.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

gotoMenuItem()

The system function **ConsoleLib.gotoMenuItem** moves the menu cursor to the specified menu item. When the function is invoked, the menu item that is specified is selected.

```
ConsoleLib.gotoMenuItem(item MenuItem in)
```

item
The menu item to which the cursor moves.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

gotoMenuItemByName()

The system function **ConsoleLib.gotoMenuItemByName** moves the menu cursor to the specified menu item. When the function is invoked, the menu item that is specified is selected.

```
ConsoleLib.gotoMenuItemByName(name STRING in)
```

name
The name of the menu item to which the cursor moves.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

hideAllMenuItems()

The system function **ConsoleLib.hideAllMenuItems** hides all menu items in the currently displayed menu.

```
ConsoleLib.hideAllMenuItems( )
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

hideErrorWindow()

The system function **ConsoleLib.hideErrorWindow** hides the error window.

```
ConsoleLib.hideErrorWindow( )
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

hideMenuItem()

The system function **ConsoleLib.hideMenuItem** hides the specified menu item so that the user cannot select it. By default all menu items are shown. The hidden item will not be activated by keystrokes.

```
ConsoleLib.hideMenuItem(item MenuItem in)
```

item
The menu item to be hidden.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

hideMenuItemByName()

The system function **ConsoleLib.hideMenuItemByName** hides the specified menu item so that the user cannot select it. By default all menu items are shown. The hidden item will not be activated by keystrokes.

```
ConsoleLib.hideMenuItemByname(name STRING in)
```

name
The name of the menu item to be hidden.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

interruptRequested

The Console UI variable **ConsoleLib.interruptRequested** indicates if an **INTR** signal has been received or simulated. If **true**, an **INTR** signal has been received. If **false**, an **INTR** signal has not been received.

Type: Boolean

Related reference
"EGL library ConsoleLib" on page 735

isCurrentField()

The system function **ConsoleLib.isCurrentField** returns **yes** if the cursor is in the field and returns **no** if the cursor is not in the field. This function is valid in an **OpenUI** statement that acts on an arrayDictionary.

```
ConsoleLib.isCurrentField(consoleField1 ConsoleField in)
returns (result BOOLEAN)
```

result
true, if the cursor is in the specified form field; otherwise **false**.

consoleField1
The name of the variable of type ConsoleField.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

isCurrentFieldByName()

The system function **ConsoleLib.isCurrentFieldByName** returns **yes** if the cursor is in the field; otherwise returns **no**.

This function is valid in an **OpenUI** statement that acts on a console form.

```
ConsoleLib.isCurrentFieldByName(name STRING in)
returns (result BOOLEAN)
```

result
true, if the cursor is in the specified form field; otherwise **false**.

name
The value of the ConsoleField name field.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

isFieldModified()

The system function **ConsoleLib.isFieldModified** identifies for **OpenUI** form/fields, whether a field has been modified during the current **OpenUI** Statement. For **OpenUI** screenarray (arrayDictionary), it returns whether the field in the current row has been modified since the cursor entered the row.

This function is valid on commands that modify fields and does not register the effect of statements that appear in a **BEFORE_OPENUI** clause. You can assign values to fields in these clauses without marking the fields as touched.

```
ConsoleLib.isFieldModified(consoleFiled1 ConsoleField in)
returns (result BOOLEAN)
```

result
true, if the the specified form field was modified; otherwise **false**.

consoleField1
The name of the variable of type ConsoleField.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

isFieldModifiedByName()

The system function **ConsoleLib.isFieldModifiedByName** identifies whether or not the contents of a named field have been modified. **ConsoleLib.isFieldModifiedByName** returns **yes** if the user changed the contents of a field and returns **no** if the user did not change the field contents.

This function is valid on commands that modify fields and does not register the effect of statements that appear in a **BEFORE_OPENUI** clause. You can assign values to fields in these clauses without marking the fields as touched.

```
ConsoleLib.isFieldModifiedByName(name STRING in)
returns (result BOOLEAN)
```

result
true, if the the specified form field was modified; otherwise **false**.

name
The value of the ConsoleField name field.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

key_accept

The system variable **ConsoleLib.key_accept** is the key for successful termination of a **OpenUI** statement. The default key is **Esc**.

Type: CHAR(32)

Related reference
"EGL library ConsoleLib" on page 735

key_deleteLine

The system variable **ConsoleLib.key_deleteLine** is the key for deleting the current row from an arrayDictionary in a console form. The default key is **F2**.

Type: CHAR(32)

Related reference
"EGL library ConsoleLib" on page 735

key_help

The system variable **ConsoleLib.key_help** is the key for showing context-sensitive help during an **OpenUI** statement. The default key is **CRTL_W**.

Type: CHAR(32)

Related reference
"EGL library ConsoleLib" on page 735

key_insertLine

The system variable **ConsoleLib.key_insertLine** identifies the keystroke used to insert a row in an arrayDictionary on a consoleForm. The default key is **F1**.

Type: CHAR(32)

Related reference
"EGL library ConsoleLib" on page 735

key_interrupt

The system variable **ConsoleLib.key_interrupt** is the key for simulating an interrupt. The default key is **CTRL_C**.

Type: CHAR(32)

Related reference
"EGL library ConsoleLib" on page 735

key_pageDown

The system variable **ConsoleLib.key_pageDown** is the key that pages forward in an arrayDictionary on a console form. The default key is **F3**.

Type: CHAR(32)

Related reference
"EGL library ConsoleLib" on page 735

key_pageUp

The system variable **ConsoleLib.key_pageUp** is the key for paging backward in an arrayDictionary on a console form. The default key is **F4.**

Type: CHAR(32)

Related reference
"EGL library ConsoleLib" on page 735

key_quit

The system variable **ConsoleLib.key_quit** is the key for leaving the program without validating user input. The default key is **CTRL_\.**

Type: CHAR(32)

Related reference
"EGL library ConsoleLib" on page 735

lastKeyTyped()

The system function **ConsoleLib.lastKeyTyped** returns the integer code of the last physical key that was pressed on the keyboard.

```
ConsoleLib.lastKeyTyped( )
returns (result INT)
```

result
An integer that represents the last key pressed.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

menuLine

The system variable **ConsoleLib.menuLine** contains the line location where menus are displayed in a window. It affects the properties of windows when they are opened.

Type: INT

Related reference
"EGL library ConsoleLib" on page 735

messageLine

The system variable **ConsoleLib.messageLine** is the window location where messages are displayed.

Type: INT

Related reference
"EGL library ConsoleLib" on page 735

messageResource

The system variable **ConsoleLib.messageResource** is the file name of the resource bundle from which help and other messages are loaded. If this variable has no value, EGL run time inspects the file identified in the Java runtime property **vgj.messages.file**.

Type: CHAR(255)

Related concepts
"Syntax diagram for EGL functions" on page 732
"Console user interface" on page 165

Related reference
"ConsoleUI parts and related variables" on page 167

"EGL library ConsoleLib" on page 735
"Java runtime properties (details)" on page 525

nextField()

The system function **ConsoleLib.nextField** moves the cursor to the next form field according the defined field travel order. This function is valid in an **openUI** statement that acts on a console form.

```
ConsoleLib.nextField( )
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

openWindow()

The system function **ConsoleLib.openWindow** makes a window visible, adds it to the top of the window stack.

```
ConsoleLib.openWindow(window1 Window inOut)
```

window1
A variable of type Window.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

openWindowByName()

The system function **ConsoleLib.openWindowByName** makes a window visible and adds it to the top of the window stack.

```
ConsoleLib.openWindowByName(name STRING in)
```

name
The value of the Window name field.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

openWindowWithForm()

The system function **ConsoleLib.openWindowWithForm** makes a window visible, adds it to the top of the window stack and displays the form in the window. The window is re-sized to fit the form.

```
ConsoleLib.openWindowWithForm(
  window1 Window inOut,
  form ConsoleForm in)
```

window1
A variable of type Window.

form
A variable of type ConsoleForm.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

openWindowWithFormByName()

The system function **ConsoleLib.openWindowWithFormByName** activates a window, makes it visible, and displays the specified console form. The window is re-sized to fit the form.

```
ConsoleLib.openWindowWithFormByName(
  windowName STRING in,
  formName STRING in)
```

windowName
The value of the Window name field.

formName
The value of the ConsoleForm name field.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

previousField()

The system function **ConsoleLib.previousField** moves the cursor to the previous form field according to the defined field tab order.

```
ConsoleLib.previousField( )
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

promptLine

The system variable **ConsoleLib.promptLine** is the default line where prompts are displayed in a window. This affects the properties of windows when they are opened.

Type: INT

Related reference
"EGL library ConsoleLib" on page 735

promptLineMode()

The system function **ConsoleLib.promptLineMode** displays the string in line mode and waits for user input, which is submitted when the user presses **Enter**.

```
ConsoleLib.promptLineMode(message String in)
returns (result STRING)
```

result
The user input.

message
The phrase to display.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

quitRequested

The system variable **ConsoleLib.quitRequested** indicates that a **QUIT** signal has been received (or simulated).If **true**, an **QUIT** signal has been received. If **false**, a **QUIT** signal has not been received.

Type: Boolean

Related reference
"EGL library ConsoleLib" on page 735

screen

The system variable **ConsoleLib.screen** automatically defines a default, borderless window. The dimensions of the screen are equal to the dimensions of the available display surface.

Type: Window

Related reference
"EGL library ConsoleLib" on page 735

scrollDownLines()

The system function **ConsoleLib.scrollDownLines** scrolls the on-screen data toward the bottom of the data.

```
ConsoleLib.scrollDownLines(numLines INT in)
```

numLines
The number of lines to scroll downward.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

scrollDownPage()

The system function **ConsoleLib.scrollDownPage** scrolls the on-screen data one page toward the bottom of the data.

```
ConsoleLib.scrollDownPage( )
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

scrollUpLines()

The system function **ConsoleLib.scrollUpLines** scrolls the on-screen data toward the top of the data.

```
ConsoleLib.scrollUpLines(numLines INT in)
```

numLines
The number of lines to scroll up.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

scrollUpPage()

The system function **ConsoleLib.scrollUpPage** scrolls the on-screen data by one page toward the top of the data.

```
ConsoleLib.scrollUpPage( )
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

setArrayLine()

The system function **ConsoleLib.setArrayLine** moves the selection to the specified program record. If necessary, the data is scrolled to make the selected record visible.

```
ConsoleLib.setArrayLine(recordNumber INT in)
```

recordNumber
The record to select.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

setCurrentArrayCount()

The system function **ConsoleLib.setCurrentArrayCount** specifies how many rows exist in a dynamic array that is bound to an on-screen arrayDictionary. This function is useful only if you invoke it before issuing the **openUI** statement that uses the arrayDictionary.

```
ConsoleLib.setCurrentArrayCount(count INT in)
```

count
The number of array entries when the openUI statement begins.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735
"openUI" on page 602

showAllMenuItems()

The system function **ConsoleLib.showAllMenuItems** shows all menu items in the currently displayed menu.

```
ConsoleLib.showAllMenuItems( )
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

showHelp()

The system function **ConsoleLib.showHelp** displays a help message. The string argument is the key for the message in the resource bundle configured with the **ConsoleLib.messageResource** field.

```
ConsoleLib.showHelp(helpKey STRING in)
```

helpKey
The key that looks up the text for a help message.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

showMenuItem()

The system function **ConsoleLib.showMenuItem** shows the specified menu item so that it can be selected by the user. By default all menu items are shown.

```
ConsoleLib.showMenuItem(item MenuItem in)
```

item
The menu item to show.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConsoleLib" on page 735

showMenuItemByName

The system function **ConsoleLib.showMenuItemByName** shows the specified menu item so that it can be selected by the user. By default all menu items are shown.

```
ConsoleLib.showMenuItemByName(name STRING in)
```

name
The value of the MenuItem name field.

Related reference
"EGL library ConsoleLib" on page 735

sqlInterrupt

For the system variable **ConsoleLib.sqlInterrupt**, if **yes**, the user can interrupt SQL statements being processed. If **no**, the user can only interrupt **OpenUI** statements. Variable **sqlInterrupt** is used in combination with the *deferInterrupt* and *deferQuit* variables.

Type: Boolean

Related reference
"EGL library ConsoleLib" on page 735

EGL library ConverseLib

The Converse library provides the functions shown in the table below.

Function	Description
clearScreen ()	Clears the screen, as is useful before the program issues a converse statement in a text application.
displayMsgNum (*msgNumber*)	Retrieves a value from the program's message table. The message is presented the next time that a form is presented by a **converse**, **display**, **print**, or **show** statement.
result = fieldInputLength (*textField*)	Returns the number of characters that the user typed in the input field when the text form was last presented. That number does not include leading or trailing blanks or nulls.
pageEject ()	Advances print-form output to the top of the next page, as is useful before the program issues a print statement.

Function	Description
validationFailed (*msgNumber*)	• If invoked in a field-validation function in a text application, **ConverseLib.validationFailed** causes the re-presentation of the received text form after all validation functions are processed. The last-invoked **ConverseLib.validationFailed** determines what message is displayed. • If invoked outside a validation function, **ConverseLib.validationFailed** presents the specified message the next time that a form is presented by a **converse**, **display**, **print**, or **show** statement. The behavior in this case is like that of **ConverseLib.displayMsgNum**.

clearScreen()

The system function **ConverseLib.clearScreen** clears the screen, as is useful before the program issues a converse statement in a text application.

```
ConverseLib.clearScreen( )
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"converse" on page 554
"EGL library ConverseLib" on page 765

displayMsgNum()

The system function **ConverseLib.displayMsgNum** retrieves a value from the program's message table. The message is presented the next time that a form is presented by a **converse**, **display**, **print**, or **show** statement.

If possible, the message presentation is on the form itself, in the field to which the form property **msgField** refers. If the form property **msgField** has no value, the message is displayed previous to the display of the form, on a separate, modal screen or on a printable page.

ConverseLib.displayMsgNum takes as its only argument a value that is compared against each cell in the first column of the program's *message table*, which is the data table to which the program's **msgTablePrefix** property refers. The message retrieved by that function is in the second column of the same row.

```
ConverseLib.displayMsgNum(msgNumber INT in)
```

msgNumber
The message is retrieved from the message table by number. The argument must be an integer literal or an item of primitive type SMALLINT or INT or the BIN equivalent.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConverseLib" on page 765

fieldInputLength()

The system function **ConverseLib.fieldInputLength** returns the number of characters that the user typed in the input field when the text form was last presented. That number does not include leading or trailing blanks or nulls.

If the field is at its originally defined state, the function returns a length of 0. For example, if the field contains the *value* property and it has not been modified during execution in any way, then the length is calculated as 0. The *set form initial* statement resets the field to its originally defined state. If the field is not at its originally defined state, then the length is calculated based on what was displayed or entered on the last converse statement.

```
ConverseLib.fieldInputLength(textField TestFormField in)
returns(result INT)
```

result
The number of characters that the user typed.

textField
The name of the text field.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library ConverseLib" on page 765

pageEject()

The system function **ConverseLib.pageEject** advances print-form output to the top of the next page, as is useful before the program issues a print statement.

```
ConverseLib.pageEject( )
```

For other details on printing, see *Print forms*.

Related concepts
"Syntax diagram for EGL functions" on page 732
"Print forms" on page 146

Related reference
"EGL library ConverseLib" on page 765
"print" on page 613

validationFailed()

The system function **ConverseLib.validationFailed** is used in either of two ways:

- If invoked in a field-validation function in a text application, **ConverseLib.validationFailed** causes the re-presentation of the received text form after all validation functions are processed. The last-invoked **ConverseLib.validationFailed** determines what message is displayed.

  If possible, the message presentation is on the form itself, in the field to which the form property **msgField** refers. If the form property **msgField** has no value, the message is displayed previous to the display of the form, on a separate, modal screen.

- If invoked outside a validation function, **ConverseLib.validationFailed** presents the specified message the next time that a form is presented by a **converse**, **display**, **print**, or **show** statement. The behavior in this case is like that of **ConverseLib.displayMsgNum**.

In any case, the value assigned to **ConverseLib.validationFailed** is stored in the system variable **ConverseVar.validationMsgNum**.

```
ConverseLib.validationFailed([msgNumber INT in])
```

msgNumber
The number of the message to display. The argument must be an integer literal or an item of primitive type SMALLINT or INT or the BIN equivalent. This number is compared against each cell in the first column of the program's *message table*, which is the data table to which the program's **msgTablePrefix** property refers. The retrieved message is in the second column of the same row.

The message number is 9999 by default.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"displayMsgNum()" on page 766
"validationMsgNum" on page 898
"EGL library ConverseLib" on page 765

EGL library DateTimeLib

The date-and-time system variables let you retrieve the system date and time in a variety of formats, as shown in the next table.

System variable	Description
result = currentDate ()	Contains the current system date in eight-digit Gregorian format (yyyyMMdd); you can assign this system variable to a variable of type DATE.
result = currentTime ()	Contains the current system time in six-digit format (HHmmss); you can assign this system variable to a variable to type TIME.
result = currentTimeStamp ()	Contains the current system time and date as a timestamp in twenty-digit Julian format (yyyyMMddHHmmssffffff); you can assign this system variable to a variable of type TIMESTAMP.
result = dateOf (*aTimeStamp*)	Returns a date derived from a variable of type TIMESTAMP.
result = dateValue (*dateAsString*)	Returns a DATE value that corresponds to an input string.
result = dateValueFromGregorian (*gregorianIntegerDate*)	Returns a DATE value that corresponds to an integer representation of a Gregorian date.
result = dateValueFromJulian (*julianIntegerDate*)	Returns a DATE value that corresponds to an integer representation of a Julian date.

System variable	Description
result = dayOf (*aTimeStamp*)	Returns a positive integer that represents a day of the month, as derived from a variable of type TIMESTAMP.
result = extend (*extensionField* [, *mask*])	Converts a timestamp, time, or date into a longer or shorter timestamp value.
result = intervalValue (intervalAsString)	Returns an INTERVAL value that reflects a string constant or literal.
result = intervalValueWithPattern (*intervalAsString*[, *intervalMask*)]	Returns an INTERVAL value that reflects a string constant or literal and is built based on an interval mask that you specify.
result = mdy (*month*, *day*, *year*)	Returns a DATE value derived from three integers that represent the month, day of the month, and year of a calendar date.
result = monthOf (*aTimeStamp*)	Returns a positive integer that represents a month, as derived from a variable of type TIMESTAMP.
result = timeOf ([*aTimeStamp*])	Returns a string that represents the time of day derived from either a TIMESTAMP variable or the system clock.
result = timeStampFrom (*tsDate tsTime*)	Contains the current system time and date as a timestamp in twenty-digit Julian format (yyyyMMddHHmmssffffff); you can assign this system variable to a variable of type TIMESTAMP.
result = timeStampValue (*timeStampAsString*)	Returns a TIMESTAMP value that reflects a string constant or literal.
result = timeStampValueWithPattern (*timeStampAsString*[, *timeStampMask*])	Returns a TIMESTAMP value that reflects a string and is built based on a timestamp mask that you specify.
result = timeValue (*timeAsString*)	Returns a TIME value that reflects a string constant or literal.
result = weekdayOf (*aTimeStamp*)	Returns a positive integer (0-6) that represents a day of the week, as derived from a variable of type TIMESTAMP.
result = yearOf (*aTimeStamp*)	Returns an integer that represents a year, as derived from a variable of type TIMESTAMP.

To set a date, time, or timestamp variable, you can assign VGVar.currentGregorianDate, DateTimeLib.currentTime, and DateTimeLib.currentTimeStamp, respectively. The functions that return formatted character text cannot be used for this purpose.

Related reference
"EGL statements" on page 83

currentDate()

The system function **DateTimeLib.currentDate** reads the system clock and returns a DATE value that represents the current calendar date. The function returns only the current date, not the time of day.

```
DateTimeLib.currentDate( )
returns (result  DATE)
```

result
A DATE value that represents the current calendar date.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"DATE" on page 38
"EGL library DateTimeLib" on page 768

currentTime()

The system function **DateTimeLib.currentTime** retrieves the current system time in six-digit format (HHmmss). The value is automatically updated each time it is referenced by your program.

```
DateTimeLib.currentTime( )
returns (result TIME)
```

result
A TIME value that represents the current system time.

You can use **DateTimeLib.currentTime** in these ways:

- As the source in an assignment or **move** statement
- As the argument in a **return** statement

The characteristics of **DateTimeLib.currentTime** are as follows:

Primitive type
TIME

Data length
6

Value saved across segments
No

Example:

```
myTime = DateTimeLib.currentTime;
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library DateTimeLib" on page 768

currentTimeStamp()

The system function **DateTimeLib.currentTimeStamp** retrieves the current system time and date as a timestamp in twenty-digit format (yyyyMMddHHmmssffffff). The value is automatically updated each time it is referenced by your program.

```
DateTimeLib.currentTimeStamp( )
returns (result TIMESTAMP)
```

result
A TIMESTAMPvalue that represents the current system time and date.

You can use **DateTimeLib.currentTimeStamp** in these ways:

- As the source in an assignment or **move** statement
- As the argument in a **return** statement

The characteristics of **DateTimeLib.currentTimeStamp** are as follows:

Primitive type
TIMESTAMP

Data length
20

Value saved across segments
No

Example:

```
myTimeStamp = DateTimeLib.currentTimeStamp;
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library DateTimeLib" on page 768

dateOf()

The system function **DateTimeLib.dateOf** returns a DATE value derived from a variable of type TIMESTAMP.

```
DateTimeLib.dateOf(aTimeStamp TIMESTAMP in)
returns (result DATE)
```

result
A DATE value.

aTimeStamp
The value from which the date is derived.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"DATE" on page 38
"EGL library DateTimeLib" on page 768

dateValue()

The function **DateTimeLib.dateValue** returns a DATE value that corresponds to a string.

```
DateTimeLib.dateValue(dateAsString STRING in)
returns (result DATE)
```

result
A variable of type DATE.

dateAsString
A string constant or literal containing digits that reflect the mask "yyyyMMdd". For details, see *DATE*.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"DATE" on page 38
"Datetime expressions" on page 483

dateValueFromGregorian()

The function **DateTimeLib.dateValueFromGregorian** returns a DATE value that corresponds to an integer representation of a Gregorian date.

```
DateTimeLib.dateValueFromGregorian(
  gregorianIntegerDate INT in)
returns (result DATE)
```

result
A variable of type DATE.

gregorianIntegerDate
A VisualAge Generator numeric value representing a Gregorian date in the format 00YYYMMDD or 00YYMMDD.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"DATE" on page 38
"EGL library DateTimeLib" on page 768

dateValueFromJulian()

The function **DateTimeLib.dateValueFromJulian** returns a DATE value that corresponds to an integer representation of a Julian date.

```
DateTimeLib.dateValueFromJulian(
  julianIntegerDate INT in)
returns (result DATE)
```

result
A variable of type DATE.

julianIntegerDate
A VisualAge Generator numeric value representing a Julian date in the format 00YYYYDDD or 00YYDD.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"DATE" on page 38
"EGL library DateTimeLib" on page 768

dayOf()

The system function **DateTimeLib.dayOf** returns a positive integer that represents a day (1-7), as derived from a variable of type TIMESTAMP.

```
DateTimeLib.dayOf(aTimeStamp TIMESTAMP in)
returns (result INT)
```

result
A positive integer that corresponds to the day of the month.

aTimeStamp
The variable from which the day is derived.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"DATE" on page 38
"EGL library DateTimeLib" on page 768

extend()

The system function **DateTimeLib.extend** converts a timestamp, time, or date into a longer or shorter timestamp value. Examples are as follows:

- If you have an input timestamp defined as "ddHH" (day and hour) and provide a timestamp mask of "ddHHmm" (day, hour, and minute)**, DateTimeLib.extend** returns an extended value that matches the mask
- If you have an input timestamp defined as "yyyyMMddHHmmss" (year, month, day, hour, minute, and second) and provide a timestamp mask "yyyy" (year), **DateTimeLib.extend** returns a shortened value that matches the mask

```
DateTimeLib.extend(
  extensionField dateOrTimeOrTimeStamp in
  [, mask outputTimeStampMask in
  ])
returns (result TIMESTAMP)
```

result

A variable of type TIMESTAMP.

extensionField

The name of a field of type TIMESTAMP, TIME, or DATE. The field contains the value to be extended or shortened.

mask

A string literal or constant that defines the mask of the timestamp value returned by the function. For details, see *TIMESTAMP*.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"Datetime expressions" on page 483

intervalValue()

The datetime value function **DateTimeLib.intervalValue** returns an INTERVAL value that reflects a string constant or literal and is built based on the default interval mask, which is *yyyyMM*.

The input string must contain six digits. The first four digits represent the number of years in the interval, and the last two represent the number of months.

If you wish to specify a mask other than *yyyyMM*, invoke **DateTimeLib.intervalValueWithPattern**.

```
DateTimeLib.intervalValue(intervalAsString STRING in)
returns (result INTERVAL)
```

result

A variable of type INTERVAL

intervalAsString

A string constant or literal that contains six digits whose meaning is indicated by the interval mask *yyyyMM*

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"Datetime expressions" on page 483
"INTERVAL" on page 39
"intervalValueWithPattern()"

intervalValueWithPattern()

The datetime value function **DateTimeLib.intervalValueWithPattern** returns an INTERVAL value that reflects a string constant or literal and (optionally) is built based on an interval mask that you specify. If the mask is *yyyy*, for example, the input string must contain four digits, and those digits represent the number of years represented in the interval.

```
DateTimeLib.intervalValueWithPattern(
  intervalAsString STRING in
  [, intervalMask STRING in
  ])
returns (result INTERVAL)
```

result
A variable of type INTERVAL.

intervalAsString
A string constant or literal that contains digits whose meaning is indicated by the interval mask.

intervalMask
Specifies an interval mask that gives meaning to each digit in the first parameter. The default mask is *yyyyMM*. For other details, see *INTERVAL*.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"Datetime expressions" on page 483
"INTERVAL" on page 39

mdy()

The mdy operator returns a DATE value derived from three integers that represent the month, day of the month, and year of a calendar date.

```
DateTimeLib.mdy(
  month INT in,
  day INT in,
  year INT in)
returns (result DATE)
```

result
A DATE value.

month
An integer in the range 1 through 12, representing the month.

day
An integer representing the day of the month in the range 1 through 28, 29, 30, or 31, depending on the month.

year
A four-digit integer representing the year.

An error results if you specify values outside the range of days and months in the calendar or if the number of operands is not three. You must enclose the three integer expression operands between parentheses, separated by commas, just as

you would if MDY( ) were a function. The third expression cannot be the abbreviation for the year. For example, 99 specifies a year in the first century, approximately 1,900 years ago.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"DATE" on page 38
"EGL library DateTimeLib" on page 768

monthOf()

The system function **DateTimeLib.monthOf** returns a positive integer that represents a month, as derived from a variable of type TIMESTAMP.

```
DateTimeLib.monthOf(aTimeStamp TIMESTAMP in)
returns (result INT)
```

result
A positive integer that represents a month.

aTimeStamp
The TIMESTAMP variable from which the month is derived.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"DATE" on page 38
"EGL library DateTimeLib" on page 768

timeOf()

The system function **DateTimeLib.timeOf** returns a string that represents the time of day derived from either a TIMESTAMP variable or the system clock.

```
DateTimeLib.timeOf([aTimeStamp TIMESTAMP in])
returns (result STRING)
```

result
The time-of-day portion of the *aTimeStamp* argument, as based on a 24-hour clock and the following format:

```
hh:mm:ss
```

hh The hour as a two-digit string.

mm
The minute as a two-digit string.

ss The second as a two-digit string.

aTimeStamp
A DATETIME value. If no value is specified, the operator returns a character string representing the current time

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"DATE" on page 38
"EGL library DateTimeLib" on page 768

timeStampFrom()

The function **DateTimeLib.timeStampFrom** returns a TIMESTAMP value that is built based on a DATE and TIME that you specify.

```
DateTimeLib.timeStampFrom(
  tsDate DATE in,
  tsTime TIME in)
returns (result TIMESTAMP)
```

result
A value of type TIMESTAMP.

tsDate
A variable of type DATE.

tsTime
A variable of type TIME.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"Datetime expressions" on page 483
"EGL library DateTimeLib" on page 768
"TIMESTAMP" on page 41

timeStampValue()

The function **DateTimeLib.timeStampValue** returns a TIMESTAMP value that reflects a string constant or literal and is built based on the default timestamp mask, which is *yyyyMMddHHmmss*.

The input string must contain fourteen digits:

- The first four digits represent the year
- The next two represent the numeric month
- The next two represent the day of the month
- The next two represent the number of hours (from *00* to *24*)
- The next two represent the number of minutes within the hour
- The last two represent the number of seconds within the minute

If you wish to specify a mask other than *yyyyMMddHHmmss*, invoke **DateTimeLib.timestampValueWithPattern**.

```
DateTimeLib.timeStampValue(timeStampAsString STRING in)
returns (result TIMESTAMP)
```

result
A variable of type TIMESTAMP.

timeStampAsString
A string constant or literal that contains fourteen digits whose meaning is indicated by the timestamp mask *yyyyMMddHHmmss*

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"Datetime expressions" on page 483
"TIMESTAMP" on page 41
"timeStampValueWithPattern()" on page 777

timeStampValueWithPattern()

The function **DateTimeLib.timeStampValueWithPattern** returns a TIMESTAMP value that reflects a string constant or literal and (optionally) is built based on a timestamp mask that you specify. If the mask is "yyyy", for example, the input string must contain four digits, and those digits represent the year value in the timestamp.

```
DateTimeLib.timeStampValueWithPattern(
  timeStampAsString STRING in
  [, timeStampMask STRING in
  ])
returns (result TIMESTAMP)
```

result
A variable of type TIMESTAMP.

timeStampAsString
A string constant or literal that contains digits whose meaning is indicated by the timestamp mask.

timeStampMask
Specifies a timestamp mask that gives meaning to each digit in the first parameter. The default mask is *yyyyMMddHHmmss*. For other details, see *TIMESTAMP*.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"Datetime expressions" on page 483
"TIMESTAMP" on page 41

timeValue()

The datetime value function **DateTimeLib.timeValue** returns a TIME value that reflects a string constant or literal.

```
DateTimeLib.timeValue(timeAsString STRING in)
returns (result TIME)
```

result
A variable of type TIME.

timeAsString
A string constant or literal containing digits that reflect the mask "HHmmss". For details, see *TIME*.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"Datetime expressions" on page 483
"TIME" on page 40

weekdayOf()

The system function **DateTimeLib.weekdayOf** returns a positive integer that represents a day of the week, as derived from a variable of type TIMESTAMP. The number 0 represents Sunday, 1 represents Monday, and so on.

```
DateTimeLib.weekdayOf(aTimeStamp TIMESTAMP in)
returns (result INT)
```

result
A positive integer from 0 to 6.

aTimeStamp
The TIMESTAMP variable from which the day is derived.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"DATE" on page 38
"EGL library DateTimeLib" on page 768

yearOf()

The system function **DateTimeLib.yearOf** returns a four-digit integer that represents a year, as derived from a variable to type TIMESTAMP.

```
DateTimeLib.yearOf(aTimeStamp TIMESTAMP in)
returns (result INT)
```

result
The integer that represents the year.

aTimeStamp
The TIMESTAMP variable from which the year is derived.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"DATE" on page 38
"EGL library DateTimeLib" on page 768

EGL library J2EELib

The next table lists the system functions in the library J2EELib.

Function	Description
clearRequestAttr (*key*)	Removes the argument that is associated with the specified key in the request object.
clearSessionAttr (*key*)	Removes the argument that is associated with the specified key in the session object.
getRequestAttr (*key*, *argument*)	Uses a specified key to retrieve an argument from the request object into a specified variable.
getSessionAttr (*key*, *argument*)	Uses a specified key to retrieve an argument from the session object into a specified variable.
setRequestAttr (*key*, *argument*)	Uses a specified key to place a specified argument in the request object.
setSessionAttr (*key*, *argument*)	Uses a specified key to place a specified argument in the session object.

clearRequestAttr()

The system function **J2EELib.clearRequestAttr** removes the argument that is associated with the specified key in the request object. This function is useful in PageHandlers and in programs that run in Web applications.

You can set an argument in the request object by using the system function J2EELib.setRequestAttr. You can retrieve the argument by using the system function J2EELib.getRequestAttr.

```
J2EELib.clearRequestAttr(key STRING in)
```

key
A string literal or an expression of type String

Related concepts
"Syntax diagram for EGL functions" on page 732
"PageHandler" on page 180

Related reference
"EGL library J2EELib" on page 778
"getRequestAttr()"
"setRequestAttr()" on page 780

clearSessionAttr()

The system function **J2EELib.clearSessionAttr** removes the argument that is associated with the specified key in the session object. This function is useful in PageHandlers and in programs that run in Web applications.

You can set an argument in the session object by using the system function J2EELib.setSessionAttr. You can retrieve the argument by using the system function J2EELib.getSessionAttr.

```
J2EELib.clearSessionAttr(key STRING in)
```

key
A string literal or an expression of type STRING

Related concepts
"Syntax diagram for EGL functions" on page 732
"PageHandler" on page 180

Related reference
"EGL library J2EELib" on page 778
"getSessionAttr()" on page 780
"setSessionAttr()" on page 781

getRequestAttr()

The system function **J2EELib.getRequestAttr** uses a specified key to retrieve an argument from the request object into a specified variable. This function is useful in PageHandlers and in programs that run in Web applications.

If an object is not found with the specified key, the target variable is unchanged. If the retrieved object is of the wrong type, an exception is thrown and the program or PageHandler terminates.

You can place an argument in the request object by using the system function **J2EELib.setRequestAttr**. The argument object placed in the servlet's request

collection is available for access as long as the servlet request is valid. Submitting a form from a page causes the creation of a new request.

```
J2EELib.getRequestAttr(
  key STRING in,
  argument attribute inOut)
```

key
A character literal or an item of any character type.

argument
An item, record, or array.

Related concepts
"Syntax diagram for EGL functions" on page 732
"PageHandler" on page 180

Related reference
"EGL library J2EELib" on page 778
"setRequestAttr()"

getSessionAttr()

The system function **J2EELib.getSessionAttr** uses a specified key to retrieve an argument from the session object into a specified variable. This function is useful in PageHandlers and in programs that run in Web applications.

If an object is not found with the specified key, the target variable is unchanged. If the retrieved object is of the wrong type, an exception is thrown and the program or PageHandler terminates.

You can place an argument in the session object by using the system function J2EELib.setSessionAttr.

```
J2EELib.getSessionAttr(
  key STRING in,
  argument attribute in)
```

key
A character literal or an item of any character type.

argument
An item, record, or array.

Related concepts
"Syntax diagram for EGL functions" on page 732
"PageHandler" on page 180

Related reference
"EGL library J2EELib" on page 778
"setSessionAttr()" on page 781

setRequestAttr()

The system function **J2EELib.setRequestAttr** uses a specified key to place a specified argument in the request object. This function is useful in PageHandlers and in programs that run in Web applications. You can retrieve the argument later by using the system function **J2EELib.getRequestAttr**.

```
J2EELib.setRequestAttr(
  key STRING in,
  argument attribute in)
```

key
A character literal or an item of any character type.

argument
An item, record, or array.

In the generated Java output, item arguments are passed as primitive Java objects (String, Integer, Decimal, and so on). Record arguments are passed as record beans. Arrays are passed as an array list of the associated type. The argument object is placed in the servlet's request collection and is available for access as long as the servlet request is valid. Submitting a form from a page causes the creation of a new request.

Related concepts
"Syntax diagram for EGL functions" on page 732
"PageHandler" on page 180

Related reference
"EGL library J2EELib" on page 778
"getRequestAttr()" on page 779

setSessionAttr()

The system function **J2EELib.setSessionAttr** uses a specified key to place a specified argument in the session object. This function is useful in PageHandlers and in programs that run in Web applications. You can retrieve the argument later by using the system function J2EELib.getSessionAttr.

```
J2EELib.setSessionAttr(
  key STRING in,
  argument attribute in)
```

key
A character literal or an item of any character type.

argument
An item, record, or array.

In the generated Java output, item arguments are passed as primitive Java objects (String, Integer, Decimal, and so on). Record arguments are passed as record beans. Arrays are passed as an array list of the associated type.

Related concepts
"Syntax diagram for EGL functions" on page 732
"PageHandler" on page 180

Related reference
"EGL library J2EELib" on page 778
"getSessionAttr()" on page 780

EGL library JavaLib

The Java access functions are listed in the table.

Function	Description
result = getField (*identifierOrClass*, *field*)	Returns the value of a specified field of a specified object or class
result = invoke (*identifierOrClass*, *method*[, *argument*])	Invokes a method on a Java object or class and may return a value

Function	Description
result = isNull (*identifier*)	Returns a value (1 for true, 0 for false) to indicate whether a specified identifier refers to a null object
result = isObjID (*identifier*)	Returns a value (1 for true, 0 for false) to indicate whether a specified identifier is in the object space
result = qualifiedTypeName(*identifier*)	Returns the fully qualified name of the class of an object in the object space
remove (*identifier*)	Removes the specified identifier from the object space and, if no other identifiers refer to the object, removes the object
removeAll ()	Removes all identifiers and objects from the object space
setField (*identifierOrClass*, *field*, *value*)	Sets the value of a field in a Java object or class
store (*storeId*, *identifierOrClass*, *method*{,*argument*})	Invokes a method and places the returned object (or null) into the object space, along with a specified identifier
storeCopy (*sourceId*, *targetID*)	Creates a new identifier based on another in the object space, so that both refer to the same object
storeField (*storeId*, *identifierOrClass*, *field*)	Places the value of a class field or object field into the object space
storeNew(*storeId*, *class*{,*argument*})	Invokes the constructor of a class and places the new object into the object space

Java access functions

The *Java access functions* are EGL system functions that allow your generated Java code to access native Java objects and classes; specifically, to access the public methods, constructors, and fields of the native code.

This EGL feature is made possible at run time by the presence of the *EGL Java object space*, which is a set of names and the objects to which those names refer. A single object space is available to your generated program and to all generated Java code that your program calls locally, whether the calls are direct or by way of another local generated Java program, to any level of call. The object space is not available in any native Java code.

To store and retrieve objects in the object space, you invoke the Java access functions. Your invocations include use of identifiers, each of which is a string that is used to store an object or to match a name that already exists in the object space. When an identifier matches a name, your code can access the object associated with the name.

Note: EGL code that includes a Java access function cannot be generated as a COBOL program.

The next sections are as follows:

- "Mappings of EGL and Java types" on page 783
- "Examples" on page 784
- "Error handling" on page 787

Mappings of EGL and Java types: Each of the arguments you pass to a method (and each value that you assign to a field) is mapped to a Java object or primitive type. Items of EGL primitive type CHAR, for example, are passed as objects of the Java String class. A cast operator is provided for situations in which the mapping of EGL types to Java types is not sufficient.

When you specify a Java name, EGL strips single- and double-byte blanks from the beginning and end of the value, which is case sensitive. The truncation precedes any cast. This rule applies to string literals and to items of type CHAR, DBCHAR, MBCHAR, or UNICODE. No such truncation occurs when you specify either a method argument or field value (for example, the string ″ my data ″ is passed to a method as is), unless you cast the value to objID or null.

The next table describes all the valid mappings.

Category of Argument		Examples	Java Type
A string literal or an item of type CHAR, DBCHAR, MBCHAR, or UNICODE	No cast	`"myString"`	java.lang.String

	Cast with objId, which indicates an identifier	`(objId)"myId"`
		
		`x = "myId";`
		`(objId)X`
			The class of the object to which the identifier refers

	Cast with null, as may be appropriate to provide a null reference to a fully qualified class	`(null)"java.`
		`lang.Thread"`
		
		`x = "java.util.`
		`HashMap";`
		`(null)x`
			The specified class
			Note: You can't pass in a null-casted array such as (null)″int[]″

	Cast with char, which means that the first character of the value is passed (each example in the next column passes an ″a″)	`(char)"abc"`
		
		`X = "abc";`
		`(char)X`
			char

An item of type FLOAT or a floating point literal	No cast	`myFloatValue`	double
An item of type HEX	No cast	`myHexValue`	byte array
An item of type SMALLFLOAT	No cast	`mySmallFloat`	float
An item of type DATE	No cast	`myDate`	java.sql.Date
An item of type TIME	No cast	`myTime`	java.sql.Time
An item of type TIMESTAMP	No cast	`myTimeStamp`	java.sql.Timestamp
An item of type INTERVAL	No cast	`myInterval`	java.lang.String
Floating point literal	No cast	-6.5231E96	double

Category of Argument		Examples	Java Type
Numeric item (or non-floating-point literal) that does not contain decimals; leading zeros are included in the number of digits for a literal	No cast, 1-4 digits	`0100`	short
	No cast, 5–9 digits	`00100`	int
	No cast, 9-18 digits	`1234567890`	long
	No cast, >18 digits	`1234567890123456789`	java.math.BigInteger
Numeric item Numeric item (or non-floating-point literal) that contains decimals; leading and trailing zeros are included in the number of digits for a literal	No cast, 1–6 digits	`3.14159`	float
	No cast, 7-18 digits	`3.14159265`	double
	No cast, >18 digits	`56789543.222`	java.math.BigDecimal

Numeric item or non-floating-point literal, with or without decimals	Cast with bigdecimal, biginteger, byte, double, float, short, int, long	`X = 42;`
		`(byte)X`
		`(long)X`
			The specified primitive type; but if the value is out of range for that type, loss of precision occurs and the sign may change

	Cast with boolean, which means that non-zero is true, zero is false	`X = 1;` `(boolean)X`	boolean

Note: To avoid losing precision, use an EGL float item for a Java double, and an EGL smallfloat item for a Java float. Using one of the other EGL types will probably result in a value being rounded.

For details on the internal format of items in EGL, see the help pages on *Primitive types*.

Examples: This section gives examples on how to use Java access functions.

Printing a date string: The following example prints a date string:

```
// call the constructor of the Java Date class and
// assign the new object to the identifier "date".
JavaLib.storeNew( (objId)"date", "java.util.Date" );

// call the toString method of the new Date object
// and assign the output (today's date) to the chaItem.
// In the absence of the cast (objId), "date"
// refers to a class rather than an object.
charItem = JavaLib.invoke( (objId)"date", "toString" );

// assign the standard output stream of the
// Java System class to the identifier "systemOut".
JavaLib.storeField( (objId)"systemOut",
  "java.lang.System", "out" );

// call the println method of the output
// stream and print today's date.
JavaLib.invoke( (objID)"systemOut","println",charItem );

// The use of "java.lang.System.out" as the first
```

```
// argument in the previous line would not have been
// valid, as the argument must either be a
// an identifier already in the object space or a class
// name. The argument cannot refer to a static field.
```

Testing a system property: The following example retrieves a system property and tests for the absence of a value:

```
// assign the name of an identifier to an item of type CHAR
valueID = "osNameProperty"

// place the value of property os.name into the
// object space, and relate that value (a Java String)
// to the identifier osNameProperty
JavaLib.store((objId)valueId, "java.lang.System",
  "getProperty", "os.name");

// test whether the property value is non-existent
// and process accordingly
myNullFlag = JavaLib.isNull( (objId)valueId );

if( myNullFlag == 1 )
  error = 27;
end
```

Working with arrays: When you work with Java arrays in EGL, use the Java class java.lang.reflect.Array, as shown in later examples and as described in the Java API documentation. You cannot use **JavaLib.storeNew** to create a Java array because Java arrays have no constructors.

You use the static method newInstance of java.lang.reflect.Array to create the array in the object space. After you create the array, you use other methods in that class to access the elements.

The method newInstance expects two arguments:

- A Class object that determines the type of array being created
- A number that specifies how many elements are in the array

The code that identifies the Class object varies according to whether you are creating an array of objects or an array of primitives. The subsequent code that interacts with the array also varies on the same basis.

Working with an array of objects: The following example shows how to create a 5-element object array that is accessible by use of the identifier ″myArray″:

```
// Get a reference to the class, for use with newInstance
JavaLib.store( (objId)"objectClass", "java.lang.Class",
  "forName", "java.lang.Object" );

// Create the array in the object space
JavaLib.store( (objId)"myArray", "java.lang.reflect.Array",
  "newInstance", (objId)"objectClass", 5 );
```

If you want to create an array that holds a different type of object, change the class name that is passed to the first invocation of **JavaLib.store**. To create an array of String objects, for example, pass ″java.lang.String″ instead of ″java.lang.Object″.

To access an element of an object array, use the get and set methods of java.lang.reflect.Array. In the following example, i and length are numeric items:

```
length = JavaLib.invoke( "java.lang.reflect.Array",
  "getLength", (objId)"myArray" );
i = 0;
```

```
while ( i < length )
  JavaLib.store( (objId)"element", "java.lang.reflect.Array",
    "get", (objId)"myArray", i );

  // Here, process the element as appropriate
  JavaLib.invoke( "java.lang.reflect.Array", "set",
    (objId)"myArray", i, (objId)"element" );
  i = i + 1;
end
```

The previous example is equivalent to the following Java code:

```
int length = myArray.length;

for ( int i = 0; i < length; i++ )
{
  Object element = myArray[i];

  // Here, process the element as appropriate

  myArray[i] = element;
}
```

Working with an array of Java primitives: To create an array that stores a Java primitive rather than an object, use a different mechanism in the steps that precede the use of java.lang.reflect.Array. In particular, obtain the Class argument to newInstance by accessing the static field TYPE of a primitive type class.

The following example creates myArray2, which is a 30-element array of integers:

```
// Get a reference to the class, for use with newInstance
JavaLib.storeField( (objId)"intClass",
  "java.lang.Integer", "TYPE");

// Create the array in the object space
JavaLib.store( (objId)"myArray2", "java.lang.reflect.Array",
  "newInstance", (objId)"intClass", 30 );
```

If you want to create an array that holds a different type of primitive, change the Class name that is passed to the invocation of **JavaLib.storeField**. To create an array of characters, for example, pass ″java.lang.Character″ instead of ″java.lang.Integer″.

To access an element of an array of primitives, use the java.lang.reflect.Array methods that are specific to a primitive type. Such methods include getInt, setInt, getFloat, setFloat, and so forth. In the following example, length, element, and i are numeric items:

```
length = JavaLib.invoke( "java.lang.reflect.Array",
  "getLength", (objId)"myArray2" );
i = 0;

while ( i < length )
  element = JavaLib.invoke( "java.lang.reflect.Array",
    "getDouble", (objId)"myArray2", i );

  // Here, process an element as appropriate

  JavaLib.invoke( "java.lang.reflect.Array", "setDouble",
    (objId)"myArray2", i, element );
  i = i + 1;
end
```

The previous example is equivalent to the following Java code:

```
int length = myArray2.length;

for ( int i = 0; i < length; i++ )
{
  double element = myArray2[i];

  // Here, process an element as appropriate

  myArray2[i] = element;
}
```

Working with collections: To iterate over a collection that is referenced by a variable called *list*, a Java program does as follows:

```
Iterator contents = list.iterator();

while( contents.hasNext() )
{
  Object myObject = contents.next();
  // Process myObject
}
```

Assume that hasNext is a numeric data and that your program related a collection to an identifier called *list*. The following EGL code is then equivalent to the Java code described earlier:

```
JavaLib.store( (objId)"contents", (objId)"list", "iterator" );
hasNext = JavaLib.invoke( (objId)"contents", "hasNext" );

while ( hasNext == 1 )
  JavaLib.store( (objId)"myObject", (objId)"contents", "next");

  // Process myObject
  hasNext = JavaLib.invoke( (objId)"contents", "hasNext" );
end
```

Converting an array to a collection: To create a collection from an array of objects, use the asList method of java.util.Arrays, as shown in the following example:

```
// Create a collection from array myArray
// and relate that collection to the identifier "list
JavaLib.store( (objId)"list", "java.util.Arrays",
  "asList", (objId)"myArray" );
```

Next, iterate over list, as shown in the preceding section.

The transfer of an array to a collection works only with an array of objects, not with an array of Java primitives. Be careful not to confuse java.util.Arrays with java.lang.reflect.Array.

Error handling: Many of the Java access functions are associated with error codes, as described in the function-specific help pages. If the value of the system variable **VGVar.handleSysLibraryErrors** is 1 when one of the listed errors occurs, EGL sets the system variable **sysVar.errorCode** to a non-zero value. If the value of **VGVar.handleSysLibraryErrors** is 0 when one of the errors occurs, the program ends.

Of particular interest is the **sysVar.errorCode** value ″00001000″, which indicates that an exception was thrown by an invoked method or as a result of a class initialization.

When an exception is thrown, EGL stores it in the object space. If another exception occurs, the second exception takes the place of the first. You can use the identifier *caughtException* to access the last exception that occurred.

In an unusual situation, an invoked method throws not an exception but an error such as OutOfMemoryError or StackOverflowError. In such a case, the program ends regardless of the value of system variable **VGVar.handleSysLibraryErrors**.

The following Java code shows how a Java program can have multiple catch blocks to handle different kinds of exceptions. This code tries to create a FileOutputStream object. A failure causes the code to set an errorType variable and to store the exception that was thrown.

```
int errorType = 0;
Exception ex = null;

try
{
  java.io.FileOutputStream fOut =
    new java.io.FileOutputStream( "out.txt" );
}
catch ( java.io.IOException iox )
{
  errorType = 1;
  ex = iox;
}
catch ( java.lang.SecurityException sx )
{
  errorType = 2;
  ex = sx;
}
```

The following EGL code is equivalent to the previous Java code:

```
VGVar.handleSysLibraryErrors = 1;
errorType = 0;

JavaLib.storeNew( (objId)"fOut",
  "java.io.FileOutputStream", "out.txt" );

if ( sysVar.errorCode == "00001000" )
  exType = JavaLib.qualifiedTypeName( (objId)"caughtException" );

  if ( exType == "java.io.IOException" )
    errorType = 1;
    JavaLib.storeCopy( (objId)"caughtException", (objId)"ex" );
  else
    if ( exType == "java.lang.SecurityException" )
      errorType = 2;
      JavaLib.storeCopy( (objId)"caughtException", (objId)"ex" );
    end
  end
end
```

Related reference

"EGL library JavaLib" on page 781
"Exception handling" on page 89
"Primitive types" on page 31
"getField()" on page 789
"isNull()" on page 793
"isObjID()" on page 794
"qualifiedTypeName()" on page 795"remove()" on page 796
"removeAll()" on page 797

“setField()” on page 798
“store()” on page 799
“storeCopy()” on page 801
“storeField()” on page 802
“storeNew()” on page 804

getField()

The system function **JavaLib.getField** returns the value of a specified field of a specified object or class. **JavaLib.getField** is one of several Java access functions.

```
JavaLib.getField(
  identifierOrClass javeObjIdOrClass in,
  field STRING in)
returns (result anyJavaPrimitive)
```

result

The result field is required and receives the value of the field specified in the second argument. The following cases apply:

- If the received value is a BigDecimal, BigInteger, byte, short, int, long, float, or double, the result field must be a numeric data type. The characteristics do not need to match the value; for example, a float may be stored in a return variable that is declared with no decimal digits. For details on handling overflow, see *VGVar.handleOverflow* and *sysVar.overflowIndicator*.
- If the received value is a boolean, the result field must be of a numeric primitive type. The value is 1 for true, 0 for false.
- If the received value is a byte array, the result field must be of type HEX. For details on mismatched lengths, see *Assignments*.
- If the received value is a String or char, the result field must be of type CHAR, DBCHAR, MBCHAR, STRING, or UNICODE--
  - If the result field is of type MBCHAR, STRING, or UNICODE, the received value is always appropriate
  - If the result field is of type CHAR, problems can arise if the received value includes characters that correspond to DBCHAR characters
  - If the result field is of type DBCHAR, problems can arise if the received value includes Unicode characters that correspond to single-byte characters

  For details on mismatched lengths, see *Assignments*.
- If the native Java method does not return a value or returns a null, error 00001004 occurs, as listed later.

identifierOrClass

This argument is one of the following entities:

- An identifier that refers to an object in the object space; or
- The fully qualified name of a Java class.

This argument is either a string literal or a variable of type CHAR, DBCHAR, MBCHAR, STRING, or UNICODE. If you are specifying an identifier of an object, the identifier must be cast to objID, as in a later example. If you intend to specify a static field in the next argument, it is recommended that you specify a class in this argument.

EGL strips single- and double-byte blanks from the beginning and end of the argument value, which is case sensitive.

field

The name of the field to read.

This argument is either a string literal or a variable of type CHAR, DBCHAR, MBCHAR, STRING, or UNICODE. Single- and double-byte blanks are stripped from the beginning and end of the string, which is case sensitive.

An example is as follows:

```
myVar = JavaLib.getField( (objId)"myID", "myField" );
```

An error during processing of **JavaLib.getField** can set **sysVar.errorCode** to a value listed in the next table.

Value in sysVar.errorCode	Description
00001000	An exception was thrown by an invoked method or as a result of a class initialization
00001001	The object was null, or the specified identifier was not in the object space
00001002	A public method, field, or class with the specified name does not exist or cannot be loaded
00001004	The method returned null, the method does not return a value, or the value of a field was null
00001005	The returned value does not match the type of the return variable
00001007	A SecurityException or IllegalAccessException was thrown during an attempt to get information about a method or field; or an attempt was made to set the value of a field that was declared final
00001009	An identifier rather than a class name must be specified; the method or field is not static

Related concepts

Related tasks

"Syntax diagram for EGL statements and commands" on page 733

Related reference

"Assignments" on page 352
"BIN and the integer types" on page 47
"EGL library JavaLib" on page 781
"Exception handling" on page 89

"invoke()" on page 791
"isNull()" on page 793
"isObjID()" on page 794
"qualifiedTypeName()" on page 795
"remove()" on page 796
"removeAll()" on page 797
"setField()" on page 798
"store()" on page 799
"storeCopy()" on page 801
"storeField()" on page 802
"storeNew()" on page 804

invoke()

The system function **JavaLib.invoke** invokes a method on a native Java object or class and may return a value. **JavaLib.invoke** is one of several Java access functions.

```
JavaLib.invoke(
  identifierOrClass javaObjIdOrClass in,
  method STRING in
  {, argument anyEglPrimitive in})
returns (result anyJavaPrimitive)
```

result

The result field, if present, receives a value from the native Java method.

If the native Java method returns a value, the result field is optional.

The following cases apply:

- If the returned value is a BigDecimal, BigInteger, byte, short, int, long, float, or double, the result field must be a numeric data type. The characteristics do not need to match the value; for example, a float may be stored in a result field that is declared with no decimal digits. For details on handling overflow, see *VGVar.handleOverflow* and *SysVar.overflowIndicator*.
- If the returned value is a boolean, the result field must be of a numeric primitive type. The value is 1 for true, 0 for false.
- If the returned value is a byte array, the result field must be of type HEX. For details on mismatched lengths, see *Assignments*.
- If the returned value is a String or char, the result field must be of type CHAR, DBCHAR, MBCHAR, STRING, or UNICODE--
  - If the result field is of type MBCHAR, STRING, or UNICODE, the returned value is always appropriate
  - If the result field is of type CHAR, problems can arise if the returned value includes characters that correspond to DBCHAR characters
  - If the result field is of type DBCHAR, problems can arise if the returned value includes Unicode characters that correspond to single-byte characters

  For details on mismatched lengths, see *Assignments*.
- If the native Java method does not return a value or returns a null, the following cases apply:
  - No error occurs in the absence of a result field
  - An error occurs at run time if a result field is present; the error is 00001004, as listed later

identifierOrClass

This argument is one of the following entities:

- An identifier that refers to an object in the object space; or
- The fully qualified name of a Java class.

This argument is either a string literal or an variable of type CHAR, DBCHAR, MBCHAR, STRING, or UNICODE. If you are specifying an identifier of an object, the identifier must be cast to objID, as in a later example. EGL strips single- and double-byte blanks from the beginning and end of the argument value, which is case sensitive.

Your code cannot invoke a method on an object until you have created an identifier for the object. A later example illustrates this point with java.lang.System.out, which refers to a PrintStream object.

method
The name of the method to call.

This argument is either a string literal or an variable of type CHAR, DBCHAR, MBCHAR, STRING, or UNICODE. Single- and double-byte blanks are stripped from the beginning and end of the string, which is case sensitive.

argument
A value passed to the method.

A cast may be required, as specified in *Java access (system words)*.

The Java type-conversion rules are in effect. No error occurs, for example, if you pass a short to a method parameter that is declared as an int.

To avoid losing precision, use an EGL float variable for a Java double, and an EGL smallfloat variable for a Java float. Using one of the other EGL types will probably result in a value being rounded.

The memory area in the invoking program does not change regardless of what the method does.

In the following example, the cast (objId) is required except as noted:

```
// call the constructor of the Java Date class and
// assign the new object to the identifier "date".
JavaLib.storeNew( (objId)"date", "java.util.Date");

// call the toString method of the new Date object
// and assign the output (today's date) to the chaItem.
// In the absence of the cast (objId), "date"
// refers to a class rather than an object.
chaItem = JavaLib.invoke( (objId)"date", "toString" );

// assign the standard output stream of the
// Java System class to the identifier "systemOut".
JavaLib.storeField( (objId)"systemOut", "java.lang.System", "out" );

// call the println method of the output
// stream and print today's date.
JavaLib.invoke( (objID)"systemOut", "println", chaItem );

// The use of "java.lang.System.out" as the first
// argument in the previous line would not have been
// valid, as the argument must either be a
// an identifier already in the object space or a class
// name. The argument cannot refer to a static field.
```

An error during processing of **JavaLib.invoke** can set **SysVar.errorCode** to a value listed in the next table.

Value in SysVar.errorCode	Description
00001000	An exception was thrown by an invoked method or as a result of a class initialization
00001001	The object was null, or the specified identifier was not in the object space
00001002	A public method, field, or class with the specified name does not exist or cannot be loaded
00001003	The EGL primitive type does not match the type expected in Java

Value in SysVar.errorCode	Description
00001004	The method returned null, the method does not return a value, or the value of a field was null
00001005	The returned value does not match the type of the return variable
00001006	The class of an argument cast to null could not be loaded
00001007	A SecurityException or IllegalAccessException was thrown during an attempt to get information about a method or field; or an attempt was made to set the value of a field that was declared final
00001009	An identifier rather than a class name must be specified; the method or field is not static

Related concepts

Related tasks
"Syntax diagram for EGL statements and commands" on page 733

Related reference
"Assignments" on page 352
"BIN and the integer types" on page 47
"EGL library JavaLib" on page 781
"Exception handling" on page 89

"getField()" on page 789
"isNull()"
"isObjID()" on page 794
"qualifiedTypeName()" on page 795
"remove()" on page 796
"removeAll()" on page 797
"setField()" on page 798
"store()" on page 799
"storeCopy()" on page 801
"storeField()" on page 802
"storeNew()" on page 804
"Primitive types" on page 31
"overflowIndicator" on page 906
"handleOverflow" on page 921

isNull()

The system function **JavaLib.isNull** returns a value (1 for true, 0 for false) to indicate whether a specified identifier refers to a null object. **JavaLib.isNull** is one of several Java access functions.

```
JavaLib.isNull(identifier javaObjId in)
returns (result INT)
```

result
A numeric field that receives one of two values: 1 for true, 0 for false. Use of a non-numeric field causes an error at validation time.

identifier

An identifier that refers to an object in the object space.

This argument is either a string literal or an item of type CHAR, DBCHAR, MBCHAR, STRING, or UNICODE. The identifier must be cast to objID. EGL strips single- and double-byte blanks from the beginning and end of the argument value, which is case sensitive.

An example is as follows:

```
// test whether an object is null
// and process accordingly
isNull = JavaLib.isNull( (objId)valueId );

if( isNull == 1 )
  error = 12;
end
```

An error during processing of **JavaLib.isNull** can set **SysVar.errorCode** to a value listed in the next table.

Value in sysVar.errorCode	Description
00001001	The specified identifier was not in the object space

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library JavaLib" on page 781
"getField()" on page 789
"invoke()" on page 791
"isObjID()"
"qualifiedTypeName()" on page 795
"remove()" on page 796
"removeAll()" on page 797
"setField()" on page 798
"store()" on page 799
"storeCopy()" on page 801
"storeField()" on page 802
"storeNew()" on page 804

Related tasks
"Syntax diagram for EGL statements and commands" on page 733

isObjID()

The system function **JavaLib.isObjID** returns a value (1 for true, 0 for false) to indicate whether a specified identifier is in the object space. **JavaLib.isObjID** is one of several Java access functions.

```
JavaLib.isObjID(identifier javaObjId in)
returns (result INT)
```

result
A numeric item that receives one of two values: 1 for true, 0 for false. Use of a non-numeric item causes an error at validation time.

identifier

An identifier that refers to an object in the object space.

This argument is either a string literal or an item of type CHAR, DBCHAR, MBCHAR, or UNICODE. The identifier must be cast to objID. EGL strips single- and double-byte blanks from the beginning and end of the argument value, which is case sensitive.

An example is as follows:

```
// test whether an object is non-existent
// and process accordingly
isPresent = JavaLib.isObjID( (objId)valueId );

if( isPresent == 0 )
  error = 27;
end
```

No run-time errors are associated with **JavaLib.isObjID**.

Related concepts
"Syntax diagram for EGL functions" on page 732
"Java access functions" on page 782

Related tasks
"Syntax diagram for EGL statements and commands" on page 733

Related reference
"EGL library JavaLib" on page 781
"getField()" on page 789
"invoke()" on page 791
"isNull()" on page 793
"qualifiedTypeName()"
"remove()" on page 796
"removeAll()" on page 797
"setField()" on page 798
"store()" on page 799
"storeCopy()" on page 801
"storeField()" on page 802
"storeNew()" on page 804

qualifiedTypeName()

The system function **JavaLib.qualifiedTypeName** returns the fully qualified name of the class of an object in the EGL Java object space. **JavaLib.qualifiedTypeName** is one of several Java access functions.

```
JavaLib.qualifiedTypeName(identifier javaObjId in)
returns (result STRING)
```

result
The result field is required and must be of type CHAR, MBCHAR, or UNICODE--

- If the result field is of type MBCHAR or UNICODE, the received value is always appropriate
- If the result field is of type CHAR, problems can arise if the received value includes characters that correspond to DBCHAR characters

For details on mismatched lengths, see *Assignments*.

identifier
An identifier that refers to an object in the object space.

This argument is either a string literal or an item of type CHAR, DBCHAR, MBCHAR, or UNICODE. The identifier must be cast to objId, as in a later example. EGL strips single- and double-byte blanks from the beginning and end of the argument value, which is case sensitive.

An example is as follows:

```
myItem = JavaLib.qualifiedTypeName( (objId)"myId" );
```

An error during processing of **JavaLib.qualifiedTypeName** can set **sysVar.errorCode** to a value listed in the next table.

Value in sysVar.errorCode	Description
00001001	The object was null, or the specified identifier was not in the object space

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library JavaLib" on page 781
"getField()" on page 789
"invoke()" on page 791
"isNull()" on page 793
"isObjID()" on page 794
"qualifiedTypeName()" on page 795
"remove()"
"removeAll()" on page 797
"setField()" on page 798
"store()" on page 799
"storeCopy()" on page 801
"storeField()" on page 802
"storeNew()" on page 804

Related tasks
"Syntax diagram for EGL statements and commands" on page 733

remove()

The system function **JavaLib.remove** removes the specified identifier from the EGL Java object space. The object related to the identifier is also removed, but only if the identifier is the only one that refers to the object. If another identifier refers to the object, the object remains in the object space and is accessible by way of that other identifier.

JavaLib.remove is one of several Java access functions.

```
JavaLib.remove(identifier javaObjId in)
```

identifier
The identifier that refers to an object. No error occurs if the identifier is not found.

This argument is either a string literal or a variable of type CHAR, DBCHAR, MBCHAR, STRING, or UNICODE. The identifier must be cast to objID, as shown in a later example. EGL strips single- and double-byte blanks from the beginning and end of the argument value, which is case sensitive.

An example is as follows:

```
JavaLib.remove( (objId)myStoredObject );
```

No run-time errors are associated with **JavaLib.remove**.

Note: By invoking the system functions **JavaLib.remove** and **JavaLib.removeAll**, your code allows the Java Virtual Machine to handle garbage collection in the EGL Java object space. If you do not invoke an system function to remove an object from the object space, the memory is not recovered during the run time of any program that has access to the object space.

Related concepts

Related tasks
"Syntax diagram for EGL statements and commands" on page 733

Related reference
"EGL library JavaLib" on page 781

"getField()" on page 789
"invoke()" on page 791
"isNull()" on page 793
"isObjID()" on page 794
"qualifiedTypeName()" on page 795
"removeAll()"
"setField()" on page 798
"store()" on page 799
"storeCopy()" on page 801
"storeField()" on page 802
"storeNew()" on page 804

removeAll()

The system function **JavaLib.removeAll** removes all identifiers and objects from the EGL Java object space. **JavaLib.removeAll** is one of several Java access functions.

```
JavaLib.removeAll( )
```

No runtime errors are associated with **JavaLib.removeAll**.

Note: By invoking the system functions **JavaLib.remove** and **JavaLib.removeAll**, your code allows the Java Virtual Machine to handle garbage collection in the EGL Java object space. If you do not invoke a system function to remove an object from the object space, the memory is not recovered during the run time of any program that has access to the object space.

Related concepts

Related tasks
"Syntax diagram for EGL statements and commands" on page 733

Related reference
"EGL library JavaLib" on page 781

"getField()" on page 789
"invoke()" on page 791
"isNull()" on page 793
"isObjID()" on page 794

“qualifiedTypeName()” on page 795
“remove()” on page 796
“setField()”
“store()” on page 799
“storeCopy()” on page 801
“storeField()” on page 802
“storeNew()” on page 804

setField()

The system function **JavaLib.setField** sets the value of a field in a native Java object or class. **JavaLib.setField** is one of several Java access functions.

```
JavaLib.setField(
  identifierOrClass javaObjId in,
  field STRING in,
  value anyEglPrimitive in)
```

identifierOrClass

This argument is one of the following entities:

- An identifier that refers to an object in the object space; or
- The fully qualified name of a Java class.

This argument is either a string literal or a variable of type CHAR, DBCHAR, MBCHAR, STRING, or UNICODE. If you are specifying an identifier of an object, the identifier must be cast to objID, as in a later example. EGL strips single- and double-byte blanks from the beginning and end of the argument value, which is case sensitive.

field

The name of the field to change.

This argument is either a string literal or a variable of type CHAR, DBCHAR, MBCHAR, STRING, or UNICODE. Single- and double-byte blanks are stripped from the beginning and end of the string, which is case sensitive.

value

The value itself.

A cast may be required, as specified in Java access (system words).

The Java type-conversion rules are in effect. No error occurs, for example, if you assign a short to a field that is declared as an int.

An example is as follows:

```
JavaLib.setField( (objID)"myId", "myField",
  (short)myNumItem );
```

An error during processing of **JavaLib.setField** can set **SysVar.errorCode** to a value listed in the next table.

Value in SysVar.errorCode	Description
00001000	An exception was thrown by an invoked method or as a result of a class initialization
00001001	The object was null, or the specified identifier was not in the object space
00001002	A public method, field, or class with the specified name does not exist or cannot be loaded

Value in SysVar.errorCode	Description
00001003	The EGL primitive type does not match the type expected in Java
00001007	A SecurityException or IllegalAccessException was thrown during an attempt to get information about a method or field; or an attempt was made to set the value of a field that was declared final
00001009	An identifier rather than a class name must be specified; the method or field is not static

Related concepts

"Syntax diagram for EGL statements and commands" on page 733

Related reference

"EGL library JavaLib" on page 781

"getField()" on page 789
"invoke()" on page 791
"isNull()" on page 793
"isObjID()" on page 794
"qualifiedTypeName()" on page 795
"remove()" on page 796
"removeAll()" on page 797
"store()"
"storeCopy()" on page 801
"storeField()" on page 802
"storeNew()" on page 804

store()

The system function **JavaLib.store** invokes a method and places the returned object (or null) into the EGL Java object space, along with a specified identifier. If the identifier is already in the object space, the action is equivalent to the following steps:

- Running **JavaLib.remove** on the identifier to remove the object that was related to that identifier
- Relating the **JavaLib.store**-returned object with the target identifier

If the method returns a Java primitive instead of an object, EGL stores an object that represents the primitive; for example, if the method returns an int, EGL stores an object of type java.lang.Integer.

JavaLib.store is one of several Java access functions.

```
JavaLib.store(
  storeId javaObjId in,
  identifierOrClass javaObjId in,
  method STRING in
  {, argument anyEglPrimitive in} )
```

storeId
The identifier to store with the returned object.

This argument is either a string literal or a variable of type CHAR, DBCHAR, MBCHAR, STRING, or UNICODE. The identifier must be cast to objID, as in a later example. EGL strips single- and double-byte blanks from the beginning and end of the argument value, which is case sensitive.

identifierOrClass

This argument is one of the following entities:

- An identifier that refers to an object in the object space; or
- The fully qualified name of a Java class.

This argument is either a string literal or an item of type CHAR, DBCHAR, MBCHAR, STRING, or UNICODE. If you are specifying an identifier of an object, the identifier must be cast to objID, as in a later example. EGL strips single- and double-byte blanks from the beginning and end of the argument value, which is case sensitive.

method

The method to invoke.

This argument is either a string literal or a variable of type CHAR, DBCHAR, MBCHAR, STRING, or UNICODE. If you are specifying an identifier of an object, the identifier must be cast to objID, as in a later example. EGL strips single- and double-byte blanks from the beginning and end of the argument value, which is case sensitive.

argument

A value passed to the method.

A cast may be required, as specified in Java access (system words).

The Java type-conversion rules are in effect. No error occurs, for example, if you pass a short to a method parameter that is declared as an int.

To avoid losing precision, use an EGL float item for a Java double, and an EGL smallfloat item for a Java float. Using one of the other EGL types will probably result in a value being rounded.

The memory area in the invoking program does not change regardless of what the method does.

An example is as follows:

```
JavaLib.store( (objId)"storeId", (objId)"myId",
  "myMethod", 36 );
```

An error during processing of **JavaLib.store** can set **sysVar.errorCode** to a value listed in the next table.

Value in sysVar.errorCode	Description
00001000	An exception was thrown by an invoked method or as a result of a class initialization
00001001	The object was null, or the specified identifier was not in the object space
00001002	A public method, field, or class with the specified name does not exist or cannot be loaded
00001003	The EGL primitive type does not match the type expected in Java

Value in sysVar.errorCode	Description
00001006	The class of an argument cast to null could not be loaded
00001007	A SecurityException or IllegalAccessException was thrown during an attempt to get information about a method or field; or an attempt was made to set the value of a field that was declared final
00001009	An identifier rather than a class name must be specified; the method or field is not static

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library JavaLib" on page 781
"getField()" on page 789
"invoke()" on page 791
"isNull()" on page 793
"isObjID()" on page 794
"qualifiedTypeName()" on page 795
"remove()" on page 796
"removeAll()" on page 797
"setField()" on page 798
"storeCopy()"
"storeField()" on page 802
"storeNew()" on page 804

storeCopy()

The system function **JavaLib.storeCopy** creates a new identifier based on another in the object space, so that both refer to the same object. If the source identifier is not in the object space, a null is stored for the target identifier and no error occurs. If the target identifier is already in the object space, the action is equivalent to the following steps:

- Running **JavaLib.remove** on the target identifier to remove the object that was related to that identifier
- Relating the source object with the target identifier

JavaLib.storeCopy is one of several Java access functions.

```
JavaLib.storeCopy(
  sourceId javaObjId in,
  targetId javaObjId in)
```

sourceId
An identifier that refers to an object in the object space or to null.

This argument is either a string literal or a variable of type CHAR, DBCHAR, MBCHAR, STRING, or UNICODE. The identifier must be cast to objId, as in a later example. EGL strips single- and double-byte blanks from the beginning and end of the argument value, which is case sensitive.

targetId
The new identifier, which refers to the same object.

This argument is either a string literal or an item of type CHAR, DBCHAR, MBCHAR, STRING, or UNICODE. The identifier must be cast to objID, as in a later example. EGL strips single- and double-byte blanks from the beginning and end of the argument value, which is case sensitive.

An example is as follows:

```
JavaLib.storeCopy( (objId)"sourceId", (objId)"targetId" );
```

No run-time errors are associated with **JavaLib.storeCopy**.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library JavaLib" on page 781
"getField()" on page 789
"invoke()" on page 791
"isNull()" on page 793
"isObjID()" on page 794
"qualifiedTypeName()" on page 795
"remove()" on page 796
"removeAll()" on page 797
"setField()" on page 798
"store()" on page 799
"storeField()"
"storeNew()" on page 804

storeField()

The system function **JavaLib.storeField** places the value of a class field or object field into the EGL Java object space. If the identifier used to store the object is already in the object space, the action is equivalent to the following steps:

- Running **JavaLib.remove** on the identifier to remove the object that was related to the identifier
- Relating the new object with the identifier

If the class or object field contains a Java primitive instead of an object, EGL stores an object that represents the primitive; for example, if the field contains an int, EGL stores an object of type java.lang.Integer.

```
JavaLib.storeField(
  storeId javaObjId in,
  identifierOrClass javaObjIdOrClass in,
  field STRING in)
```

storeId
The identifier to store with the object.

This argument is either a string literal or an item of type CHAR, DBCHAR, MBCHAR, or UNICODE. The identifier must be cast to objID, as in a later example. EGL strips single- and double-byte blanks from the beginning and end of the argument value, which is case sensitive.

identifierOrClass

This argument is one of the following entities:

- An identifier that refers to an object in the object space; or
- The fully qualified name of a Java class.

This argument is either a string literal or an item of type CHAR, DBCHAR, MBCHAR, or UNICODE. If you are specifying an identifier of an object, the identifier must be cast to objID, as in a later example. If you intend to specify a static field in the next argument, it is recommended that you specify a class in this argument.

EGL strips single- and double-byte blanks from the beginning and end of the argument value, which is case sensitive.

field
The name of the field that refers to an object.

This argument is either a string literal or an item of type CHAR, DBCHAR, MBCHAR, or UNICODE. Single- and double-byte blanks are stripped from the beginning and end of the string, which is case sensitive.

An example is as follows:

```
JavaLib.storeField( (objId)"myStoreId",
  (objId)"myId", "myField");
```

An error during processing of **JavaLib.storeField** can set **sysVar.errorCode** to a value listed in the next table.

Value in sysVar.errorCode	Description
00001000	An exception was thrown by an invoked method or as a result of a class initialization
00001001	The object was null, or the specified identifier was not in the object space
00001002	A public method, field, or class with the specified name does not exist or cannot be loaded
00001007	A SecurityException or IllegalAccessException was thrown during an attempt to get information about a method or field; or an attempt was made to set the value of a field that was declared final
00001009	An identifier rather than a class name must be specified; the method or field is not static

Related concepts
“Syntax diagram for EGL functions” on page 732

Related reference
“EGL library JavaLib” on page 781
“getField()” on page 789
“invoke()” on page 791
“isNull()” on page 793
“isObjID()” on page 794
“qualifiedTypeName()” on page 795
“remove()” on page 796
“removeAll()” on page 797
“setField()” on page 798

“store()” on page 799
“storeCopy()” on page 801
“storeNew()”

Related tasks
“Syntax diagram for EGL statements and commands” on page 733

storeNew()

The system function **JavaLib.storeNew** invokes the constructor of a class and places the new object into the EGL Java object space. If the identifier is already in the object space, the action is equivalent to the following steps:

- Running **JavaLib.remove** on the identifier to remove the object previously associated with the identifier
- Relating the new object with the identifier

JavaLib.storeNew is one of several Java access functions.

```
JavaLib.storeNew(
  storeId javaObjId in,
  class STRING in
  {, argument anyEglPrimitive in})
```

storeId
The identifier to store with the new object.

This argument is either a string literal or an item of type CHAR, DBCHAR, MBCHAR, or UNICODE. The identifier must be cast to objID, as in a later example. EGL strips single- and double-byte blanks from the beginning and end of the argument value, which is case sensitive.

class

The fully qualified name of a Java class.

This argument is either a string literal or an item of type CHAR, DBCHAR, MBCHAR, or UNICODE. EGL strips single- and double-byte blanks from the beginning and end of the argument value, which is case sensitive.

argument
A value passed to the constructor.

A cast may be required, as specified in *Java access (system words)*.

The Java type-conversion rules are in effect. No error occurs, for example, if you pass a short to a constructor parameter that is declared as an int.

To avoid losing precision, use an EGL float item for a Java double, and an EGL smallfloat item for a Java float. Using one of the other EGL types will probably result in a value being rounded.

The memory area in the invoking program does not change regardless of what the constructor does.

An example is as follows:

```
JavaLib.storeNew( (objId)"storeId", "myClass", 36 );
```

An error during processing of **JavaLib.storeNew** can set **sysVar.errorCode** to a value listed in the next table.

Value in sysVar.errorCode	Description
00001000	An exception was thrown by an invoked method or as a result of a class initialization
00001001	The object was null, or the specified identifier was not in the object space
00001002	A public method, field, or class with the specified name does not exist or cannot be loaded
00001003	The EGL primitive type does not match the type expected in Java
00001006	The class of an argument cast to null could not be loaded
00001007	A SecurityException or IllegalAccessException was thrown during an attempt to get information about a method or field; or an attempt was made to set the value of a field that was declared final
00001008	The constructor cannot be called; the class name refers to an interface or abstract class

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library JavaLib" on page 781
"getField()" on page 789
"invoke()" on page 791
"isNull()" on page 793
"isObjID()" on page 794
"qualifiedTypeName()" on page 795
"remove()" on page 796
"removeAll()" on page 797
"setField()" on page 798
"store()" on page 799
"storeCopy()" on page 801
"storeField()" on page 802

Related tasks
"Syntax diagram for EGL statements and commands" on page 733

EGL library LobLib

The next table lists the functions in the library LobLib.

System function/Invocation	Description
attachBlobToFile(*blobVariable*, *fileName*)	Copies the data referenced by a variable of type BLOB into a specified file.
attachBlobToTempFile(*blobVariable*)	Copies the data referenced by a variable of type BLOB into a unique, temporary system file.

System function/Invocation	Description
attachClobToFile(*clobVariable*, *fileName*)	Copies the data referenced by a variable of type CLOB into a specified file.
attachClobToTempFile(*clobVariable*)	Copies the data referenced by a variable of type CLOB into a unique, temporary system file.
freeBlob(*blobVariable*)	Releases the resources used by a variable of type BLOB.
freeClob(*clobVariable*)	Releases the resources used by a variable of type CLOB.
result = getBlobLen(*blobVariable*)	Returns the number of bytes in the value referenced by a variable of type BLOB.
result = getClobLen(*clobVariable*)	Returns the number of characters referenced by a variable of type CLOB.
result = getStrFromClob(*clobVariable*)	Returns a string that corresponds to the value referenced by a variable of type CLOB.
result = getSubStrFromClob(*clobVariable*, *pos*, *length*)	Returns a substring from the value referenced by a variable of type CLOB.
loadBlobFromFile(*blobVariable*, *fileName*)	Copies the data from a specified file to a memory area referenced by a variable of type BLOB.
loadClobFromFile(*blobVariable*, *fileName*)	Copies the data from a specified file to a memory area referenced by a variable of type CLOB.
setClobFromString(*clobVariable*, *str*)	Copies a string into a memory area referenced by a variable of type CLOB.
setClobFromStringAtPosition(*clobVariable*, *pos*, *str*)	Copies a string into a memory area referenced by a variable of type CLOB, starting at a specified position in the memory area.
truncateBlob(*blobVariable*, *length*)	Truncates the value referenced by a variable of type BLOB.
truncateClob(*clobVariable*, *length*)	Truncates the value referenced by a variable of type CLOB.
updateBlobToFile(*blobVariable*, *fileName*)	Copies the data referenced by a variable of type BLOB into a specified file.
updateClobToFile(*blobVariable*, *fileName*)	Copies the data referenced by a variable of type CLOB into a specified file.

attachBlobToFile()

The system function **LobLib.attachBlobToFile** copies the data referenced by a variable of type BLOB into a specified file. This function cannot be used in program generated for COBOL.

```
LobLib.attachBlobToFile(
  blobVariable BLOB inOut,
  fileName STRING in)
```

blobVariable
The variable of type BLOB.

fileName
The name of the file. The name is fully qualified or is relative to the directory from which the program is invoked.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"BLOB" on page 46
"EGL library LobLib" on page 805

attachBlobToTempFile()

The system function **LobLib.attachBlobToTempFile** copies the data referenced by a variable of type BLOB into a unique, temporary system file. This function minimizes the memory used at run time.

```
LobLib.attachBlobToTempFile(blobVariable BLOB in)
```

blobVariable
The variable of type BLOB.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"BLOB" on page 46
"EGL library LobLib" on page 805

attachClobToFile()

The system function **LobLib.attachClobToFile** copies the data referenced by a variable of type CLOB into a specified file. This function cannot be used in program generated for COBOL.

```
LobLib.attachClobToFile(
  clobVariable CLOB inOut,
  fileName STRING in)
```

clobVariable
The variable of type CLOB.

fileName
The name of the file. The name is fully qualified or is relative to the directory from which the program is invoked.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"CLOB" on page 45
"EGL library LobLib" on page 805

attachClobToTempFile()

The system function **LobLib.attachClobToTempFile** copies the data referenced by a variable of type CLOB into a unique, temporary system file. This function minimizes the memory used at run time.

```
LobLib.attachClobToTempFile(clobVariable CLOB in)
```

clobVariable
The variable of type CLOB.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"CLOB" on page 45
"EGL library LobLib" on page 805

freeBlob()

The system function **LobLib.freeBlob** releases any resources used by a variable of type BLOB.

```
LobLib.freeBlob(blobVariable BLOB inOut)
```

blobVariable
The variable of type BLOB.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"BLOB" on page 46
"EGL library LobLib" on page 805

freeClob()

The system function **LobLib.freeClob** releases the resources used by a variable of type CLOB.

```
LobLib.freeClob(clobVariable CLOB inOut)
```

clobVariable
The variable of type CLOB.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"CLOB" on page 45
"EGL library LobLib" on page 805

getBlobLen()

The system function **LobLib.getBlobLen** returns the number of bytes in the value referenced by a variable of type BLOB.

```
LobLib.getBlobLen(blobVariable BLOB in)
returns (result BIGINT)
```

result
The number of bytes.

blobVariable
The variable of type BLOB.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"BLOB" on page 46
"EGL library LobLib" on page 805

getClobLen()

The system function **LobLib.getClobLen** returns the number of characters referenced by a variable of type CLOB.

```
LobLib.getClobLen(clobVariable CLOB in)
returns (result BIGINT)
```

result
The number of characters.

clobVariable
The variable of type CLOB.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"CLOB" on page 45
"EGL library LobLib" on page 805

getStrFromClob()

The system function **LobLib.getStrFromClob** returns a string that corresponds to the value referenced by a variable of type CLOB.

```
LobLib.getStrFromClob(clobVariable CLOB in)
returns (result STRING)
```

result
The returned string.

clobVariable
The variable of type CLOB.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"CLOB" on page 45
"EGL library LobLib" on page 805

getSubStrFromClob()

The system function **LobLib.getSubStrFromClob** returns a substring from the value referenced by a variable of type CLOB.

```
LobLib.getSubStrFromClob(
  clobVariable CLOB in,
  pos BIGINT in,
  length BIGINT in)
returns (result STRING)"
```

result
A value of type STRING.

clobVariable
The variable of type CLOB.

pos
Identifies the numeric position of the character that starts the substring. The first character in the CLOB variable is at position 1.

length
Identifies the number of characters in the substring.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"CLOB" on page 45
"EGL library LobLib" on page 805

loadBlobFromFile()

The system function **LobLib.loadBlobFromFile** copies the data from a specified file to a memory area referenced by a variable of type BLOB. This function cannot be used in program generated for COBOL.

```
LobLib.loadBlobFromFile(
  blobVariable BLOB inOut,
  fileName STRING in)
```

blobVariable
The variable of type BLOB.

fileName
The name of the file. The name is fully qualified or is relative to the directory from which the program is invoked.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"BLOB" on page 46
"EGL library LobLib" on page 805

loadClobFromFile()

The system function **LobLib.loadClobFromFile** copies the data from a specified file to a memory area referenced by a variable of type CLOB. This function cannot be used in program generated for COBOL.

```
LobLib.loadClobFromFile(
  clobVariable CLOB inOut,
  fileName STRING in)
```

clobVariable
The variable of type CLOB.

fileName
The name of the file. The name is fully qualified or is relative to the directory from which the program is invoked.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"CLOB" on page 45
"EGL library LobLib" on page 805

setClobFromString()

The system function **LobLib.setClobFromString** copies a string into a memory area referenced by a variable of type CLOB.

```
LobLib.setClobFromString(
  clobVariable CLOB inOut,
  str STRING in)
```

clobVariable
The variable of type CLOB.

str The string to be copied.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"CLOB" on page 45
"EGL library LobLib" on page 805

setClobFromStringAtPosition()

The system function **LobLib.setClobFromStringAtPosition** copies a string into the memory area referenced by a variable of type CLOB, starting at a specified position in the memory area.

```
LobLib.setClobFromStringAtPosition(
  clobVariable CLOB inOut,
  pos BIGINT in
  str STRING in)
```

clobVariable
The variable of type CLOB.

pos
The character position in the value referenced by *clobVariable*. The first character in the CLOB variable is at position 1.

str The string to be copied.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"CLOB" on page 45
"EGL library LobLib" on page 805

truncateBlob()

The system function **LobLib.truncateBlob** truncates the value referenced by a variable of type BLOB.

```
LobLib.truncateBlob(
  blobVariable BLOB inOut,
  length BIGINT in)
```

blobVariable
A variable of type BLOB.

length
The number of bytes in the output.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"BLOB" on page 46
"EGL library LobLib" on page 805

truncateClob()

The system function **LobLib.truncateClob** truncates the value referenced by a variable of type CLOB.

```
LobLib.truncateClob(
  clobVariable CLOB inOut,
  length BIGINT in)
```

clobVariable
A variable of type CLOB.

length
The number of bytes (not characters) in the output.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"CLOB" on page 45
"EGL library LobLib" on page 805

updateBlobToFile()

The system function **LobLib.updateBlobToFile** copies the data referenced by a variable of type BLOB into a specified file. If the file exists, the function first erases the content of the file; otherwise, the function creates the file. This function cannot be used in program generated for COBOL.

```
LobLib.updateBlobToFile(
  blobVariable BLOB inOut,
  fileName STRING in)
```

blobVariable
The variable of type BLOB.

fileName
The name of the file. The name is fully qualified or is relative to the directory from which the program is invoked.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"BLOB" on page 46
"EGL library LobLib" on page 805

updateClobToFile()

The system function **LobLib.updateClobToFile** copies the data referenced by a variable of type CLOB into a specified file. If the file exists, the function first erases the content of the file; otherwise, the function creates the file. This function cannot be used in program generated for COBOL.

```
LobLib.updateClobToFile(
  clobVariable CLOB inOut,
  fileName STRING in)
```

clobVariable
The variable of type CLOB.

fileName
The name of the file. The name is fully qualified or is relative to the directory from which the program is invoked.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"CLOB" on page 45
"EGL library LobLib" on page 805

EGL library MathLib

The next table lists the functions in the system library MathLib.

Note: The field *numericField* is of type BIGINT, BIN, DECIMAL, HEX, INT, NUM, NUMC, PACF, SMALLINT, FLOAT, or SMALLFLOAT.

A field of type HEX (length 8) is assumed to be a single-precision, 4-byte floating-point number that is native to the run-time environment; and a field of type HEX (length 16) is assumed to be a double-precision, 8-byte floating-point number that is native to the run-time environment.

System function/Invocation	Description
result = abs (*numericField*)	Returns absolute value of *numericField*
result = acos (*numericField*)	Returns arccosine of *numericField*
result = asin (*numericField*)	Returns arcsine of *numericField*
result = atan (*numericField*)	Returns arctangent of *numericField*
result = atan2 (*numericField1*, *numericField2*)	Computes the principal value of the arc tangent of *numericField1*/*numericField2*, using the signs of both arguments to determine the quadrant of the return value
result = ceiling (*numericField*)	Returns smallest integer not less than *numericField*
result = compareNum (*numericField1*, *numericField2*)	Returns a result (-1, 0, or 1) that indicates whether *numericField1* is less than, equal to, or greater than *numericField2*
result = cos (*numericField*)	Returns cosine of *numericField*
result = cosh (*numericField*)	Returns hyperbolic cosine of *numericField*
result = exp (*numericField*)	Returns exponential value of *numericField*
result = floatingAssign (*numericField*)	Returns *numericField* as a double-precision floating-point number
result = floatingDifference (*numericField1*, *numericField2*)	Returns the difference between *numericField1* and *numericField2*

System function/Invocation	Description
result = floatingMod (*numericField1*, *numericField2*)	Calculates the floating point remainder of *numericField1* divided by *numericField2*, with the result having the same sign as *numericField1*
result = floatingProduct (*numericField1*, *numericField2*)	Returns product of *numericField1* and *numericField2*
result = floatingQuotient (*numericField1*, *numericField2*)	Returns quotient of *numericField1* divided by *numericField2*
result = floatingSum (*numericField1*, *numericField2*)	Returns sum of *numericField1* and *numericField2*
result = floor (*numericField*)	Returns the largest integer not greater than *numericField*
result = frexp (*numericField*, *integer*)	Splits a number into a normalized fraction in the range of .5 to 1 (which is the returned value) and a power of 2 (which is returned in *integer*
result = Ldexp (*numericField*, *integer*)	Returns *numericField* multiplied by 2 to the power of *integer*
result = log (*numericField*)	Returns the natural logarithm of *numericField*
result = log10 (*numericField*)	Returns the base 10 logarithm of *numericField*
result = maximum (*numericField1*, *numericField2*)	Returns the greater of *numericField1* and *numericField2*
result = minimum (*numericField1*, *numericField2*)	Returns the lesser of *numericField1* and *numericField2*
result = modf (*numericField1*, *numericField2*)	Splits *numericField1* into integral and fractional parts, both with the same sign as *numericField1*; places the integral part in *numericField2*; and returns the fractional part
result = pow (*numericField1*, *numericField2*)	Returns *numericField1* raised to the power of *numericField2*
result = precision (*numericField*)	Returns the maximum precision (in decimal digits) for *numericField*
result = round (*numericField*[, *integer*]) result = mathLib.round(*numericExpression*)	Rounds a number or expression to a nearest value (for example, to the nearest thousands) and returns the result
result = sin (*numericField*)	Returns sine of *numericField*
result = sinh (*numericField*)	Returns hyperbolic sine of *numericField*
result = sqrt (*numericField*)	Returns the square root of *numericField* if *numericField* is greater than or equal to zero
result = stringAsDecimal (*numberAsText*)	Accepts a character value (like *"98.6"*) and returns the equivalent value of type DECIMAL
result = stringAsFloat (*numberAsText*)	Accepts a character value (like *"98.6"*) and returns the equivalent value of type FLOAT
result = stringAsInt (*numberAsText*)	Accepts a character value (like *"98"*) and returns the equivalent value of type BIGINT
result = tan (*numericField*)	Returns the tangent of *numericField*

System function/Invocation	Description
result = tanh (*numericField*)	Returns the hyperbolic tangent of *numericField*

abs()

The system function **MathLib.abs** returns the absolute value of a number.

```
MathLib.abs(numericField mathLibNumber in)
returns (result mathLibTypeDependentResult)
```

result
Any numeric or HEX item, as described in *Mathematical (system words)*. The absolute value of *numericItem* is converted to the format of *result* and returned in *result*.

numericField
Any numeric item or HEX item, as described in *Mathematical (system words)*.

MathLib.abs works on every target system. In relation to Java programs, EGL uses one of the abs() methods in the Java StrictMath class so that the run-time behavior is the same for every Java Virtual Machine.

Example:

```
myItem = -5;
result = MathLib.abs(myItem);  // result = 5
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library MathLib" on page 813

acos()

The system function **MathLib.acos** returns the arccosine of an argument, in radians.

```
MathLib.acos(numericField mathLibNumber in)
returns (result mathLibTypeDependentResult)
```

result
Any numeric or HEX item, as described in *Mathematical (system words)*. The returned value (between 0.0 and pi) is in radians and is converted to the format of *result*.

numericField
Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating point before the calculation occurs. If the value is not between -1 and 1, an error occurs.

MathLib.acos works on every target system. In relation to Java programs, EGL uses the acos() method in the Java StrictMath class so that the run-time behavior is the same for every Java Virtual Machine.

Example:

```
result = MathLib.acos(myItem);
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library MathLib" on page 813

asin()

The system function **MathLib.asin** returns the arcsine of a number that is in the range of -1 to 1. The result is in radians and is in the range of -pi/2 to pi/2.

```
MathLib.asin(numericField mathLibNumber in)
returns (result mathLibTypeDependentResult)
```

result
Any numeric or HEX item, as described in *Mathematical (system words)*. The value returned by the MathLib.asin function is converted to the format of *result* and returned in *result*.

numericField
Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating point before the mathLib.asin function is called.

Example:

```
result = MathLib.asin(myItem);
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library MathLib" on page 813

atan()

The system function **MathLib.atan** returns the arctangent of a number. The result is in radians and is in the range of -pi/2 and pi/2.

```
MathLib.atan(numericField mathLibNumber in)
returns (result mathLibTypeDependentResult)
```

result
Any numeric or HEX item, as described in *Mathematical (system words)*. The value returned by **MathLib.atan** is converted to the format of *result*.

numericField
Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating point before **MathLib.atan** is called.

Example:

```
result = MathLib.atan(myItem);
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library MathLib" on page 813

atan2()

The system function **MathLib.atan2** computes the principal value of the arc tangent of y/x, using the signs of both arguments to determine the quadrant of the return value. The result is in radians and is in the range of -pi to pi.

```
MathLib.atan2(
  numericField1 mathLibNumber in,
  numericField2 mathLibNumber in)
returns (result mathLibTypeDependentResult)
```

result
Any numeric or HEX item, as described in *Mathematical (system words)*. The value returned by **MathLib.atan2** is converted to the format of *result* and returned in *result*.

numericField1
Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating point before **MathLib.atan2** is called. *numericField1* is the y value.

numericField2
Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating point before **MathLib.atan2** is called. *numericField2* is the x value.

Example:

```
myItemY = 1;
myItemX = 5;

// returns pi/2
result = MathLib.atan2(myItemY, myItemX);
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library MathLib" on page 813

ceiling()

The system function **MathLib.ceiling** returns the smallest integer not less than a specified number.

```
MathLib.ceiling(numericField mathLibNumber in)
returns (result mathLibTypeDependentResult)
```

result
Any numeric or HEX item, as described in *Mathematical (system words*). The smallest integer not less than *numericItem* is converted to the format of *result* and returned in *result*.

numericField
Any numeric or HEX item, as described in *Mathematical (system words)*.

Example:

```
myItem = 4.5;
result = MathLib.ceiling(myItem); // result = 5
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library MathLib" on page 813

compareNum()

The system function **MathLib.compareNum** returns a result (-1, 0, or 1) that indicates whether the first of two numbers is less than, equal to, or greater than the second.

```
MathLib.compareNum(
  numericField1 mathLibNumber in,
  numericField2 mathLibNumber in)
returns (result mathLibTypeDependentResult)
```

result
Item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places. This item receives one of the following values:

-1 *numericField1* is less than *numericField2*.

0 *numericField1* is equal to *numericField2*.

1 *numericField1* is greater than *numericField2*.

numericField1
Any numeric or HEX item, as described in *Mathematical (system words)*.

numericField2
Any numeric or HEX item, as described in *Mathematical (system words)*.

Example:

```
myItem01 = 4
myItem02 = 7

result = MathLib.compareNum(myItem01,myItem02);

// result = -1
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library MathLib" on page 813

cos()

The system function **MathLib.cos** returns the cosine of a number. The returned value is in the range of -1 to 1.

```
MathLib.cos(numericField mathLibNumber in)
returns (result mathLibTypeDependentResult)
```

result
Any numeric or HEX item, as described in *Mathematical (system words)*. The value returned by **MathLib.cos** is converted to the format of *result* and returned in *result*.

numericField
Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before **MathLib.cos** is called.

Example:

```
result = MathLib.cos(myItem);
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library MathLib" on page 813

cosh()

The system function **MathLib.cosh** returns the hyperbolic cosine of a number.

```
MathLib.cosh(numericField mathLibNumber in)
returns (result mathLibTypeDependentResult)
```

result
Any numeric or HEX item, as described in *Mathematical (system words)*. The value returned by mathLib.cosh is converted to the format of *result* and returned in *result*.

numericField
Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before mathLib.cosh is called.

Example:

```
result = MathLib.cosh(myItem);
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library MathLib" on page 813

exp()

The system function **MathLib.exp** returns *e* raised to the power of a number.

```
MathLib.exp(numericField mathLibNumber in)
returns (result mathLibTypeDependentResult)
```

result
Any numeric or HEX item, as described in *MathLib*. The value returned by **MathLib.exp** is converted to the format of *result* and returned in *result*.

numericField
Any numeric or HEX item, as described in *MathLib*. The item is converted to double-precision floating-point before **MathLib.exp** is called.

Example:

```
result = MathLib.exp(myItem);
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library MathLib" on page 813

floatingAssign()

The system function **MathLib.floatingAssign** returns *numericItem* as a double-precision floating-point number. The function assigns the value of BIN, DECIMAL, NUM, NUMC, or PACKF items to floating-point numbers that are defined as HEX items, and vice versa.

```
MathLib.floatingAssign(numericField mathLibNumber in)
returns (result mathLibTypeDependentResult)
```

result
Any numeric or HEX item, as described in *Mathematical (system words)*. The floating-point number is converted to the format of *result* and returned in *result*.

numericField
Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before being assigned to the result.

Example:

```
result = MathLib.floatingAssign(myItem);
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library MathLib" on page 813

floatingDifference()

The system function **MathLib.floatingDifference** subtracts the second of two numbers from the first and returns the difference. The function is implemented using double-precision floating-point arithmetic.

```
MathLib.floatingDifference(
   numericField1 mathLibNumber in,
   numericField2 mathLibNumber in)
 returns (result mathLibTypeDependentResult)
```

result
Any numeric or HEX item, as described in *Mathematical (system words)*. The difference is converted to the format of *result* and returned in *result*.

numericField1
Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before the difference is calculated.

numericField2
Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before the difference is calculated.

Example:

```
result = MathLib.floatingDifference(myItem01,myItem02);
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library MathLib" on page 813

floatingMod()

The system function **MathLib.floatingMod** returns the floating-point remainder of one number divided by another. The result has the same sign as the numerator. A domain exception is raised if the denominator equals zero.

```
MathLib.floatingMod(
  numericField1 mathLibNumber in,
  numericField2 mathLibNumber in)
returns (result mathLibTypeDependentResult)
```

result
Any numeric or HEX item, as described in *Mathematical (system words)*. The floating-point remainder is converted to the format of *result* and returned in *result*.

numericField1
Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before *result* is calculated.

numericField2
Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before *result* is calculated.

Example:

```
result = MathLib.floatingMod(myItem01,myItem02);
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library MathLib" on page 813

floatingProduct()

The system function **MathLib.floatingProduct** returns the product of two numbers. The function is implemented using double-precision floating-point arithmetic.

```
MathLib.floatingProduct(
  numericField1 mathLibNumber in,
  numericField2 mathLibNumber in)
returns (result mathLibTypeDependentResult)
```

result
Any numeric or HEX item, as described in *Mathematical (system words)*. The product is converted to the format of *result* and returned in *result*.

numericField1
Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before *result* is calculated.

numericField2
Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before *result* is calculated.

Example:

```
result = MathLib.floatingProduct(myItem01,myItem02);
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library MathLib" on page 813

floatingQuotient()

The system function **MathLib.floatingQuotient** returns the quotient of one number divided by another. A domain exception is raised if the denominator equals zero. The function is implemented using double-precision floating-point arithmetic.

```
MathLib.floatingQuotient(
  numericField1 mathLibNumber in,
  numericField2 mathLibNumber in)
returns (result mathLibTypeDependentResult)
```

result
Any numeric or HEX item, as described in *Mathematical (system words)*. The quotient is converted to the format of *result* and returned in *result*.

numericField1
Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before the quotient is calculated.

numericField1
Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before the quotient is calculated.

Example:

```
result = MathLib.floatingQuotient(myItem01,myItem02);
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library MathLib" on page 813

floatingSum()

The system function **MathLib.floatingSum** returns the sum of two numbers. The function is implemented using double-precision floating-point arithmetic.

```
MathLib.floatingSum(
  numericField1 mathLibNumber in,
  numericField2 mathLibNumber in)
returns (result mathLibTypeDependentResult)
```

result
Any numeric or HEX item, as described in *Mathematical (system words)*. The sum is converted to the format of *result* and returned in *result*.

numericField1
Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before the sum is calculated.

numericField2
Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before the sum is calculated.

Example:

```
result = MathLib.floatingSum(myItem01,myItem02);
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library MathLib" on page 813

floor()

The system function **MathLib.floor** returns the largest integer not greater than a specified number.

```
MathLib.floor(numericField mathLibNumber in)
 returns (result mathLibTypeDependentResult)
```

result
Any numeric or HEX item, as described in *Mathematical (system words)*. The largest integer not greater than*numericField* is converted to the format of *result* and returned in *result*.

numericField
Any numeric or HEX item, as described in *Mathematical (system words)*.

Example:

```
myItem = 4.6;
result = MathLib.floor(myItem); // result = 4
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library MathLib" on page 813

frexp()

The system function **MathLib.frexp** splits a number into a normalized fraction in the range of .5 to 1 (which is returned as the *result*) and a power of 2 (which is returned in *exponent*).

```
MathLib.frexp(
  numericField mathLibNumber in,
  exponent mathLibInteger inOut)
returns (result mathLibTypeDependentResult)
```

result
Any numeric or HEX item, as described in *Mathematical (system words)*. The floating-point fraction is converted to the format of *result* and returned in *result*.

numericField
Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before *result* is calculated.

exponent
Item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

Example:

```
result = MathLib.frexp(myItem,myInteger);
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library MathLib" on page 813

Ldexp()

The system function **MathLib.Ldexp** returns the value of a specified number that is multiplied by the following value: two to the power of *exponent*.

```
MathLib.Ldexp(
  numericField mathLibNumber in,
  exponent mathLibInteger in)
returns (result mathLibTypeDependentResult)
```

result
Any numeric or HEX item, as described in *Mathematical (system words)*. The calculated value is converted to the format of *result* and returned in *result*.

numericField
Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before *result* is calculated.

exponent
Item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

Example:

```
result = MathLib.Ldexp(myItem,myInteger);
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library MathLib" on page 813

log()

The system function **MathLib.log** returns the natural logarithm of a number.

```
MathLib.log(numericField mathLibNumber in)
returns (result mathLibTypeDependentResult)
```

result
Any numeric or HEX item, as described in *Mathematical (system words)*. The value returned by the mathLib.log function is converted to the format of *result* and returned in *result*.

numericField
Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before *result* is calculated.

Example:

```
result = MathLib.log(myItem);
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library MathLib" on page 813

log10()

The system function **MathLib.log10** returns the base 10 logarithm of a number.

```
MathLib.log10(numericField mathLibNumber in)
returns (result mathLibTypeDependentResult)
```

result
Any numeric or HEX item, as described in *Mathematical (system words)*. The value returned by the log10 function is converted to the format of *result* and returned in *result*.

numericField
Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before *result* is calculated.

Example:

```
result = MathLib.log10(myItem);
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library MathLib" on page 813

maximum()

The system function **MathLib.maximum** returns the greater of two numbers.

```
MathLib.maximum(
   numericField1 mathLibNumber in,
   numericField2 mathLibNumber in)
  returns (result mathLibTypeDependentResult)
```

result
Any numeric or HEX item, as described in *Mathematical (system words)*. The greater of two numbers is converted to the format of *result* and returned in *result*.

numericField1
Any numeric or HEX item, as described in *Mathematical (system words)*.

numericField2
Any numeric or HEX item, as described in *Mathematical (system words)*.

Example:

```
result = MathLib.maximum(myItem01,myItem02);
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library MathLib" on page 813

minimum()

The system function **MathLib.minimum** returns the lesser of two numbers.

```
MathLib.minimum(
  numericField1 mathLibNumber in,
  numericField2 mathLibNumber in)
returns (result mathLibTypeDependentResult)
```

result
Any numeric or HEX item, as described in *Mathematical (system words)*. The lesser of two numbers is converted to the format of *result* and returned in *result*.

numericField1
Any numeric or HEX item, as described in *Mathematical (system words)*.

numericField2
Any numeric or HEX item, as described in *Mathematical (system words)*.

Example:

```
result = MathLib.minimum(myItem01,myItem02);
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library MathLib" on page 813

modf()

The system function **MathLib.modf** splits a number into integral and fractional parts, both with the same sign as the number. The fractional part is returned in *result* and the integral part is returned in *numericField2*.

```
MathLib.modf(
  numericField1 mathLibNumber in,
  numericField2 mathLibNumber inOut)
returns (result mathLibTypeDependentResult)
```

result
Any numeric or HEX item, as described in *Mathematical (system words)*. The fractional part of *numericField1* is converted to the format of *result* and returned in *result*.

numericField1
Any numeric or HEX item, as described in *Mathematical (system words)*.

numericField2
Any numeric or HEX item, as described in *Mathematical (system words)*. The integral part of *numericField1* is converted to the format of *numericField2* and returned in *numericField2*.

Example:

```
result = MathLib.modf(myItem01,myItem02);
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library MathLib" on page 813

pow()

The system function **MathLib.pow** returns a number raised to the power of a second number. A domain exception is raised if on pow(x,y) the value of x is negative and y is non-integral, or the value of x is 0.0 and y is negative.

```
MathLib.pow(
  numericField1 mathLibNumber in,
  numericField2 mathLibNumber in)
returns (result mathLibTypeDependentResult)
```

result
Any numeric or HEX item, as described in *Mathematical (system words)*. The result of the mathLib.pow function is converted to the format of *result* and returned in *result*.

numericField1
Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before *result* is calculated.

numericField2
Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before *result* is calculated.

Example:

```
result = MathLib.pow(myItem01,myItem02);
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library MathLib" on page 813

precision()

The system function **MathLib.precision** returns the maximum precision (in decimal digits) for a number. For floating-point numbers (8-digit HEX for standard-precision floating-point number or 16-digit HEX for double-precision floating-point number), the precision is the maximum number of decimal digits that can be represented in the number for the system on which the program is running.

```
MathLib.precision(numericField mathLibNumber in)
returns (result INT)
```

result
An item that receives the precision of *numericItem*. The *result* item is defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

numericField
Any numeric or HEX item, as described in *Mathematical (system words)*.

Example:

```
result = MathLib.precision(myItem);
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library MathLib" on page 813

round()

The system function **MathLib.round** rounds a number or expression to a nearest value (for example, to the nearest thousands) and returns the result.

```
MathLib.round(
  numericField mathLibNumber in
  [, powerOf10 mathLibInteger in
  ])
returns (result mathLibTypeDependentResult)
MathLib.round(numericExpression anyNumericExpression in
returns (result mathLibTypeDependentResult)
```

result
Any numeric or HEX item, as described in *Mathematical (system words)*. The value produced by the rounding operation is converted to the format of *result* and returned in *result*.

The maximum supported length in this case is 31 rather than 32 because rounding occurs as follows:

- Add five to the digit in *result* at a precision one higher than the precision of the result digit
- Truncate the result

A numeric overflow occurs at run time if more than 31 digits are used in the calculation and if EGL cannot determine the violation at development time.

numericField
Any numeric or HEX item, as described in *Mathematical (system words)*.

numericExpression
A numeric expression other than simply a numeric item. If you specify an operator, you cannot specify a value for *powerOf10*.

You cannot use **MathLib.round** with the remainder operator (%).

powerOf10
An integer that determines the value to which the number is rounded:

- If the integer is positive, the number is rounded to a nearest value equal to 10 to the power of *powerOf10*. If integer is 3, for example, the number is rounded to the nearest thousands.
- The same is true if the integer is zero or negative; in that case, the number is rounded to the specified number of decimal places.

If you do not specify *powerOf10*, **MathLib.round** rounds to the number of decimal places in *result*.

The integer is defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

Examples: In the next example, item `balance` is rounded to the nearest thousand:

```
balance = 12345.6789;
rounder = 3;
balance = MathLib.round(balance, rounder);
// The value of balance is now 12000.0000
```

In the next example, a `rounder` value of -2 is used to round `balance` to two decimal places:

```
balance = 12345.6789;
rounder = -2;
balance = mathLib.round(balance, rounder);
// The value of balance is now 12345.6800
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library MathLib" on page 813

sin()

The system function **MathLib.sin** that returns the sine of a number. The result is in the range of -1 to 1.

```
MathLib.sin(numericField mathLibNumber in)
returns (result mathLibTypeDependentResult)
```

result
Any numeric or HEX item, as described in *Mathematical (system words)*. The value returned by the **MathLib.sin** function is converted to the format of *result* and returned in *result*.

numericField
Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before *result* is calculated.

Example:

```
result = MathLib.sin(myItem);
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library MathLib" on page 813

sinh()

The system function **MathLib.sinh** returns the hyperbolic sine of a number.

```
MathLib.sinh(numericField mathLibNumber in)
returns (result mathLibTypeDependentResult)
```

result
Any numeric or HEX item, as described in *Mathematical (system words)*. The value returned by the **MathLib.sinh** function is converted to the format of *result* and returned in *result*.

numericField
Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before *result* is calculated.

Example:

```
result = MathLib.sinh(myItem);
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library MathLib" on page 813

sqrt()

The math function **MathLib.sqrt** returns the square root of a number. The function operates on any number that is greater than or equal to zero.

```
MathLib.sqrt(numericField mathLibNumber in)
returns (result mathLibTypeDependentResult)
```

result
Any numeric or HEX item, as described in *Mathematical (system words)*. The value returned by the **MathLib.sqrt** function is converted to the format of *result* and returned in *result*.

numericField
Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before *result* is calculated.

Example:

```
result = MathLib.sqrt(myItem);
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library MathLib" on page 813

stringAsDecimal()

The system function **MathLib.stringAsDecimal** accepts a character value (like *"98.6"*) and returns the equivalent value of type DECIMAL.

```
MathLib.stringAsDecimal(numberAsText STRING in)
returns (result DECIMAL)
```

result
A value of type DECIMAL. The receiving field can have any decimal position and any length.

EGL allows as many as 32 digits on either side of the decimal point. If you are generating Java code, the decimal point (if any) is specific to the locale.

For details on the implications of assigning numeric values to fields of different types, see *Assignments*.

numberAsText
A character field or literal string, which can include an initial sign character.

Example:

```
myField = "-5.243";

// result = -5.243
result = MathLib.stringAsDecimal(myField);
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"Assignments" on page 352
"EGL library MathLib" on page 813

stringAsFloat()

The system function **MathLib.stringAsFloat** accepts a character value (like *"98.6"*) and returns the equivalent value of type FLOAT.

```
MathLib.stringAsFloat(numberAsText STRING in)
returns (result FLOAT)
```

result
A value of type FLOAT. The receiving field can have any decimal position and any length. If you are generating Java code, the decimal point (if any) is specific to the locale.

For details on the implications of assigning numeric values to fields of different types, see *Assignments*.

numberAsText
A character field or literal string, which can include an initial sign character.

Example:

```
myField = "-5.243";

// result = -5.243
result = MathLib.stringAsFloat(myField);
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"Assignments" on page 352
"EGL library MathLib" on page 813

stringAsInt()

The system function **MathLib.stringAsInt** accepts a character value (like *"98"*) and returns the equivalent value of type BIGINT.

```
MathLib.stringAsInt(numberAsText STRING in)
returns (result BIGINT)
```

result
A value of type BIGINT.

For details on the implications of assigning numeric values to fields of different types, see *Assignments*.

numberAsText
A character field or literal string, which can include an initial sign character.

Example:

```
myField = "-5";

// result = -5
result = MathLib.stringAsInt(myField);
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library MathLib" on page 813

tan()

The system function **MathLib.tan** returns the tangent of a number.

```
MathLib.tan(numericField mathLibNumber in)
returns (result mathLibTypeDependentResult)
```

result
Any numeric or HEX item, as described in *Mathematical (system words)*. The value returned by the **MathLib.tan** function is converted to the format of *result* and returned in *result*.

numericField
Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before *result* is calculated.

Example:

```
result = MathLib.tan(myItem);
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library MathLib" on page 813

tanh()

The system function **MathLib.tanh** returns the hyperbolic tangent of a number. The result is in the range of -1 to 1.

```
MathLib.tanh(numericField mathLibNumber in)
returns (result mathLibTypeDependentResult)
```

result
Any numeric or HEX item, as described in *Mathematical (system words)*. The value returned by the **MathLib.tanh** function is converted to the format of *result* and returned in *result*.

numericField
Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before *result* is calculated.

Example:

```
result = MathLib.tanh(myItem);
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library MathLib" on page 813

recordName.resourceAssociation

When your program does an I/O operation against a record, the I/O is done on the physical file whose name is in the record-specific variable *recordName*.**resourceAssociation**. The variable is initialized in accordance with the resourceAssociation part used at generation time; for details, see *Resource associations and file types*. You can change the system resource name at run time by placing a different value in **resourceAssociation**.

In most cases, you must use the syntax *recordName*.**resourceAssociation**. You do not need to specify a record name, however, if EGL can determine the record that you intended, as is true in each of these cases:

- I/O is only performed against one record in the program
- **resourceAssociation** is used in a function that performs I/O against only one record
- I/O is performed against multiple records in the program, but all records have the same file name; in this case, the first record that appears as an I/O object is used as the implicit qualifier.

You can use **resourceAssociation** as any of the following:

- The source or target operand of an assignment statement
- An item in a logical expression in a **case**, **if**, or **while** statement
- The argument in a **return** or **exit** statement

The characteristics of **resourceAssociation** are as follows:

Primitive type
CHAR

Data length
Varies by file type

Saved across segment?
Yes

Definition considerations

The value moved into *recordName*.**resourceAssociation** must be a valid system resource name for the system and file type that were specified when the program was generated. If more than one record specifies the same file name, modification of **resourceAssociation** for any record with that file name changes the setting of **resourceAssociation** for all records in the program with the same file name.

If a system resource identified in the setting of **resourceAssociation** is open when that record-specific variable is modified, the system resource that *was* in that variable is closed in the following circumstance: an I/O option runs against a record that has the same EGL file name as the record that qualifies **resourceAssociation**.

If two programs are using the same EGL file name, each of the record-specific **resourceAssociation** variables must contain the same value. Otherwise the previously opened system resource is closed when a new one is opened.

A comparison of **resourceAssociation** with another value tests true only if the match is exact. If you initialize **resourceAssociation** with a lowercase value, for example, the lowercase value matches only a lowercase value.

The value that you place in **resourceAssociation** remains unchanged for purposes of comparison.

Files shared across programs: You can set the system resource name either at generation or at runtime:

At generation time
If two programs in the same run unit access the same logical file, you must specify the same system resource name for the logical file at generation to ensure that both programs access the same physical file at run time.

At run time
If you use *recordName*.**resourceAssociation**, each program that accesses the file must set **resourceAssociation** for the file. If two programs in the same run unit access the same logical file, each program must set **resourceAssociation** to the same system resource name to ensure that both programs access the same physical file at run time.

If a system resource is shared by multiple programs, each program that accesses the resource must set **resourceAssociation** to refer to the same resource. Also, if two programs in the same run unit access the same logical file, each program must set **resourceAssociation** to the same system resource name at generation time to ensure that both programs access the same system resource at run time.

MQ records: The system resource name for MQ records defines the queue manager name and queue name. Specify the name in the following format:

```
queueManagerName:queueName
```

queueManagerName
Name of the queue manager.

queueName
Name of the queue.

As shown, the names are separated with a colon. However, *queueManagerName* and the colon can be omitted. The system resource name is used as the initial value for the record-specific **resourceAssociation** item and identifies the default queue associated with the record. For further details, see *MQSeries support*.

Target platforms

Platform	Compatibility considerations
iSeries COBOL	The filetype must be SEQ or VSAM. The value can be moved to **resourceAssociation** in one of the following ways: **LIB/FILE MEMBER** Explicitly specify Library, File and Member **LIB/FILE** The first member in the file is used **FILE MEMBER** *LIBL is used to find the file **FILE** *LIBL is used to find the file, and the first member in that file is used When you modify the value in **resourceAssociation**, the iSeries OVRDBF command has this effect: 1. Closes the old file 2. Performs an override to the new value 3. Opens the new file The value set in **resourceAssociation** is propagated from the call level and is changed to all its subordinate call levels. The value is not propagated if the file previously was opened by the program.
Java platforms	None.

Example

```
if (process == 1)
  myrec.resourceAssociation = "myFile.txt";
else
  myrec.resourceAssociation = "myFile02.txt";
end
```

Related concepts
"MQSeries support" on page 247
"Resource associations and file types" on page 286

Related reference

EGL library ReportLib

ReportLib, the EGL report library, is a system library that establishes a framework containing all of the components that are needed to interact with the JasperReports library. The EGL report library includes the following components:

- Functions, variables, and constants that are used for these purposes:
  - To interact with JasperReports library functions
  - To define, set, and retrieve the data source for a report
  - To export a filled report to different file formats
  - To manipulate the contents of the report and process report data
- Records containing names of files that store the report design, filled report, and exported report.
- The report

The report library includes the following functions:

System function/Invocation	Description
addReportParameter(*report*, *parameterString*, *parameterValue*)	Adds a value to the parameter list of the report
fillReport(*report*, *source*)	Fills the report using the specified data source
exportReport(*report*, *format*)	Exports the filled report in the specified format
resetReportParameters(*report*)	Removes all of the parameters used for a particular report

The following functions are invoked only within report handlers:

System function/Invocation	Description
addReportData(*rd*, *dataSetName*)	Adds the report data object with the specified name to the current Report Handler.
result = getReportData(*dataSetName*)	Retrieves the report data record with the specified name. The returned value is of type ReportData.
result = getReportParameter(*parameter*)	Returns the value of the specified parameter from the report that is being filled.
result = getFieldValue(*fieldName*)	Returns the value of the specified field value for the row currently being processed. The returned value is of type ANY.
result = getReportVariableValue(*variable*)	Returns the value of the specified variable from the report that is being filled. The returned value is of type ANY.
setReportVariableValue(*variable*, *value*)	Sets the value of the specified variable to the provided value.

Note: If you delete an EGL report, you must remove all references to the report.

Related concepts
"Data sources" on page 196
"EGL report creation process overview" on page 194
"EGL reports overview" on page 193

addReportData()

The system function **ReportLib.addReportData** makes the variable of type ReportData available in either of two ways:

- By invoking **ReportLib.getReportData**
- By invoking the report handler method getDataSource in the design file

```
ReportLib.addReportData(
  rd ReportData in,
  dataID STRING in)
```

rd A variable of type reportData

dataID
An arbitrary name that can be used to access the variable

Related concepts
"EGL reports overview" on page 193
"EGL report creation process overview" on page 194
"Syntax diagram for EGL functions" on page 732

Related reference
"addReportParameter()"
"EGL library ReportLib" on page 834
"exportReport()"
"fillReport()" on page 837

addReportParameter()

The syntax diagram for the **ReportLib.addReportParameter** function is as follows:

```
ReportLib.addReportParameter(
  report Report in,
  parameterString STRING in,
  parameterValue any in)
```

report
The name of the report

parameterString
The name of the parameter

parameterValue
The value of the parameter

Before filling a report, EGL can pass a set of parameters that either establish values to be used in the report or override parameters specified in the XML report design. The **ReportLib.addReportParameter** function adds the value of the specified parameter to the parameter list of the report.

Note: See JasperReports documentation for information on JasperReports parameters and data types.

Related concepts
"Syntax diagram for EGL functions" on page 732
EGL report overview
EGL report creation process overview

Related reference
EGL report library
ReportLib.fillReport function
ReportLib.exportReport function
ReportLib.resetReportParameters function

exportReport()

The system function **ReportLib.exportReport** exports the filled report in the format you specify.

The following diagram illustrates the syntax of that function:

```
ReportLib.exportReport(
  report Report in,
  format ExportFormat in)
```

report
The report being exported.

format
The format and file extension of the exported report.

The values are of the enumeration **ExportFormat**:

csv
The output show one value separated from the next with a comma; **csv** stands for comma-separated values.

html
The output is in HTML format.

pdf
The output is in Adobe Acrobat PDF format.

text
The output is in ASCII text format.

Related concepts
"EGL reports overview" on page 193
"EGL report creation process overview" on page 194
"Enumerations in EGL" on page 471
"Syntax diagram for EGL functions" on page 732

Related tasks
"Exporting Reports" on page 211

Related reference
"addReportParameter()" on page 836
"EGL library ReportLib" on page 834
"fillReport()"
"resetReportParameters()" on page 840

fillReport()

The syntax diagram for the **ReportLib.fillReport** function is as follows:

```
ReportLib.fillReport(
  report Report in,
  source DataSource in)
```

report
The report to be filled with data.

source
The source of the data that is used to fill the report.

Consider this example, which shows how a variable of type **reportData** is associated with the report:

```
eglReport     Report;
eglReportData ReportData;
eglReport.reportData = eglReportData;
```

source indicates which field to use in the variable of type **ReportData**. Each value of *source* is not a field name, but a value in the enumeration **DataSource**:

databaseConnection
Use the variable that is referenced in the **connectionName** field of the **reportData** variable, as in this example:

```
eglReportData.connectionName = "mycon";
```

In this case, the SQL statement that accesses data is in the report design file, which is created outside of EGL.

reportData
Use the variable that is referenced in the **data** field of the **reportData** variable, as in this example:

```
// an array of records, with data
myRecords customerRecord[];

eglReportData.data = myRecords;
```

sqlStatement
Use the SQL statement identified in the **sqlStatement** field of the **reportData** variable, as in this example:

```
mySQLString = "Select * From MyTable";
eglReportData.sqlStatement = mySQLString;
```

Following is an example invocation:

```
ReportLib.fillReport (eglReport, DataSource.sqlStatement);
```

Related concepts
"EGL reports overview" on page 193
"EGL report creation process overview" on page 194
"Enumerations in EGL" on page 471
"Syntax diagram for EGL functions" on page 732

Related reference
"Data sources" on page 196
"EGL library ReportLib" on page 834
"addReportParameter()" on page 836
"exportReport()" on page 836
"resetReportParameters()" on page 840

getFieldValue()

The **ReportLib.getFieldValue** function returns the value of the specified field for the row currently being processed.

```
ReportLib.getFieldValue(fieldName STRING in)
returns (result ANY)
```

result
The value of the specified field

fieldName
The name of the specified field

Related concepts
"Syntax diagram for EGL functions" on page 732
"EGL reports overview" on page 193
"EGL report creation process overview" on page 194

Related reference
"addReportParameter()" on page 836
"fillReport()" on page 837
"EGL library ReportLib" on page 834
"exportReport()" on page 836

getReportData()

The system function **ReportLib.getReportData** retrieves the report data by using a name specified in **ReportLib.addReportData**.

```
ReportLib.getReportData(dataID STRING in)
returns (result ReportData)
```

result
The value of type ReportData

dataID
A name assigned to the variable during an invocation of **ReportLib.addReportData**

Related concepts
"Syntax diagram for EGL functions" on page 732
"EGL reports overview" on page 193
"EGL report creation process overview" on page 194

Related reference
"addReportParameter()" on page 836
"EGL library ReportLib" on page 834
"exportReport()" on page 836
"fillReport()" on page 837

getReportParameter()

The **ReportLib.getReportParameter** function returns the value of the specified parameter from the report that is being filled.

```
ReportLib.getReportParameter(parameter STRING in)
returns (result ANY)
```

result
The value of the parameter

parameter
The name of the parameter

Related concepts
"Syntax diagram for EGL functions" on page 732
"EGL reports overview" on page 193
"EGL report creation process overview" on page 194

Related reference
"EGL library ReportLib" on page 834
"addReportParameter()" on page 836
"fillReport()" on page 837
"exportReport()" on page 836

getReportVariableValue()

The system function **ReportLib.getReportVariableValue** returns the value of the specified variable from the report that is being filled.

```
ReportLib.getReportVariableValue(variable STRING in)
returns (result ANY)
```

result
The returned value

variable
The variable of interest

Related concepts
"Syntax diagram for EGL functions" on page 732
"EGL reports overview" on page 193
"EGL report creation process overview" on page 194

Related reference
"addReportParameter()" on page 836
"EGL library ReportLib" on page 834
"exportReport()" on page 836
"fillReport()" on page 837

resetReportParameters()

The syntax diagram for the **ReportLib.resetReportParameters** function is as follows:

```
ReportLib.resetReportParameters(report Report in)
```

report
The name of the report that contains the parameters you want to remove.

The **ReportLib.resetReportParameters** function removes all of the EGL parameters used for a particular report.

Related concepts
"EGL reports overview" on page 193
"EGL report creation process overview" on page 194
"Syntax diagram for EGL functions" on page 732

Related reference
"addReportParameter()" on page 836
"EGL library ReportLib" on page 834
"exportReport()" on page 836
"fillReport()" on page 837

setReportVariableValue()

The **ReportLib.setReportVariableValue** function sets the value of the specified variable to a value provided to the function.

```
ReportLib.setReportVariableValue(
  variable STRING  in,
  value Any in)
```

variable
The variable to set

value
The value to assign

Related concepts
"EGL report creation process overview" on page 194
"EGL reports overview" on page 193
"Syntax diagram for EGL functions" on page 732

Related reference
"addReportParameter()" on page 836

"EGL library ReportLib" on page 834
"fillReport()" on page 837
"exportReport()" on page 836

EGL library StrLib

The next table shows the system functions in the library **StrLib** and is followed by tables that show the variables and constants in that library.

System function and invocation	Description
result = characterAsInt (*text*)	Converts a character string into an integer string corresponding to the first character in the character expression.
result = clip (*text*)	Deletes trailing blank spaces and nulls from the end of returned character strings.
result = compareStr (*target*, *targetSubstringIndex*, *targetSubstringLength*, *source*, *sourceSubstringIndex*, *sourceSubstringLength*)	Compares two substrings in accordance with their ASCII or EBCDIC order at run time and returns a value (-1, 0, or 1) to indicate which is greater.
result = concatenate (*target* , *source*)	Concatenates *target* and *source*; places the new string in *target*; and returns an integer that indicates whether *target* was long enough to contain the new string
result = concatenateWithSeparator (*target*, *source*, *separator*)	Concatenates *target* and *source*, inserting *separator* between them; places the new string in *target*; and returns an integer that indicates whether *target* was long enough to contain the new string
copyStr (*target*, *targetSubstringIndex*, *targetSubstringLength*, *source*, *sourceSubstringIndex*, *sourceSubstringLength*)	Copies one substring to another
result = findStr (*source*, *sourceSubstringIndex*, *sourceSubstringLength*, *searchString*)	Searches for the first occurrence of a substring within a string
result = formatDate (*dateValue* [, *dateFormat*])	Formats a date value and returns a value of type STRING. The default format is the format specified in the current locale.
result = formatNumber (*numericExpression*, *numericFormat*)	Returns a number as a formatted string.
result = formatTime (*timeValue* [, *timeFormat*])	Formats a parameter into a time value and returns a value of type STRING. The default format is the format specified in the current locale.
result = formatTimeStamp (*timeStampValue* [, *timeStampFormat*])	Formats a parameter into a timestamp value and returns a value of type STRING. The DB2 format is the default format.
result = getNextToken (*target*, *source*, *sourceSubstringIndex*, *sourceStringLength*, *characterDelimiter*)	Searches a string for the next token and copies the token to *target*
result = integerAsChar (*integer*)	Converts an integer string into a character string.
result = lowerCase (*text*)	Converts all uppercase values in a character string to lowercase values. Numeric and existing lowercase values are not affected.

System function and invocation	Description
setBlankTerminator (*target*)	Replaces a null terminator and any subsequent characters in a string with spaces, so that a string value returned from a C or C++ program can operate correctly in an EGL-generated program
setNullTerminator (*target*)	Changes all trailing spaces in a string to nulls
setSubStr (*target*, *targetSubstringIndex*, *targetSubstringLength*, *source*)	Replaces each character in a substring with a specified character
result =spaces (*characterCount*)	Returns a string of a specified length.
result = strLen (*source*)	Returns the number of bytes in an item, excluding any trailing spaces or nulls
result = textLen (*source*)	Returns the number of bytes in a text expression, excluding any trailing spaces or nulls
result = upperCase (*characterItem*)	Converts all lowercase values in a character string to uppercase values. Numeric and existing uppercase values are not affected.

The next table shows the system variables in the library **StrLib**.

System variable	Description
defaultDateFormat	Specifies the value of **defaultDateFormat**, which is one of several masks that can be used to create the string returned by the function **StrLib.formatDate**.
defaultMoneyFormat	Specifies the value of **defaultMoneyFormat**, which is one of several masks that can be used to create the string returned by the function **StrLib.formatNumber**.
defaultNumericFormat	Specifies the value of **defaultNumericFormat**, which is one of several masks that can be used to create the string returned by the function **StrLib.formatNumber**.
defaultTimeFormat	Specifies the value of **defaultTimeFormat**, which is one of several masks that can be used to create the string returned by the function **StrLib.formatTime**.
defaultTimestampFormat	Specifies the value of **defaultTimestampFormat**, which is one of several masks that can be used to create the string returned by the function **StrLib.formatTimestamp**.

The next table shows the system constants in the library **StrLib**. All are of type STRING.

System variable	Description
db2TimestampFormat	The pattern *yyyy-MM-dd-HH.mm.ss.ffffff*, which is the IBM DB2 default timestamp format.
eurDateFormat	The pattern *dd.MM.yyyy*, which is the IBM European standard date format.
eurTimeFormat	The pattern *HH.mm.ss*, which is the IBM European standard time format.
isoDateFormat	The pattern *yyyy-MM-dd*, which is the date format specified by the International Standards Organization (ISO).
isoTimeFormat	The pattern *HH.mm.ss*, which is the time format specified by the International Standards Organization (ISO).
jisDateFormat	The pattern *yyyy-MM-dd*, which is the Japanese Industrial Standard date format.
jisTimeFormat	The pattern *HH:mm:ss*, which is the Japanese Industrial Standard time format.
odbcTimestampFormat	The pattern *yyyy-MM-dd HH:mm:ss.ffffff*, which is the ODBC timestamp format.
usaDateFormat	The pattern *MM/dd/yyyy*, which is the IBM USA standard date format.
usaTimeFormat	The pattern *hh:mm AM*, which is the IBM USA standard time format.

Related reference
"formatDate()" on page 851
"formatNumber()" on page 851
"formatTime()" on page 852
"formatTimeStamp()" on page 853

characterAsInt()

The string-formatting function **StrLib.characterAsInt** converts a character string into an integer string corresponding to the first character in the character expression.

```
StrLib.characterAsInt(text STRING in)
returns (result INT)
```

result
A variable of type INT.

text
A literal, variable, or expression that returns a character string of type CHAR.

To convert an integer string into a character string, use the **StrLib.integerAsChar** string-formatting function.

Related reference
"EGL library StrLib" on page 841
"integerAsChar()" on page 856

clip()

The string-formatting function **StrLib.clip** deletes trailing blank spaces and nulls from the end of returned character strings.

```
StrLib.clip(text STRING in)
returns (result STRING)
```

result
A character string.

text
A literal, variable, or expression that returns a character string of type CHAR.

Related reference
"EGL library StrLib" on page 841

compareStr()

The system function **StrLib.compareStr** compares two substrings in accordance with their ASCII or EBCDIC order at run time.

```
StrLib.compareStr(
  target VagText in,
  targetSubstringIndex INT in,
  targetSubstringLength INT in,
  source VagText in,
  sourceSubstringIndex INT in,
  sourceSubstringLength INT in  )
returns (result INT)
```

result
Numeric item that receives one of the following values (defined as type INT or the equivalent: type BIN with length 9 and no decimal places) returned by the function:

-1 The substring based on *target* is less than the substring based on *source*

0 The substring based on *target* is equal to the substring based on *source*

1 The substring based on *target* is greater than the substring based on *source*

target
String from which a target substring is derived. Can be an item or a literal.

targetSubStringIndex
Identifies the starting byte of the substring in *target*, given that the first byte in *target* has the index value 1. This index can be an integer literal. Alternatively, this index can be an item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

targetSubStringLength
Identifies the number of bytes in the substring that is derived from *target*. The length can be an integer literal. Alternatively, this index can be an item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

source
String from which a source substring is derived. Can be an item or a literal.

sourceSubStringIndex
Identifies the starting byte of the substring in *source*, given that the first byte in *source* has the index value of 1. This index can be an integer literal. Alternatively, this index can be an item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

sourceSubStringLength
Identifies the number of bytes in the substring that is derived from *source*. The length can be an integer literal. Alternatively, this index can be an item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

A byte-to-byte binary comparison of the substring values is performed. If the substrings are not the same length, the shorter substring is padded with spaces before the comparison.

Definition considerations: The following values are returned in **sysVar.errorCode**:

8 Index less than 1 or greater than string length.

12 Length less than 1.

20 Invalid double-byte index. Index for a DBCHAR or UNICODE string points to middle of double-byte character

24 Invalid double-byte length. Length in bytes for a DBCHAR or UNICODE string is odd (double-byte lengths must always be even).

Example:

```
target = "123456";
source = "34";
result =
  StrLib.compareStr(target,3,2,source,1,2);
// result = 0
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library StrLib" on page 841

concatenate()

The system function **StrLib.concatenate** concatenates two strings.

```
StrLib.concatenate(
  target VagText inOut,
  source VagText in)
returns (result INT)
```

result
Numeric item that receives one of the following values (defined as type INT or the equivalent: type BIN with length 9 and no decimal places) returned by the function:

-1 Concatenated string is too long to fit in the target item and the string was truncated, as described later

0 Concatenated string fits in the target item

target
Target item

source
Source item or literal

When two strings are concatenated, the following occurs:

1. Any trailing spaces or nulls are deleted from the target string.

2. The source string is appended to the string produced by step 1 on page 845.
3. If the string produced by step 2 is longer than the target string item, it is truncated. If it is shorter than the target item, it is padded with blanks.

Example:

```
phrase = "and/  "; // CHAR(7)
or     = "or";
result =
  StrLib.concatenate(phrase,or);
if (result == 0)
  print phrase;  // phrase = "and/or "
end
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library StrLib" on page 841

concatenateWithSeparator()

The system function **StrLib.concatenateWithSeparator** concatenates two strings, inserting a separator string between them. If the initial length of the target string is zero (not counting trailing blanks and nulls), the separator is omitted and the source string is copied to the target string.

```
StrLib.concatenateWithSeparator(
  target VagText inOut,
  source VagText in,
  separator VagText in)
returns (result INT)
```

result
Numeric item that receives one of the following values (defined as type INT or the equivalent: type BIN with length 9 and no decimal places) returned by the function: :

0 Concatenated string fits in target item.

-1 Concatenated string is too long to fit in the target item and the string was truncated, as described later

target
Target item.

source
Source item or literal.

separator
Separator item or literal.

Trailing spaces and nulls are truncated from *target*; then, the *separator* string and *source* are appended to the truncated value. If the concatenation is longer than the target allows, truncation occurs. If the concatenation is shorter than the target allows, the concatenated value is padded with spaces.

Example:

```
phrase = "and";   // CHAR(7)
or     = "or";
result =
```

```
    StrLib.concatenateWithSeparator(phrase,or,"/");
  if (result == 0)
    print phrase;  // phrase = "and/or "
  end
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library StrLib" on page 841

copyStr()

The system function **StrLib.copyStr** copies one substring to another.

```
StrLib.copyStr(
  target VagText inOut,
  targetSubstringIndex INT in,
  targetSubstringLength INT in,
  source VagText in,
  sourceSubstringIndex INT in,
  sourcetSubstringLength INT in)
```

target
String from which a target substring is derived. Can be an item or a literal.

targetSubstringIndex
Identifies the starting byte in *target*, given that the first byte in *target* has the value 1. This index can be an integer literal. Alternatively, this index can be an item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

targetSubstringLength
Identifies the number of bytes in the substring that is derived from *target*. The length can be an integer literal. Alternatively, the length can be an item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

source
String from which a source substring is derived. Can be an item or a literal.

sourceSubstringIndex
Identifies the starting byte of the substring in *source*, given that the first byte in *source* has the value 1. This index can be an integer literal. Alternatively, this index can be an item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

sourceSubstringLength
Identifies the number of bytes in the substring that is derived from *source*. The length can be an integer literal. Alternatively, the length can be an item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

If the source is longer than the target, the source is truncated. If the source is shorter than the target, the source value is padded on the right with spaces.

Definition considerations: The following values are returned in **sysVar.errorCode**:

8 Index less than 1 or greater than string length.

12 Length less than 1.

20 Invalid double-byte index. Index for a DBCHAR or UNICODE string points to middle of double-byte character.

24 Invalid double-byte length. Length in bytes for a DBCS or UNICODE string is odd (double-byte lengths must always be even).

Example:

```
target = "120056";
source = "34";
StrLib.copyStr(target,3,2,source,1,2);
// target = "123456"
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library StrLib" on page 841

defaultDateFormat

The system variable **StrLib.defaultDateFormat** specifies the value of **defaultDateFormat**, which is one of several masks that can be used to create the string returned by the function **StrLib.formatDate**.

The initial value of **StrLib.defaultDateFormat** is the value of the Java run-time property **vgj.default.dateFormat**. If that property is not set, the initial value of **StrLib.defaultDateFormat** is *MM/dd/yyyy*.

For details on the characteristics of a time mask, see *Date, time, and timestamp specifiers*.

Type: STRING

Related reference
"Date, time, and timestamp format specifiers" on page 42
"EGL library StrLib" on page 841
"formatDate()" on page 851
"Java runtime properties (details)" on page 525

defaultMoneyFormat

The system variable **StrLib.defaultMoneyFormat** specifies the value of **defaultMoneyFormat**, which is one of several masks that can be used to create the string returned by the function **StrLib.formatNumber**.

The initial value of **StrLib.defaultMoneyFormat** is the value of the Java run-time property **vgj.default.moneyFormat**. If that property is not set, the initial value of **StrLib.defaultMoneyFormat** is an empty string.

For details on the characteristics of a numeric mask, see *formatNumber()*.

Type: STRING

Related reference
"EGL library StrLib" on page 841
"formatNumber()" on page 851
"Java runtime properties (details)" on page 525

defaultNumericFormat

The system variable **StrLib.defaultNumericFormat** specifies the value of **defaultNumericFormat**, which is one of several masks that can be used to create the string returned by the function **StrLib.formatNumber**.

The initial value of **StrLib.defaultNumericFormat** is the value of the Java run-time property **vgj.default.numericFormat**. If that property is not set, the initial value of **StrLib.defaultNumericFormat** is an empty string.

For details on the characteristics of a numeric mask, see *formatNumber()*.

Type: STRING

Related reference
"EGL library StrLib" on page 841
"formatNumber()" on page 851
"Java runtime properties (details)" on page 525

defaultTimeFormat

The system variable **StrLib.defaultTimeFormat** specifies the value of **defaultTimeFormat**, which is one of several masks that can be used to create the string returned by the function **StrLib.formatTime**. The variable is not used in any other context.

The initial value of **StrLib.defaultTimeFormat** is the value of the Java run-time property **vgj.default.timeFormat**. If that property is not set, the initial value of **StrLib.defaultTimeFormat** is *HH:mm:ss*.

For details on the characteristics of a time mask, see *Date, time, and timestamp specifiers*.

Type: STRING

Related reference
"Date, time, and timestamp format specifiers" on page 42
"EGL library StrLib" on page 841
"formatTime()" on page 852
"Java runtime properties (details)" on page 525

defaultTimestampFormat

The system variable **StrLib.defaultTimestampFormat** specifies the value of **defaultTimestampFormat**, which is one of several masks that can be used to create the string returned by the function **StrLib.formatTimestamp**.

The initial value of **StrLib.defaultTimestampFormat** is the value of the Java run-time property **vgj.default.timestampFormat**. If that property is not set, the initial value of **StrLib.defaultTimestampFormat** is an empty string.

For details on the characteristics of a timestamp mask, see *Date, time, and timestamp specifiers*.

Type: STRING

Related reference
“Date, time, and timestamp format specifiers” on page 42
“EGL library StrLib” on page 841
“formatTimeStamp()” on page 853
“Java runtime properties (details)” on page 525

findStr()

The system function **StrLib.findStr** searches for the first occurrence of a substring in a string.

```
StrLib.findStr(
  source VagText in,
  sourceSubstringIndex INT inOut,
  sourceSubstringLength INT in,
  searchString VagText in)
returns (result INT)
```

result
Numeric item that receives one of the following values (defined as type INT or the equivalent: type BIN with length 9 and no decimal places) returned by the function:

-1 Search string was not found

0 Search string was found

source
String from which a source substring is derived. Can be an item or a literal.

sourceSubstringIndex
Identifies the starting byte for the substring in *source*, given that the first byte in *source* has the index value of 1. This index can be an item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

sourceStringLength
Identifies the number of bytes in the substring that is derived from *source*. This index can be an item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

searchString
String item or literal to be searched for in the source substring. Trailing blanks or nulls are truncated from the search string before searching begins.

If *searchString* is found in the source substring, *sourceSubstringIndex* is set to indicate its location (the byte of the source where the matching substring begins). Otherwise, *sourceSubstringIndex* is not changed.

Definition considerations: The following values are returned in sysVar.errorCode:

8 Index less than 1 or greater than string length.

12 Length less than 1.

20 Invalid double-byte index. Index for a DBCHAR or UNICODE string points to middle of double-byte character.

24 Invalid double-byte length. Length in bytes for a DBCHAR or UNICODE string is odd (double-byte lengths must always be even).

Example:

```
source = "123456";
sourceIndex = 1
sourceLength = 6
```

```
search = "34";
result =
  StrLib.findStr(source,sourceIndex,sourceLength,"34");
// result = 0, sourceIndex = 3
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library StrLib" on page 841

formatDate()

The system function **StrLib.formatDate** formats a date value and returns a value of type STRING.

```
StrLib.formatDate(
  dateValue DATE in
  [, dateFormat STRING in])
returns (result STRING)
```

result
A variable of type STRING.

dateValue
The value to be formatted. Can be any expression that resolves to a date value; for example, the system variable **VGVar.currentGregorianDate**.

dateFormat
Identifies the date format, as described in *Date, time, and timestamp specifiers*. If you specify no value for *dateFormat*, EGL run time uses the date format in the Java locale.

You can use a string, the system variable **StrLib.defaultDateFormat** (as described in *defaultDateFormat*), or any of these constants:

StrLib.eurDateFormat
The pattern *dd.MM.yyyy*, which is the IBM European standard date format

StrLib.isoDateFormat
The pattern *yyyy-MM-dd*, which is the date format specified by the International Standards Organization (ISO)

StrLib.jisDateFormat
The pattern *yyyy-MM-dd*, which is the Japanese Industrial Standard date format

StrLib.usaDateFormat
The pattern *MM/dd/yyyy*, which is the IBM USA standard date format

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"DATE" on page 38
"Date, time, and timestamp format specifiers" on page 42
"defaultDateFormat" on page 848
"EGL library StrLib" on page 841

formatNumber()

The string function **StrLib.formatNumber** returns a number as a formatted string.

```
StrLib.formatNumber(
  numericExpression anyNumericExpression in,
  numericFormat STRING in)
returns (result STRING)
```

result
A variable of type STRING.

numericExpression
The numeric value to be formatted. Can be any expression that resolves to a number.

numericFormat
A string that defines how the number is to be formatted. Details are in the next table. The string is required, but you can use the system variable **StrLib.defaultMoneyFormat** or **StrLib.defaultNumericFormat**. For details on those variables, see *defaultMoneyFormat* and *defaultNumericFormat*.

Valid characters are as follows:

A placeholder for a digit.

***** Use an asterisk (*) as the fill character for a leading zero.

& Use a zero as the fill character for a leading zero.

Use a space as the fill character for a leading zero.

< Left justify the number.

, Use a locale-dependent numeric separator unless the position contains a leading zero.

. Use a locale-dependent decimal point.

- Use a minus sign (-) for values less than 0; use a space for values greater than or equal to 0.

+ Use a minus sign for values less than 0; use a plus sign (+) for values greater than or equal to 0.

(Precede negative values with a left parenthesis, as appropriate in accounting.

) Place a right parenthesis after a negative value, as appropriate in accounting.

$ Precede the value with the locale-dependent currency symbol.

@ Place the locale-dependent currency symbol after the value.

Related reference
"defaultMoneyFormat" on page 848
"defaultNumericFormat" on page 849
"EGL library StrLib" on page 841

formatTime()

The datetime function **StrLib.formatTime** formats a time value and returns a value of type STRING.

```
StrLib.formatTime(
  aTime Time in
  [, timeFormat STRING in
  ])
returns (result STRING)
```

result
A variable of type STRING.

aTime
The value to be formatted. Can be any expression that resolves to a time value; for example, the system variable **DateTimeLib.currentTime**.

timeFormat
Identifies the time format, as described in *Date, time, and timestamp specifiers*. If you specify no value for *timeFormat*, EGL run time uses the time format in the Java locale.

You can use a string, the system variable **StrLib.defaultTimeFormat** (as described in *defaultTimeFormat*), or any of these constants:

eurTimeFormat
The pattern *HH.mm.ss*, which is the IBM European standard time format.

isoTimeFormat
The pattern *HH.mm.ss*, which is the time format specified by the International Standards Organization (ISO).

jisTimeFormat
The pattern *HH:mm:ss*, which is the Japanese Industrial Standard time format.

usaTimeFormat
The pattern *hh:mm AM*, which is the IBM USA standard time format.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"Date, time, and timestamp format specifiers" on page 42
"defaultTimeFormat" on page 849
"EGL library StrLib" on page 841
"TIME" on page 40

formatTimeStamp()

The datetime formatting function **StrLib.formatTimeStamp** formats a parameter into a timestamp value and returns a value of type STRING.

```
StrLib.formatTimeStamp(
  aTimeStamp TimeStamp in
  [, timeStampFormat STRING in
  ])
returns (result STRING)
```

result
A variable of type STRING.

aTimeStamp
The TIMESTAMP value to be formatted. Can be any expression that resolves to a TIMESTAMP value; for example, the system variable **DateTimeLib.currentTimeStamp**.

timeStampFormat
Identifies the date format, as described in *Date, time, and timestamp specifiers*. The default format is **db2TimeStampFormat** (as described later).

You can use a string, the system variable **StrLib.defaultTimestampFormat** (as described in *defaultTimestampFormat*), or one of these constants:

db2TimeStampFormat
The pattern *yyyy-MM-dd-HH.mm.ss.ffffff*, which is the IBM DB2 default timestamp format.

odbcTimeStampFormat
The pattern *yyyy-MM-dd HH:mm:ss.ffffff*, which is the ODBC timestamp format.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"currentTimeStamp()" on page 770
"Date, time, and timestamp format specifiers" on page 42
"defaultTimestampFormat" on page 849
"EGL library StrLib" on page 841
"TIMESTAMP" on page 41

getNextToken()

The system function **StrLib.getNextToken** searches a substring for a token and copies that token to a target item.

Tokens are strings separated by delimiter characters. For example, if the characters space (" ") and comma (",") are defined as delimiters, the string `"CALL PROGRAM ARG1,ARG2,ARG3"` can be broken down into the five tokens `"CALL"`, `"PROGRAM"`, `"ARG1"`, `"ARG2"`, and `"ARG3"`.

```
StrLib.getNextToken(
  target VagText inOut,
  source VagText in,
  sourceSubstringIndex INT inOut,
  sourceSubstringLength INT inOut,
  characterDelimiter VagText in)
returns (result INT)
```

result
An item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places. The value is one of these:

+n	Number of characters in the token. The token is copied from the substring under review to the target item.
0	No token was in the substring under review.
-1	The token was truncated when copied to the target item.

target
Target item of type CHAR, DBCHAR, HEX, MBCHAR, or UNICODE.

source
Source item of type CHAR, DBCHAR, HEX, MBCHAR, or UNICODE. May be a literal of any of those types other than UNICODE.

sourceSubstringIndex
Identifies the starting byte at which to begin searching for a delimiter, given that the first byte in *source* has the value 1. *sourceSubstringIndex* can be an item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places. If a token is found, the value in *sourceSubstringIndex* is changed to the index of the first character that follows the token.

sourceSubstringLength
Indicates the number of bytes in the substring under review. *sourceSubstringLength* can be an item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places. If a token is found, the value in *sourceSubstringLength* is changed to the number of bytes in the substring that begins after the returned token.

characterDelimiter
One or more delimiter characters, with no characters separating one from the next. May be an item of type CHAR, DBCHAR, HEX, MBCHAR, or UNICODE. May be a literal of any of those types other than UNICODE.

You can invoke a sequence of calls to retrieve each token in a substring without resetting the values for *sourceSubstringIndex* and *sourceSubstringLength*, as shown in a later example.

Error conditions: The following values are returned in **SysVar.errorCode**:

8 *sourceSubstringIndex* is less than 1 or is greater than number of bytes in the substring under review.

12 *sourceSubstringLength* is less than 1.

20 The value in *sourceSubstringIndex* for a DBCHAR or UNICODE string refers to the middle of a double-byte character.

24 The value in *sourceSubstringLength* for a DBCHAR or UNICODE string is odd (double-byte lengths must always be even).

Example:

```
Function myFunction()
  myVar myStructurePart;
  myRecord myRecordPart;

  i = 1;
  myVar.mySourceSubstringIndex = 1;
  myVar.mySourceSubstringLength = 29;

  while (myVar.mySourceSubstringLength > 0)
    myVar.myResult = StrLib.getNextToken( myVar.myTarget[i],
      "CALL PROGRAM arg1, arg2, arg3",
        myVar.mySourceSubstringIndex,
        myVar.mySourceSubstringLength, " ," );

    if (myVar.myResult > 0)
      myRecord.outToken = myVar.myTarget[i];
      add myRecord;
      set myRecord empty;
      i = i + 1;
    end
  end
end

Record myStructurePart
  01 myTarget CHAR(80)[5];
  01 mySource CHAR(80);
  01 myResult myBinPart;
  01 mySourceSubstringIndex INT;
  01 mySourceSubstringLength BIN(9,0);
  01 i myBinPart;
end

Record myRecordPart
  serialRecord:
    fileName="Output"
  end
  01 outToken CHAR(80);
end
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library StrLib" on page 841

integerAsChar()

The string-formatting function **StrLib.integerAsChar** converts an integer string into a character string.

```
StrLib.integerAsChar(integer INT in)
returns (result STRING)
```

result
A variable of type STRING.

integer
A literal, variable or expression that returns an integer of type BIGINT, INT or SMALLINT.

To convert a character string into an integer string, use the **StrLib.characterAsInt** string-formatting function.

Related reference
"EGL library StrLib" on page 841
"characterAsInt()" on page 843

lowerCase()

The string-formatting function **StrLib.lowerCase** converts all uppercase values in a character string to lowercase values. Numeric and existing lowercase values are not affected.

```
StrLib.lowerCase(text STRING in)
returns (result STRING)
```

result
A variable of type STRING.

text
A literal, variable, or expression that returns a character string of type CHAR.

The **StrLib.lowerCase** function has no effect on double-byte characters in items of type DBCHAR or MBCHAR.

To convert lowercase values to uppercase values, use the **StrLib.upperCase** string-formatting function.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library StrLib" on page 841
"upperCase()" on page 859

setBlankTerminator()

The system function **StrLib.setBlankTerminator** changes a null terminator and any subsequent characters to spaces. **StrLib.setBlankTerminator** changes a string value returned from a C or C++ program to a character value that can operate correctly in an EGL program.

```
StrLib.setBlankTerminator(target VagText inOut)
```

target
The target string item. If no null is found in *targetString*, the function has no effect.

Example:

```
StrLib.setBlankTerminator(target);
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library StrLib" on page 841

setNullTerminator()

The system function **StrLib.setNullTerminator** changes all trailing spaces in a string to nulls. You can use **StrLib.setNullTerminator** to convert an item before passing it to a C or C++ program that expects a null-terminated string as an argument.

```
StrLib.setNullTerminator(target VagText inOut)
```

target
String to be converted

The target string is searched for trailing spaces and nulls. Any spaces found are changed to nulls.

Definition considerations: The following value can be returned in **sysVar.errorCode**:

16 Last byte of string is not a space or null

Example:

```
StrLib.setNullTerminator(myItem01);
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library StrLib" on page 841

setSubStr()

The system function **StrLib.setSubStr** replaces each character in a substring with a specified character.

```
StrLib.setSubStr(
  target VagText inOut,
  targetSubstringIndex INT in,
  targetSubstringLength INT in,
  source)
```

target
Item that is changed.

targetSubstringIndex
Identifies the starting byte of the substring in *target*, given that the first byte in *target* has the index value of 1. This index can be an integer literal. Alternatively, this index can be an item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

targetSubstringLength
Identifies the number of bytes in the substring that is derived from *target*. The length can be an integer literal. Alternatively, the length can be an item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

source
If the target item is CHAR, MBCHAR, or HEX, the source item must be a one-byte CHAR, MBCHAR, or HEX item or a CHAR literal. If the target is a DBCHAR or UNICODE item, the source must be a single-character DBCHAR or UNICODE item.

Definition considerations: The following values are returned in SysVar.errorCode:

8 Index less than 1 or greater than string length

12 Length less than 1

20 Invalid double-byte index. Index for a DBCHAR or UNICODE string points to middle of double-byte character

24 Invalid double-byte length. Length in bytes for a DBCHAR or UNICODE string is odd (double-byte lengths must always be even)

Example:

```
StrLib.setSubStr(target,12,5," ");
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library StrLib" on page 841

spaces()

The system function **StrLib.spaces** returns a string composed of a specified number of spaces.

```
StrLib.spaces(characterCount INT in)
returns (result STRING)
```

result
A variable of type STRING.

characterCount
The length of the string of spaces to be returned.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library StrLib" on page 841

strLen()

The system function **StrLib.strLen** returns the number of bytes in an item, excluding any trailing spaces and nulls.

```
StrLib.strLen(source VagText in)
returns (result INT)
```

result
An item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

source
String item or literal to be measured.

Example:

```
length = StrLib.strLen(source);
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library StrLib" on page 841

textLen()

The system function **StrLib.textLen** returns the number of characters in a text expression, excluding any trailing spaces and nulls.

```
StrLib.textLen(source STRING in)
returns (result INT)
```

result
An item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

source
The text expression of interest.

Example:

```
length = StrLib.textLen(source);
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library StrLib" on page 841
"Text expressions" on page 492

upperCase()

The string-formatting function **StrLib.upperCase** converts all lowercase values in a character string to uppercase values. Numeric and existing uppercase values are not affected.

```
StrLib.upperCase(text STRING in)
returns (result STRING)
```

result
A variable of type STRING.

text
A literal, variable, or expression that returns a character string of type CHAR.

The **StrLib.upperCase** function has no effect on double-byte characters in items of type DBCHAR or MBCHAR.

To convert a character string to lowercase, use the **StrLib.lowerCase** string-formatting function.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library StrLib" on page 841
"lowerCase()" on page 856

EGL library SysLib

Function	Description
beginDatabaseTransaction([*database*])	Begins a relational-database transaction, but only when the EGL run time is not committing changes automatically.
result = bytes(*field*)	Returns the number of bytes in a named area of memory.
calculateChkDigitMod10 (*text*, *checkLength*, *result*)	Places a modulus-10 check digit in a character item that begins with a series of integers.
calculateChkDigitMod11 (*text*, *checkLength*, *result*)	Places a modulus-11 check digit in a character item that begins with a series of integers.
callCmd (*commandString*[, *modeString*])	Runs a system command and waits until the command finishes.
commit()	Saves updates that were made to databases, MQSeries message queues, and CICS recoverable files since the last commit. A generated Java program or wrapper also saves the updates done by a remote, CICS-based COBOL program (including updates to CICS recoverable files), but only when the call to the remote COBOL program involves a client-controlled unit of work, as described in *luwControl in callLink element*.
result = conditionAsInt (*booleanExpression*)	Accepts a logical expression (like *myVar == 6*) , returning a 1 if the expression is true, a 0 if the expression is false.
connect (*database*, *userID*, *password*[, *commitScope*[, *disconnectOption*[, *isolationLevel*[, *commitControl*]]]])	Closes all cursors, releases locks, ends any existing connection, and connects to the database.
convert (*target*, *direction*, *conversionTable*)	Converts data between EBCDIC (host) and ASCII (workstation) formats or performs code-page conversion within a single format.
defineDatabaseAlias (*alias*, *database*)	Creates an alias that can be used to establish a new connection to a database to which your code is already connected.
disconnect ([*database*])	Disconnects from the specified database or (if no database is specified) from the current database.
disconnectAll ()	Disconnects from all the currently connected databases.
errorLog ()	Copies text into the error log that was started by the system function **SysLib.startLog**.

Function	Description
result = getCmdLineArg (*index*)	Returns the specified argument from the list of arguments with which the EGL program was involved. The specified argument is returned as a string value.
result = getCmdLineArgCount ()	Returns the number of arguments that were used to start the main EGL program.
result = getMessage (*key* [, *insertArray*])	Returns a message from the file that is referenced in the Java runtime property vgj.message.file.
result = getProperty(propertyName)	Retrieves the value of a Java runtime property. If the specified property is not found, the function returns a null string ("").
loadTable (*filename*, *insertintoClause*[, *delimiter*])	Loads data from a file into a relational database. The function is available only for EGL-generated Java programs.
result = maximumSize (*arrayName*)	Returns the maximum number of rows that can be in a dynamic array of data items or records; specifically, the function returns the value of the array property **maxSize**.
queryCurrentDatabase (*product*, *release*)	Returns the product and release number of the currently connected database.
rollback ()	Reverses updates that were made to databases and MQSeries message queues since the last commit. That reversal occurs in any EGL-generated application.
setCurrentDatabase (*database*)	Makes the specified database the currently active one.
setError (*itemInError*, *msgKey*{, *itemInsert*}) setError (this, *msgKey*{, *itemInsert*}) setError (*msgText*)	Associates a message with an item in a PageHandler or UI record or with the PageHandler or UI record as a whole. The message is placed at the location of a JSF message or messages tag in the JSP and is displayed when the related Web page is displayed.
setLocale (*languageCode*, *countryCode*[, *variant*])	Used in PageHandlers and in programs that run in a Web application.
setRemoteUser (*userID*, *passWord*)	Sets the userid and password that are used on calls to remote programs from Java programs.
result = size (*arrayName*)	Returns the number of rows in the specified data table or the number of elements in the specified array. The array may be a structure-item array, a static array of data items or records, or a dynamic array of data items or records.
startCmd (*commandString*[, *modeString*])	Runs a system command and does not wait until the command finishes.
startLog (*logFile*)	Opens an error log. Text is written into that log every time your program invokes **SysLib.errorLog**.

Function	Description
startTransaction (*termID*[, *prID*[, *termID*]])	Invokes a main program asynchronously, associates that program with a printer or terminal device, and passes a record. If the receiving program is generated by EGL, the record is used to initialize the input record; if the receiver is produced by VisualAge Generator, the record is used to initialize the working storage.
unloadTable (*filename*, *selectStatement*[, *delimiter*])	Unloads data from a relational database into a file. The function is available only for EGL-generated Java programs.
verifyChkDigitMod10 (*input*, *checkLength*, *result*)	Verifies a modulus-10 check digit in a character item that begins with a series of integers.
verifyChkDigitMod11 (*input*, *checkLength*, *result*)	Verifies a modulus-11 check digit in a character item that begins with a series of integers.
wait (*timeInSeconds*0	Suspends execution for the specified number of seconds.

beginDatabaseTransaction()

The system function **SysLib.beginDatabaseTransaction** begins a relational-database transaction, but only when the EGL run time is not committing changes automatically. If changes are being committed automatically, the function has no effect.

The function is valid only for EGL-generated Java output.

```
SysLib.beginDatabaseTransaction(
  [database STRING in])
```

database
A database name that was specified in SysLib.connect or VGLib.connectionService. Use a literal or variable of a character type.

If you do not specify a connection, the function affects the current connection.

When you invoke **SysLib.beginDatabaseTransaction**, a transaction begins at the next I/O operation that uses the specified connection; and the transaction ends when a commit or rollback occurs, as described in *Logical unit of work*. After the commit or rollback, the EGL run time resumes committing changes automatically.

For details on automatic commits, see *SysLib.connect* and *sqlCommitControl*.

Related concepts
"Syntax diagram for EGL functions" on page 732
"Logical unit of work" on page 288
"SQL support" on page 213

Related reference
"sqlCommitControl" on page 384
"connect()" on page 867
"connectionService()" on page 888

bytes()

The system function **SysLib.bytes** returns the number of bytes in a named area of memory.

```
SysLib.bytes(field fixedFieldOrArray in)
returns (result INT)
```

result
A numeric item that receives the number of bytes in *field*. Two cases are notable:

- If *field* is an array, the function returns the number of bytes in one element
- If *field* is an SQL record, the function returns the number of bytes in the record, including the extra bytes; for details see *SQL record internals*

field
An array, item, or record

Example():

```
result = SysLib.bytes(myItem);
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library SysLib" on page 860
"Primitive types" on page 31
"SQL record internals" on page 726

calculateChkDigitMod10()

The system function **SysLib.calculateChkDigitMod10** places a modulus-10 check digit in a character item that begins with a series of integers.

```
SysLib.calculateChkDigitMod10(
  text anyChar inOut,
  checkLength SMALLINT in,
  result SMALLINT inOut)
```

text
A character item that begins with a series of integers. The item must include an additional position for the check digit, which goes immediately to the right of the other integers.

checkLength
An item that contains the number of characters that you want to use from the *text* item, including the position used for the check digit. This item has 4 digits and is either of type SMALLINT or is of a type BIN, with no decimal places.

result
An item that receives one of two values:

- 0, if the check digit was created
- 1, if the check digit was not created

This item has 4 digits and is either of type SMALLINT or is of a type BIN, with no decimal places.

You can use **SysLib.calculateChkDigitMod10** in a function-invocation statement.

Example: In the following example, myInput is an item of type CHAR and contains the value 1734289; myLength is an item of type SMALLINT and contains the value 7; and myResult is an item of type SMALLINT:

```
SysLib.verifyChkDigitMod10 (myInput, myLength, myResult);
```

An algorithm is used to derive the modulus-10 check digit, and in all cases the number at the check-digit position is not considered. The algorithm is described in relation to the example values:

1. Multiply the units position of the input number by 2 and multiply every alternate position, moving right to left, by 2:

   ```
   8 x 2 = 16
   4 x 2 = 8
   7 x 2 = 14
   ```

2. Add the digits of the products (16814) to the input-number digits (132) that were not multiplied by 2:

   ```
   1 + 6 + 8 + 1 + 4 + 1 + 3 + 2 = 26
   ```

3. To get the check digit, subtract the sum from the next-highest number ending in 0:

   ```
   30 - 26 = 4
   ```

   If the subtraction yields 10, the check digit is 0.

In this example, the original characters in myInput become these:

```
1734284
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library SysLib" on page 860

calculateChkDigitMod11()

The system function **SysLib.calculateChkDigitMod11** places a modulus-11 check digit in a character item that begins with a series of integers.

```
SysLib.calculateChkDigitMod11(
  text anyChar inOut,
  checkLength SMALLINT in,
  result SMALLINT inOut)
```

text
A character item that begins with a series of integers. The item must include an additional position for the check digit, which goes immediately to the right of the other integers.

checkLength
An item that contains the number of characters that you want to use from the *text* item, including the position used for the check digit. This item has 4 digits and is either of type SMALLINT or is of a type BIN, with no decimal places.

result
An item that receives one of two values:

- 0, if the check digit was created
- 1, if the check digit was not created

This item has 4 digits and is either of type SMALLINT or is of a type BIN, with no decimal places.

You can use **SysLib.calculateChkDigitMod11** in a function-invocation statement.

Example: In the following example, myInput is an item of type CHAR and contains the value 56621869; myLength is an item of type SMALLINT and contains the value 8; and myResult is an item of type SMALLINT:

```
SysLib.verifyChkDigitMod (myInput, myLength, myResult);
```

An algorithm is used to derive the modulus-11 check digit, and in all cases the number at the check-digit position is not considered. The algorithm is described in relation to the example values:

1. Multiply the digit at the units position of the input number by 2, at the tens position by 3, at the hundreds position by 4, and so on, but let myLength ″ 1 be the largest number used as a multiplier; and if more digits are in the input number, begin the sequence again using 2 as a multiplier:

   ```
   6 x 2 = 12
   8 x 3 = 24
   1 x 4 = 4
   2 x 5 = 10
   6 x 6 = 36
   6 x 7 = 42
   5 x 2 = 10
   ```

2. Add the products of the first step and divide the sum by 11:

   ```
   (12 + 24 + 4 + 10 + 36 + 42 + 10) / 11
   = 138 / 11
   = 12 remainder 6
   ```

3. To get the check digit, subtract the remainder from 11 to get the self-checking digit:

   ```
   11 - 6 = 5
   ```

   If the remainder is 0 or 1, the check digit is 0.

In this example, the original characters in myInput become these:

```
56621865
```

Related concepts
“Syntax diagram for EGL functions” on page 732

Related reference
“EGL library SysLib” on page 860

callCmd()

The system function **SysLib.callCmd** runs a system command and waits until the command finishes.

```
SysLib.callCmd(
  commandString STRING in
  [, modeString STRING in
  ])
```

commandString
Identifies the operating-system command to invoke.

modeString
The *modeString* can be any character or string item. The item can be in either of two modes:

- *form*: in which each character of input becomes available to the program as it is typed, i.e., every key stroke is passed directly to the command specified.
- *line*: in which input is not available until after the newline character key is used, i.e., no information is sent to the command specified until the ENTER key is pressed, and then the entire line typed is sent to the command.

The system command that is being executed must be visible to the running program. For example, if you execute **callCmd**(″mySpecialProgram.exe″), the program ″mySpecialProgram.exe″ must be in a directory pointed to by the environment variable PATH. You may also specify the complete directory location, for example **callCmd**(″program files/myWork/mySpecialProgram.exe″).

The **SysLib.callCmd** function is supported only in Java environments.

Use the **SysLib.startCmd** function to run a system command that does not wait until the command finishes.

Related concepts
“Syntax diagram for EGL functions” on page 732

Related reference
“EGL library SysLib” on page 860
“startCmd()” on page 882

commit()

The system function **SysLib.commit** saves updates that were made to databases and MQSeries message queues since the last commit. A generated Java program or wrapper also saves the updates done by a remote, CICS-based COBOL program (including updates to CICS recoverable files), but only when the call to the remote COBOL program involves a client-controlled unit of work, as described in *luwControl in callLink element*.

```
SysLib.commit( )
```

In most cases, EGL performs a single-phase commit that affects each recoverable manager in turn. On CICS for z/OS, however, **SysLib.commit** results in a CICS SYNCPOINT, which performs a two-phase commit that is coordinated across all resource managers.

SysLib.commit releases the scan position and the update locks in any file or databases, but an exception is in effect for COBOL programs, when the option cursorWithHold is used during database access; for details on the exception, see *prepare* and *open*.

When you use **SysLib.commit** with MQ records, the following statements apply:

- Message queue updates are recoverable only if the *Include message in transaction* option is selected in MQ record part.
- Both message **get**s and **add**s are affected by **commit** and **rollback** for recoverable messages. If a **rollback** is issued following a **get** for a recoverable message, the message is placed back on the input queue so that the input message is not lost when the transaction fails to complete successfully. Also, if a **rollback** is issued following an **add** for a recoverable message, the message is deleted from the queue.

You can enhance performance by avoiding unnecessary use of **SysLib.commit**. For details on when an implicit commit occurs, see *Logical unit of work*.

Special considersations for iSeries COBOL: If the program issued SQL statements, **SysLib.commit** results in an SQL COMMIT WORK. If the program has not issued SQL requests, **SysLib.commit** results in the equivalent of an iSeries COMMIT command.

Example:

```
sysLib.commit();
```

Related concepts
"Syntax diagram for EGL functions" on page 732
"Logical unit of work" on page 288
"MQSeries support" on page 247
"Run unit" on page 721
"SQL support" on page 213

Related reference
"commitOnConverse" on page 894
"segmentedMode" on page 898
"EGL library SysLib" on page 860
"luwControl in callLink element" on page 403
"open" on page 598
"prepare" on page 611

conditionAsInt()

The system function **SysLib.conditionAsInt** accepts a logical expression (like *myVar == 6*) , returning a 1 if the expression is true, a 0 if the expression is false.

```
SysLib.conditionAsInt(logicalExpression AnyLogicalExpression in)
returns (result SMALLINT)
```

result
A value of type SMALLINT.

logicalExpression
A logical expression, as described in *Logical expressions*.

Example:

```
myField = -5;

// result = 0
result = SysLib.conditionAsInt(myField == 6);
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library SysLib" on page 860
"Logical expressions" on page 484

connect()

The system function **SysLib.connect** allows a program to connect to a database at run time. This function does not return a value.

```
SysLib.connect(
  database STRING in,
  userID STRING in,
  password STRING in
  [, commitScope enumerationCommitScope in
  [, disconnectOption enumerationDisconnectOption in
  [, isolationLevel enumerationIsolationLevel in
  [, commitControl enumerationCommitControlOption in
  ]]]] )
```

database
Identifies a database.

If your code is running as a COBOL program, the following statements apply:

- Setting *database* to RESET returns to the connection status that is in effect at the beginning of a run unit, as described in *Default database*.
- Otherwise, the value of *database* must be a value in the location column in the SYSIBM.LOCATIONS table, which is in the DB2 UDB subsystem.
- In either case, the system function **SysLib.connect** closes all cursors, releases locks, ends any existing connection, and connects to the database. Despite these actions, invoke **SysLib.commit** or **SysLib.rollback** before invoking **SysLib.connect**.

If your code is running as a Java program, the following statements apply:

- Setting *database* to RESET reconnects to the default database, but if the default database is not available, the connection status remains unchanged; for further details, see *Default database*.
- Otherwise, the physical database name is found by looking up the property **vgj.jdbc.database.***server*, where *server* is the name of the database specified on the **SysLib.connect** call. If this property is not defined, the database name that is specified on the **SysLib.connect** call is used as is.
- The format of the database name is different for J2EE connections as compared with non-J2EE connections:
  - If you generated the program for a J2EE environment, use the name to which the datasource is bound in the JNDI registry; for example, jdbc/MyDB. This situation occurs if build descriptor option **J2EE** was set to YES.
  - If you generated the program for a non-J2EE JDBC environment, use a connection URL; for example, jdbc:db2:MyDB. This situation occurs if option **J2EE** was set to NO.

userID

UserID used to access the database. The argument must be an item of type CHAR and length 8, and a literal is valid. The argument is required, but is ignored for COBOL generation. For background information, see *Database authorization and table names*.

password

Password used to access the database. The argument must be an item of type CHAR and length 8, and a literal is valid. The argument is required, but is ignored for COBOL generation.

commitScope

This parameter is meaningful only if you are generating Java output. The value is one of the following words, and you cannot use quotes and cannot use a variable:

type1 (the default)

Only a *one*-phase commit is supported. A new connection closes all cursors, releases locks, and ends any existing connection; nevertheless, invoke **SysLib.commit** or **SysLib.rollback** before making a type1 connection.

If you use type1 as the value of *commitScope*, the value of parameter *disconnectOption* must be the word *explicit*, as is the default.

type2

A connection to a database does not close cursors, release locks, or end an existing connection. Although you can use multiple connections to read from multiple databases, you should update only one database in a unit of work because only a one-phase commit is available.

twophase
Identical to type2.

disconnectOption
This parameter is meaningful only if you are generating Java output. The value is one of the following words, and you cannot use quotes and cannot use a variable:

explicit (the default)
The connection remains active after the program invokes **SysLib.commit** or **SysLib.rollback**. To release connection resources, a program must issue SysLib.disconnect.

If you use type1 as the value of *commitScope*, the value of parameter *disconnectOption* must be set (or allowed to default) to the word *explicit*.

automatic
A commit or rollback ends an existing connection.

conditional
A commit or rollback automatically ends an existing connection unless a cursor is open and the hold option is in effect for that cursor. For details on the hold option, see *open*.

isolationLevel
This parameter indicates the level of independence of one database transaction from another, as is meaningful only if you are generating Java output.

The following words are in order of increasing strictness, and as before, you cannot use quotes and cannot use a variable:

- **readUncommitted**
- **readCommitted**
- **repeatableRead**
- **serializableTransaction** (the default value)

For details, see the JDBC documentation from Sun Microsystems, Inc.

commitControl
This parameter specifies whether a commit occurs after every change to the database. This parameter is ignored in EGL-generated COBOL code.

Valid values are as follows:

- **noAutoCommit** (the default) means that the commit is not automatic, which usually results in faster execution. For details on the rules of commit and rollback in this case, see *Logical unit of work*.
- **autoCommit** means that updates take effect immediately.

You can switch from autoCommit to noAutoCommit temporarily. For details, see *SysLib.beginDatabaseTransaction*.

Definition considerations: **SysLib.connect** sets the following system variables:

- VGVar.sqlerrd[3]
- SysVar.sqlca
- SysVar.sqlcode
- VGVar.sqlerrmc (available in COBOL code only)
- VGVar.sqlWarn[2]
- VGVar.sqlWarn[7] (available in COBOL code only)

Example:

```
SysLib.connect(myDatabase, myUserid, myPassword);
```

Related concepts
"Logical unit of work" on page 288
"Run unit" on page 721
"SQL support" on page 213

Related tasks
"Syntax diagram for EGL statements and commands" on page 733
"Setting up a J2EE JDBC connection" on page 341
"Understanding how a standard JDBC connection is made" on page 245

Related reference
"Database authorization and table names" on page 453
"Default database" on page 234
"EGL library SysLib" on page 860

"Java runtime properties (details)" on page 525
"open" on page 598
"sqlDB" on page 384
"beginDatabaseTransaction()" on page 862
"disconnect()" on page 873
"sqlca" on page 909
"sqlcode" on page 910
"sqlerrd" on page 923
"sqlerrmc" on page 924
"sqlWarn" on page 925

convert()

The system function **SysLib.convert** converts data between EBCDIC (host) and ASCII (workstation) formats or performs code-page conversion within a single format. You can use **SysLib.convert** as the function name in a function invocation statement.

```
SysLib.convert(
  target anyFixedItemOrRecordOrFormVariable inout,
  direction enumerationConversionDirection in,
  conversionTable CHAR(8) in)
```

target
Name of the record, data item, or form that has the format you want to convert. The data is converted in place based on the item definition of the lowest-level items (items with no substructure) in the target object.

Variable-length records are converted only for the length of the current record. The length of the current record is calculated using the record's numElementsItem or is set from the record's lengthItem. A conversion error occurs and the program ends if the variable-length record ends in the middle of a numeric field or a DBCHAR character.

direction
Direction of conversion. "R" and "L" (including the quotation marks) are the only valid values. Required if *conversionTable* is specified; optional otherwise.

- "**R**" Default value. The data is assumed to be in remote format and is converted to local format.
- "**L**" Data is assumed to be in local format and is converted to remote format (as defined in the conversion table).

conversionTable
Data item or literal (eight characters, optional) that specifies the name of the conversion table to be used for data conversion. The default value is the conversion table associated with the national language code specified when the program was generated.

Definition considerations: You can use the linkage options part to request that automatic data conversion be generated for remote calls, to start remote asynchronous transactions, or for remote file access. Automatic conversion is always performed using the data structure defined for the argument being converted. If an argument has multiple formats, do not request automatic conversion. Instead, code the program to explicitly call **SysLib.convert** with redefined record declarations that correctly map the current values of the argument.

Example:

```
Record RecordA
   record_type char(3);
   item1 char(20);
end

Record RecordB
   record_type char(3);
   item2 bigint;
   item3 decimal(7);
   item4 char(8);
end

Program ProgramX type basicProgram
   myRecordA RecordA;
   myRecordB RecordB {redefines = "myRecordA"};
   myConvTable char(8);

   function main();
      myConvTable = "ELACNENU"; // conversion table for US English
      if (myRecordA.record_type == "00A")
         SysLib.convert(myRecordA, "L", myConvTable);
      else;
         SysLib.convert(myRecordB, "L", myConvTable);
      end
     call ProgramY myRecordA;
   end
end
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"Data conversion" on page 454
"EGL library SysLib" on page 860
"callConversionTable" on page 902

defineDatabaseAlias()

The system function **SysLib.defineDatabaseAlias** creates an alias that can be used to establish a new connection to a database to which your code is already connected. Once established, the alias can be used in any of these functions:

- SysLib.connect
- SysLib.disconnect
- SysLib.beginDatabaseTransaction

- SysLib.setCurrentDatabase
- VGLib.connectionService

The alias can also be used in the **connectionName** field of a variable of type **ReportData**.

The function is valid only for EGL-generated Java output.

```
SysLib.defineDatabaseAlias(
  alias STRING in,
  database STRING in)
```

alias
A string literal or variable that acts as an alias of the connection identified in the second parameter. The alias is case-insensitive.

database
A database name that was specified in SysLib.connect or VGLib.connectionService. Use a literal or variable of a character type.

If you do not specify a connection, the function affects the current connection.

Examples are as follows:

```
// Connect to a database with alias "alias",
// which becomes the current connection.
defineDatabaseAlias( "alias", "database" );
connect( "alias", "user", "pwd" );

// Make two connections to the same database.
String db = "database";
defineDatabaseAlias( "alias1", db );
defineDatabaseAlias( "alias2", db );
connect( "alias1", "user", "pwd" );
connect( "alias2", "user", "pwd" );

// Another way to make two connections
// to the same database.
defineDatabaseAlias( "alias", "database" );
connect( "alias", "user", "pwd" );
connect( "database", "user", "pwd" );

// An alias is defined but not used. The second
// connect() does not create a new connection.
defineDatabaseAlias( "alias", "database" );
connect( "database", "user", "pwd" );
connect( "database", "user", "pwd" );

// Use of an alias (which is case-insensitive)
// when disconnecting.
defineDatabaseAlias( "alias", "database" );
connect( "aLiAs", "user", "pwd" );
disconnect( "ALIAS" );

// The next disconnect call fails because the
// connection is called "alias" not "database".
defineDatabaseAlias( "alias", "database" );
connect( "alias", "user", "pwd" );
disconnect( "database" );

// An alias may change. After the next call,
// "alias" refers to "firstDatabase"
defineDatabaseAlias( "alias", "firstDatabase" );

// After  the next call,
// "alias" refers to "secondDatabase".
```

```
defineDatabaseAlias( "alias", "secondDatabase" );

// The last call would have failed
// if a connection was in place with "alias".
```

Related concepts
“Syntax diagram for EGL functions” on page 732
“SQL support” on page 213

Related reference
“beginDatabaseTransaction()” on page 862“connect()” on page 867
“disconnect()”
“setCurrentDatabase()” on page 879
“connectionService()” on page 888

disconnect()

The system function **SysLib.disconnect** disconnects from the specified database or (if no database is specified) from the current database.

```
SysLib.disconnect(
  [database STRING in
  ])
```

database
A database name that was specified in SysLib.connect or VGLib.connectionService. Use a literal or variable of a character type.

Before disconnecting, invoke SysLib.commit or SysLib.rollback.

Related concepts
“Syntax diagram for EGL functions” on page 732

Related reference
“EGL library SysLib” on page 860
“commit()” on page 866
“connect()” on page 867
“rollback()” on page 878
“connectionService()” on page 888

disconnectAll()

The system function **SysLib.disconnectAll** disconnects from all the currently connected databases.

Before disconnecting, invoke **SysLib.commit** or **SysLib.rollback**.

```
SysLib.disconnectAll( )
```

Related concepts
“Syntax diagram for EGL functions” on page 732

Related reference
“EGL library SysLib” on page 860
“connect()” on page 867
“connectionService()” on page 888

errorLog()

The system function **SysLib.errorLog** copies text into the error log that was started by the system function **SysLib.startLog**.

```
SysLib.errorLog(text STRING in)
```

text
The value to be placed in the error log.

Log entries include the date and time when the entry was written.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library SysLib" on page 860
"startLog()" on page 883

getCmdLineArg()

The system function **SysLib.getCmdLineArg** returns the specified argument from the list of arguments with which the EGL program was invoked. The specified argument is returned as a string value.

```
SysLib.getCmdLineArg(index INT in)
returns (result STRING)
```

result
The *result* can be any character item.

index
The *index* can be any integer item.

- If index = 0, the command name is returned.
- If index = *n*, the nth argument name is returned.
- If *n* is greater than the argument count, a blank is returned.

The following code example loops through the argument list:

```
count int;
argument char(20);

count = 0;
argumentCount = SysLib.getCmdLineArgCount();

while (count < argumentCount)
   argument = SysLib.getCmdLineArg(count)
   count = count + 1;
end
```

The **SysLib.getCmdLineArg** function is supported only in Java environments.

Use the **SysLib.getCmdLineArgCount** function to get the number of arguments or parameters that were passed to the main EGL program at the time of its invocation.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library SysLib" on page 860
"getCmdLineArgCount()"

getCmdLineArgCount()

The system function **SysLib.getCmdLineArgCount** returns the number of arguments that were used to start the main EGL program.

```
SysLib.getCmdLineArgCount( )
returns (result INT)
```

result
The *result* is the number of arguments.

The following code example loops through the argument list:

```
count int;
argument char(20);

count = 0;
argumentCount = SysLib.getCmdLineArgCount();

while (count < argumentCount)
   argument = SysLib.getCmdLineArg(count)
   count = count + 1;
end
```

The **SysLib.getCmdLineArgCount** function is supported only in Java environments.

Use the **SysLib.getCmdLineArg** function to get the specified argument from the list of arguments with which the EGL program was involved.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library SysLib" on page 860
"getCmdLineArg()" on page 874

getMessage()

The system function **SysLib.getMessage** returns a message from the file that is referenced in the Java runtime property vgj.message.file. You can specify inserts for inclusion in the message. After retrieving the message, you can display it in a text form, print form, console form, Web page, or log file.

```
SysLib.getMessage(
  key STRING in
  [, insertArray STRING[] in])
returns (result STRING)
```

result
A field of type STRING.

key
A character field or literal of type STRING. This parameter provides the key into the properties file that is used at run time. If the key is blank, the message is a concatenation of message inserts.

insertArray
An array of type STRING. Each element contains an insert for inclusion in the message being retrieved.

In the message text, the substitution symbol is an integer surrounded by braces, as in this example from a properties file:

```
VGJ0216E = {0} is not a valid date mask for {1}.
```

The first element in *insertArray* is assigned to the placeholder numbered zero, the second is assigned to the placeholder numbered one, and so forth.

The format of the file referenced by Java runtime property vgj.messages.file is the same as for any Java properties file. For details on that format, see *Program properties file*.

Related concepts
"Syntax diagram for EGL functions" on page 732
"Java runtime properties" on page 327
"Program properties file" on page 329

Related reference
"EGL library SysLib" on page 860
"Java runtime properties (details)" on page 525

getProperty()

The system function **SysLib.getProperty** retrieves the value of a Java runtime property. If the specified property is not found, the function returns a null string ("").

```
SysLib.getProperty(propertyName STRING in)
returns (result STRING)
```

result
A field of type STRING

propertyName
A character variable or constant, or a string literal

Related concepts
"Syntax diagram for EGL functions" on page 732
"Java runtime properties" on page 327

Related reference
"EGL library SysLib" on page 860"Java runtime properties (details)" on page 525

loadTable()

The system function **SysLib.loadTable** loads data from a file into a relational database. The function is available only for EGL-generated Java programs.

```
SysLib.loadTable(
  fileName STRING in,
  insertIntoClause STRING in
  [, delimiter STRING in
  ])
```

fileName
The name of the file. The name is fully qualified or is relative to the directory from which the program is invoked.

insertIntoClause
Specify the table and rows that will provide the data. Use the syntax of an INSERT clause in an SQL INSERT statement, as in this example:

```
"INSERT INTO myTable(column1, column2)"
```

A clause like the following is sufficient if the file includes values for all table columns in column order:

```
"INSERT INTO myTable"
```

delimiter
Specifies the symbol that separates one value from the next in the file. (One row of data must be separated from the next by the newline character.)

The default symbol for *delimiter* is the value in the Java runtime property **vgj.default.databaseDelimiter**; and the default value for that property is a pipe (|).

The following symbols are not available:

- Hexadecimal characters (*0* through *9*, *a* through *f*, *A* through *F*)
- Backslash (\)
- The newline character or *CONTROL-J*

To unload information from a relational database table and insert it into a file, use the **SysLib.unloadTable** function.

Related reference
"EGL library SysLib" on page 860
"Java runtime properties (details)" on page 525
"unloadTable()" on page 884

maximumSize()

The system function **SysLib.maximumSize** returns the maximum number of rows that can be in a dynamic array; specifically, the function returns the value of the array property **maxSize**.

```
SysLib.maximumSize(arrayName anyArray in)
returns (result INT)
```

result Maximum number of rows.

arrayName
Name of the dynamic array.

Definition considerations: The item to which the value is returned must be of type INT or the following equivalent: type BIN with length 9 and no decimal places.

The array name may be qualified by a package name, a library name, or both

An error occurs if you reference an item or record that is not a dynamic array.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"Arrays" on page 69
"EGL library SysLib" on page 860

queryCurrentDatabase()

The system function **SysLib.queryCurrentDatabase** returns the product and release number of the currently connected database.

```
SysLib.queryCurrentDatabase(
  product CHAR(8) inOut,
  release CHAR(8) inOut)
```

product
Receives the database product name. The argument must be an item of type CHAR and length 8.

To determine the string that will be received when your code connects to a particular database, review the product documentation for the database or driver; or run your code in a test environment and write the received value to a file.

release
Receives the database release level. The argument must be an item of type CHAR and length 8.

To determine the string that will be received when your code connects to a particular database, review the product documentation for the database or driver; or run your code in a test environment and write the received value to a file.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library SysLib" on page 860

rollback()

The system function **SysLib.rollback** reverses updates that were made to databases and MQSeries message queues since the last commit. That reversal occurs in any EGL-generated application.

```
SysLib.rollback( )
```

A rollback occurs automatically when a program ends as a result of an error condition.

Definition considerations: When you use **SysLib.rollback** with MQ records, the following statements apply:

- Message queue updates are recoverable only if the *Include message in transaction* option is selected in MQ record part.
- Both message **scans** and **add**s are affected by **commit** and **rollback** for recoverable messages. If a **rollback** is issued following a **scan** for a recoverable message, the message is placed back on the input queue so that the input message is not lost when the transaction fails to complete successfully. Also, if a **rollback** is issued following an **add** for a recoverable message, the message is deleted from the queue.

Target platforms:

Platform	Compatibility considerations
iSeries, USS, Windows 2000, Windows NT	Reverses changes to relational databases and MQSeries message queues, as well as changes made to remote server programs that were called using client-controlled unit of work.

Example:

```
SysLib.rollback();
```

Related concepts
"Syntax diagram for EGL functions" on page 732
"Logical unit of work" on page 288
"MQSeries support" on page 247
"SQL support" on page 213

Related reference
"EGL library SysLib" on page 860

setCurrentDatabase()

The system function **SysLib.setCurrentDatabase** makes the specified database the currently active one.

```
SysLib.setCurrentDatabase(database STRING in)
```

database
A database name that was specified in SysLib.connect or VGLib.connectionService. Use a literal or variable of a character type.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library SysLib" on page 860
"connect()" on page 867
"connectionService()" on page 888

setError()

The system function **SysLib.setError** associates a message with an item in a PageHandler or with the PageHandler as a whole. The message is placed at the location of a JSF message or messages tag in the JSP and is displayed when the related Web page is displayed.

If a validation function invokes **SysLib.setError**, the Web page is re-displayed automatically when the function ends.

```
SysLib.setError(
  itemInError anyPageItem in,
  msgKey STRING in
  {, itemInsert sysLibItemInsert in})
SysLib.setError(
  this enumerationThis in,
  msgKey STRING in
  {, itemInsert sysLibItemInsert in})
SysLib.setError(msgText STRING in)
```

itemInError
The name of the PageHandler item that is in error.

this
Refers to the PageHandler from which **SysLib.setError** is issued. In this case, the message is not specific to an item, but is associated with the PageHandler as a whole. For details on **this**, see *References to variables and constants*.

msgKey
A character item or literal (type CHAR or MBCHAR) that provides the key into the message resource bundle or properties file used at run time. If the key is blank, the message is a concatenation of message inserts.

itemInsert
The character item or literal that is included as an insert to the output message. The substitution symbol in message text is an integer surrounded by braces, as in this example:

```
Invalid file name {0}
```

msgText
The character item or literal that you can specify if you do not specify other arguments. The text is associated with the page as a whole.

You can associate multiple messages with an item or PageHandler. The EGL run time displays the messages when the page is re-displayed. If control is forwarded (specifically, if the PageHandler runs a **forward** statement), those messages are lost.

Related concepts
"PageHandler" on page 180
"References to variables in EGL" on page 55

Related tasks
"Syntax diagram for EGL statements and commands" on page 733

Related reference
"EGL library SysLib" on page 860
"forward" on page 566

setLocale()

The system function **SysLib.setLocale** is used in PageHandlers. The function sets the Java locale, which determines these aspects of run-time behavior:

- The human language used for labels and messages
- The default date and time formats

You might present a list of languages on a Web page, for example, and set the Java locale based on the user's selection. The new Java locale is in use until one of the following occurs:

- You invoke **SysLib.setLocale** again; or
- The browser session ends; or
- A new Web page is presented otherwise.

In the cases mentioned, the next Web page reverts (by default) to the Java locale specified in the browser.

If the user submits a form or clicks a link that opens a new window, the Java locale in the original window is unaffected by the locale in the new window.

SysLib.setLocale conforms to the JDK 1.1 and 1.2 API documentation for class java.util.Locale. See ISO 639 for language codes and ISO 3166 for country codes.

```
SysLib.setLocale(
  languageCode CHAR(2) in,
  countryCode CHAR(2) in
  [, variant CHAR(2) in])
```

languageCode
A two-character language code specified as a literal or contained in an item of type CHAR. Only language codes that are defined by ISO 639 are valid.

countryCode
A two-character country code specified as a literal or contained in an item of type CHAR. Only country codes that are defined by ISO 3166 are valid.

variant
A variant, which is a code specified as a literal or contained in an item of type

CHAR. This code is not part of a Java specification but depends on the browser and other aspects of the user environment.

Related concepts
"Syntax diagram for EGL functions" on page 732
"PageHandler" on page 180

Related reference
"EGL library SysLib" on page 860

setRemoteUser()

The system function **SysLib.setRemoteUser** sets the userid and password that are used on calls to remote programs from Java programs.

```
SysLib.setRemoteUser(
  userID STRING in,
  password STRING in)
```

userID
The user ID on the remote system.

password
The password on the remote system.

When the linkage option part, callLink element, property remoteComType is CICSJ2C, CICSECI, or JAVA400 on a remote call, authorization is based on the values (if non-blank) that are passed to **SysLib.setRemoteUser**. If a value is blank or not specified, the value is sought in the file **csouidpwd.properties**, which includes the properties CSOUID (for the user ID) and CSOPWD (for the password). If you use neither approach, EGL run-time makes the call without a username and password.

Before invoking **SysLib.setRemoteUser**, your code can issue Java access functions that display a dialog box to prompt the user for the user ID and password. You can use one or both values in **csouidpwd.properties** as a default that takes effect when the user does not provide the information.

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
csouidpwd.properties file for remote calls
"EGL library SysLib" on page 860
"remoteComType in callLink element" on page 408

size()

The system function **SysLib.size** returns the number of rows in the specified data table or the number of elements in the specified array. The array may be a structure-item array or a dynamic array of data items or records.

```
SysLib.size(arrayName anyArray in)
returns (result INT)
```

result
The number of rows in the specified data table or the number of elements in the specified array.

arrayName
Name of the array or data table.

Definition considerations: The item to which the value is returned must be of type INT or the following equivalent: type BIN with length 9 and no decimal places.

If the array name (*arrayName*) is in a substructured element of another array, the returned value is the number of occurrences for the structure item itself, not the total number of occurrences in the containing structure (see *Examples* section).

The array name may be qualified by a package name, a library name, or both

An error occurs if you reference an item or record that is not an array.

Examples: This example uses the value returned by **SysLib.size** to control a loop:

```
// Calculate the sum of an array of numbers
sum = 0;
i = 1;
myArraySize = SysLib.size(myArray);

while (i <= myArraySize)
  sum = myArray[i] + sum;
  i = i + 1;
end
```

Next, consider the following record part:

```
Record myRecordPart
  10 siTop CHAR(40)[3];
    20 siNext CHAR(20)[2];
end
```

Given that you create a record based on myRecordPart, you can use **SysLib.size(siNext)** to determine the occurs value for the subordinate array:

```
// Sets count to 2
count = SysLib.size(myRecord.siTop.siNext);
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"Arrays" on page 69
"EGL library SysLib" on page 860

startCmd()

The system function **SysLib.startCmd** runs a system command and does not wait until the command finishes.

```
SysLib.startCmd(
  commandString STRING in
  [, modeString STRING in
  ])
```

commandString
Identifies the operating-system command to invoke.

modeString
The *modeString* can be any character or string item. The item can be in either of two modes:

- *form*: in which each character of input becomes available to the program as it is typed, i.e., every key stroke is passed directly to the command specified.

- *line*: in which input is not available until after the newline character is used, i.e., no information is sent to the command specified until the ENTER key is pressed, and then the entire line typed is sent to the command.

The following code example

```
example from Arlan here...
```

The system command that is being executed must be visible to the running program. For example, if you execute **callCmd**(″mySpecialProgram.exe″), the program ″mySpecialProgram.exe″ must be in a directory pointed to by the environment variable PATH. You may also specify the complete directory location, for example **callCmd**(″program files/myWork/mySpecialProgram.exe″).

The **SysLib.startCmd** function is supported only in Java environments.

Use the **SysLib.callCmd** function to run a system command which waits until the command finishes.

Related concepts
“Syntax diagram for EGL functions” on page 732

Related reference
“EGL library SysLib” on page 860
dummy change as necessary

startLog()

The system function **SysLib.startLog** opens an error log. Text is written into that log every time your program invokes **SysLib.errorLog**.

```
SysLib.startLog(logFile STRING in)
```

logFile
The error log.

Related concepts
“Syntax diagram for EGL functions” on page 732

Related reference
“EGL library SysLib” on page 860
“errorLog()” on page 873

startTransaction()

The system function **SysLib.startTransaction** invokes a main program asynchronously, associates that program with a printer or terminal device, and passes a record. If the receiving program is generated by EGL, the record is used to initialize the input record; if the receiver is produced by VisualAge Generator, the record is used to initialize the working storage.

This function is not supported in programs that are generated as iSeries COBOL programs.

The default behavior of this function is to start a program that resides in the same Java package. To change that behavior, specify an asynchLink element in the linkage options part that is used to generate the invoking program.

A Java program can transfer only to another Java program on the same machine.

```
SysLib.startTransaction(
  request anyBasicRecord in
  [, prID startTransactionPrId in
  [, termID CHAR(4) in ]])
```

request
The name of a basic record, which must have the following format:

- The first 2 bytes (of type SMALLINT or of type BIN without decimals) contain the length of the data to be passed to the started transaction, plus 10 for the two fields (including this one) that are not passed. The value cannot exceed 32767 bytes.If the target is a Java program, the value cannot exceed 32767 bytes; but if the target is a COBOL program on iSeries, the value cannot exceed 4095 bytes.
- The next 8 bytes (of type CHAR) are also not passed, but contain the name of the program to be started.
- The remaining part of the request record is passed.

prID
This optional 4-byte item of type CHAR is used only for transferring to a COBOL program on iSeries. The item contains the value of the output queue used for the asynchronous job, and the default value is VGEN. The output queue must be defined before the program runs **ConverseVar.printerAssociation**.

termID
This optional 4-byte item of type CHAR is ignored if specified.

For COBOL programs, EGL Server for iSeries provides support for SysLib.startTransaction by way of two command language (CL) programs:

CREATX
Acts as follows:

- Gets the current job number
- Sends the user data to the data queue VGCREATX
- Starts the job CREATXJOB to start the CL program CREATXPP

CREATXPP
Acts as follows:

- Uses the job number as the key to retrieve data from the data queue VGCREATX
- Calls the asynchronous CL program specified in the user record bytes 3 through 11

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"asynchLink element" on page 355
"EGL library SysLib" on page 860
"errorCode" on page 903
"printerAssociation" on page 896
"transfer" on page 627

unloadTable()

The system function **SysLib.unloadTable** unloads data from a relational database into a file. The function is available only for EGL-generated Java programs.

```
SysLib.unloadTable(
  fileName STRING in,
  selectStatement STRING in
  [, delimiter STRING in
  ])
```

fileName
The name of the file. The name is fully qualified or is relative to the directory from which the program is invoked.

selectStatement
Specify the criteria for selecting data from the relational database. Use the syntax of an SQL SELECT statement without including host variables; for example:

```
"SELECT column1, column2 FROM myTABLE
WHERE column3 > 10"
```

delimiter
Specifies the symbol that will separate one value from the next in the file. (One row of data must be separated from the next by the newline character.)

The default symbol for *delimiter* is the value in the Java runtime property **vgj.default.databaseDelimiter**; and the default value for that property is a pipe (|).

The following symbols are not available:

- Hexadecimal characters (*0* through *9*, *a* through *f*, *A* through *F*)
- Backslash (\)
- The newline character or *CONTROL-J*

To load information from a file and insert it into a relational database table, use the **SysLib.loadTable** function.

Related reference
"EGL library SysLib" on page 860
"loadTable()" on page 876
"Java runtime properties (details)" on page 525

verifyChkDigitMod10()

The system function **SysLib.verifyChkDigitMod10** verifies a modulus-10 check digit in a character item that begins with a series of integers.

```
SysLib.verifyChkDigitMod10(
  text anyChar in,
  checkLength SMALLINT in,
  result SMALLINT inOut)
```

text
A character item that begins with a series of integers. The item include an additional position for the check digit, which is immediately to the right of the other integers.

checkLength
An item that contains the number of characters that you want to use from the *text* item, including the position used for the check digit. This item has 4 digits and is either of type SMALLINT or is of a type BIN, with no decimal places.

result
An item that receives one of two values:

- 0, if the calculated check digit matches the value in *text*
- 1, if the calculated check digit does not match that value

This item has 4 digits and is either of type SMALLINT or is of a type BIN, with no decimal places.

You can use **SysLib.verifyChkDigitMod10** in a function-invocation statement; or as an item validator in a text form.

Example: In the following example, myInput is an item of type CHAR and contains the value 1734284; myLength is an item of type SMALLINT and contains the value 7; and myResult is an item of type SMALLINT:

```
SysLib.verifyChkDigitMod10 (myInput, myLength, myResult);
```

An algorithm is used to derive the modulus-10 check digit, and in all cases the number at the check-digit position is not considered; but when the algorithm is complete, the calculated value is compared with the number at the check-digit position.

The algorithm is described in relation to the example values:

1. Multiply the units position of the input number by 2 and multiply every alternate position, moving right to left, by 2:

   ```
   8 x 2 = 16
   4 x 2 = 8
   7 x 2 = 14
   ```

2. Add the digits of the products (16814) to the input-number digits (132) that were not multiplied by 2:

   ```
   1 + 6 + 8 + 1 + 4 + 1 + 3 + 2 = 26
   ```

3. To get the check digit, subtract the sum from the next-highest number ending in 0:

   ```
   30 - 26 = 4
   ```

   If the subtraction yields 10, the check digit is 0.

In this example, the calculated check digit matches the value in the check-digit position, and the value of myResult is 0.

Related reference
"EGL library SysLib" on page 860
"Validation properties" on page 63

verifyChkDigitMod11()

The system function **SysLib.verifyChkDigitMod11** verifies a modulus-11 check digit in a character item that begins with a series of integers.

```
SysLib.verifyChkDigitMod11(
  text anyChar in,
  checkLength SMALLINT in,
  result SMALLINT inOut)
```

text
A character item that begins with a series of integers. The item include an additional position for the check digit, which is immediately to the right of the other integers.

checkLength
An item that contains the number of characters that you want to use from the *text* item, including the position used for the check digit. This item has 4 digits and is either of type SMALLINT or is of a type BIN, with no decimal places.

result
An item that receives one of two values:

- 0, if the calculated check digit matches the value in *text*
- 1, if the calculated check digit does not match that value

This item has 4 digits and is either of type SMALLINT or is of a type BIN, with no decimal places.

You can use **SysLib.verifyChkDigitMod11** in a function-invocation statement; or as an item validator in a text form.

Example: In the following example, myInput is an item of type CHAR and contains the value 56621869; myLength is an item of type SMALLINT and contains the value 8; and myResult is an item of type SMALLINT:

```
sysLib.verifyChkDigitMod11 (myInput, myLength, myResult);
```

An algorithm is used to derive the modulus-11 check digit, and in all cases the number at the check-digit position is not considered; but when the algorithm is complete, the calculated value is compared with the number at the check-digit position. The algorithm is described in relation to the example values:

1. Multiply the digit at the units position of the input number by 2, at the tens position by 3, at the hundreds position by 4, and so on, but let myLength ″ 1 be the largest number used as a multiplier; and if more digits are in the input number, begin the sequence again using 2 as a multiplier:

   ```
   6 x 2 = 12
   8 x 3 = 24
   1 x 4 = 4
   2 x 5 = 10
   6 x 6 = 36
   6 x 7 = 42
   5 x 2 = 10
   ```

2. Add the products of the first step and divide the sum by 11:

   ```
   (12 + 24 + 4 + 10 + 36 + 42 + 10) / 11
   = 138 / 11
   = 12 remainder 6
   ```

3. To get the check digit, subtract the remainder from 11 to get the self-checking digit:

   ```
   11 - 6 = 5
   ```

   If the remainder is 0 or 1, the check digit is 0.

In this example, the calculated check digit matches the value in the check-digit position, and the value of myResult is 0.

Related reference
“EGL library SysLib” on page 860
“Validation properties” on page 63

wait()

The system function **SysLib.wait** suspends execution for the specified number of seconds.

```
SysLib.wait(timeInSeconds BIN(9,2) in)
```

timeInSeconds
The time can be any numeric item or literal. Fractions of a second down to hundredths of seconds are honored if the number is not an integer.

You can use **SysLib.wait** when two asynchronously running programs need to communicate through a record in a shared file or database. One program might need to suspend processing until the other program updates the information in the shared record.

Example

```
SysLib.wait(15); // waits for 15 seconds
```

Related concepts
"Syntax diagram for EGL functions" on page 732

Related reference
"EGL library SysLib" on page 860

EGL library VGLib

The VGLib functions are shown below:

System function/Invocation	Description
connectionService(*userID*, *password*, *serverName* [, *product*, *release* [, *connectionOption*]])	Provides two benefits: • Allows a program to connect or disconnect to a database at run time. • Receives (optionally) the database product name and release level. You can use the received information in a **case**, **if**, or **while** statement so that run-time processing is dependent on characteristics of the database.
result = getVAGSysType()	Identifies the target system in which the program is running.

connectionService()

The system function **VGLib.connectionService** provides two benefits:

- Allows a program to connect or disconnect to a database at run time.
- Receives (optionally) the database product name and release level. You can use the received information in a **case**, **if**, or **while** statement so that run-time processing is dependent on characteristics of the database.

When you use **VGLib.connectionService** to create a new connection, specify the isolation level by setting the system variable **VGVar.sqlIsolationLevel**.

VGLib.connectionService is for use only in programs migrated from VisualAge Generator and EGL 5.0. The function is supported (at development time) if the EGL preference **VisualAge Generator compatibility** is selected or (at generation time) if the build descriptor option **VAGCompatibility** is set to *yes*.

For new programs, use these system functions instead:

- SysLib.connect
- SysLib.disconnect
- SysLib.disconnectAll
- SysLib.queryCurrentDatabase
- SysLib.setCurrentDatabase

VGLib.connectionService does not return a value.

```
VGLib.connectionService(
  userID CHAR(8) in,
  password CHAR(8) in,
  serverName CHAR(18) in
  [, product CHAR(8) inOut,
     release CHAR(8) inOut
  [, connectionOption STRING in
  ]])
```

userID
UserID used to access the database. The argument must be an item of type CHAR and length 8; a literal is not valid. The argument is required, but is ignored for COBOL generation. For background information, see *Database authorization and table names*.

password
Password used to access the database. The argument must be an item of type CHAR and length 8; a literal is not valid. The argument is required, but is ignored for COBOL generation.

serverName
Specifies a connection and uses that connection to assign values to the arguments *product* and *release*, if those arguments are included in the invocation of **VGLib.connectionService**.

The argument *serverName* is required and must be an item of type CHAR and length 18. Any of the following values are valid:

blanks (no content)
If a connection is in place, **VGLib.connectionService** maintains that connection. If a connection is not in place, the result (other than to assign values) is to return to the connection status that is in effect at the beginning of a run unit, as described in *Default database*.

RESET
In relation to COBOL, RESET commits changes, closes cursors, releases locks, disconnects from the current database, and returns to the connection status that is in effect at the beginning of a run unit, as described in *Default database*. Nevertheless, if you intend to specify RESET in this case, invoke **SysLib.commit** or **SysLib.rollback** before invoking **VGLib.connectionService**.

In relation to a Java program, RESET reconnects to the default database; but if the default database is not available, the connection status remains unchanged.

For further details, see *Default database*.

serverName
Identifies a database.

If your code is running as a COBOL program, the following statements apply:

- The value of serverName must be a value in the location column in the SYSIBM.LOCATIONS table, which is in the DB2 UDB subsystem
- Specifying a database closes all cursors, releases locks, ends any existing connection, and connects to the database; nevertheless, if you intend to specify a value for **serverName**, invoke **SysLib.commit** or **SysLib.rollback** before invoking **VGLib.connectionService**

If your code is running as a Java program, the following statements apply:

- The physical database name is found by looking up the property **vgj.jdbc.database.***server*, where *server* is the name of the server specified on the **VGLib.connectionService** call. If this property is not defined, the server name that is specified on the **VGLib.connectionService** call is used as is.
- The format of the database name is different for J2EE connections as compared with non-J2EE connections:
  - If you generated the program for a J2EE environment, use the name to which the datasource is bound in the JNDI registry; for example, jdbc/MyDB. This situation occurs if build descriptor option **J2EE** was set to YES.
  - If you generated the program for a non-J2EE JDBC environment, use a connection URL; for example, jdbc:db2:MyDB. This situation occurs if option **J2EE** was set to NO.

product
Receives the database product name. The argument, if any, must be an item of type CHAR and length 8.

To determine the string that will be received when your code connects to a particular database, review the product documentation for the database or driver; or run your code in a test environment and write the received value to a file.

release
Receives the database release level. The argument, if any, must be an item of type CHAR and length 8.

To determine the string that will be received when your code connects to a particular database, review the product documentation for the database or driver; or run your code in a test environment and write the received value to a file.

connectionOption
Valid values are as follows:

D1E
D1E is the default. The *1* in the option name indicates that only a *one*-phase commit is supported, and the *E* indicates that any disconnect must be *explicit*. In this case, a commit or rollback has no effect on an existing connection.

If you are generating a non-CICS COBOL program, you can access only one database at a time, and you must explicitly request a disconnect to release the resources from a previous connection (if any) or to connect to a different database.

If you are generating a Java program, the following statements apply:

- A connection to a database does not close cursors, release locks, or end an existing connection. If the run unit is already connected to the same database, however, the effect is equivalent to specifying DISC then D1E.
- You can use multiple connections to read from multiple databases, but you should update only one database in a unit of work because only a one-phase commit is available.

D1A

The *1* in the option name indicates that only a *one*-phase commit is supported, and the *A* indicates that any disconnect is *automatic*. Characteristics of this option are as follows:

- You can connect to only one database at a time
- A commit, rollback, or connection to a database ends an existing connection

DISC
Disconnect from the specified database. Disconnecting from a database causes a rollback and releases locks, but only for that database.

DCURRENT
Disconnect from the currently connected database. Disconnecting from a database causes a rollback and releases locks, but only for that database.

DALL
Disconnect from all connected databases. Disconnecting from all databases causes a rollback in those database, but not in other recoverable resources.

SET
Set a connection current. (By default, the connection most recently made in the run unit is current.)

The following values are supported for compatibility with VisualAge Generator, but are equivalent to D1E: R, D1C, D2A, D2C, D2E.

Definition considerations: **VGLib.connectionService** sets the following system variables:

- VGVar.sqlerrd
- SysVar.sqlca
- SysVar.sqlcode
- VGVar.sqlerrmc (available in COBOL code only)
- VGVar.sqlWarn

Example:

```
VGLib.connectionService(myUserid, myPassword,
  myServerName, myProduct, myRelease, "D1E");
```

Related concepts
"Syntax diagram for EGL functions" on page 732
"Logical unit of work" on page 288
"Run unit" on page 721
"SQL support" on page 213

Related tasks
"Syntax diagram for EGL statements and commands" on page 733
"Setting up a J2EE JDBC connection" on page 341
"Understanding how a standard JDBC connection is made" on page 245

Related reference
"Database authorization and table names" on page 453
"Default database" on page 234
"EGL library VGLib" on page 888

"Java runtime properties (details)" on page 525
"sqlDB" on page 384
"sqlca" on page 909
"sqlcode" on page 910
"sqlerrd" on page 923

“sqlerrmc” on page 924
“sqlIsolationLevel” on page 924
“sqlWarn” on page 925

getVAGSysType()

The system function **VGLib.getVAGSysType** identifies the target system in which the program is running. The function is supported (at development time) if the program property **VAGCompatibility** is selected or (at generation time) if the build descriptor option **VAGCompatibility** is set to *yes*.

If the generated output is a Java wrapper, **VGLib.getVAGSysType** is not available. Otherwise, the function returns the character value that would have been returned by the VisualAge Generator EZESYS special function word. If the current system was not supported by VisualAge Generator, the function returns the uppercase, string equivalent of the code returned by **SysVar.systemType**.

```
VGLib.getVAGSysType( )
returns (result CHAR(8))
```

result
A character string that contains the system type code, as shown in the next table.

VGLib.getVAGSysType returns the VisualAge Generator equivalent of the value in **SysVar.systemType**.

Value in sysVar.systemType	Value returned by VGLib.getVAGSysType
AIX	″AIX″
DEBUG	″ITF″
ISERIESC	″OS400″
ISERIESJ	″OS400″
LINUX	″LINUX″
USS	″OS390″
WIN	″WINNT″

The value returned by **VGLib.getVAGSysType** can be used only as a character string; you cannot use the returned value with the operands *is* or *not* in a logical expression, as you can with **sysVar.systemType**:

```
// valid ONLY for sysVar.systemType
if sysVar.systemType is AIX
  call myProgram;
end
```

The only place that **VGLib.getVAGSysType** can be used is as the source in an assignment or **move** statement.

The characteristics of **VGLib.getVAGSysType** are as follows:

Primitive type
CHAR

Data length
8 (padded with blanks)

Is value always restored after a converse?
Yes

It is recommended that you use **sysVar.systemType** instead of **VGLib.getVAGSysType**.

Definition considerations: The value of **VGLib.getVAGSysType** does not affect what code is validated at generation time. For example, the following **add** statement is validated even if you are generating for Windows:

```
mySystem CHAR(8);
mySystem = VGLib.getVAGSysType();
if (mySystem == "AIX")
  add myRecord;
end
```

To avoid validating code that will never run in the target system, move the statements that you do not want to validate to a second program; then, let the original program call the new program conditionally:

```
mySystem CHAR(8);
mySystem = VGLib.getVAGSysType();

if (mySystem == "AIX")
  call myAddProgram myRecord;
end
```

An alternative way to solve the problem is available, but only if you use **sysVar.systemType** instead of **VGLib.getVAGSysType**; for details, see *eliminateSystemDependentCode*.

Related reference
"EGL library VGLib" on page 888
"eliminateSystemDependentCode" on page 370
"systemType" on page 911

System variables outside of EGL libraries

A variable contained in an EGL library is global to the run unit. Other system variables have different scoping characteristics and are categorized as follows:

ConverseVar
Variables that are useful primarily in textUI applications.

SysVar
Variables that are useful for general purposes.

VGVar
Variables that are useful primarily in applications migrated from VisualAge Generator.

If you are referring to the system variable when you have another, same-named identifier in scope, you must include the category name as a qualifier. For example, you must specify **ConverseVar.eventKey** rather than **eventKey** if a second variable named **eventKey** is in scope. If a same-named identifier is not in scope, the qualifier is optional.

Related concepts
"References to variables in EGL" on page 55
"Scoping rules and ″this″ in EGL" on page 53

Related reference
"ConverseVar"
"SysVar" on page 899
"VGVar" on page 915

ConverseVar

The qualifier **ConverseVar** can precede the name of each EGL system variable listed in the next table. These variables are useful primarily in textUI applications.

System variable	Description
commitOnConverse	Specify whether a commit and a release of resources occurs in a text application, before a non-segmented program issues a converse. The default value is 0 (meaning *no*) for non-segmented programs and 1 (meaning *yes*) for segmented programs.
eventKey	Identifies the key that the user pressed to return a form to an EGL text program.
printerAssociation	Allows you to specify, at run time, the output destination when you print a print form.
segmentedMode	Used in a text application to change the effect of the converse statement, but the variable is ignored for this purpose in called programs.
validationMsgNum	Contains the value assigned by **ConverseLib.validationFailed** in a text application, so you can determine if a validation function reported an error.

Related concepts
"References to variables in EGL" on page 55
"Scoping rules and "this" in EGL" on page 53

Related reference
"System variables outside of EGL libraries" on page 893

commitOnConverse

The system variable **ConverseVar.commitOnConverse** specifies whether a commit and a release of resources occurs in a text application, before a non-segmented program issues a converse. The default value is 0 (meaning *no*) for non-segmented programs and 1 (meaning *yes*) for segmented programs.

You can use **ConverseVar.commitOnConverse** in any of these ways:

- As the source or target of an assignment or **move** statement
- As the variable in a logical expression used in a **case**, **if**, or **while** statement
- As the argument in a **return** or **exit** statement

Other characteristics of **ConverseVar.commitOnConverse** are as follows:

Primitive type
NUM

Data length
1

Is value always restored after a converse?
Yes

For details on using this variable, see *Segmentation*.

Related concepts
"Segmentation in text applications" on page 149

Related reference
"converse" on page 554
"System variables outside of EGL libraries" on page 893

eventKey

The system variable **ConverseVar.eventKey** identifies the key that the user pressed to return a form to an EGL text program. The value is reset each time that the program runs the **converse** statement.

If the EGL code has no input form, the initial value of **ConverseVar.eventKey** is **ENTER**.

The following values are valid (whether uppercase, lowercase, or a combination):
- **ENTER**
- **BYPASS** (which refers to any of the keys that were specified as bypass keys for the form; or if none were specified for the form, any of the keys that were specified as bypass keys for the formGroup; or if none were specified for the formGroup, any of the keys that were specified as bypass keys for the program)
- **PA1** through **PA3**
- **PF1** through **PF24** (as also used for F1 through F24)
- **PAKEY** (for any PA key)
- **PFKEY** (for any PF or F key)

Note: PA keys are always treated as bypass keys.

You can use **ConverseVar.eventKey** as an operand in an **if** or **while** statement.

The characteristics of this system variable are as follows:

Primitive type
CHAR

Data length
1

Value saved across segments
No

ConverseVar.eventKey is not valid in a batch program.

Example: The comparison operator for ConverseVar.eventKey is either *is* or *not*, as in this example:

```
if (ConverseVar.eventKey IS PF3)
  exit program(0);
end
```

Related reference
"Logical expressions" on page 484
"System variables outside of EGL libraries" on page 893

printerAssociation

The system variable **ConverseVar.printerAssociation** allows you to specify, at run time, the output destination when you print a print form.

You can use this variable in any of these ways:

- As the source or target in an assignment or move statement
- As a comparison value in a logical expression
- As the value in a return statement

Characteristics of **ConverseVar.printerAssociation** are as follows:

Primitive type
CHAR

Data length
Varies by file type

Is value always restored after a converse?
Yes

ConverseVar.printerAssociation is initialized to the system resource name specified during generation or for debugging. If a program passes control to another program, the value of **ConverseVar.printerAssociation** is set to the default value for the receiving program.

Even when multiple print jobs are allowed for a given print form, the close statement closes only the file related to the current value of **ConverseVar.printerAssociation**.

Details specific to Java output: For Java output, you set **ConverseVar.printerAssociation** to a two-part string with an intervening colon:

```
jobID:destination
```

jobID A sequence of characters (without a colon) that uniquely identifies each print job. The characters are case sensitive (*job01* is different from *JOB01*), and you can reuse *jobID* after a print job closes.

You can use different jobs to promote a different kind of output or a different ordering of output, depending on the flow of events in your code. Consider the following sequence of EGL statements, for example:

```
ConverseVar.printerAssociation = "job1";
print form1;
ConverseVar.printerAssociation = "job2";
print form2;
ConverseVar.printerAssociation = "job1";
print form3;
```

When the program ends, two print jobs are created:

- form1 followed by form3
- form2 alone

destination
The printer or file that receives the output.

The string *destination* is optional and is ignored if the print job is still open. The following statements apply if the string is absent:

- You can omit the colon that precedes *destination*
- In most cases, the program shows a print preview dialog from which the user can specify a printer or a file for output. The exception occurs if the curses library is used on UNIX; in that case, the print job goes to the default printer.

The following statements apply to the setting of *destination* when you are generating for Windows 2000/NT/XP:

- To send output to the default printer, do as follows--
  - Specify a value that matches the **fileName** property in the resource associations part.
  - Change the Java run-time properties so that *spool* (rather than *seqws*) is the value of the related file type. For example, in the resource associations part, if the value of the **fileName** property is *myFile* and the value of **systemName** is *printer*, you must change the settings of Java run-time properties so that vgj.ra.myFile.fileType is set to *spool* rather than *seqws*. After your change, the properties are as follows:

    ```
    vgj.ra.myFile.systemName=printer
    vgj.ra.myFile.fileType=spool
    ```
- To send output to a file, specify a value that matches the **fileName** property in the resource associations part, when *seqws* is the value of the related **fileType** property in the resource associations part. The **systemName** property is the resource associations part contains the name of the operating-system file that receives the output.
- Do not specify the value *printer* as the value of *destination*. If you do, the print preview dialog is displayed to the user, but that behavior may change in later versions of EGL.

The following statements apply to the setting of *destination* when you are generating for UNIX:

- To send output to the default printer (regardless of whether the curses library is in use), specify a value that matches the **fileName** property in the resource associations part, when *spool* is the value of the related **fileType** property in the resource associations part.
- To send output to a file, specify a value that matches the **fileName** property in the resource associations part, when *seqws* is the value of the related **fileType** property in the resource associations part. The **systemName** property in the resource associations part contains the name of the operating-system file that receives the output.
- Do not specify the value *printer* as the value of *destination*. If you do (and if the curses library is not in use), the print preview dialog is displayed to the user, but that behavior may change in later versions of EGL.

Details specific to COBOL output for iSeries: In relation to iSeries COBOL, set the system variable **ConverseVar.printerAssociation** to the value of a **fileName** property in the resource associations part that is used at generation time. The file type must be of type SEQ and not of type SPOOL.

Multiple print jobs are not supported for COBOL programs that are generated for iSeries, and when **ConverseVar.printerAssociation** is set, the EGL run time closes

the old file (to complete the previous output of data); uses the iSeries command OVRPRTF to override the file name; and opens the new file.

Prior to its use, the value in **ConverseVar.printerAssociation** is folded to uppercase; but the value in the system variable itself remains unchanged. The value of **ConverseVar.printerAssociation** tests true when compared against a lowercase version if the system variable was initialized with a lowercase version.

The value set in **ConverseVar.printerAssociation** is propagated from the call level and changed to all the subordinate call levels. The value is not propagated, however if the program opened the file previously.

segmentedMode

The system variable **ConverseVar.segmentedMode** is used in a text application to change the effect of the converse statement, but the variable is ignored for this purpose in called programs. For background information, see *Segmentation*.

Values of **ConverseVar.segmentedMode** are as follows:

1 The next **converse** statement runs in segmented mode.

0 The next **converse** statement runs in non-segmented mode.

The default value is 0 for non-segmented programs and 1 for segmented programs. The variable is reset to the default after the **converse** statement runs.

You can use this variable in any of these ways:

- As the source or destination in an assignment or move statement
- As the count value in a **move...for count** statement
- As a comparison value in a logical expression
- As the value in a return statement

Characteristics of **ConverseVar.segmentedMode** are as follows:

Primitive type
NUM

Data length
1

Is value restored after a converse?
No

Related concepts
"Segmentation in text applications" on page 149

Related reference
"System variables outside of EGL libraries" on page 893

validationMsgNum

The system variable **ConverseVar.validationMsgNum** contains the value assigned by **ConverseLib.validationFailed** in a text application, so you can determine if a validation function reported an error. The value is reset to zero in each of the following cases:

- The program initializes

- The program issues a converse, display, or print statement
- The program reissues a converse statement to display a text form as the result of a validation error

You can use **ConverseVar.validationMsgNum** in these ways:

- As the source or target of an assignment or **move** statement (also allowed in the "for count" of a **move** statement)
- As the variable in a logical expression
- As the argument in a **return** or **exit** statement

The characteristics of **ConverseVar.validationMsgNum** are as follows:

Primitive type
INT

Is value always restored after a converse?
No

Example

```
/*Keep the first message number that was set
  during validation routines */
if (ConverseVar.validationMsgNum > 0)
    ConverseLib.validationFailed(10);
end
```

Related reference
"converse" on page 554
"validationFailed()" on page 767
"display" on page 556
"print" on page 613
"System variables outside of EGL libraries" on page 893

SysVar

The qualifier **SysVar** can precede the name of each EGL system variable listed in the next table. These variables are useful for general purposes.

System variable	Description
arrayIndex	Contains a number: • The number of the first element in an array that matches the search condition of a simple logical expression with an **in** operator. • Zero, if no array element matches the search condition. • The number of the last element modified in the target array after a **move ... for count** statement.

System variable	Description
callConversionTable	Contains the name of the conversion table that is used to convert data when your program does the following at run time: • Passes arguments in a call to a program on a remote system • Passes arguments when invoking a remote program by way of the system function sysLib.startTransaction • Accesses a file at a remote location
errorCode	Receives a status code after any of the following events: • The invocation of a call statement, if that statement is in a try block • An I/O operation on an indexed, MQ, relative, or serial file • The invocation of almost any system function in these cases-- – The invocation is within a try block; or – The program is running in VisualAge Generator Compatibility mode and **VGVar.handleSysLibraryErrors** is set to 1
formConversionTable	Contains the name of the conversion table that is used for bidirectional text conversion when an EGL-generated Java program acts as follows: • Shows a text or print form that includes a series of Hebrew or Arabic characters; or • Shows a text form that accepts a series of Hebrew or Arabic characters from a user.
overflowIndicator	Is set to 1 when arithmetic overflow occurs. By checking the value of this variable, you can test for overflow conditions.
remoteSystemID	Contains the system name for the location of a remote entity: a program, VSAM file, CICS transaction, or transient data queue.
returnCode	Contains an external return code, as set by your program and made available to the operating system.
sessionID	Contains an ID that is specific to the Web application server session.
sqlca	Contains the entire SQL communication area (SQLCA).
sqlcode	Contains the return code for the most recently completed SQL I/O operation. The code is obtained from the SQL communications area (SQLCA) and can vary with the relational database manager.

System variable	Description
sqlState	Contains the SQL state value for the most recently completed SQL I/O operation. The code is obtained from the SQL communications area (SQLCA) and can vary with the relational database manager.
systemType	Identifies the target system in which the program is running.
terminalID	In relation to COBOL code on iSeries, is initialized to blanks; and if the code is interactive, the variable is reset to the terminal device name received from a query of the attributes of the active job. In relation to Java code, is initialized from the Java Virtual Machine system property *user.name* and is blank if the property cannot be retrieved.
transactionID	As described in the topic *transactionID*.
transferName	Allows you to specify, at run time, the name of the program or transaction to which you want to transfer.
userID	Contains a user identifier in environments where one is available.

Related concepts
"References to variables in EGL" on page 55
"Scoping rules and ″this″ in EGL" on page 53

Related reference
"System variables outside of EGL libraries" on page 893

arrayIndex

The system variable **SysVar.arrayIndex** contains a number:

- The number of the first element in an array that matches the search condition of a simple logical expression with an **in** operator, as shown in a later example.
- Zero, if no array element matches the search condition.
- The number of the last element modified in the target array after a **move ... for count** statement.

You can use **SysVar.arrayIndex** as any of these:

- As an array subscript to access the matching row or array element
- As the source or target in an assignment or **move** statement
- As the count value in a **move ... for count** statement
- As a variable in a logical expression
- As the argument in an **exit** or **return** statement

The characteristics of **SysVar.arrayIndex** are as follows:

Primitive type
BIN

Data length
4

Is value always restored after a converse?
Only in a non-segmented text program; for details see *Segmentation*

Example: Assume that the record *myRecord* is based on the following part:

```
Record mySerialRecPart
  serialRecord:
    fileName = "myFile"
  end
  10 zipCodeArray   CHAR(9)[100];
  10 cityStateArray CHAR(30)[100];
end
```

Furthermore, assume that the arrays are initialized with zip codes and city-and-state combinations.

The following code sets the variable *currentCityState* to the city and state that corresponds to the specified zip code:

```
currentZipCode = "27540";
if (currentZipCode in myRecord.zipCodeArray)
  currentCityState = myRecord.cityStateArray[SysVar.arrayIndex];
end
```

After the **if** statement, **SysVar.arrayIndex** contains the index of the first *zipCodeArray* element that contains the value of ″27540″. If ″27540″ is not found in *zipCodeArray*, the value of **SysVar.arrayIndex** is 0.

Related concepts
“Segmentation in text applications” on page 149

Related reference
“Arrays” on page 69
“in operator” on page 518
“Logical expressions” on page 484
“System variables outside of EGL libraries” on page 893

callConversionTable

The system variable **SysVar.callConversionTable** contains the name of the conversion table that is used to convert data when your program does the following at run time:

- Passes arguments in a call to a program on a remote system
- Passes arguments when invoking a remote program by way of the system function SysLib.startTransaction
- Accesses a file at a remote location

The conversion occurs when the data is being moved between EBCDIC-based and ASCII-based systems or between systems that use different code pages. Conversion is possible only if the linkage options part used at generation time specifies **PROGRAMCONTROLLED** as the value of property **conversionTable** in the callLink or asynchLink element. Conversion does not occur, however, if **PROGRAMCONTROLLED** is specified but **SysVar.callConversionTable** is blank.

Characteristics: The characteristics of **SysVar.callConversionTable** are as follows:

Primitive type
CHAR

Data length
8

Value saved across segments?
Yes

Definition considerations: You should use **SysVar.callConversionTable** to switch conversion tables in a program or to turn data conversion on or off in a program.

SysVar.callConversionTable is initialized to blanks. To cause conversion to occur, make sure that the linkage options part includes the value **PROGRAMCONTROLLED**, as described earlier, and move the name of a conversion table to the system variable. You can set **SysVar.callConversionTable** to an asterisk (*) to use the default conversion table for the default national language code. For Java, this setting references the default locale on the target system provided the locale is mapped to one of the languages that can be specified for the **targetNLS** build descriptor option. For COBOL, this setting references the default national language code you specified when you installed EGL Server for iSeries.

Conversion is performed on the system that originates the call, invocation, or file access. When you define multiple levels of a record structure, conversion is performed on the lowest level items (the items with no substructure).

You can use **SysVar.callConversionTable** in these ways:
- As the source or target operand in an assignment or **move** statement
- As a variable in a logical expression
- As an argument in a **return** or **exit** statement

A comparison of **SysVar.callConversionTable** with another value tests true only if the match is exact. If you initialize **SysVar.callConversionTable** with a lowercase value, for example, the lowercase value matches only a lowercase value.

The value that you place in **SysVar.callConversionTable** remains unchanged for purposes of comparison.

Example:
```
  SysVar.callConversionTable = "ELACNENU";
// conversion table for US English COBOL generation
```

Related reference
"Data conversion" on page 454

"startTransaction()" on page 883
"System variables outside of EGL libraries" on page 893
"targetNLS" on page 389

errorCode
The system variable **SysVar.errorCode** receives a status code after any of the following events:
- The invocation of a call statement, if that statement is in a try block
- An I/O operation on an indexed, MQ, relative, or serial file
- The invocation of almost any system function in these cases--
  - The invocation is within a try block; or
  - The program is running in VisualAge Generator Compatibility mode and **VGVar.handleSysLibraryErrors** is set to 1

The **SysVar.errorCode** values associated with a given system function are described in relation to the system function, not in the current topic.

You can use **SysVar.errorCode** in these ways:

- As the source or target in an assignment or **move** statement
- As a variable in a logical expression
- In a function invocation, as an argument associated with an in, out, or inOut parameter

SysVar.errorCode is set to 0 if the call, I/O, or system function invocation is successful.

The characteristics of **SysVar.errorCode** are as follows:

Primitive type
CHAR

Data length
8

Is value always restored after a converse?
Yes

For an overview that includes details on **SysVar.errorCode**, see *Exception handling*.

Definition considerations: If you are generating Java code, the list of possible **SysVar.errorCode** values is provided in *EGL Java run-time error codes*.

I/O errors and COBOL code: In relation to COBOL code, the following rules apply:

- If you set the build descriptor option **sysCodes** to YES and perform an I/O operation on a resource other than a database, **SysVar.errorCode** contains a return code that is specific to the type of resource; for example, specific to a VSAM file. To interpret this code, refer to the appropriate manual for the resource.
- If you set that build description option to NO, however, the file-related return codes are independent of the type of resource.

The next table indicates some of the COBOL file status codes that can be returned, along with the value that is placed in **SysVar.errorCode** if you generate COBOL output with the build descriptor option **sysCodes** set to NO. Also shown is the EGL I/O error value, which is unaffected by the value in **sysCodes**.

COBOL file status code (as placed in SysVar.errorCode when sysCodes is set to YES)	SysVar.errorCode value when sysCodes is set to NO	EGL I/O error value (a blank in this column means "not applicable")
00000000, 00000005, 00000007	00000000	
00000002	00000103	duplicate, ioError
00000004 (var record format)	00000000	
00000004 (other)	00000220	format, hardIOError, ioError
00000010, 00000014, 00000046	00000102	endOfFile, ioError
00000022	00000206	ioError, unique
00000023 (start)	00000102	endOfFile, ioError
00000023 (other)	00000205	noRecordFound, ioError

COBOL file status code (as placed in SysVar.errorCode when sysCodes is set to YES)	SysVar.errorCode value when sysCodes is set to NO	EGL I/O error value (a blank in this column means ″not applicable″)
00000024, 00000034 (access method not relative or relative key not 0)	0000025A	full, hardIOError, ioError
00000035	00000251	fileNotFound, hardIOError, ioError
00000038	00000218	fileNotAvailable, hardIOError, ioError
00000039, 00000095	00000220	format, ioError
0000009D (iSeries COBOL only)	00000381	deadlock, hardIIOError, ioError

The next table shows the setting for **SysVar.errorCode** when the run-time system returns other COBOL file status codes.

Type of request	SysVar.errorCode value when sysCodes is set to NO	EGL I/O error value
open	00000500	ioError, hardIOError
close or unlock	00000989	ioError, hardIOError
read or start	00000987	ioError, hardIOError
write	00000988	ioError, hardIOError

Example:

```
if (SysVar.errorCode == "00000008")
 exit program;
end
```

Related reference

“EGL Java runtime error codes” on page 935
“Exception handling” on page 89
“System variables outside of EGL libraries” on page 893
“try” on page 628
“handleSysLibraryErrors” on page 922

formConversionTable

The system variable **SysVar.formConversionTable** contains the name of the conversion table that is used for bidirectional text conversion when an EGL-generated Java program acts as follows:

- Shows a text or print form that includes a series of Hebrew or Arabic characters; or
- Shows a text form that accepts a series of Hebrew or Arabic characters from a user.

Characteristics: The characteristics of **SysVar.formConversionTable** are as follows:

Primitive type
CHAR

Data length
8

Value saved across segments?
Yes

Related reference
"Bidirectional language text" on page 458
"Data conversion" on page 454

"System variables outside of EGL libraries" on page 893

overflowIndicator

The system variable **SysVar.overflowIndicator** is set to 1 when arithmetic overflow occurs. By checking the value of this variable, you can test for overflow conditions.

After detection of an overflow condition, **SysVar.overflowIndicator** is not reset automatically. You must include code in your program to reset **SysVar.overflowIndicator** to 0 before performing any calculations that may trigger overflow checks.

You can use **SysVar.overflowIndicator** in these ways:

- As the source or target in an assignment or **move** statement (also allowed in the "for count" of a **move** statement)
- As a variable in a logical expression
- As the argument in an **exit** or **return** statement

The characteristics of **SysVar.overflowIndicator** are as follows:

Primitive type
NUM

Data length
1

Is value always restored after a converse?
Yes

Example:

```
SysVar.overflowIndicator = 0;
VGVar.handleOverflow = 2;
a = b;
if (SysVar.overflowIndicator == 1)
  add errorrecord;
end
```

Related reference
"Assignments" on page 352

"System variables outside of EGL libraries" on page 893
"handleOverflow" on page 921

remoteSystemID

The system variable **SysVar.remoteSystemID** contains the system name for the location of a remote program. This variable does not support dynamic definition of programs, but does support dynamic selection from a predefined set of locations.

SysVar.remoteSystemID is initialized to blanks and must be set before doing any call that requires use of this variable.

If you generate a COBOL program, any value in **SysVar.remoteSystemID** is folded to uppercase. Regardless of the target language, however, any comparison of **SysVar.remoteSystemID** and a character string is case-sensitive and is based on the value assigned to the variable. The comparison in the following code resolves to false, for example:

```
sysVar.remoteSystemID = "myWin";

// resolves to false
if (sysVar.remoteSystemID == "MYWIN")
  record1.resourceAssociation = "myCorp.txt";
end
```

You can use **SysVar.remoteSystemID** in most places where an item is allowed: as the target or source in an assignment statement, as a value passed to a system function, as an item in a logical expression, or as the argument in a return statement.

The characteristics of **SysVar.remoteSystemID** are as follows:

Primitive type
CHAR

Data length
8 (padded with blanks)

Value saved across segments?
Yes

Access of remote programs: The value of **SysVar.remoteSystemID** provides access of the remote program only if the linkage options part, callLink element, property **location** is set to PROGRAMCONTROLLED. For details on the meaning of **SysVar.remoteSystemID** for remote programs, see the description of *system name* in *location in callLink element*.

Target platforms:

Platform	Compatibility considerations
iSeries COBOL	Not supported

Example:

```
sysVar.remoteSystemID = "myWIN";

// resolves to true
if (sysVar.remoteSystemID == "myWIN")
  record1.resourceAssociation = "myCorp.txt";
end
```

Related concepts
"Linkage options part" on page 291

Related tasks
"Editing the asynchLink element of a linkage options part" on page 296
"Editing the callLink element of a linkage options part" on page 294

Related reference
"asynchLink element" on page 355

"location in callLink element" on page 402

“startTransaction()” on page 883
“System variables outside of EGL libraries” on page 893
“transferToProgram element” on page 926

returnCode

The system variable **SysVar.returnCode** contains an external return code, as set by your program and made available to the operating system. It is not possible to pass return codes from one EGL program to another. A non-zero return code does not cause EGL to run an onException block, for example.

The initial value of **SysVar.returnCode** is zero. For Java output, the value must be in the range of -2147483648 to 2147483647, inclusive. For COBOL output, the value must be in the range of 0 to 512, inclusive.

In relation to Java code,**SysVar.returnCode** is meaningful only for a main text program (which runs outside of J2EE) or a main batch program (which runs either outside of J2EE or in a J2EE application client). The purpose of **SysVar.returnCode** in this context is to provide a code for the command file or exec that invokes the program. If the program ends with an error that is not under the program’s control, the EGL run time ignores the setting of **SysVar.returnCode** and attempts to return the value 693.

In relation to COBOL code, the following statements apply:

- **SysVar.returnCode** contains a code that in most cases is provided to the operating system and to any caller that is not an EGL-generated program. If the program ends with an error that is not handled by your code, the EGL run time attempts to return a value greater than 512.
- **SysVar.returnCode** is implemented using the COBOL **RETURN-CODE** special register.

You can use **SysVar.returnCode** in these ways:

- As the source or target in an assignment or **move** statement (also allowed in the ″for count″ of a **move** statement)
- As a variable in a logical expression
- As the argument in an **exit** or **return** statement

The characteristics of **SysVar.returnCode** are as follows:

Primitive type
BIN

Data length
9

Is value always restored after a converse?
Yes

Example:

```
SysVar.returnCode = 6;
```

Related reference

“System variables outside of EGL libraries” on page 893

sessionID

In Web applications, the system variable **SysVar.sessionID** contains an ID that is specific to the Web application server session. You can use the **SysVar.sessionID** value as a key value to access file or database information shared between programs.

Outside of Web applications, the following statements apply:

- The system variable **SysVar.sessionID** contains a system-dependent user identifier or terminal identifier for your program
- **SysVar.sessionID** is supported for this use only for compatibility with products that preceded EGL (specifically, for CSP releases prior to CSP 370AD Version 4 Release 1). It is recommended that you use **SysVar.userID** or **SysVar.terminalID** instead.

You can use **SysVar.sessionID** in these ways:

- As the source in an assignment or **move** statement
- As a variable in a logical expression
- As the argument in a **return** statement

The characteristics of **SysVar.sessionID** are as follows:

Primitive type
CHAR

Data length
8 (padded with blanks if the value has less than 8 characters)

Is value always restored after a converse?
Yes

SysVar.sessionID is initialized from the Java Virtual Machine system property *user.name*; and if the property cannot be retrieved, **SysVar.sessionID** is blank.

In relation to COBOL code for iSeries, **SysVar.sessionID** is the logon user ID and equivalent to the **SysVar.userID**.

Example:

```
myItem = SysVar.sessionID;
```

Related reference
"System variables outside of EGL libraries" on page 893
"terminalID" on page 913
"userID" on page 914

sqlca

The system variable **SysVar.sqlca** contains the entire SQL communication area (SQLCA). As noted later, the current values of a subset of fields in the SQLCA are available to you after your code accesses a relational database.

You can use **SysVar.sqlca** in these ways:

- As the source or target in an assignment or **move** statement
- In a function invocation, as an argument associated with an in, out, or inOut parameter
- As a variable in a logical expression
- As the argument in an **exit** or **return** statement

In order to refer to specific fields in the SQLCA, you must move **SysVar.sqlca** to a base record. The record must have a structure as specified in the SQLCA description for your database management system. Use the base record if you pass the SQLCA contents to a remote program so that the contents will be converted correctly to the remote system data format.

For specific information about the fields that are available in **SysVar.sqlca**, refer to the following topics:

- VGVar.sqlerrd
- SysVar.sqlcode
- VGVar.sqlerrmc (refreshed by the database management system only in COBOL code; not in Java code or in the EGL Debugger)
- SysVar.sqlState
- VGVar.sqlWarn

The characteristics of **SysVar.sqlca** are as follows:

Primitive type
HEX

Data length
272 (136 bytes)

Is value always restored after a converse?
Only in a non-segmented text program; for details see *Segmentation*

Example:

```
myItem = SysVar.sqlca;
```

Related concepts
"Segmentation in text applications" on page 149
"SQL support" on page 213

Related reference
"System variables outside of EGL libraries" on page 893
"sqlcode"
"sqlState" on page 911
"sqlerrd" on page 923
"sqlerrmc" on page 924
"sqlWarn" on page 925

sqlcode

The system variable **SysVar.sqlcode** contains the return code for the most recently completed SQL I/O operation. The code is obtained from the SQL communications area (SQLCA) and can vary with the relational database manager.

You can use **SysVar.sqlcode** in these ways:

- As the source or target in an assignment or **move** statement (also allowed in the "for count" of a **move** statement)
- In a function invocation, as an argument associated with an in, out, or inOut parameter
- As a variable in a logical expression
- As the argument in an **exit** or **return** statement

The characteristics of **SysVar.sqlcode** are as follows:

Primitive type
BIN

Data length
9

Is value always restored after a converse?
Only in a non-segmented text program; for details see *Segmentation*

Example:

```
rcitem = SysVar.sqlcode;
```

Related concepts
"Segmentation in text applications" on page 149
"SQL support" on page 213

Related reference

"System variables outside of EGL libraries" on page 893

sqlState

The system variable **SysVar.sqlState** contains the SQL state value for the most recently completed SQL I/O operation. The code is obtained from the SQL communications area (SQLCA) and can vary with the relational database manager.

You can use **SysVar.sqlState** in these ways:

- As the source or target in an assignment or **move** statement
- In a function invocation, as an argument associated with an in, out, or inOut parameter
- As a variable in a logical expression
- As the argument in an **exit** or **return** statement

The characteristics of **SysVar.sqlState** are as follows:

Primitive type
CHAR

Data length
5

Is value always restored after a converse?
Only in a non-segmented text program; for details see*Segmentation*

Example:

```
rcitem = SysVar.sqlState;
```

Related concepts
"Segmentation in text applications" on page 149
"SQL support" on page 213

Related reference
"System variables outside of EGL libraries" on page 893

systemType

The system variable **SysVar.systemType** identifies the target system in which the program is running. If the generated output is a Java wrapper, **SysVar.systemType** is not available. Otherwise, the valid values are as follows:

aix For AIX

debug For the EGL Debugger

hp For HP-UX

iseriesj
For iSeries Java programs

iseriesc
For iSeries COBOL programs

linux For Linux (on Intel-based hardware)

solaris For Solaris

win For Windows 2000/NT/XP

You can use **SysVar.systemType** in these ways:

- As the source in an assignment or **move** statement
- As a variable in a logical expression
- As the argument in a **return** statement

The characteristics of **SysVar.systemType** are as follows:

Primitive type
CHAR

Data length
8 (padded with blanks)

Is value always restored after a converse?
Yes

Use **SysVar.systemType** instead of **VGLib.getVAGSysType**.

Definition considerations: The value of **SysVar.systemType** does not affect what code is validated at generation time. For example, the following **add** statement is validated even if you are generating for Windows:

```
if (sysVar.systemType IS AIX)
  add myRecord;
end
```

To avoid validating code that will never run in the target system, take either of the following actions:

- Set the build descriptor option **EliminateSystemDependentCode** to YES. In the current example, the **add** statement is not validated if you set that build descriptor option to YES. Be aware, however, that the generator can eliminate system-dependent code only if the logical expression (in this case, `SysVar.systemType IS AIX`) is simple enough to evaluate at generation time.
- Alternatively, move the statements that you do not want to validate to a second program; then, let the original program call the new program conditionally:

  ```
  if (SysVar.systemType IS AIX)
    call myAddProgram myRecord;
  end
  ```

Example:

```
if (SysVar.systemType is WIN)
  call myAddProgram myRecord;
end
```

Related reference
"eliminateSystemDependentCode" on page 370
"System variables outside of EGL libraries" on page 893
"getVAGSysType()" on page 892

terminalID

In relation to COBOL code on iSeries, **SysVar.terminalID** is initialized to blanks; and if the code is interactive, the variable is reset to the terminal device name received from a query of the attributes of the active job.

In relation to Java code, **SysVar.terminalID** (like **SysVar.sessionID**) is initialized from the Java Virtual Machine system property *user.name*, and if the property cannot be retrieved, **SysVar.terminalID** is blank.

You can use **SysVar.terminalID** in these ways:

- As the source in an assignment or **move** statement
- As a variable in a logical expression
- As the argument in a **return** statement

The characteristics of **SysVar.terminalID** are as follows:

Primitive type
CHAR

Data length
10 for iSeries COBOL, otherwise 8,and padded with blanks if the value has less than the maximum number of characters

Is value always restored after a converse?
Yes

Example:

```
myItem10 = SysVar.terminalID;
```

transactionID

Thevariable is not used; but if the program was invoked by a **transfer** statement of the form *transfer to program*, the variable contains the name of the transferring program.

You can use this variable in any of these ways:

- As the source or destination in an assignment or move statement
- As a comparison value in a logical expression
- As the value in a return statement

Characteristics of **SysVar.transactionID** are as follows:

Primitive type
CHAR

Data length
8

Is value always restored after a converse?
Yes

Related concepts
"Segmentation in text applications" on page 149

Related reference

"System variables outside of EGL libraries" on page 893

transferName

The system variable **SysVar.transferName** allows you to specify, at run time, the name of the program or transaction to which you want to transfer.

You can use this variable in any of these ways:

- As the source or target in an assignment or move statement
- As a program or transaction name in a transfer statement
- As a comparison value in a logical expression
- As the value in a return statement

Characteristics of **SysVar.transferName** are as follows:

Primitive type
CHAR

Data length
8

Is value always restored after a converse?
Yes

Related reference
"System variables outside of EGL libraries" on page 893
"transfer" on page 627

userID

The system variable **SysVar.userID** contains a user identifier in environments where one is available.

You can use **SysVar.userID** in these ways:

- As the source in an assignment or **move** statement
- As a variable in a logical expression
- As the argument in a **return** statement

The characteristics of **SysVar.userID** are as follows:

Primitive type
CHAR

Data length
8 (padded with blanks if the value has less than 8 characters)

Is value always restored after a converse?
Yes

SysVar.userID is initialized from the Java Virtual Machine system property *user.name*; and if the property cannot be retrieved, **SysVar.userID** is blank.

In relation to COBOL code for iSeries, **SysVar.userID** contains the user ID specified at sign-on.

Example:

```
myItem = SysVar.userID;
```

VGVar

The qualifier **VGVar** can precede the name of each EGL system variable listed in the next table. These variables are useful primarily in applications migrated from VisualAge Generator.

System variable	Description
currentFormattedGregorianDate	Contains the current system date in long Gregorian format.
currentFormattedJulianDate	Contains the current system date in long Julian format.
currentFormattedTime	Contains the current system time in HH:mm:ss format.
currentGregorianDate	Contains the current system date in eight-digit Gregorian format (yyyyMMdd).
currentJulianDate	Contains the current system date in seven-digit Julian format (yyyyDDD). Avoid using this variable, which exists to support code migration from VisualAge Generator to EGL.
currentShortGregorianDate	Contains the current system date in six-digit Gregorian format (yyMMdd). Avoid using this variable, which exists to support code migration from VisualAge Generator to EGL.
currentShortJulianDate	Contains the current system date in five-digit Julian format (yyDDD). Avoid using this variable, which exists to support code migration from VisualAge Generator to EGL.
handleHardIOErrors	Controls whether a program continues to run after a hard error occurs on an I/O operation in a try block.
handleOverflow	Controls error processing after an arithmetic overflow.
handleSysLibraryErrors	Specifies whether the value of system variable **SysVar.errorCode** is affected by the invocation of a system function.
mqConditionCode	Contains the completion code from an MQSeries API call following an **add** or **scan** I/O operation for an MQ record.
sqlerrd	Six-element array, where each element contains the corresponding SQL communication area (SQLCA) value that was returned from the last SQL I/O option.
sqlerrmc	Contains the substitution variables for the error message associated with the return code in **SysVar.sqlcode**.

System variable	Description
sqlIsolationLevel	Indicates the level of independence of one database transaction from another, and is meaningful only if you are generating Java output.
sqlWarn	Eleven-element array, where each element contains a warning byte returned in the SQL communications area (SQLCA) for the last SQL I/O operation and where the index is one greater that the warning number in the SQL SQLCA description.

Related concepts
"References to variables in EGL" on page 55
"Scoping rules and "this" in EGL" on page 53

Related reference
"System variables outside of EGL libraries" on page 893

currentFormattedGregorianDate

The system variable **VGVar.currentFormattedGregorianDate** contains the current system date in long Gregorian format. The value is automatically updated each time system variable is referenced by your program.

For COBOL programs, the system administrator for EGL run-time services sets the format at installation.

For Java programs, the format is in this Java run-time property:

```
vgj.datemask.gregorian.long.NLS
```

NLS
The NLS (national language support) code specified in the Java run-time property **vgj.nls.code**. The code is one of those listed for the targetNLS build descriptor option. Uppercase English (code ENP) is not supported.

For additional details on **vgj.nls.code**, see *Java run-time properties (details)*.

The format specified in **vgj.datemask.gregorian.long.*NLS*** includes dd (for numeric day), MM (for numeric month), and yyyy (for numeric year), with characters other than d, M, y, or digits used as separators. You can specify the format in the **dateMask** build descriptor option, and the default format is specific to the locale.

You can use **VGVar.currentFormattedGregorianDate** as the source in an **assignment** or **move** statement or as the argument in a **return** or **exit** statement.

Make sure that this Gregorian long date format is the same as the date format specified for the SQL database manager. Matching the two formats enables **VGVar.currentFormattedGregorianDate** to produce dates in the format expected by the database manager.

The characteristics of **VGVar.currentFormattedGregorianDate** are as follows:

Primitive type
CHAR

Data length
10

Value saved across segments
No

Example:
```
myDate = VGVar.currentFormattedGregorianDate;
```

Related concepts
"Build descriptor part" on page 275
"Java runtime properties" on page 327

Related tasks
"Editing Java run-time properties in a build descriptor" on page 284

Related reference
"EGL library DateTimeLib" on page 768
"Java runtime properties (details)" on page 525
"System variables outside of EGL libraries" on page 893
"targetNLS" on page 389

currentFormattedJulianDate

The system variable **VGVar.currentFormattedJulianDate** contains the current system date in long Julian format. The value is automatically updated each time the system variable is referenced by your program

For COBOL programs, the system administrator for EGL run-time services sets the format at installation.

For Java programs, the format is in this Java run-time property:
```
vgj.datemask.julian.long.NLS
```

NLS
The NLS (national language support) code specified in the Java run-time property **vgj.nls.code**. The code is one of those listed for the targetNLS build descriptor option. Uppercase English (code ENP) is not supported.

For additional details on **vgj.nls.code**, see *Java run-time properties (details)*.

The format specified in **vgj.datemask.julian.long.*NLS*** includes DDD (for numeric day) and yyyy (for numeric year), with characters other than D, y, or digits used as separators. You can specify the format in the **dateMask** build descriptor option, and the default format is specific to the locale.

You can use **VGVar.currentFormattedJulianDate** as the source in an assignment or **move** statement or as the argument in a **return** or **exit** statement.

The characteristics of **VGVar.currentFormattedJulianDate** are as follows:

Primitive type
CHAR

Data length
8

Value saved across segments
No

Example:
```
myDate = VGVar.currentFormattedJulianDate;
```

Related concepts
"Build descriptor part" on page 275
"Java runtime properties" on page 327

Related tasks
"Editing Java run-time properties in a build descriptor" on page 284

Related reference
"EGL library DateTimeLib" on page 768
"Java runtime properties (details)" on page 525
"System variables outside of EGL libraries" on page 893
"targetNLS" on page 389

currentFormattedTime

The system variable **VGVar.currentFormattedTime** contains the current system time in HH:mm:ss format. The value is automatically updated each time it is referenced by your program.

You can use **VGVar.currentFormattedTime** in these ways:

- As the source in an assignment or **move** statement
- As the argument in an **exit** or **return** statement

The characteristics of **VGVar.currentFormattedTime** are as follows:

Primitive type
CHAR

Data length
8

Value saved across segments
No

Example:

```
timeField = VGVar.currentFormattedTime;
```

Related reference
"EGL library DateTimeLib" on page 768
"System variables outside of EGL libraries" on page 893

currentGregorianDate

The system variable **VGVar.currentGregorianDate** contains the current system date in eight-digit Gregorian format (yyyyMMdd).

The **VGVar.currentGregorianDate** value is updated automatically before each reference. The value is numeric and contains no separator characters.

You can use **VGVar.currentGregorianDate** as the source in an **assignment** or **move** statement or as the argument in a **return** or **exit** statement.

The characteristics of **VGVar.currentGregorianDate** are as follows:

Primitive type
DATE

Data length
8

Value saved across segments
No

Example:

```
myDate = VGVar.currentGregorianDate
```

Related reference
"EGL library DateTimeLib" on page 768
"System variables outside of EGL libraries" on page 893

currentJulianDate

The system variable **VGVar.currentJulianDate** contains the current system date in seven-digit Julian format (yyyyDDD). Avoid using this variable, which exists to support code migration from VisualAge Generator to EGL.

The value is numeric, contains no separator characters, and is updated automatically before each reference.

You can use **VGVar.currentJulianDate** as the source in an **assignment** or **move** statement or as the argument in a **return** or **exit** statement.

The characteristics of **VGVar.currentJulianDate** are as follows:

Primitive type
NUM

Data length
7

Value saved across segments
No

Example:

```
myDay = VGVar.currentJulianDate;
```

Related reference
"EGL library DateTimeLib" on page 768
"System variables outside of EGL libraries" on page 893

currentShortGregorianDate

The system variable **VGVar.currentShortGregorianDate** contains the current system date in six-digit Gregorian format (yyMMdd). Avoid using this variable, which exists to support code migration from VisualAge Generator to EGL.

The **VGVar.currentShortGregorianDate** value is automatically updated each time it is referenced by the program. The returned value is numeric and contains no separator characters.

You can use **VGVar.currentShortGregorianDate** as the source in an **assignment** or **move** statement or as the argument in a **return** or **exit** statement.

The characteristics of **VGVar.currentShortGregorianDate** are as follows:

Primitive type
NUM

Data length
6

Value saved across segments
No

Example:

```
myDay = VGVar.currentShortGregorianDate;
```

Related reference
"EGL library DateTimeLib" on page 768
"System variables outside of EGL libraries" on page 893

currentShortJulianDate

The system variable **VGVar.currentShortJulianDate** contains the current system date in five-digit Julian format (yyDDD). Avoid using this variable, which exists to support code migration from VisualAge Generator to EGL.

The value is numeric, contains no separator characters, and is automatically updated each time it is referenced by your program.

You can use **VGVar.currentShortJulianDate** as the source in an **assignment** or **move** statement or as the argument in a **return** or **exit** statement.

The characteristics of **VGVar.currentShortJulianDate** are as follows:

Primitive type
NUM

Data length
5

Value saved across segments
No

Example:

```
myDay = VGVar.currentShortJulianDate;
```

Related reference
"EGL library DateTimeLib" on page 768
"System variables outside of EGL libraries" on page 893

handleHardIOErrors

The system variable **VGVar.handleHardIOErrors** controls whether a program continues to run after a hard error occurs on an I/O operation in a try block. The default value is 1, unless you set the program property **handleHardIOErrors** to *no*, which sets the variable to 0. (That property is also available for other generatable logic parts.) For background information, see *Exception handling*.

You can use **VGVar.handleHardIOErrors** in any of these ways:

- As the source or target of an assignment or **move** statement (also allowed in the "for count" of a **move** statement)
- As the variable in a logical expression used in a **case**, **if**, or **while** statement
- As the argument in a **return** or **exit** statement

The characteristics of **VGVar.handleHardIOErrors** are as follows:

Primitive type
NUM

Data length
1

Is value always restored after a converse?
Yes

Example

```
VGVar.handleHardIOErrors  = 1;
```

Related reference
"Exception handling" on page 89

"System variables outside of EGL libraries" on page 893

handleOverflow

The system variable **VGVar.handleOverflow** controls error processing after an arithmetic overflow. Two types of overflow conditions are detected:

- *User variable overflow* occurs when the result of an arithmetic operation or assignment to a numeric item causes a significant value (not decimal positions) to be lost due to the length of the item.
- *Maximum value overflow* occurs when the result of an arithmetic operation is greater than 18 digits.

You can set **VGVar.handleOverflow** to one of the following values. (The default setting is 0.)

Value	Effect on user overflow	Effect on maximum value overflow
0	The program sets the system variable **SysVar.overflowIndicator** to 1 and continues	The program ends with an error message
1	The program ends with an error message	The program ends with an error message
2	The program sets the system variable **SysVar.overflowIndicator** to 1 and continues	The program sets the system variable **SysVar.overflowIndicator** to 1 and continues

You can use **VGVar.handleOverflow** in these ways:

- As the source or target in an assignment or **move** statement (also allowed in the "for count" of a **move** statement)
- As a variable in a logical expression
- As the argument in an **exit** or **return** statement

The characteristics of **VGVar.handleOverflow** are as follows:

Primitive type
NUM

Data length
1

Is value always restored after a converse?
Yes

Example:

```
VGVar.handleOverflow = 2;
```

Related reference
"Assignments" on page 352

"System variables outside of EGL libraries" on page 893
"overflowIndicator" on page 906

handleSysLibraryErrors

The system variable **VGVar.handleSysLibraryErrors** specifies whether the value of system variable **SysVar.errorCode** is affected by the invocation of a system function. However, **VGVar.handleSysLibraryErrors** is available only when VisualAge Generator compatibility is in effect, as explained in *Compatibility with VisualAge Generator*.

For details and restrictions, see *Exception handling*.

You can use **VGVar.handleSysLibraryErrors** in these ways:

- As the source or target in an assignment or **move** statement
- As a variable in a logical expression
- As the argument in an **exit** or **return** statement

The characteristics of **VGVar.handleSysLibraryErrors** are as follows:

Primitive type
NUM

Data length
1

Is value always restored after a converse?
Only in a non-segmented text program; for details see *Segmentation*

Example:

```
VGVar.handleSysLibraryErrors = 1;
```

Related concepts
"Compatibility with VisualAge Generator" on page 428
"Segmentation in text applications" on page 149

Related reference
"Exception handling" on page 89
"System variables outside of EGL libraries" on page 893
"errorCode" on page 903

mqConditionCode

The system variable **VGVar.mqConditionCode** contains the completion code from an MQSeries API call following an **add** or **scan** I/O operation for an MQ record. Valid values and their related meanings are as follows:

00 OK

01 WARNING

02 FAILED

You can use **VGVar.mqConditionCode** in these ways:

- As the source or target in an assignment or **move** statement (also allowed in the ″for count″ of a **move** statement)
- As a variable in a logical expression
- As the argument in an **exit** or **return** statement

The characteristics of **VGVar.mqConditionCode** are as follows:

Primitive type
NUM

Data length
2

Is value always restored after a converse?
Yes

Example:

```
add MQRecord;
if (VGVar.mqConditionCode == 0)
  // continue
else
  exit program;
end
```

Related concepts
“MQSeries support” on page 247

Related reference
“Exception handling” on page 89
“System variables outside of EGL libraries” on page 893

sqlerrd

The system array **VGVar.sqlerrd** is a 6-element array, where each element contains the corresponding SQL communication area (SQLCA) value that was returned from the last SQL I/O option. The value in VGVar.sqlerrd[3], for example, is the third value and indicates the number of rows processed for some SQL requests.

Of the elements in **VGVar.sqlerrd**, only **VGVar.sqlerrd[3]** is refreshed by the database management system for Java code or at debugging time.

You can use a **VGVar.sqlerrd** element in these ways:

- As the source or target in an assignment or **move** statement
- As the value in the **for** ***count*** clause of a **move** statement
- In a function invocation, as an argument associated with an in, out, or inOut parameter
- As a variable in a logical expression
- As the argument in an **exit** or **return** statement

The characteristics of each element in the **VGVar.sqlerrd** array are as follows:

Primitive type
BIN

Data length
9

Is value always restored after a converse?
Only in a non-segmented text program; for details see *Segmentation*

Example:

```
myItem = VGVar.sqlerrd[3];
```

Related concepts
"Segmentation in text applications" on page 149
"SQL support" on page 213

Related reference

"System variables outside of EGL libraries" on page 893

sqlerrmc

The system variable **VGVar.sqlerrmc** contains the error message associated with the return code in **SysVar.sqlcode**. **VGVar.sqlerrmc** is obtained from the SQL communications area (SQLCA) and can vary with the relational database manager.

VGVar.sqlerrmc has no meaning for the JDBC environment.

You can use **VGVar.sqlerrmc** in these ways:

- As the source or target in an assignment or **move** statement
- As a variable in a logical expression
- In a function invocation, as an argument associated with an in, out, or inOut parameter
- As the argument in an **exit** or **return** statement

The characteristics of **VGVar.sqlerrmc** are as follows

Primitive type
CHAR

Data length
70

Is value always restored after a converse?
Only in a non-segmented text program; for details see *Segmentation*

Definition considerations: **VGVar.sqlerrmc** is defined as a fixed-length text string.

Example:

```
myItem = VGVar.sqlerrmc;
```

Related concepts
"Segmentation in text applications" on page 149
"SQL support" on page 213

Related reference
"sqlca" on page 909
"System variables outside of EGL libraries" on page 893

sqlIsolationLevel

The system variable **VGVar.sqlIsolationLevel** indicates the level of independence of one database transaction from another, and is meaningful only if you are generating Java output.

For an overview of isolation level and of the phrases *repeatable read* and *serializable transaction*, see the JDBC documentation available from Sun Microsystems, Inc.

VGVar.sqlIsolationLevel is for use only in programs migrated from VisualAge Generator and EGL 5.0. The function is supported (at development time) if the EGL preference **VisualAge Generator Compatibility** is selected or (at generation time) if the build descriptor option **VAGCompatibility** is set to *yes*.

For new development, set the SQL isolation level in the **SysLib.connect**.

The following values of **VGVar.sqlIsolationLevel** are in order of increasing strictness:

0 (the default)
Repeatable read

1 Serializable transaction

You can use this variable in any of these ways:

- As the source or target in an assignment or move statement
- In a function invocation, as an argument associated with an in, out, or inOut parameter
- As a comparison value in a logical expression
- As the value in a return statement

Characteristics of **SysVar.transactionID** are as follows:

Primitive type
NUM

Data length
1

Is value always restored after a converse?
Yes

Related reference
"connect()" on page 867
"System variables outside of EGL libraries" on page 893

sqlWarn

The system array **VGVar.sqlWarn** is an 11-element array, where each element contains a warning byte returned in the SQL communications area (SQLCA) for the last SQL I/O operation and where the index is one greater that the warning number in the SQL SQLCA description. The system variable **VGVar.sqlWarn[2]**, for example, refers to SQLWARN1, which indicates whether characters in an item were truncated in the I/O operation.

Of the elements in **VGVar.sqlWarn**, only the system variable **VGVar.sqlWarn[2]** is refreshed by the database management system for Java code or at debugging time.

You can use **VGVar.sqlWarn** in these ways:

- As the source or target in an assignment or **move** statement
- As the value in the **for** ***count*** clause of a **move** statement
- As a variable in a logical expression
- In a function invocation, as an argument associated with an in, out, or inOut parameter

- As the argument in an **exit** or **return** statement

The characteristics of each element in the **VGVar.sqlWarn** array are as follows:

Primitive type
CHAR

Data length
1

Is value always restored after a converse?
Only in a non-segmented text program; for details see *Segmentation*

Definition considerations: **VGVar.sqlWarn[2]** contains *W* if the last SQL I/O operation caused the database manager to truncate character data items because of insufficient space in the program's host variables. You can use logical expressions to test whether the values in specific host variables were truncated. For details, see the references to **trunc** in *Logical expressions*.

When the host variable is a number, no truncation warning is given. Fractional parts of a number are truncated with no indication. When DB2 UDB is used, if the non-fractional part of a number does not fit into a user variable, the database manager returns -304 in **sysVar.sqlcode**.

Also when DB2 is used, **VGVar.sqlWarn[7]** contains *W* if an adjustment was made to correct a result that was not valid from an arithmetic operation on date or time values.

Example: In the following example, *my-char-field* is a field in the SQL row record just processed and *lost-data* is a function that sets an error message indicating that information for *my-char-field* was truncated.

```
if (VGVar.sqlWarn[2] == 'W')
  if (my-char-field is trunc)
    lost-data();
  end
end
```

Related concepts
"Segmentation in text applications" on page 149
"SQL support" on page 213

Related reference

"Logical expressions" on page 484
"System variables outside of EGL libraries" on page 893

transferToProgram element

A *transferToProgram* element of a linkage options part specifies how a generated COBOL program transfers control and ends processing.

The element includes these properties:

- fromPgm
- toPgm
- linkType
- alias (as is necessary if your code is transferring to a program whose run-time name is different from the name of the related program part)

You can avoid specifying a **transferToProgram** element when the target program is generated with VisualAge Generator or (in the absence of an alias) with EGL.

Related concepts
"Linkage options part" on page 291
"Run unit" on page 721

Related tasks
"Adding a linkage options part to an EGL build file" on page 294
"Editing the transfer-related elements of a linkage options part" on page 297

Related reference
"alias in transfer-related linkage elements" on page 929
"fromPgm in transferToProgram element"
"linkType in transferToProgram element"
"toPgm in transfer-related linkage elements" on page 928

fromPgm in transferToProgram element

The linkage options part, transferToProgram element, property **fromPgm** specifies the name of a program part:

- If the target system is CICS for z/OS, the program part is the one that issues the transfer statement.
- If the target system is z/OS but not CICS and if any of the programs in the run unit issue a transfer statement, the property **fromPgm** is assigned the name of the first program in the run unit. For an example, see *transferToProgram element*.

The value of the **fromPgm** property is required and cannot include an asterisk (*).

Related concepts
"Linkage options part" on page 291
"Run unit" on page 721

Related tasks
"Adding a linkage options part to an EGL build file" on page 294
"Editing the transfer-related elements of a linkage options part" on page 297

Related reference
"toPgm in transfer-related linkage elements" on page 928
"transfer" on page 627
"transferToProgram element" on page 926
"linkType in transferToProgram element"

linkType in transferToProgram element

The linkage options part, transferToProgram element, property **linkType** specifies the type of linkage to generate in relation to a transfer statement of type *transfer to program*. Valid values are as follows:

Dynamic (the default)
In programs that run on CICS for z/OS, an XCTL implements the transfer statement. In programs that run on z/OS outside of CICS, a dynamic COBOL call is generated in the first program in the run unit, and the EGL run-time handles processing so that the transfer simulates the behavior of a CICS-based program.

The target program is assumed to be produced by EGL or by VisualAge Generator.

Static
In programs that run on CICS for z/OS, an XCTL implements the transfer statement. In programs that run on z/OS outside of CICS, the following statements apply:

- A static COBOL call is generated
- The EGL run-time handles processing so that the transfer simulates the behavior of a CICS-based program
- The value *Static* is required for target programs that call PL/I programs or that call programs that call PL/I programs.

The target program is assumed to be produced by EGL or by VisualAge Generator.

ExternallyDefined
Specify the value *ExternallyDefined* if you are transferring to a program that was not produced by EGL or VisualAge Generator. In all COBOL target systems, an XCTL implements the transfer statement.

If the program property **VAGCompatibility** is set to *yes*, you can specify ExternallyDefined in the transfer statement, as noted in *Compatibility with VisualAge Generator*. It is recommended that the value be specified in the transferToProgram element instead, but the value is in effect if specified in either place.

Related concepts
"Compatibility with VisualAge Generator" on page 428
"Linkage options part" on page 291

Related tasks
"Adding a linkage options part to an EGL build file" on page 294
"Editing the transfer-related elements of a linkage options part" on page 297

Related reference
"fromPgm in transferToProgram element" on page 927
"toPgm in transfer-related linkage elements"
"transfer" on page 627
"transferToProgram element" on page 926

toPgm in transfer-related linkage elements

In the transfer-related elements of the linkage options part, the required property **toPgm** specifies the name of the program part (or of the non-EGL program) that receives control.

If the function word sysVar.transferName is specified as the target in a transfer statement, do not specify that system variable in the related **toPgm** property. Instead, specify the program name that will be in sysVar.transferName when the program runs.

Related concepts
"Linkage options part" on page 291

Related tasks
"Adding a linkage options part to an EGL build file" on page 294
"Editing the transfer-related elements of a linkage options part" on page 297

Related reference
"transferName" on page 914
"transfer" on page 627

transferToTransaction element

A *transferToTransaction* element of a linkage options part specifies how a generated program transfers control to a transaction and ends processing. The element includes the property toPgm and may include these properties:

- alias, as is necessary if your code is transferring to a program whose run-time name is different from the name of the related program part
- externallyDefined, as is necessary if your code is transferring to a program that was not generated with EGL or VisualAge Generator

You can avoid specifying a **transferToTransaction** element when the target program is generated with VisualAge Generator or (in the absence of an alias) with EGL.

Related concepts
"Linkage options part" on page 291

Related tasks
"Adding a linkage options part to an EGL build file" on page 294
"Editing the transfer-related elements of a linkage options part" on page 297

Related reference
"alias in transfer-related linkage elements"
"externallyDefined in transferToTransaction element" on page 930
"fromPgm in transferToProgram element" on page 927
"linkType in transferToProgram element" on page 927
"toPgm in transfer-related linkage elements" on page 928

alias in transfer-related linkage elements

In the transfer-related elements of the linkage options part, the property **alias** specifies the run-time name of the program that is identified in property **toPgm**.

The value of this property must match the alias (if any) you specified when declaring the program to which you are transferring. If you did not specify an alias when declaring that program, either set the property **alias** to the name of the program part or do not set the property at all.

Related concepts
"Linkage options part" on page 291

Related tasks
"Adding a linkage options part to an EGL build file" on page 294
"Editing the transfer-related elements of a linkage options part" on page 297

Related reference
"transferToProgram element" on page 926

"transferToProgram element" on page 926
"toPgm in transfer-related linkage elements" on page 928

externallyDefined in transferToTransaction element

The linkage options part, transferToTransaction element, property **externallyDefined** indicates whether you are transferring to a program that was produced by software other than EGL or VisualAge Generator. Valid values are **no** (the default) and **yes**.

If you specify **yes**, an XCTL implements the transfer statement in all COBOL target systems.

If the program property **VAGCompatibility** is set to *yes*, you can specify externallyDefined in the transfer statement, as noted in *Compatibility with VisualAge Generator*. It is recommended that the value be specified in the transferToTransaction element instead, but the value is in effect if specified in either place.

Related concepts
"Compatibility with VisualAge Generator" on page 428

Related tasks
"Adding a linkage options part to an EGL build file" on page 294
"Editing the transfer-related elements of a linkage options part" on page 297

Related reference
"transfer" on page 627

Use declaration

This section describes the use declaration, followed by details on how to write the declaration:

- "In a program or library part" on page 931
- "In a formGroup part" on page 933
- "In a pageHandler part" on page 934

Background

The use declaration allows you to easily reference data areas and functions in parts that are separately generated. A program, for instance, can issue a use declaration that allows for easy reference to a data table, library, or form group, but only if those parts are visible to the program part. For details on visibility, see *References to parts*.

In most cases, you can reference data areas and functions from another part regardless of whether a use declaration is in effect. For example, if you are writing a program and do not have a use declaration for a library part called `myLib`, you can access the library variable called `myVar` as follows:

```
myLib.myVar
```

If you include the library name in a use declaration, however, you can reference the variable as follows:

```
myVar
```

The previous, short form of the reference is valid only if the symbol `myVar` is unique for every variable and structure item that is global to the program. (If the symbol is not unique, an error occurs.) Also, the symbol `myVar` refers to an item in the library only if a local variable or parameter does not have the same name. (A local data area takes precedence over a same-named, program-global data area.)

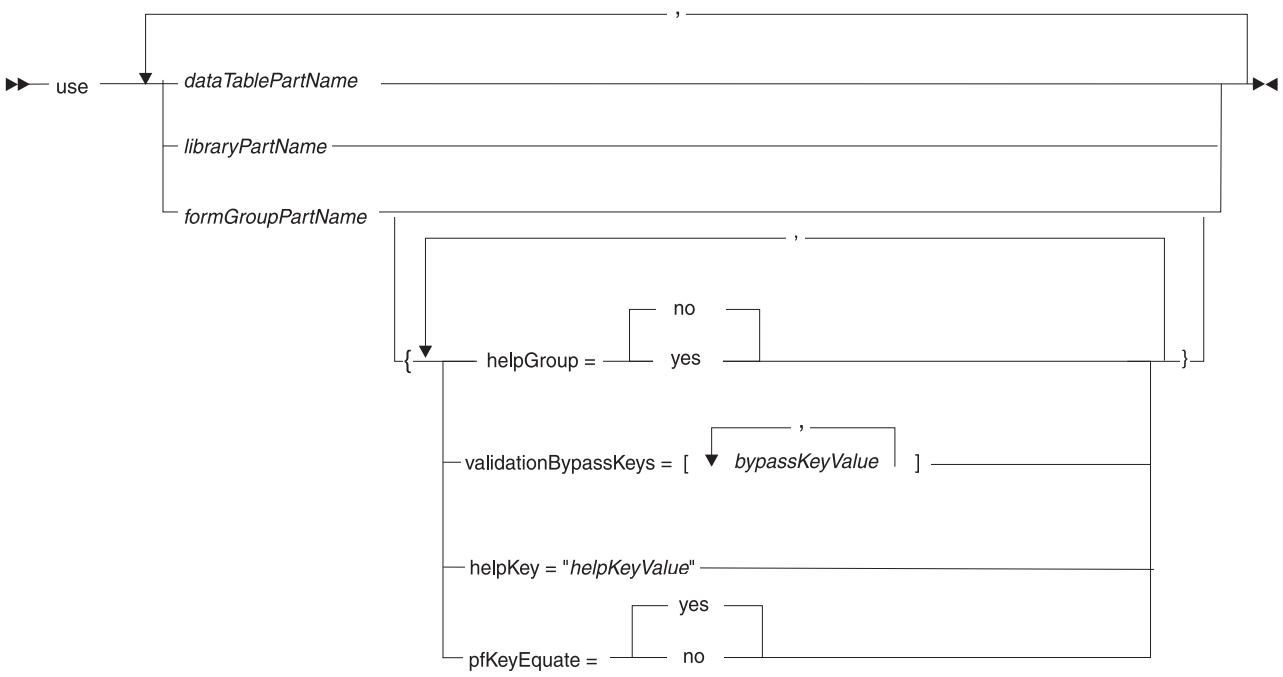
A use declaration is required in these situations:

- A program or library that uses any of the forms in a given formGroup part must have a use declaration for that formGroup part
- A formGroup part must have a use declaration for a form that is required by the program or library but is not embedded in the formGroup part
- If you have declared a function at the top level of an EGL source file rather than physically inside a container (a program, PageHandler, or library), that function can invoke library functions only if the following situation is in effect:
  - The container includes a use statement that refers to the library
  - In the invoking function, the property **containerContextDependent** is set to *yes*

Each name specified in the use declaration may be qualified by a package name, library name, or both.

In a program or library part

Each use declaration in a program or library must be external to any function. The syntax for the declaration is as follows:



dataTablePartName
Name of a dataTable part that is visible to the program or library.

A reference in a use declaration is unnecessary for a dataTable part that is referenced in the program property **msgTablePrefix**.

You cannot override properties of a dataTable part in the use declaration.

For an overview of dataTable parts, see *DataTable part*.

libraryPartName
Name of a library part that is visible to the program or library.

You cannot override properties of the library part in the use declaration.

For an overview of library parts, see *Library part of type basicLibrary* and *Library part of type nativeLibrary*.

formGroupPartName
Name of a formGroup part that is visible to the program or library. For an overview of form groups, see*FormGroup part*.

A program that uses any of the forms in a given formGroup part must have a use declaration for that formGroup part.

No overrides occur for form-level properties. If a property like **validationBypassKeys** is specified in a form, for example, the value in the form is in effect at run time. If a form-level property is not specified in the form, however, the situation is as follows:

- EGL run time uses the value in the program's use declaration
- If no value is specified in the program's use declaration, EGL run time uses the value (if any) in the form group

The properties that follow let you change behaviors when a form group is accessed by a specific program.

helpGroup = no, helpGroup = yes
Specifies whether to use the formGroup part as a help group. The default is *no*.

validationBypassKeys = [*bypassKeyValue***]**
Identifies a user keystroke that causes the EGL run time to skip input-field validations. This property is useful for reserving a keystroke that ends the program quickly. Each *bypassKeyValue* option is as follows:

pf*n*
The name of an F or PF key, including a number between 1 and 24, inclusive.

Note: Function keys on a PC keyboard are often *f* keys such as f1, but EGL uses the IBM *pf* terminology so that (for example) f1 is called pf1.

If you specify multiple keys, separate one from the next with a comma.

helpKey = *"helpKeyValue"*
Identifies a user keystroke that causes the EGL run time to present a help form to the user. The *helpKeyValue* option is as follows:

pf*n*
The name of an F or PF key, including a number between 1 and 24, inclusive.

Note: Function keys on a PC keyboard are often *f* keys such as f1, but EGL uses the IBM *pf* terminology so that (for example) f1 is called pf1.

pfKeyEquate = yes, pfKeyEquate = no
Specifies whether the keystoke that is registered when the user presses a high-numbered function key (PF13 through PF24) is the same as the keystroke that is registered when the user presses a function key that is lower by 12. The default is *yes*. For details, see *pfKeyEquate*.

In a formGroup part

In a formGroup part, a use declaration refers to a form that is specified outside the form group. This kind of declaration allows multiple form groups to share the same form.

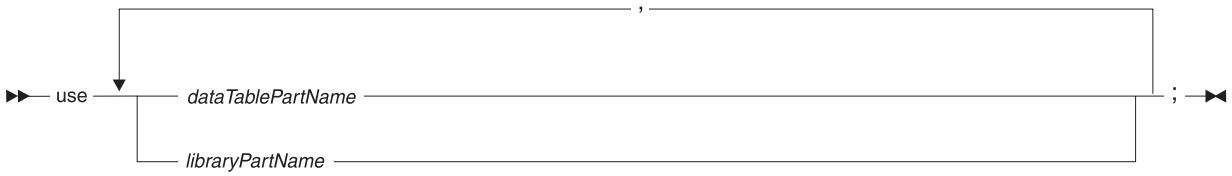
The syntax for a use declaration in a formGroup part is as follows:

formPartName
Name of a form part that is visible to the form group. For an overview of forms, see *Form part*.

You cannot override properties of a form part in the use declaration of a formGroup part.

In a pageHandler part

Each use declaration in a pageHandler part must be external to any function. The syntax for the declaration is as follows:



dataTablePartName
Name of a dataTable part that is visible to the pageHandler part.

You cannot override properties of a dataTable part in the use declaration.

For an overview of dataTable parts, see *DataTable part*.

libraryPartName
Name of a library part that is visible to the pageHandler part.

You cannot override properties of the library part in the use declaration.

For an overview of library parts, see *Library part*.

Related concepts
"DataTable" on page 137
"FormGroup part" on page 143
"Form part" on page 144
"Library part of type basicLibrary" on page 133
"Library part of type basicLibrary" on page 133
"References to parts" on page 20

Related reference
"pfKeyEquate" on page 666

EGL Java runtime error codes

When an error occurs at Java run time, EGL places an error code in the system variable sysVar.errorCode and in most cases presents a message that has the same identifier as the error code. You can cause a customized message to be displayed in place of the EGL message; for details, see *Message customization for EGL Java run time*.

The error situations are as follows:

- A failure occurs during a remote call, an EJB call, a commit, or a rollback. In those cases, the message identifier begins with CSO.
- An error occurs in a Web application. In a subset of those cases, the message identifier begins with EGL.
- An error occurs during a local call, during access of a file or database, or during execution of one the following system functions--
  - Math functions
  - String functions
  - sysLib.convert

  In those cases, the message identifier begins with VGJ.
- An error occurs in a Java access function. In that case, the error code includes only numbers, and no message is displayed.

The error codes that are assigned by the Java access functions are shown in the next table. The other error codes are shown in the next sections.

Value in sysVar.errorCode	Description
00001000	An exception was thrown by an invoked method or as a result of a class initialization.
00001001	The object was null, or the specified identifier was not in the object space.
00001002	A public method, field, or class with the specified name does not exists or cannot be loaded.
00001003	The EGL primitive type does not match the type expected in Java.
00001004	The method returned null, the method does not return a value, or the value of a field was null.
00001005	The returned value does not match the type of the return item.
00001006	The class of an argument cast to null could not be loaded.
00001007	A SecurityException or IllegalAccessException was thrown during an attempt to get information about a method or field, or an attempt was made to set the value of a field that was declared final.
00001008	The constructor cannot be called; the class name refers to an interface or abstract class.
00001009	An identifier rather than a class name must be specified; the method or field is not static.

Related reference
"I/O error values" on page 522

"Message customization for EGL Java run time" on page 641
"errorCode" on page 903

EGL Java run-time error code CSO7000E

CSO7000E: An entry for the specified called program %1 cannot be found in the linkage properties file %2.

Explanation

The message occurs in this situation:

- When the calling program was generated, property **remoteBind** was set to RUNTIME in the linkage options part, in the **callLink** element for the called program; and
- An entry for the specified called program cannot be found at run time, in the linkage properties file. The reason may be one of the following:
  - The linkage properties file cannot be found.
  - The file was found, but an entry for the called program is not in that file.
  - An incorrect linkage properties file was specified.

User Response

Do as follows:

- If the program is being called from a Java wrapper, the linkage properties file must be named *link*.properties, where *link* is the name of the linkage options part used at generation. Make sure the file exists, has an entry for the called program, and is in a directory or archive specified in the CLASSPATH variable.
- If the program is being called from a program running in the J2EE environment, the linkage properties file can be identified by the cso.linkageOptions.*link* environment variable in the deployment descriptor, where *link* is the name of the linkage options part used at generation. If the environment variable is not set, the linkage properties file must be named *link*.properties, where *link* is the name of the linkage options part used at generation. Make sure the file exists, has an entry for the called program, and is in a directory or archive specified in CLASSPATH.
- If the program is being called from a program not running in the J2EE environment, the situation is as follows:
  - The linkage properties file can be identified by the cso.linkageOptions.*link* property, where *link* is the name of the linkage options part used at generation. If the property is not set, the linkage properties file may be named *link*.properties, where *link* is the name of the linkage options part used at generation. In these two cases, make sure the file exists, has an entry for the called program, and is in a directory or archive specified in CLASSPATH.
  - If the linkage properties file cannot be found, the linkage properties must be in the program properties file; in that case, make sure the program properties file includes an entry for the called program and that the program properties file is in a directory or archive specified in CLASSPATH.

For other details, see the EGL help pages on the **callLink** element, on Java run-time properties, and on setting up the environment.

If the problem persists, do as follows:

1. Record the message number and the message text.

   Note: The error message includes the following important information:
   - Where the error occurred
   - The type of internal error
2. Record the situation in which this message occurs.
3. For further instructions on how to report possible defects to the IBM Support Center, refer to the *EGL Installation Guide*.

EGL Java run-time error code CSO7015E

CSO7015E: Cannot open the linkage properties file %1.

Explanation

The linkage properties file cannot be opened because the file is locked or cannot be found.

User Response

Make sure that the linkage properties file is not locked by another process and that the file resides in a directory or archive specified in your CLASSPATH.

EGL Java run-time error code CSO7016E

CSO7016E: The properties file csouidpwd.properties ould not be read. Error: %1

Explanation

The file was found but there was an error reading from it.

User Response

Use the Error portion of the message to diagnose and correct the problem.

EGL Java run-time error code CSO7020E

CSO7020E: The conversion table %1 is not valid.

Explanation

A conversion table that handles bidirectional text is invalid or cannot be loaded.

User Response

The conversion table must reside in a directory or archive specified in the CLASSPATH. For details on developing the conversion table, see the help page on bidirectional text.

EGL Java run-time error code CSO7021E

CSO7021E: The client text attribute tag %2 in conversion table %1 is not valid.

Explanation

The conversion table file is not valid.

User Response

Correct the file and run the program again.

EGL Java run-time error code CSO7022E

CSO7022E: The server text attribute tag %2 in conversion table %1 is not valid.

Explanation

The conversion table file is not valid.

User Response

Correct the file and run the program again.

EGL Java run-time error code CSO7023E

CSO7023E: The value %3 for Arabic option tag %2 in conversion table %1 is not valid.

Explanation

The conversion table file is not valid.

User Response

Correct the file and run the program again.

EGL Java run-time error code CSO7024E

CSO7024E: The value %3 for Wordbreak option tag %2 in conversion table %1 is not valid.

Explanation

The conversion table file is not valid.

User Response

Correct the file and run the program again.

EGL Java run-time error code CSO7026E

CSO7026E: The value %3 for Roundtrip option tag %2 in conversion table %1 is not valid.

Explanation

The conversion table file is not valid.

User Response

Correct the file and run the program again.

EGL Java run-time error code CSO7045E

CSO7045E: Error obtaining the address of entry point %1 within the shared library %2. RC = %3.

Explanation

An error was encountered in obtaining the address of the entry point within the shared library.

User Response

Make sure that the referenced shared library is the correct shared library to be loaded. If so, make sure the shared library is built correctly.

EGL Java run-time error code CSO7050E

CSO7050E: An error occurred in remote program %1, date %2, time %3

Explanation

An error occurred in a called program, and the program stopped running.

User Response

Use the date and time stamp on this message to associate the message with any diagnostic messages logged at the remote location. Check those diagnostic messages for further details.

EGL Java run-time error code CSO7060E

CSO7060E: An error was encountered while loading the shared library %1. The return code is %2.

Explanation

An error was encountered while loading the shared library.

User Response

Make sure the shared library resides in a directory specified in your PATH or LIBPATH environment variable. Make sure that the shared library is built correctly.

EGL Java run-time error code CSO7080E

CSO7080E: The specified protocol %1 is not valid.

Explanation

The specified protocol in the linkage is unrecognized.

User Response

Consult the documentation and specify a valid protocol.

EGL Java run-time error code CSO7160E

CSO7160E: An error occurred in remote program %1, date %2, time %3, on system %4.

Explanation

The Java program that you are running calls a remote program on the specified system, which failed in execution at the date and time specified.

User Response

Check the remote server log for a more detailed description in problem analysis.

EGL Java run-time error code CSO7161E

CSO7161E: Run unit ended due to an application error on system %1 trying to call program %2. %3

Explanation

An error occurred at the remote server that causes the remote run unit to terminate abnormally when executing the remote program. Diagnostic messages preceding this message in the server job log explain the nature of error. If available, additional information may be included with the message text.

User Response

Check the error messages logged on the server system to determine what to do to fix the original problem.

EGL Java run-time error code CSO7162E

CSO7162E: Invalid password or user ID supplied for connecting to system %1. Java exception message received: %2.

Explanation

The password or user ID supplied to connect to the remote system is not set or not valid.

User Response

Verify that the connection is set. Verify that the user ID and password supplied to the remote system are correct, and try again.

EGL Java run-time error code CSO7163E

CSO7163E: Remote access security error to system %1 for user %2. Java exception message received: %3

Explanation

The specified user currently connecting to the system does not have sufficient authority or does not have access to the remote resource on the specified system.

User Response

Verify that the user connecting to the remote machine has the proper authority to connect to the remote machine and to execute the remote server program.

EGL Java run-time error code CSO7164E

CSO7164E: Remote connection error to system %1. Java exception message received: %2

Explanation

An error occurred when communicating or connecting to the remote system.

User Response

Check that the remote server is available; then retry. If this does not work, contact the remote host's system administrator to determine the actual problem.

EGL Java run-time error code CSO7165E

CSO7165E: Commit failed on system %1. %2

Explanation

A commit operation failed on the remote system.

User Response

Diagnose the problem by reviewing the detailed message, which is shown here as %2.

EGL Java run-time error code CSO7166E

CSO7166E: Rollback failed on system %1. %2

Explanation

A rollback operation failed on the remote system.

User Response

Diagnose the problem by reviewing the detailed message, which is shown here as %2.

EGL Java run-time error code CSO7360E

CSO7360E: AS400Toolbox execution error: %1, %2 while calling program %3 on system %4

Explanation

The Java program or applet that you are running uses the Java400 protocol to call a remote server program. An unexpected exception was caught while attempting to call the server program. The message text consists of the name of the AS400 Toolbox exception followed by the message returned with the exception.

User Response

Use the AS400 Toolbox error message provided to analyze the cause of the problem.

EGL Java run-time error code CSO7361E

CSO7361E: EGL OS/400® Host Services error. Required files not found on system %1.

Explanation

The Java program or applet that you are running uses the Java400 protocol to call a remote server program. An exception is raised when the remote catcher is not found or is not in the proper library on the server.

User Response

Check that EGL OS/400 Host Services is properly installed on the remote system. Apply the latest PTFs if available.

EGL Java run-time error code CSO7488E

CSO7488E: Unknown TCP/IP hostname: %1

Explanation

An UnknownHostException was thrown during an attempt to connect to the remote TCP/IP listener program.

User Response

Do as follows:

- Add the property cso.serverLinkage.*xxx*.location to the run-time linkage properties file, where *xxx* is the name of the called program or is an application name, as described in the EGL reference-type help page on the linkage properties file. The value of the property is a valid TCP/IP host name.
- Alternatively, set the TCP/IP host name at generation time and regenerate the program:
  - In the linkage options part, in the callLink element for the called program, set property **location** to the TCP/IP host name
  - If you wish to finalize linkage options only at run time, set property **remoteBind** to RUNTIME and generate with build descriptor option **genProperties** set to YES

For other details, see the EGL help pages on the callLink element, on the linkage properties file, and on setting up the environment.

EGL Java run-time error code CSO7489E

CSO7489E: The linkage information used to call the program is inconsistent or missing.

Explanation

The program was unable to determine how the program should be called.

User Response

Supply all required linkage information. The information that is required depends on the desired type of call. Refer to the help pages on the linkage options part, particularly on the callLink element.

EGL Java run-time error code CSO7610E

CSO7610E: An error was encountered while calling CICS ECI to commit a unit of work. The CICS return code is %1.

Explanation

A commit request was issued by the client but was not successful. An error was encountered while calling CICS External Call Interface to commit a logical unit of work.

User Response

Please refer to the appropriate CICS documentation for the corrective actions for the specified error.

EGL Java run-time error code CSO7620E

CSO7620E: An error was encountered while calling the CICS ECI to rollback a unit of work. The CICS return code is %1.

Explanation

A rollback request was issued by the client but was not successful. An error was encountered while calling CICS External Call Interface to rollback a logical unit of work.

User Response

Please refer to the appropriate CICS documentation for the corrective actions for the specified error.

EGL Java run-time error code CSO7630E

CSO7630E: An error was encountered while ending the remote procedure call to a CICS server. The CICS return code is %1.

Explanation

An attempt was made to commit all open logical units of work before ending the EGL remote procedure call to a CICS server but was not successful. This request

was made via the CICS External Call Interface.

User Response

Refer to the appropriate CICS documentation for the corrective actions for the specified error.

EGL Java run-time error code CSO7640E

CSO7640E: %1 is an invalid value for the ctgport entry.

Explanation

The value of ctgport must be an integer.

User Response

Use the correct ctgport number.

EGL Java run-time error code CSO7650E

CSO7650E: An error was encountered calling program %1 using the CICS ECI. Return code: %2. CICS system identifier: %3.

Explanation

An error was returned from a CICS External Call Interface (ECI) function call when attempting to call a remote server program.

The system identifier is the name of the CICS system where the server program was to run. If blank, the system is specified in the CICS program definition for the program or in the CICS client initialization file. The return code is the CICS return code.

User Response

Correct the problem indicated by the return code.

For a complete explanation of the return code or if the return code is not documented above, refer to the CICS ECI documentation for your system for information on the corrective actions.

Return code values are associated with symbols in the CICS ECI include files faaecih.h or cics_eci.h.

EGL Java run-time error code CSO7651E

CSO7651E: An error was encountered calling program %1 using the CICS ECI. Return code: -3 (ECI_ERR_NO_CICS). CICS system identifier: %2.

Explanation

An error was returned from a CICS External Call Interface (ECI) function call when attempting to call a remote server program.

The system identifier is the name of the CICS system where the server program was to run. If blank, the system is specified in the CICS program definition for the program or in the CICS client initialization file. The return code is the CICS return code.

The CICS return code has the following meaning:

- -3 - ECI_ERR_NO_CICS

  Client or server system not available

User Response

Correct the problem indicated by the return code.

For a complete explanation of the return code or if the return code is not documented above, refer to the CICS ECI documentation for your system for information on the corrective actions.

Return code values are associated with symbols in the CICS ECI include files faaecih.h or cics_eci.h.

EGL Java run-time error code CSO7652E

CSO7652E: An error was encountered calling program %1 using the CICS ECI. Return code: -4 (ECI_ERR_CICS_DIED). CICS system identifier: %2.

Explanation

An error was returned from a CICS External Call Interface (ECI) function call when attempting to call a remote server program.

The system identifier is the name of the CICS system where the server program was to run. If blank, the system is specified in the CICS program definition for the program or in the CICS client initialization file. The return code is the CICS return code.

The CICS return code has the following meaning:

- -4 - ECI_ERR_CICS_DIED

  Server system no longer available

User Response

Correct the problem indicated by the return code.

For a complete explanation of the return code or if the return code is not documented above, refer to the CICS ECI documentation for your system for information on the corrective actions.

Return code values are associated with symbols in the CICS ECI include files faaecih.h or cics_eci.h.

EGL Java run-time error code CSO7653E

CSO7653E: An error was encountered calling program %1 using the CICS ECI. Return code: -6 (ECI_ERR_RESPONSE_TIMEOUT). CICS system identifier: %2.

Explanation

An error was returned from a CICS External Call Interface (ECI) function call when attempting to call a remote server program.

The system identifier is the name of the CICS system where the server program was to run. If blank, the system is specified in the CICS program definition for the program or in the CICS client initialization file. The return code is the CICS return code.

The CICS return code has the following meaning:

- -6 - ECI_ERR_RESPONSE_TIMEOUT

  Response time out. Time limit is specified in environment variable CSOTIMEOUT.

User Response

Correct the problem indicated by the return code.

For a complete explanation of the return code or if the return code is not documented above, refer to the CICS ECI documentation for your system for information on the corrective actions.

Return code values are associated with symbols in the CICS ECI include files faaecih.h or cics_eci.h.

EGL Java run-time error code CSO7654E

CSO7654E: An error was encountered calling program %1 using the CICS ECI. Return code: -7 (ECI_ERR_TRANSACTION_ABEND). CICS system identifier: %2. Abend code: %3.

Explanation

An error was returned from a CICS External Call Interface (ECI) function call when attempting to call a remote server program.

The system identifier is the name of the CICS system where the server program was to run. If blank, the system is specified in the CICS program definition for the program or in the CICS client initialization file. The return code is the CICS return code.

The CICS return code has the following meaning:

- -7 - ECI_ERR_TRANSACTION_ABEND

  Abnormal termination on server. Common ABEND codes are:

  – AEI0 - Server program not defined

  – AEI1 - Server transaction not defined

User Response

Correct the problem indicated by the return code.

For a complete explanation of the return code or if the return code is not documented above, refer to the CICS ECI documentation for your system for information on the corrective actions.

Return code values are associated with symbols in the CICS ECI include files faaecih.h or cics_eci.h.

EGL Java run-time error code CSO7655E

CSO7655E: An error was encountered calling program %1 using the CICS ECI. Return code: -22 (ECI_ERR_UNKNOWN_SERVER). CICS system identifier: %2.

Explanation

An error was returned from a CICS External Call Interface (ECI) function call when attempting to call a remote server program.

The system identifier is the name of the CICS system where the server program was to run. If blank, the system is specified in the CICS program definition for the program or in the CICS client initialization file. The return code is the CICS return code.

The CICS return code has the following meaning:

- -22 - ECI_ERR_UNKNOWN_SERVER

  Server system not defined

User Response

Correct the problem indicated by the return code.

For a complete explanation of the return code or if the return code is not documented above, refer to the CICS ECI documentation for your system for information on the corrective actions.

Return code values are associated with symbols in the CICS ECI include files faaecih.h or cics_eci.h.

EGL Java run-time error code CSO7656E

CSO7656E: An error was encountered calling program %1 using the CICS ECI. Return code: -27 (ECI_ERR_SECURITY_ERROR). CICS system identifier: %2.

Explanation

An error was returned from a CICS External Call Interface (ECI) function call when attempting to call a remote server program.

The system identifier is the name of the CICS system where the server program was to run. If blank, the system is specified in the CICS program definition for the program or in the CICS client initialization file. The return code is the CICS return code.

The CICS return code has the following meaning:

- -27 - ECI_ERR_SECURITY_ERROR

  User ID or password not valid

User Response

Correct the problem indicated by the return code.

For a complete explanation of the return code or if the return code is not documented above, refer to the CICS ECI documentation for your system for information on the corrective actions.

Return code values are associated with symbols in the CICS ECI include files faaecih.h or cics_eci.h.

EGL Java run-time error code CSO7657E

CSO7657E: An error was encountered calling program %1 using the CICS ECI. Return code: -28 (ECI_ERR_MAX_SYSTEMS). CICS system identifier: %2.

Explanation

An error was returned from a CICS External Call Interface (ECI) function call when attempting to call a remote server program.

The system identifier is the name of the CICS system where the server program was to run. If blank, the system is specified in the CICS program definition for the program or in the CICS client initialization file. The return code is the CICS return code.

The CICS return code has the following meaning:

- -28 - ECI_ERR_MAX_SYSTEMS
  Maximum number of servers reached

User Response

Correct the problem indicated by the return code.

For a complete explanation of the return code or if the return code is not documented above, refer to the CICS ECI documentation for your system for information on the corrective actions.

Return code values are associated with symbols in the CICS ECI include files faaecih.h or cics_eci.h.

EGL Java run-time error code CSO7658E

CSO7658E: An error was encountered calling program %1 on system %2 for user %3. CICS ECI call returned RC %4 and Abend Code %5.

Explanation

A non-zero return code was returned on a CICS ECI call made from the gateway to the specified system on behalf of the user identified in the message.

User Response

Correct the problem indicated by the return code.

For a complete explanation of the return code, refer to the CICS ECI documentation for your system for information on the corrective actions.

Return code values are associated with symbols in the CICS ECI include files faaecih.h or cics_eci.h.

EGL Java run-time error code CSO7659E

CSO7659E: An exception occurred on the flow of an ECI Request to CICS system %1. Exception: %2

Explanation

An unexpected exception occurred in the flow method when attempting to send the ECI Request from the gateway to the CICS system identified in the message.

User Response

Examine the exception string that was returned. If you are unable to determine the cause of the problem from the exception, please contact IBM Support for assistance.

EGL Java run-time error code CSO7669E

CSO7669E: An error was encountered when connecting to CTG. CTG Location: %1, CTG Port: %2. Exception: %3

Explanation

An unexpected exception occurred when connecting to the CICS Transaction Gateway.

User Response

Examine the exception string that was returned. If you are unable to determine the cause of the problem from the exception, please contact IBM Support for assistance.

EGL Java run-time error code CSO7670E

CSO7670E: An error was encountered when disconnecting from CTG. CTG Location: %1, CTG Port: %2. Exception: %3

Explanation

An unexpected exception occurred when disconnecting from the CICS Transaction Gateway.

User Response

Examine the exception string that was returned. If you are unable to determine the cause of the problem from the exception, please contact IBM Support for assistance.

EGL Java run-time error code CSO7671E

CSO7671E: When using CICSSSL protocol, both ctgKeyStore and ctgKeyStorePassword must be specified.

Explanation

Required values were not specified so the call cannot be completed.

User Response

Make sure that both ctgKeyStore and ctgKeyStorePassword are specified.

EGL Java run-time error code CSO7816E

CSO7816E: A socket exception occurred when the gateway attempted to connect to server with hostname %1 and port %2 for userid %4. Exception was: %3

Explanation

The socket call to create and connect a socket from the gateway to the server system identified in the message failed with the exception shown.

The EGL gateway attempted a socket call to create and connect a TCP/IP socket for a server call. The socket call failed with the exception indicated in the message.

User Response

Examine the exception information to determine a reason why a socket call from the gateway failed. If you are unable to determine the cause of the problem by examining the exception information, please contact IBM Support for assistance.

EGL Java run-time error code CSO7819E

CSO7819E: An unexpected exception occurred on function %2. Exception: %1

Explanation

The EGL gateway received an unexpected exception from the function identified in the message. An internal error may have occurred.

User Response

If you are unable to determine the source of the problem from examining the exception information, please contact IBM Support for assistance.

EGL Java run-time error code CSO7831E

CSO7831E: The client's buffer was too small for the amount of data being passed on the call. Ensure that the cumulative size of the parameters being passed does not exceed the maximum allowed which is 32567 bytes.

Explanation

The buffer established by the client cannot be made as large as the cumulative size of the parameters being passed to the remote called program.

User Response

Ensure that the cumulative size of the parameters being passed does not exceed the maximum allowed which is 32567 bytes. If they do not exceed the maximum and this error occurs, please report the error to IBM Support Center.

EGL Java run-time error code CSO7836E

CSO7836E: The client has received notification that the server is unable to start the remote called program. Reason code: %1.

Explanation

The server is unable to run the remote called program and has returned a reason code for problem determination.

User Response

Reason codes are as follows:

- 2 - Server was unable to load the class for the called program. The server trace file may show more specific information. Make sure that the class is available to the server.

  This problem may result from improper conversion of the class name passed to the server. Review the help page on data conversion to verify that the correct conversion table was specified in the linkage options part, in the callLink element for the called program, in property conversionTable.
- 3 - The called program was ended because of an error. The server trace file may show more specific information.

For any reason code not listed above or if you are unable to determine the cause of the failure, contact IBM support.

EGL Java run-time error code CSO7840E

CSO7840E: The client received notification from the server that the remote called program failed with return code %1.

Explanation

The remote called program ran but ended with a non-zero return code. The problem is in the program rather than in communications.

User Response

Examine or trace the called program to determine why it completed with a non-zero return code.

EGL Java run-time error code CSO7885E

CSO7885E: A TCP/IP read function failed on a call for userid %2 to hostname %1. Exception returned was: %3

Explanation

The EGL gateway received an exception when attempting a TCP/IP read function.

User Response

Examine the exception information returned in order to determine the cause of the problem. If you are unable to determine why the failure occurred, please contact IBM Support for assistance.

EGL Java run-time error code CSO7886E

CSO7886E: A TCP/IP write function failed on a call for userid %2 to hostname %1. Exception returned was: %3

Explanation

The EGL gateway received an exception when attempting a TCP/IP write function.

User Response

Examine the exception information returned in order to determine the cause of the problem. If you are unable to determine why the failure occurred, please contact IBM Support for assistance.

EGL Java run-time error code CSO7955E

CSO7955E: %1, %2

Explanation

An unexpected Java exception was caught.

The message text shows the name of the Java exception followed by the Java message that was thrown with the exception.

User Response

Review the message and respond as appropriate.

EGL Java run-time error code CSO7957E

CSO7957E: Conversion table name %1 is not valid for Java data conversion.

Explanation

You are using a generated Java class to call a program and have incorrectly specified a conversion table to convert Java data to the format used by the called program.

User Response

Review the help page on data conversion to determine the conversion table name, which you specify in the linkage options part, in the callLink element for the called program, in property conversionTable.

EGL Java run-time error code CSO7958E

CSO7958E: The native code did not provide an object of type CSOPowerServer to the Java wrapper, as is needed to convert data between the Java wrapper and the EGL-generated program.

Explanation

The native Java code invoked the call or execute method of a Java wrapper without first instantiating an object of class CSOPowerServer and providing that object to the wrapper.

User Response

Review the help pages on the Java wrapper for details on accessing EGL middleware, as is always required for data conversion.

EGL Java run-time error code CSO7966E

CSO7966E: The code page encoding %1 was not found for the conversion table %2.

Explanation

The conversion table specified in the linkage options requires an encoding not available in the Java Virtual Machine (JVM) being used.

User Response

Review the help page on data conversion to determine the correct conversion table name, which you specify in the linkage options part, in the callLink element for the called program, in property conversionTable. If you specified the correct conversion table, make sure that the JVM that you are using is supported by the Java run-time environment of EGL.

If the previous steps do not reveal the problem, consider whether the installation of your JVM is flawed or whether your Java Virtual machine does not support all encodings. In these cases, refer to the documentation of your JVM vendor or contact the JVM vendor for assistance.

If you encountered the error when running an applet client in a browser, the error occurred at the PowerServer SessionManager used by the client applet. In this case, refer to the documentation for the JVM that the SessionManager is running on or contact the JVM vendor.

EGL Java run-time error code CSO7968E

CSO7968E: Host %1 is not known or could not be found.

Explanation

No remote system specified in the linkage.

User Response

The remote system must be specified in the linkage part's location field.

EGL Java run-time error code CSO7970E

CSO7970E: Could not load the required EGL shared library %1, reason: %2

Explanation

The shared library for is required to complete the operation, but it could not be loaded.

User Response

Make sure that the shared library is on the system. It must be included in the environment variable that specifies the shared library path, PATH or LIBPATH.

EGL Java run-time error code CSO7975E

CSO7975E: The properties file %1 could not be opened.

Explanation

The properties file required by the program could not be opened. The name of the properties file may be specified on the command line when the program is started. If no name is given when the program is started, the following name is used by default:

```
tcpiplistener.properties
```

Either the properties file does not exist, or it exists but could not be opened.

User Response

Ensure that the properties file exists and that the program has the proper permissions to read it, then run the program again.

EGL Java run-time error code CSO7976E

CSO7976E: The trace file %1 could not be opened. The exception is %2 The message is as follows: %3

Explanation

An exception occurred when the program tried to open the trace output file.

User Response

Correct the problem and re-run the program.

EGL Java run-time error code CSO7977E

CSO7977E: The program properties file does not contain a valid setting for the %1 property, which is required.

Explanation

The property is not defined in the program properties file.

User Response

Add the property to the program properties file and re-run the program. For details, see the help page on Java run-time properties.

EGL Java run-time error code CSO7978E

CSO7978E: An unexpected exception occurred. The exception is %1 The message is as follows: %2

Explanation

The program encountered an error.

User Response

Correct the problem and re-run the program.

EGL Java run-time error code CSO7979E

CSO7979E: Unable to create an InitialContext. Exception is %1

Explanation

The exception was thrown from the constructor of javax.naming.InitialContext. The program needs to create the InitialContext object to access the J2EE environment settings.

User Response

Use the text of the exception and the documentation of your J2EE environment to correct the problem.

EGL Java run-time error code CSO8000E

CSO8000E: The password entered to the Gateway has expired. %1

Explanation

The EGL GatewayServlet received an expired password exception when attempting to authenticate the user with the provided password.

User Response

Examine the exception information returned in order to determine the cause of the problem. Correct the problem by proving a new password.

EGL Java run-time error code CSO8001E

CSO8001E: The password entered to the Gateway is not valid. %1

Explanation

The EGL GatewayServlet received an invalid password exception when attempting to authenticate the user with the provided password.

User Response

Examine the exception information returned in order to determine the cause of the problem. Correct the problem by proving a new password.

EGL Java run-time error code CSO8002E

CSO8002E: The userid entered to the Gateway is not valid. %1

Explanation

The EGL GatewayServlet received an invalid userid exception when attempting to authenticate the user with the provided userid.

User Response

Examine the exception information returned in order to determine the cause of the problem. Correct the problem by proving a new userid.

EGL Java run-time error code CSO8003E

CSO8003E: Null entry for %1

Explanation

Null entry has been detected.

User Response

Examine the exception information returned in order to determine the cause of the problem. Correct the problem by providing required entry.

EGL Java run-time error code CSO8004E

CSO8004E: The gateway received an unknown security error.

Explanation

The EGL GatewayServlet received an unknown security exception when attempting to authenticate the user with the provided user information.

User Response

Examine the exception information returned in order to determine the cause of the problem. Correct the problem by proving new user information. If you are unable to determine why the failure occured, please contact IBM Support for assistance.

EGL Java run-time error code CSO8005E

CSO8005E: Error occurred when changing the password. %1

Explanation

The EGL GatewayServlet received an error when attempting to change the provided password.

User Response

Examine the exception information returned in order to determine the cause of the problem. Correct the problem by proving new password. If you are unable to determine why the failure occured, please contact IBM Support for assistance.

EGL Java run-time error code CSO8100E

CSO8100E: Unable to get a connection factory. Exception is %1

Explanation

The exception was thrown during a look-up of the connection factory that is used on a call when the value of the remoteComType property is CICSJ2C. The remoteComType property is in the linkage options part, in the callLink element for the called program.

The name of the connection factory begins java:comp/env/, followed by the value that you set in the location property of the same callLink element.

User Response

Make sure that the connection factory is defined properly in the J2EE environment and that the value for the location property is correct in the callLink element for the called program.

EGL Java run-time error code CSO8101E

CSO8101E: Unable to get a connection. Exception is: %1

Explanation

The exception was thrown by the getConnection method of the ConnectionFactory object that was used to make a call when the value of the remoteComType property is CICSJ2C. The remoteComType property is in the linkage options part, in the callLink element for the called program.

User Response

The Connection Factory or Resource Adapter may not be defined or onfigured properly. Diagnose the problem by referencing the text of the exception, the documentation of your Resource Adapter, and the documentation of your J2EE environment.

EGL Java run-time error code CSO8102E

CSO8102E: Unable to get an Interaction. Exception is: %1

Explanation

The exception was thrown by the createInteraction method of the Connection object that is used to make a call when the value of the remoteComType property is CICSJ2C. The remoteComType property is in the linkage options part, in the callLink element for the called program.

User Response

The Connection Factory or Resource Adapter may not be defined or configured properly. Diagnose the problem by using the text of the exception, the documentation of your Resource Adapter, and the documentation of your J2EE environment.

EGL Java run-time error code CSO8103E

CSO8103E: Unable to set an interaction verb. Exception is %1

Explanation

The exception was thrown by the setInteractionVerb method of the ECIInteractionSpec object that is used to make a call when the value of the remoteComType property is CICSJ2C. The remoteComType property is in the linkage options part, in the callLink element for the called program.

User Response

The Connection Factory or Resource Adapter may not be defined or configured properly. Diagnose the problem by using the text of the exception, the documentation of your Resource Adapter, and the documentation of your J2EE environment.

EGL Java run-time error code CSO8104E

CSO8104E: An error occurred during an attempt to communicate with CICS. Exception is %1

Explanation

The exception was thrown by the execute method of the Interaction object that is used to make a call when the value of the remoteComType property is CICSJ2C. The remoteComType property is in the linkage options part, in the callLink element for the called program.

User Response

The Connection Factory or Resource Adapter may not be defined or configured properly. Diagnose the problem by using the text of the exception, the documentation of your Resource Adapter, and the documentation of your J2EE environment. Additional information may be in the gateway log or in a log file on the remote system.

EGL Java run-time error code CSO8105E

CSO8105E: Unable to close an Interaction or Connection. Exception is %1

Explanation

The exception was thrown by the close method of a Connection or Interaction object that is used to make a call when the value of the remoteComType property is CICSJ2C. The remoteComType property is in the linkage options part, in the callLink element for the called program.

User Response

The Connection Factory or Resource Adapter may not be defined or configured properly. Diagnose the problem by using the text of the exception, the documentation of your Resource Adapter, and the documentation of your J2EE environment.

EGL Java run-time error code CSO8106E

CSO8106E: Unable to get a LocalTransaction for client unit of work. Exception is %1

Explanation

The exception was thrown by the getLocalTransaction method of a Connection object that is used to make a call in this situation:

- The value of the remoteComType property is CICSJ2C
- The value of the luwControl property is CLIENT

Those properties are in the linkage options part, in the callLink element for the called program.

User Response

The Connection Factory or Resource Adapter may not be defined or configured properly. Diagnose the problem by using the text of the exception, the documentation of your Resource Adapter, and the documentation of your J2EE environment.

EGL Java run-time error code CSO8107E

CSO8107E: Unable to set the timeout value on a CICSJ2C call. Exception is %1

Explanation

The exception was thrown by the setExecuteTimeout method of an ECIInterationSpec object that is used to make a call when the value of the remoteComType property is CICSJ2C. The remoteComType property is in the linkage options part, in the callLink element for the called program.

User Response

The Connection Factory or Resource Adapter may not be defined or configured properly. Diagnose the problem by using the text of the exception, the documentation of your Resource Adapter, and the documentation of your J2EE environment.

EGL Java run-time error code CSO8108E

CSO8108E: An error occurred during an attempt to communicate with CICS.

Explanation

The execute method of the Interaction object that is used to make the call returned false. The call did not complete successfully.

User Response

The Connection Factory or Resource Adapter may not be defined or configured properly. Diagnose the problem by using the text of the exception, the documentation of your Resource Adapter, and the documentation of your J2EE environment. Additional information may be in the gateway log or in a log file on the remote system.

EGL Java run-time error code CSO8109E

CSO8109E: The timeout value %1 is invalid. It must be a number.

Explanation

An invalid value was specified for the timeout.

User Response

Either do not specify a timeout value, or specify a number.

EGL Java run-time error code CSO8110E

CSO8110E: The parmForm linkage property must be set to COMMPTR to call program %1 as there is at least one parameter that is a dynamic array.

Explanation

The parmForm must be COMMPTR because one of the parameters is a dynamic array.

User Response

Change the parmForm to COMMPTR.

EGL Java run-time error code CSO8180E

CSO8180E: The linkage specified a DEBUG call within a J2EE server. The call was not made on a J2EE server, the J2EE server is not in debug mode, or the J2EE server was not enabled for EGL debugging.

Explanation

The DEBUG call cannot be completed.

User Response

If the call is not being made on a J2EE server, the TCP/IP hostname of the machine running the EGL debugger must be specified in the location field of the linkage. If the call is being made on a J2EE server, make sure that it was started in debug mode and make sure that the EGL Debugger jar files were added to it.

EGL Java run-time error code CSO8181E

CSO8181E: Cannot contact the EGL debugger at hostname %1 and port %2. Exception is %3

Explanation

The DEBUG call cannot be completed because the EGL debugger could not be contacted.

User Response

Make sure an EGL Listener is running in the EGL debugger at the specified hostname and port.

EGL Java run-time error code CSO8182E

CSO8182E: An error occurred while communicating with the EGL debugger at hostname %1 and port %2. Exception is %3

Explanation

Communication between the EGL debugger and the calling program failed.

User Response

Use the information in the exception message to correct the problem.

EGL Java run-time error code CSO8200E

CSO8200E: Array wrapper %1 cannot be expanded beyond its maximum size. The error occured in method %2.

Explanation

The maximum size of the array was exceeded.

User Response

Check the size and maximum size of the array before attempting to add to it.

EGL Java run-time error code CSO8201E

CSO8201E: %1 is an invalid index for array wrapper %2. Maximum size: %3. Current size: %4

Explanation

The index is outside the bounds of the array.

User Response

Use a valid index.

EGL Java run-time error code CSO8202E

CSO8202E: %1 is not a valid maximum size for array wrapper %2.

Explanation

The property maxSize must be greater than or equal to zero.

User Response

Do not set the property maxSize to a negative number.

EGL Java run-time error code CSO8203E

CSO8203E: %1 is an invalid object type to add to an array wrapper of type %2.

Explanation

The contents of the array must match its definition.

User Response

Change the type of objects that the array stores, or do not attempt to store that type of object in the array.

EGL Java run-time error code CSO8204E

CSO8204E: Cannot pass an Any, Dictionary, ArrayDictionary, Blob, Clob, or Ref variable as a parameter.

Explanation

The types listed may not be used as parameters on a call statement. In addition, types that contain the listed types may not be used as parameters.

User Response

Do not pass that kind of parameter to the called program.

EGL Java run-time error code EGL0650E

EGL0650E: The %1RequestAttr function failed with key, %2. Error: %3

Explanation

The EGL GetRequestAttr or SetRequestAttr function failed when invoked with the given key.

User Response

Use the Error part of this message to diagnose and correct the problem. Make sure the function is used within a PageHandler function.

EGL Java run-time error code EGL0651E

EGL0651E: The %1SessionAttr function failed with key, %2. Error: %3

Explanation

The EGL GetSessionAttr or SetSessionAttr function failed when invoked with the given key.

User Response

Use the Error part of this message to diagnose and correct the problem. Make sure the function is invoked within a PageHandler function.

EGL Java run-time error code EGL0652E

EGL0652E: The forward statement failed with label, %1. Error: %2

Explanation

Control could not be forwarded to the given label.

User Response

Use the Error part of this message to diagnose and correct the problem. Make sure the the EGL object which associated with the label is generated correctly and that the label is defined in the application configuration file.

EGL Java run-time error code EGL0653E

EGL0653E: Failed to create Bean from EGL object, %1. Error: %2

Explanation

Could not create an access bean from EGL record or PageHandler definition.

User Response

Use the Error part of this message to diagnose and correct the problem.

EGL Java run-time error code EGL0654E

EGL0654E: The SetError function failed with item, %1, key, %2. Error: %3

Explanation

The SetError function failed when invoked with the given message key.

User Response

Use the Error part of this message to diagnose and correct the problem. Make sure the item has an error entry in the JSP and the key is defined in the message resource file.

EGL Java run-time error code EGL0655E

EGL0655E: Failed to copy data from Bean to EGL record, %1. Error: %2

Explanation

An attempt to move data from the form bean to the record failed.

User Response

Use the Error part of this message to diagnose and correct the problem. Make sure the bean definition matches with the record definition.

EGL Java run-time error code EGL0656E

EGL0656E: Cannot assign array of size %1 to static array of size %2.

Explanation

The sizes of the arrays must match.

User Response

Check the EGL array definitions and make sure the array sizes are the same.

EGL Java run-time error code EGL0657E

EGL0657E: Processing of an onPageLoad parameter failed. Error: %1.

Explanation

An error occurred when EGL tried to receive values into the parameters of the onPageLoad function.

User Response

Use the Error part of this message to diagnose and correct the problem. Make sure the type definition of the passed value matches the type defined for the parameter in the onPageLoad function.

EGL Java run-time error code VGJ0001E

VGJ0001E: Maximum value overflow from %1.

Explanation

During an arithmetic calculation, either a value was divided by zero or an intermediate result exceeded 18 significant digits. The program ends unless system variable **VGVar.handleOverflow** is set to 2.

User Response

Perform one or more of the following actions:

- Correct the logic of your program to avoid the error.
- Define the program logic to handle the overflow condition; use the system variables **VGVar.handleOverflow** and **overflowIndicator**.

EGL Java run-time error code VGJ0002E

VGJ0002E: Error %1 occurred. The message text for this error could not be found in the message file %2.

Explanation

The message file may be corrupt or from an older release of EGL.

User Response

Complete one of the following instructions:

- If you extracted class files from the file fda6.jar, verify that the classes you have are at the same release or maintenance level as the classes in that file. If you find a mismatch, replace the older classes with the correct version.
- Reinstall fda6.jar from EGL.

If the problem persists, do as follows:

1. Record the message number and the message text.

   Note: The error message includes the following important information:
   - Where the error occurred
   - The type of internal error
2. Record the situation in which this message occurs.
3. For further instructions on how to report possible defects to the IBM Support Center, refer to the product's installation manual.

EGL Java run-time error code VGJ0003E

VGJ0003E: An internal error occurred at location %1.

Explanation

This error can occur only when system constraints or requirements were not satisfied or when EGL program parts were used improperly. The location specified in the error is used only for IBM diagnostic purposes.

User Response

Check the program setup and restart the system. If the problem persists, do as follows:

1. Record the message number and the message text.

   Note: The error message includes the following important information:
   - Where the error occurred
   - The type of internal error
2. Record the situation in which this message occurs.
3. For further instructions on how to report possible defects to the IBM Support Center, refer to the product's installation manual.

EGL Java run-time error code VGJ0004I

VGJ0004I: The error occurred in %1, function %2.

Explanation

This message accompanies another message when an error occurs. It identifies the program or record where the error occurred, as well as the function that was executing at the time.

User Response

None.

EGL Java run-time error code VGJ0005I

VGJ0005I: The error occurred in %1.

Explanation

This message accompanies another message and identifies the program or record where an error occurred.

User Response

None.

EGL Java run-time error code VGJ0006E

VGJ0006E: An error occurred during an I/O operation. %1

Explanation

An I/O operation failed, and the EGL statement has no try statement to deal with the error.

User Response

If you want the program to handle the error, set handleHardIOErrors to 1 and put the I/O statement in a try statement, as in the following example:

```
VGVar.handleHardIOErrors  = 1;

if (userRequest == "A")
  try
    add record1
  onException
    myErrorHandler(12);
  end
end
```

EGL Java run-time error code VGJ0007E

VGJ0007E: Minimum value overflow from %1.

Explanation

The arithmetic operation has produced a result tha is beyond the minimum value allowed for the data type.

User Response

Adjust the arithmetic expression accordingly.

EGL Java run-time error code VGJ0008E

VGJ0008E: A recoverable resource error occurred. %1

Explanation

There was an error while closing, committing, or rolling back a recoverable resource.

User Response

Use the information in the error message to correct the problem.

EGL Java run-time error code VGJ0009E

VGJ0009E: No field with identifier %1 could be found in %2.

Explanation

A dynamic access failed because the specified field does not exist.

User Response

Do not access nonexistent fields.

EGL Java run-time error code VGJ0010E

VGJ0010E: The assignment to %1 failed: incompatible assignment source %2.

Explanation

The source's type is not one that may be assigned to the target.

User Response

Ensure that the source and target types are compatible when assigning values.

EGL Java run-time error code VGJ0011E

VGJ0011E: Cannot resolve the value of %1 to a primitive type.

Explanation

The variable was used as a data item, but it is not a data item.

User Response

Change the program so that it does not use the variable as if it were a data item.

EGL Java run-time error code VGJ0012E

VGJ0012E: Could not evaluate an arithmetic expression: incompatible types in %1.

Explanation

The types of the values in the expression are incompatible.

User Response

Change the program to use compatible types in the expression.

EGL Java run-time error code VGJ0013E

VGJ0013E: The set statement failed: %1 cannot be set to the %2 state.

Explanation

The specified state is not supported for the variable.

User Response

Change the program so that it does not attempt this operation.

EGL Java run-time error code VGJ0014E

VGJ0014E: %1 cannot be subscripted. It is not an array.

Explanation

The variable was used as an array, but it is not an array.

User Response

Change the program so that it does not use the variable as an array.

EGL Java run-time error code VGJ0015E

VGJ0015E: %1, %2

Explanation

There was an error. The exception and its message are used as inserts to this message.

User Response

Use the information from the message inserts to correct the problem.

EGL Java run-time error code VGJ0016E

VGJ0016E: Any variable %1 has not been given a value.

Explanation

The variable was used before having been assigned a value.

User Response

Change the program to assign a value to the variable before using it.

EGL Java run-time error code VGJ0017E

VGJ0017E: Ref variable %1 is Nil.

Explanation

The variable must reference a value before it can be used.

User Response

Before using the variable, give it a value.

EGL Java run-time error code VGJ0018E

VGJ0018E: Cannot perform dynamic access on structured record %1.

Explanation

Dynamic access is not permitted on a structured record.

User Response

Do not use dynamic access on the structured record.

EGL Java run-time error code VGJ0019E

VGJ0019E: %1 cannot be copied.

Explanation

An operation attempted to copy something that may not be copied, or the attempt to make a copy failed.

User Response

EGL Java run-time error code VGJ0020E

VGJ0020E: The variable named %1 cannot be used as a %2.

Explanation

The variable's type does not allow it to be used as if it were of the specified type.

User Response

Change the program so that it does not use the variable as if it were a different type.

EGL Java run-time error code VGJ0021E

VGJ0021E: %1 cannot be tested for the %2 state.

Explanation

The error occurred in an IS or NOT expression. The variable on the left side of the expression does not support the state that was specified as the right side of the expression.

User Response

Remove or modify the expression.

EGL Java run-time error code VGJ0050E

VGJ0050E: An exception occurred while loading program %1. Exception: %2 Message: %3

Explanation

The program's class could not be loaded.

User Response

Use the exception message to diagnose and fix the problem. The most common cause of this error is that the jar file or the directory containing the program's class file is not listed in the CLASSPATH environment variable.

EGL Java run-time error code VGJ0055E

VGJ0055E: An error occurred on a call to program %1. The error code was %2 (%3).

Explanation

The error occurred during a call to a local Java program.

User Response

Use the exception message to diagnose and fix the problem.

EGL Java run-time error code VGJ0056E

VGJ0056E: Called program %1 expected %2 parameters but was passed %3.

Explanation

The wrong number of parameters was passed to a called program.

User Response

Rewrite the calling program or the called program so that both expect the same number of parameters to be passed.

EGL Java run-time error code VGJ0057E

VGJ0057E: An exception occurred while passing parameters to called program %1. Exception: %2 Message: %3

Explanation

An error occurred during a call to a Java program. The error may have happened before or after the program began.

User Response

Use the exception and its message to diagnose and fix the problem.

EGL Java run-time error code VGJ0058E

VGJ0058E: Properties file %1 could not be loaded.

Explanation

The program's properties file could not be loaded. The name of the properties file is obtained from the system property vgj.properties.file.

User Response

Ensure that vgj.properties.file has the correct file name and that the properties file is in a Jar file or directory listed in the CLASSPATH environment variable.

EGL Java run-time error code VGJ0060E

VGJ0060E: StartTransaction to class %1 failed. The exception is %2.

Explanation

The exception was thrown while the program was attempting to start a new JVM to run the specified server class as a new transaction. The property vgj.java.command specifies the command used to start a new JVM. The default command is java.

User Response

Ensure that the property vgj.java.command has the correct value, and that your program has permission to create a new process.

Put the startTransaction statement inside a try statement to prevent this from being a fatal error. When the startTransaction fails within a try statement, an error code will be stored in the errorCode system variable.

EGL Java run-time error code VGJ0062E

VGJ0062E: One or more parameters passed to MQ program %1 was of the wrong type. %2

Explanation

An exception was thrown while attempting to call the MQ program. The parameters are incorrect.

User Response

Consult the MQ program's documentation and the exception's message to correct the error.

EGL Java run-time error code VGJ0064E

VGJ0064E: Program %1 expected text form %2 but it was given text form %3 on a show statement.

Explanation

Both programs must use the same text form.

User Response

Modify the programs to use the same text form and regenerate.

EGL Java run-time error code VGJ0100E

VGJ0100E: The data of %1 is not in %2 format.

Explanation

The data in the item is in an unexpected format. Another item may have written over the specified item.

User Response

Correct the program logic to avoid the error.

EGL Java run-time error code VGJ0104E

VGJ0104E: %1 is not a valid index for subscript %2 of %3.

Explanation

One of the subscripts used with a multidimensional array is invalid. A subscript value must be between one and the number of occurrences defined for the subscripted item.

User Response

Ensure that the index value is a valid subscript for the subscripted item.

EGL Java run-time error code VGJ0105E

VGJ0105E: %1 is not a valid index for %2.

Explanation

A subscript value must be between one and the number of occurrences defined for the subscripted item.

User Response

Ensure that the index value is a valid subscript for the subscripted item.

EGL Java run-time error code VGJ0106E

VGJ0106E: User overflow during assignment of %1 to %2.

Explanation

The target of an assignment is not large enough to hold the result without truncating significant digits. The value of system variable VGVar.handleOverflow is 1, which causes the program to end.

User Response

Do as follows:

- Increase the number of significant digits in the target; or
- Define the program logic to handle the overflow condition; use the system variables VGVar.handleOverflow and overflowIndicator.

EGL Java run-time error code VGJ0108E

VGJ0108E: HEX item %1 was assigned nonhexadecimal value %2.

Explanation

HEX items can receive only hexadecimal digits.

User Response

Make sure that the source value includes only hexadecimal digits.

EGL Java run-time error code VGJ0109E

VGJ0109E: HEX item %1 was assigned nonhexadecimal value from %2: %3.

Explanation

HEX items can receive only hexadecimal digits.

User Response

Make sure that the source in the assignment contains only hexadecimal digits.

EGL Java run-time error code VGJ0110E

VGJ0110E: HEX item %1 was compared to nonhexadecimal value: %2.

Explanation

HEX items can be compared only to hexadecimal digits.

User Response

Make sure that the comparison value includes only hexadecimal digits.

EGL Java run-time error code VGJ0111E

VGJ0111E: HEX item %1 was compared to nonhexadecimal value from %2: %3.

Explanation

HEX items can be compared only to hexadecimal digits.

User Response

Make sure that the comparison value contains only hexadecimal digits.

EGL Java run-time error code VGJ0112E

VGJ0112E: NUM item %1 was assigned nonnumeric value: %2.

Explanation

NUM items can be assigned only numeric values. Such values contain digits and may have leading and trailing spaces, a decimal point, and a leading sign. The decimal point is allowed in between two digits, immediately before the first digit, or immediately after the last digit.

User Response

Make sure that the source value is numeric.

EGL Java run-time error code VGJ0113E

VGJ0113E: NUM item %1 was assigned nonnumeric value from %2: %3.

Explanation

NUM items can be assigned only numeric values. Such values contain digits and may have leading and trailing spaces, a decimal point, and a leading sign. The decimal point is allowed in between two digits, immediately before the first digit, or immediately after the last digit.

User Response

Make sure that the source value is numeric.

EGL Java run-time error code VGJ0114E

VGJ0114E: The value of item %1 (%2) is not valid as a subscript.

Explanation

The value has too many digits to be a subscript for any element in the array. A subscript value must be between one and the number of occurs declared for the structure item.

User Response

Make sure that the index value is a valid subscript for the array.

EGL Java run-time error code VGJ0115E

VGJ0115E: %1 cannot be assigned a string. The string was %2.

Explanation

The item cannot be assigned a string.

User Response

Do not assign a string to the item.

EGL Java run-time error code VGJ0116E

VGJ0116E: %1 cannot be assigned a number. The number was %2.

Explanation

The item cannot be assigned a number.

User Response

Do not assign a number to the item.

EGL Java run-time error code VGJ0117E

VGJ0117E: %1 cannot be converted to a long.

Explanation

The item cannot be converted to a long.

User Response

Complete the following steps:

1. Record the message number and the message text.

   Note: The error message includes the following important information:
   - Where the error occurred
   - The type of internal error
2. Record the situation in which this message occurs.
3. For further instructions on how to report possible defects to the IBM Support Center, refer to the *EGL Installation Guide*.

EGL Java run-time error code VGJ0118E

VGJ0118E: %1 cannot be converted to a number.

Explanation

The item item cannot be converted to a number.

User Response

Do not use the item in a place where a number is required.

EGL Java run-time error code VGJ0119E

VGJ0119E: %1 is not a valid number.

Explanation

While using the debugger, the user attempted to set the value of a numeric item, but the new value is not a number.

User Response

Use a numeric value.

EGL Java run-time error code VGJ0120E

VGJ0120E: %1 is not a valid value for the starting index of the substring operator on item %2.

Explanation

The starting index cannot be less than 1 or more than the length of the item.

User Response

Use a valid index for the starting index of the substring operator.

EGL Java run-time error code VGJ0121E

VGJ0121E: %1 is not a valid value for the ending index of the substring operator on item %2.

Explanation

The ending index cannot be less than 1 or more than the length of the item.

User Response

Use a valid index for the ending index of the substring operator.

EGL Java run-time error code VGJ0122E

VGJ0122E: The ending index of the substring operator on item %1 is %2, which cannot be less than the starting index, which is %3.

Explanation

The ending index of the substring operator cannot be less than the starting index.

User Response

Make sure that the starting index is less than or equal to the ending index.

EGL Java run-time error code VGJ0123E

VGJ0123E: The substring operator failed: %1 cannot be used as a string value.

Explanation

The variable does not support the substring operator.

User Response

Change the program so that it does not use the substring operator on the variable.

EGL Java run-time error code VGJ0124E

VGJ0124E: %1 cannot be assigned a record. The record was %2.

Explanation

The data item's type does not allow assignments from records.

User Response

Change the program so that it does not assign a record to the data item.

EGL Java run-time error code VGJ0125E

VGJ0125E: %1 cannot be used as a field.

Explanation

The variable is not a field.

User Response

Change the program so that it does not use the variable as a field.

EGL Java run-time error code VGJ0126E

VGJ0126E: Incompatible types in comparison of %1 to %2.

Explanation

The types of the values are incompatible in a comparison.

User Response

Ensure that the comparison uses compatible types.

EGL Java run-time error code VGJ0127E

VGJ0127E: %1 cannot be assigned a date or time value. The value was %2.

Explanation

The item cannot be assigned a date or time value.

User Response

Do not assign a date or time value to the item.

EGL Java run-time error code VGJ0140E

VGJ0140E: Array function %1 failed because there was an attempt to expand array %2 beyond its maximum size.

Explanation

The array cannot hold any more values.

User Response

Modify the program to check the size of the array before attempting to add to it.

EGL Java run-time error code VGJ0141E

VGJ0141E: %1 is an invalid index for array %2. Current size: %3. Max size: %4

Explanation

The index is out for range for the array.

User Response

Modify the program to use a valid array index.

EGL Java run-time error code VGJ0142E

VGJ0142E: The maximumSize of array %1 cannot be changed. Expected %2 got %3.

Explanation

The array was passed on a call statement. The corresponding array in the called program had a different maximumSize.

User Response

Change one of the programs so that both use an array with the same maximumSize.

EGL Java run-time error code VGJ0143E

VGJ0143E: %1 is not a valid size for array %2.

Explanation

The array was passed on a call statement. The called program changed the array's size to a value that is less than zero or larger than the value of the property maxSize.

User Response

Change the programs so they use the same value for the property maxSize.

EGL Java run-time error code VGJ0144E

VGJ0144E: %1 failed for array %2. Too many sizes were specified.

Explanation

The specified function failed. Its argument is an array of sizes, which contained too many elements.

User Response

Correct the argument.

EGL Java run-time error code VGJ0145E

VGJ0145E: %1 failed for array %2. The sizes must be numeric data items.

Explanation

The specified function failed. Its argument should be an array of numeric data items, but it was not.

User Response

Correct the argument.

EGL Java run-time error code VGJ0146E

VGJ0146E: %1 failed for array %2. The size given was less than zero.

Explanation

The specified function failed. Its argument is an array of sizes. One of the sizes was less than zero, but this is not allowed.

User Response

Correct the argument.

EGL Java run-time error code VGJ0147E

VGJ0147E: %1 failed for array %2. The maxSize given is less than the current size.

Explanation

The array's maxSize cannot be changed to a value that is less than its current size.

User Response

Correct the argument.

EGL Java run-time error code VGJ0160E

VGJ0160E: Math function %1 failed with error code 8 (domain error).

Explanation

An argument to the function is not valid.

User Response

Do as follows:

- Change the program logic to ensure that the arguments to the function are valid, as per the function's documentation; or
- Call the function in a try statement, or set VGVar.handleSysLibraryErrors to 1 before calling the function, so that the program can handle the error.

EGL Java run-time error code VGJ0161E

VGJ0161E: Math function %1 failed with error code 8 (domain error).

Explanation

The argument must be between -1 and 1.

User Response

Do as follows:

- Change the program logic to ensure that the argument passed to the function is between -1 and 1; or
- Call the function in a try statement, or set VGVar.handleSysLibraryErrors to 1 before calling the function, so that the program can handle the error.

EGL Java run-time error code VGJ0162E

VGJ0162E: Math function atan2 failed with error code 8 (domain error).

Explanation

Both arguments cannot be zero.

User Response

Do as follows:

- Change the program logic to ensure that at least one argument passed to the function is not zero; or

- Call the function in a try statement, or set VGVar.handleSysLibraryErrors to 1 before calling the function, so that the program can handle the error.

EGL Java run-time error code VGJ0163E

VGJ0163E: Math function %1 failed with error code 8 (domain error).

Explanation

The second argument must not be zero.

User Response

Do as follows:

- Change the program logic to ensure that the second argument is not zero; or
- Call the function in a try statement, or set VGVar.handleSysLibraryErrors to 1 before calling the function, so that the program can handle the error.

EGL Java run-time error code VGJ0164E

VGJ0164E: Math function %1 failed with error code 8 (domain error).

Explanation

The argument must be greater than zero.

User Response

Do as follows:

- Change the program logic to ensure that the argument passed to the function is greater than zero; or
- Call the function in a try statement, or set VGVar.handleSysLibraryErrors to 1 before calling the function, so that the program can handle the error.

EGL Java run-time error code VGJ0165E

VGJ0165E: Math function pow failed with error code 8 (domain error).

Explanation

If the first argument is zero, the second must be greater than zero.

User Response

Do as follows:

- Change the program logic to ensure that if the first argument passed to the function is zero, the second argument is greater than zero; or
- Call the function in a try statement, or set VGVar.handleSysLibraryErrors to 1 before calling the function, so that the program can handle the error.

EGL Java run-time error code VGJ0166E

VGJ0166E: Math function pow failed with error code 8 (domain error).

Explanation

If the first argument is less than zero, the second must be an integer.

User Response

Do as follows:

- Change the program logic to ensure that if the first argument passed to the function is less than zero, the second argument is an integer; or
- Call the function in a try statement, or set VGVar.handleSysLibraryErrors to 1 before calling the function, so that the program can handle the error.

EGL Java run-time error code VGJ0167E

VGJ0167E: Math function sqrt failed with error code 8 (domain error).

Explanation

The argument must be greater than or equal to zero.

User Response

Do as follows:

- Change the program logic to ensure that the argument passed to the function is greater than or equal to zero; or
- Call the function in a try statement, or set VGVar.handleSysLibraryErrors to 1 before calling the function, so that the program can handle the error.

EGL Java run-time error code VGJ0168E

VGJ0168E: Math function %1 failed with error code 12 (range error).

Explanation

An intermediate or final result cannot be represented as a double precision floating point number or with the precision of the result item.

User Response

Do as follows:

- Change the program logic to ensure that the target item is large enough to hold the result value; or
- Change the program logic so that the arguments to the function have values that do not cause this problem; or
- Call the function in a try statement, or set VGVar.handleSysLibraryErrors to 1 before calling the function, so that the program can handle the error.

EGL Java run-time error code VGJ0200E

VGJ0200E: String function %1 failed with error code 8.

Explanation

The index must be between 1 and the length of the string.

User Response

Do as follows:

- Change the program logic to ensure that the index-related argument to the function ranges between 1 and the length of the string; or
- Call the function in a try statement, or set VGVar.handleSysLibraryErrors to 1 before calling the function, so that the program can handle the error.

EGL Java run-time error code VGJ0201E

VGJ0201E: String function %1 failed with error code 12.

Explanation

The length must be greater than zero.

User Response

Do as follows:

- Change the program logic to ensure that the length arguments passed to the function have values that are greater than zero; or
- Call the function in a try statement, or set VGVar.handleSysLibraryErrors to 1 before calling the function, so that the program can handle the error.

EGL Java run-time error code VGJ0202E

VGJ0202E: String function setNullTerminator failed with error code 16.

Explanation

The last byte of the target string must be a blank or null character.

User Response

Do as follows:

- Change the program logic to ensure that the last byte of the target string is a blank or null character; or
- Call the function in a try statement, or set VGVar.handleSysLibraryErrors to 1 before calling the function, so that the program can handle the error.

EGL Java run-time error code VGJ0203E

VGJ0203E: String function %1 failed with error code 20.

Explanation

The index of a DBCHAR or UNICODE substring must be odd so that the index identifies the first byte of a character.

User Response

Do as follows:

- Change the program logic to ensure that the index arguments passed to the function are valid; or

- Call the function in a try statement, or set VGVar.handleSysLibraryErrors to 1 before calling the function, so that the program can handle the error.

EGL Java run-time error code VGJ0204E

VGJ0204E: String function %1 failed with error code 24.

Explanation

The length of a DBCHAR or UNICODE substring must be even to refer to a whole number of characters.

User Response

Do as follows:

- Change the program logic to ensure that the length arguments passed to the function have valid values; or
- Call the function in a try statement, or set VGVar.handleSysLibraryErrors to 1 before calling the function, so that the program can handle the error.

EGL Java run-time error code VGJ0215E

VGJ0215E: %1 was passed the nonnumeric string %2.

Explanation

Every character in the portion of the string defined by the length argument must be numeric.

User Response

Change the program logic so the characters in the portion of the string defined by the length argument are numeric.

EGL Java run-time error code VGJ0216E

VGJ0216E: %1 is not a valid date mask for %2.

Explanation

The date mask defined in the properties file for use with the function is not valid.

The valid characters for a date mask are as follows:

D, M, Y
D for Day, M for Month, Y for Year

Separator character
Any nonnumeric, single-byte character except D, M, or Y.

Valid date masks can be in any of the following formats:

- Long Gregorian

  The long version of the Gregorian mask must contain the following parts in any order:

  YYYY 4-digit year

MM 2-digit numeric month

DD 2-digit numeric day of month

The mask parts must be separated by any nonnumeric single-byte character except D, M, or Y.

For example, a mask of YYYY/MM/DD is used to display August 25, 1997 as 1997/08/25.

- Long Julian

  The long version of the Julian mask must contain the following parts in any order:

  YYYY 4-digit year

  DDD 3-digit numeric day of year

  The mask parts must be separated by any single-byte nonnumeric character except D, M, or Y.

  For example, a mask of DDD-YYYY can be used to display August 25, 1997 as 237-1997.

User Response

Change the date mask property to a valid value and restart the program. If no date mask property is defined, a default date mask will be used.

The date masks can be set using the properties vgj.datemask.gregorian.long.*NNN* and vgj.datemask.julian.long.*NNN*, where *NNN* is the current NLS code.

EGL Java run-time error code VGJ0217E

VGJ0217E: An error occurred in the convert function with argument %1: %2

Explanation

The attempt to convert the data of the argument failed. The reason for the failure is included in the message.

User Response

Use the error message to diagnose and correct the problem.

EGL Java run-time error code VGJ0218E

VGJ0218E: GetMessage failed. Could not find the message for key %1.

Explanation

No message was found for the key that was passed to the getMessage system function.

User Response

Either add the message, or use a different key.

EGL Java run-time error code VGJ0250E

VGJ0250E: Could not retrieve item %1 from containing part %2.

Explanation

An internal error occurred.An attempt was made to access an item with the specified index in the record or table.

User Response

Do as follows:

1. Record the message number and the message text.

   Note: The error message includes the following important information:
   - Where the error occurred
   - The type of internal error
2. Record the situation in which this message occurs.
3. For further instructions on how to report possible defects to the IBM Support Center, refer to the *EGL Installation Guide*.

EGL Java run-time error code VGJ0300E

VGJ0300E: Table file for table %1 could not be loaded. Could not find a file named either %2 or %3.

Explanation

Neither of the named files could be found in any of the resource locations. All resource locations are searched for the first file. If no such file exists, all resource locations are searched for the second file.

Resource locations differ depending on the mechanism that was used to locate the table file.

If the error was encountered in an applet, resource locations refer to locations on the server machine and can vary depending on the implementation of the Java Virtual Machine. However, all implementations should search the directory on the server specified by the CODEBASE value. This value is set by the APPLET tag in the HTML file containing the applet. If no CODEBASE value is specified, it defaults to the directory on the web server containing the HTML file.

If the error was encountered in an application, valid resource locations are as follows:

- The directory that the Java Virtual Machine was started in (the working directory for the executable).
- Any directory specified in the CLASSPATH for the application being run. Specification of this value is system-dependent. On some systems, it can be specified as an environment variable. All systems allow it to be specified when invoking the Java Virtual Machine using the -classpath option. See the documentation that came with your copy of the Java Virtual Machine for more information on the value of CLASSPATH.

User Response

First, locate the table file and make sure the permissions necessary to access it are set.

If the error occurred from within an applet or if the error occurred from within an application and you do not want to modify the existing set of resource locations, copy the table file into a valid resource location.

Otherwise, complete one of the following instructions:

- If the Java interpreter will use the value of the CLASSPATH environment variable, add the directory containing the table file to the current value of CLASSPATH.
- Specify the directory containing the table file by using the -classpath option when invoking the Java interpreter. If specifying the -classpath option overrides the value of the CLASSPATH environment variable, you need to specify the path to the Java run-time classes (e.g. classes.zip or rt.jar) in addition to any directories you add as resource locations.

Additional diagnostic information may become available if you enable program trace.

EGL Java run-time error code VGJ0301E

VGJ0301E: Table file %1 for table %2 could not be loaded because an incorrect number of bytes was returned during the read operation on the table header.

Explanation

One of the following conditions exists:

- The table file has become corrupt.
- The table file was not generated with EGL or VisualAge Generator.

User Response

Regenerate the table.

Additional diagnostic information may become available if you enable program trace.

If the problem persists, do as follows:

1. Record the message number and the message text.

   Note: The error message includes the following important information:
   - Where the error occurred
   - The type of internal error
2. Record the situation in which this message occurs.
3. For further instructions on how to report possible defects to the IBM Support Center, refer to the *EGL Installation Guide*.

EGL Java run-time error code VGJ0302E

VGJ0302E: Table file %1 for table %2 could not be loaded because an unexpected magic number was encountered during inspection of the table header.

Explanation

One of the following conditions exists:
- The table file has become corrupt.
- The table file was not generated with EGL or VisualAge Generator.

User Response

Regenerate the table.

Additional diagnostic information may become available if you enable program trace.

If the problem persists, do as follows:
1. Record the message number and the message text.

   Note: The error message includes the following important information:
   - Where the error occurred
   - The type of internal error
2. Record the situation in which this message occurs.
3. For further instructions on how to report possible defects to the IBM Support Center, refer to the *EGL Installation Guide*.

EGL Java run-time error code VGJ0303E

VGJ0303E: Table file %1 for table %2 could not be loaded because an internal I/O error occurred during a read or close operation.

Explanation

One of the following conditions exists:
- The table file has become corrupt.
- The table file was not generated with EGL or VisualAge Generator.

User Response

Regenerate the table.

Additional diagnostic information may become available if you enable program trace.

If the problem persists, do as follows:
1. Record the message number and the message text.

   Note: The error message includes the following important information:
   - Where the error occurred
   - The type of internal error
2. Record the situation in which this message occurs.
3. For further instructions on how to report possible defects to the IBM Support Center, refer to the *EGL Installation Guide*.

EGL Java run-time error code VGJ0304E

VGJ0304E: Table file %1 for table %2 could not be loaded because an incorrect number of bytes was returned during the read operation on the table data.

Explanation

One of the following conditions exists:

- The table file was regenerated after its columns were changed but the program that is attempting to load the table was not regenerated. Generating only the table after changing the column definition causes an inconsistency to exist between the definition in the table file and the definition in the table class file, which is only generated during run-time code generation.
- The table file has become corrupt.
- The table file was not generated with EGL or VisualAge Generator.

User Response

Do as follows:

- If the column definition has not been changed, regenerate the table.
- If the column definition has been changed, either remove the change and regenerate the table or regenerate the run-time code for the program that uses the table.

Additional diagnostic information may become available if you enable program trace.

If the problem persists, do as follows:

1. Record the message number and the message text.

   Note: The error message includes the following important information:
   - Where the error occurred
   - The type of internal error
2. Record the situation in which this message occurs.
3. For further instructions on how to report possible defects to the IBM Support Center, refer to the *EGL Installation Guide*.

EGL Java run-time error code VGJ0305E

VGJ0305E: Table file %1 for table %2 could not be loaded. The data encountered in the table file for item %3 is not in the correct format. The corresponding data format error is: %4

Explanation

One of the following conditions exists:

- The table file was regenerated after its columns were changed but the applet or application that is attempting to load the table was not regenerated. Generating only the table after changing the column definition causes an inconsistency to exist between the definition in the table file and the definition in the table class file, which is only generated during run-time code generation.
- The table file has become corrupt.

- The table file was not generated with Rational Application Developer for z/OS or with VisualAge Generator.

User Response

Do as follows:

- If the column definition has not been changed, regenerate the table.
- If the column definition has been changed, either remove the change and regenerate the table or regenerate the run-time code for the program that uses the table.

If the problem persists, do as follows:

1. Record the message number and the message text.

   Note: The error message includes the following important information:
   - Where the error occurred
   - The type of internal error
2. Record the situation in which this message occurs.
3. For further instructions on how to report possible defects to the IBM Support Center, refer to the *EGL Installation Guide*.

EGL Java run-time error code VGJ0306E

VGJ0306E: Table file %1 for table %2 could not be loaded because the data in the table file is for a different type of table than table %2.

Explanation

One of the following conditions exists:

- The table file was regenerated after its columns were changed but the applet or application that is attempting to load the table was not regenerated. Generating only the table after changing the column definition causes an inconsistency to exist between the definition in the table file and the definition in the table class file, which is only generated during run-time code generation.
- The table file has become corrupt.
- The table file was not generated with EGL or VisualAge Generator.

User Response

Do as follows:

- If the table type has not been changed, regenerate the table.
- If the table type has been changed, either edit the table definition so that it is of the correct type and regenerate the table or regenerate the run-time code for the program that uses the table.

Additional diagnostic information may become available if you enable program trace.

If the problem persists, do as follows:

1. Record the message number and the message text.

   Note: The error message includes the following important information:

- Where the error occurred
- The type of internal error

2. Record the situation in which this message occurs.
3. For further instructions on how to report possible defects to the IBM Support Center, refer to the *EGL Installation Guide*.

EGL Java run-time error code VGJ0307E

VGJ0307E: Table file %1 for table %2 could not be loaded because table file %1 is a VisualAge Generator C++ table file and is not in big-endian format.

Explanation

Table files generated by the VisualAge Generator C++ generator can only be used with Java programs if the byte-ordering used to encode numeric data within the table is big-endian.

User Response

Regenerate the table in big-endian format or as a Java platform-independent table.

To regenerate the table in big-endian format, use VisualAge Generator to generate the table for a C++ target system that is big-endian (e.g. AIX). To regenerate the table as a Java platform-independent table, generate the table for a Java target system with VisualAge Generator or EGL.

Additional diagnostic information may become available if you enable program trace.

EGL Java run-time error code VGJ0308E

VGJ0308E: Table file %1 for table %2 could not be loaded. Table file %1 is a VisualAge Generator C++ table file, and the character encoding used in the table (%3) is not supported on the run-time system.

Explanation

Table files generated by the VisualAge Generator C++ generator can be used with Java programs only if the type of character encoding used for data within the table is the same type of encoding used by the run-time system.

User Response

Do as follows:

1. Determine the character encoding used on your system. Java programs use either the ASCII or EBCDIC character encodings. Most workstations use the ASCII encoding. Most host platforms use the EBCDIC encoding. If you do not know the encoding used on your system, contact your system administrator.
2. Regenerate the table using the correct character encoding or as a Java platform-independent table.

   To regenerate the table using the correct character encoding, use VisualAge Generator to generate the table for your target system or another C++ target

system that uses the same character encoding. To regenerate the table as a Java platform-independent table, generate the table for a Java target system with VisualAge Generator or EGL.

Additional diagnostic information may become available if you enable program trace.

EGL Java run-time error code VGJ0315E

VGJ0315E: A shared table entry for table %1 could not be found during the table unloading process.

Explanation

An internal error occurred.

User Response

Do as follows:

1. Record the message number and the message text.

   Note: The error message includes the following important information:
   - Where the error occurred
   - The type of internal error
2. Record the situation in which this message occurs.
3. For further instructions on how to report possible defects to the IBM Support Center, refer to the *EGL Installation Guide*.

EGL Java run-time error code VGJ0320E

VGJ0320E: An edit routine with table %1 failed while comparing the table column %2 and the field %3.

Explanation

The table column and the field have types that are not valid for comparison.

User Response

Do one of the following:

- Ensure that the types of the column and the field are valid for comparison by doing the following:
  1. Correct either the type of the column or the type of the field so that the comparison will be valid.
  2. Regenerate the program.
  3. Run the program.
- Modify your program to use a different table for the edit routine such that the comparison of the column and the field will be valid.

Refer to the trace output for more information.

EGL Java run-time error code VGJ0330E

VGJ0330E: Could not find a message with ID %1 in the message table %2.

Explanation

This error can occur during the following operations:

- Lookup of the the value for a form's msgField.
- Lookup of the value with the identifier specified as an edit message.

One of the following conditions exists:

- A message with this ID does not exist in the message table.
- The table file or message resource bundle for the table has become corrupt.

User Response

Do one of the following:

- Ensure that a message with the message ID exists by doing the following:
  1. Add a message to the table with the message ID if it does not already exist.
  2. Regenerate the table.
  3. Run the program.
- Modify your program to use a different message that is already defined in the table.
- Modify your program to use a different message table that contains a message with the message ID.

EGL Java run-time error code VGJ0331E

VGJ0331E: Message table file %1 could not be loaded.

Explanation

The class for the program's message table could not be loaded, or an instance of the class could not be created.

User Response

Ensure the message table has been generated.

EGL Java run-time error code VGJ0350E

VGJ0350E: An error occurred on a call to program %1. The error code was %2.

Explanation

A remote or EJB call to the specified program failed.

User Response

Additional diagnostic information may become available if you enable program trace.

EGL Java run-time error code VGJ0351E

VGJ0351E: commit failed: %1

Explanation

The resources could not be committed.

User Response

Additional diagnostic information may become available if you enable program trace.

EGL Java run-time error code VGJ0352E

VGJ0352E: rollBack failed: %1

Explanation

The resources could not be rolled back.

User Response

Additional diagnostic information may become available if you enable program trace.

EGL Java run-time error code VGJ0400E

VGJ0400E: An invalid parameter index, %1, was used for function %2.

Explanation

This is an internal error.

User Response

Contact IBM support.

EGL Java run-time error code VGJ0401E

VGJ0401E: An invalid parameter descriptor was detected for function %1, parameter %2.

Explanation

This is an internal error.

User Response

Contact IBM support.

EGL Java run-time error code VGJ0402E

VGJ0402E: The type of the value used for parameter %1 of function or program %2 is invalid.

Explanation

The value cannot be passed as a parameter, because the type of the value is incompatible with the type of the parameter.

User Response

Do one of the following:

- Change the definition of the parameter to match the type of the value.
- Change the type of the value to match the definition of the parameter.

EGL Java run-time error code VGJ0403E

VGJ0403E: An error occured while running script %1. The exception text is %2.

Explanation

The script caused an exception to be thrown.

User Response

Correct the program logic to avoid the error.

EGL Java run-time error code VGJ0416E

VGJ0416E: An error occurred on a call to program %1. The error code was %2 (%3).

Explanation

An exception was thrown during an attempt to run the called program. The problem may be due to one of the following conditions:

- The program may not have permission to create a new process.
- The called program may not exist.
- The called program may not be found in the system path.

User Response

Do as follows:

1. Verify that the program has permission to create a new process.
2. Verify that the called program exists.
3. Verify that the called program can be found in the system path.

If the problem persists, do as follows:

1. Record the message number and the message text.

   Note: The error message includes the following important information:
   - Where the error occurred
   - The type of internal error
2. Record the situation in which this message occurs.
3. For further instructions on how to report possible defects to the IBM Support Center, refer to the *EGL Installation Guide*.

EGL Java run-time error code VGJ0450E

VGJ0450E: I/O operation %1 with I/O object %2 failed for this reason: %3.

Explanation

An EGL I/O statement failed outside of a try statement, or when the value of system variable sysVar.handleHardIoErrors was zero.

User Response

Review the error message and respond as appropriate.

EGL Java run-time error code VGJ0500E

VGJ0500E: No input received for required field - enter again.

Explanation

No data was typed in the field. The field is defined as required.

User Response

Enter data in the field, or press a bypass edit key to bypass the edit check. Blanks will not satisfy the data input requirement for any type of field. In addition, zeros will not satisfy the data input requirement for numeric fields. The program continues.

EGL Java run-time error code VGJ0502E

VGJ0502E: Data type error in input - enter again.

Explanation

The data in the field is not valid numeric data. The field was defined as numeric.

User Response

Enter only numeric data in this field, or press a bypass edit key to bypass the edit check. In either situation, the program continues.

EGL Java run-time error code VGJ0503E

VGJ0503E: Number of allowable significant digits exceeded - enter again.

Explanation

Data was entered into a numeric field that is defined with decimal places, a sign, currency symbol, or numeric separator edits. The input data exceeds the number of significant digits that can be displayed within the editing criteria. The number entered is too large. The number of significant digits cannot exceed the field length, minus the number of decimal places, minus the places required for editing characters.

User Response

Enter a number with fewer significant digits.

EGL Java run-time error code VGJ0504E

VGJ0504E: Input not within defined range - enter again.

Explanation

The data in the field is not within the range of valid data defined for this item.

User Response

Enter data that is within the defined range or press a bypass edit key to bypass the edit check. In either case, the program will continue.

EGL Java run-time error code VGJ0505E

VGJ0505E: Input minimum length error - enter again.

Explanation

The data in the field does not contain enough characters to meet the required minimum length.

User Response

Enter the required number of characters to meet the minimum length or press a bypass edit key to bypass the edit check. In either case, the program will continue.

EGL Java run-time error code VGJ0506E

VGJ0506E: Table edit validity error - enter again.

Explanation

The data in the field does not meet the table edit requirement defined for the variable field.

User Response

Enter data that conforms to the table edit requirement or press a bypass edit key to bypass the edit check. In either case, the program will continue.

EGL Java run-time error code VGJ0507E

VGJ0507E: Modulus check error on input - enter again.

Explanation

The data in the field does not meet the modulus check requirement defined for the variable field.

User Response

Enter data that conforms to the modulus check defined for the variable field or press a bypass edit key to bypass the edit check. In either case, the program will continue.

EGL Java run-time error code VGJ0508E

VGJ0508E: Input not valid for defined date or time format %1.

Explanation

The data in the field, defined with a date edit, does not meet the requirements of the format specification.

User Response

Enter the date in the correct format shown in the message.

EGL Java run-time error code VGJ0510E

VGJ0510E: Input not valid for boolean field.

Explanation

The value typed in the field does not conform to the boolean check. Input into a boolean field must be either 'Y' or 'N' for character fields and either 1 or 0 for numeric fields.

User Response

Enter a 'Y' or 'N' for a character field or a 1 or 0 for a numeric field, or press the bypass edit key to bypass the edit check. In either case, the program will continue.

EGL Java run-time error code VGJ0511E

VGJ0511E: Edit table %1 is not defined for %2.

Explanation

A user message was requested but a user message table prefix was not defined for the program.

User Response

Have the program developer do one of the following:

- Add the message table prefix to the program specification and generate the program again.
- Remove the user message number from the field edit and generate again.

EGL Java run-time error code VGJ0512E

VGJ0512E: Hexadecimal data is not valid.

Explanation

The data in the variable field must be in hexadecimal format. One or more of the characters you entered does not occur in the following set: a b c d e f A B C D E F 0 1 2 3 4 5 6 7 8 9

User Response

Enter only hexadecimal characters in the variable field. The characters are left-justified and padded with the character 0. Embedded blanks are not permitted.

EGL Java run-time error code VGJ0513E

VGJ0513E: Value entered is invalid as it does not match the pattern that is set.

Explanation

Entered a value that does not match the pattern

User Response

Enter the value as specified in the pattern.

EGL Java run-time error code VGJ0514E

VGJ0514E: Input maximum length error - enter again.

Explanation

Exceeded specified maximum length set for this field.

User Response

Do not exceed the maximum length specified.

EGL Java run-time error code VGJ0516E

VGJ0516E: Input not within defined list - enter again.

Explanation

Input not within defined list - enter again.

User Response

Input not within defined list - enter again.

EGL Java run-time error code VGJ0517E

VGJ0517E: Date/Time format specified %1 is invalid.

Explanation

The date or time format specified is invalid.

User Response

Change the format to conform to the rules specified in the help topic *Date, time, and timestamp format specifiers*.

Related reference
"Date, time, and timestamp format specifiers" on page 42

EGL Java run-time error code VGJ0600E

VGJ0600E: Unable to get linkage for program, %1.

Explanation

An entry for the specified program cannot be found in the CSO properties file because of one of the following reasons:

- An incorrect properties file was specified in the GatewayServlet configuration.
- The entry for the program was not specified in the CSO properties file.
- The CSO properties file is not in the directory specified in the GatewayServlet configuration.

User Response

Contact the web server administrator to make sure that the following are performed:

- Make sure the GatewayServlet configuration specifies the correct CSO properties file using the linkageTable initialization parameter.
- Make sure that the program is defined in the CSO properties file.

EGL Java run-time error code VGJ0601E

VGJ0601E: An exception occurred while attempting to call entry point program, %1. Exception: %2. Message: %3.

Explanation

An unexplained error occurred while attempting to call the entry point program. The exception and message will define the error further. An entry point page or program gives the user a menu of programs which can be started using the GatewayServlet.

User Response

Contact the web server administrator to make sure that the entry point page or the entry program are specified correctly in the GatewayServlet configuration.

EGL Java run-time error code VGJ0603E

VGJ0603E: The bean, %1, is invalid.

Explanation

The Page Bean or the bean name is invalid.

User Response

Contact the web server administrator to make sure that the bean name is correct and that the Page Bean and the Java Server Page are deployed and made available to the GatewayServlet.

EGL Java run-time error code VGJ0604E

VGJ0604E: An exception occurred while attempting to load bean, %1. Exception: %2. Message: %3.

Explanation

An unexplained error occurred while trying to load the Page Bean. The exception and message will define the error further.

User Response

Contact the web server administrator to make sure that the bean name is correct and that the Page Bean and the Java Server Page are deployed and made available to the GatewayServlet.

EGL Java run-time error code VGJ0607E

VGJ0607E: A version mismatch has occurred between the server, %1, and bean, %2.

Explanation

The version of the User Interface Record Bean does not match the verion of the User Interface Record used by the server program. For proper operation, the versions must be compatible.

User Response

Contact the program developer and generate both the program and user interface record beans. Contact the web server administrator to make sure that the user interface record bean is deployed to the proper location.

EGL Java run-time error code VGJ0608E

VGJ0608E: An error occurred while attempting to set data in the bean, %1. Exception: %2. Message: %3.

Explanation

An exception occurred while trying to set the record data from the server application into the User Interface Record Bean. The exception and message are included to help determine the problem.

User Response

Use the exception and message included in the message for problem determination.

EGL Java run-time error code VGJ0609I

VGJ0609I: A gateway session is being bound for user, %1.

Explanation

This informational message appears on the application server's stdout or stderr. The message appears whenever a web session is created for the user.

User Response

No response is required.

EGL Java run-time error code VGJ0610I

VGJ0610I: A gateway session is being unbound for user, %1.

Explanation

This informational message appears on the application server's stdout or stderr. The message appears whenever a web session has ended for the user. A session will end after a period of inactivity or if a severe error occurs that terminates the session.

User Response

No response is required.

EGL Java run-time error code VGJ0611E

VGJ0611E: Unable to establish a connection with the SessionIDManager.

Explanation

The GatewayServlet was unable to connect to the SessionIDManager. The SessionIDManager is the component which gives session ids for gateway users. A session id is obtained for each active session and is used by the server program for saving and restoring application data.

The SessionIDManager is a separate application which listens for connects and requests for ids. When a session ends, the SessionIDManager will make the session id available to other sessions. The SessionIDManager must be active in order to run the GatewayServlet.

User Response

Contact your web server administrator to start the SessionIDManager. If already started, the location of the SessionIDManager must be set in the GatewayServlet's configuration.

EGL Java run-time error code VGJ0612I

VGJ0612I: A gateway session is connected to the SessionIDManager for user, %1.

Explanation

This informational message appears in the web server's stdout or stderr. A session has connected to the SessionIDManager successfully in order to obtain a session id. The session id is used by the server program to save and restore program data.

User Response

No response is required.

EGL Java run-time error code VGJ0614E

VGJ0614E: A required parameter, %1, is missing from the GatewayServlet configuration.

Explanation

A required parameter was not specified in the servlet configuration. The GatewayServlet will not run without these parameters.

User Response

Contact the web server administrator to make sure that the GatewayServlet is properly configured. Reference your application server documentation to determine how to configure servlet parameters.

EGL Java run-time error code VGJ0615E

VGJ0615E: Web transaction %1 is not allowed to run on this instance of the EGL Action Invoker.

Explanation

There was a problem creating or retrieving the GatewayRequestHandler for the program.

User Response

Ensure that the named application has been generated and deployed to the server.

EGL Java run-time error code VGJ0616E

VGJ0616E: The gateway parameter %1 does not specify valid class: %2

Explanation

The class identified in the specified gateway property could not be loaded or instantiated.

User Response

Ensure that the named class has been deployed to the server and specified correctly in the gateway properties file.

EGL Java run-time error code VGJ0617E

VGJ0617E: Please provide valid public user information in the gateway properties file.

Explanation

The public user name or password specified in the gateway properties file is invalid.

User Response

Ensure that the public user name and password values in the gateway properties file are correct.

EGL Java run-time error code VGJ0700E

VGJ0700E: An error occurred during database connection: %1.

Explanation

An error occurred during an attempt to connect to a database. The error message ends with text from the database management system.

User Response

Review the error message and respond as appropriate. Additional diagnostic information may become available if you enable program trace.

EGL Java run-time error code VGJ0701E

VGJ0701E: A database connection must be established prior to an SQL I/O operation.

Explanation

An SQL I/O operation was attempted before a database connection was established.

User Response

An SQL I/O operation is valid only after the program creates a database connection. The program can create a default connection based on a program property and can override the default by running the connect system function. Review the EGL help pages for details on program properties and on setting up database access.

EGL Java run-time error code VGJ0702E

VGJ0702E: An error occurred during SQL I/O operation %1. %2.

Explanation

An error occurred during the specified SQL I/O operation. The message ends with text from the database management system.

User Response

Review the message and respond as appropriate.

Additional diagnostic information may become available if you enable program trace.

EGL Java run-time error code VGJ0703E

VGJ0703E: An error occurred during setup for SQL I/O operation %1. %2.

Explanation

An error occurred during setup for the specified SQL I/O operation.

User Response

Review the message and respond as appropriate.

Additional diagnostic information may become available if you enable program trace.

EGL Java run-time error code VGJ0705E

VGJ0705E: An error occurred while disconnecting database %1. %2.

Explanation

An error occurred during an attempt to disconnect from the specified database. The error message ends with text from the database management system.

User Response

Review the message and respond as apropriate.

Additional diagnostic information may become available if you enable program trace.

EGL Java run-time error code VGJ0706E

VGJ0706E: Cannot set connection to database %1. The connection does not exist.

Explanation

An error occurred during an attempt to set the connection to the specified database. The connection can be set only to an active database connection within the transaction.

User Response

Make sure that the name of the database matches one of the active database connections established for the transaction.

Additional diagnostic information may become available if you enable program trace.

EGL Java run-time error code VGJ0707E

VGJ0707E: An SQL I/O sequence error occurred on %1.

Explanation

A sequence error may occur in these cases:

- An EGL replace or delete occurs but was not preceded by a setupd or update statement against the same SQL record
- An EGL scan occurs but was not preceded by a setupd or setinq statement against the same SQL record

The message identifies the last I/O operation that the program attempted, whether replace, delete, or scan.

User Response

Make sure that the order of EGL statements is correct.

Additional diagnostic information may become available if you enable program trace.

EGL Java run-time error code VGJ0708E

VGJ0708E: Error while loading the JDBC driver classes: %1

Explanation

An error occurred while loading the JDBC driver classes, which are necessary for SQL I/O.

User Response

Ensure that the JDBC driver classes are specified correctly in the property vgj.jdbc.drivers. If more than one is needed, separate their names with a semicolon. Also ensure that the classes can be found somewhere in the classpath.

EGL Java run-time error code VGJ0709E

VGJ0709E: A statement (%1) used a prepared statement that has not been prepared.

Explanation

The prepared statement named in the error message does not exist. Prepared statements are created by calling the EGL prepare statement.

User Response

Correct the program logic by adding a prepare before the prepared statement is used.

EGL Java run-time error code VGJ0710E

VGJ0710E: A %1 statement used a result set that is closed or does not exist.

Explanation

The result set used by the statement cannot be used because it is not open or does not exist.

User Response

Correct the program logic to avoid using invalid result sets.

EGL Java run-time error code VGJ0711E

VGJ0711E: An error occurred while connecting to database %1: %2

Explanation

A connection could not be established to the database named in the message.

User Response

Use the Error part of this message to diagnose and correct the problem.

EGL Java run-time error code VGJ0712E

VGJ0712E: Cannot connect to the default database. The name of the default database was not specified.

Explanation

The name of the default database was not specified, so the program cannot connect to it.

User Response

The name of the default database can be specified in several ways. One of the properties vgj.jdbc.default.database.programName (where programName is the name of the program) and vgj.jdbc.default.database must be set. The value of that property may be the actual name of the default database, or it may be the default database's logical name. When a logical name is used, another property must be set: vgj.jdbc.database.logicalName. The value of this property must be the actualy name of the default database.

EGL Java run-time error code VGJ0713E

VGJ0713E: GET failed because result set %1 was not opened with scroll.

Explanation

Only GET NEXT is allowed when the OPEN statement does not specify SCROLL.

User Response

Add the scroll option to the open statement where the result set is created.

EGL Java run-time error code VGJ0750E

VGJ0750E: The I/O driver for file %1 could not be created. %2

Explanation

A failure occurred during creation of the I/O driver for the specified file. This error can occur at the following times:

- On the first I/O operation for a record that is related to the specified file; or
- On the first access of the system variable resourceAssociation for a record that is related to the specified file.

The end of the message indicates the reason for the failure.

User Response

Review the error message and respond as appropriate.

EGL Java run-time error code VGJ0751E

VGJ0751E: The fileType property for file %1 could not be found in the Java run-time property vgj.ra.fileName.fileType.

Explanation

You need to set the following run-time property to a valid file type:

```
vgj.ra.fileName.fileType
```

fileName
Name of the file specified in the message. This file name is a logical file name that is associated with an EGL record.

For an MQ record, the value is mq; for a serial record, the value is seqws. The source of the value is the generation-time resource associations part; specifically, the association element for the file, property **fileType**.

User Response

Do as follows:

- Add the run-time fileType property to the run-time properties file or deployment descriptor; or
- Set the fileType value at generation time and regenerate the program:
  - In the file-name-specific association element of the resource associations part, set the property **fileType**
  - In the build descriptor used at generation, set the option **genProperties** to GLOBAL

  For other details, see the EGL help pages on the association element, on Java run-time properties, and on setting up the environment.

EGL Java run-time error code VGJ0752E

VGJ0752E: An invalid fileType %1 was specified for file %2 in the resource associations part.

Explanation

You need to set the following run-time property to a valid file type:

```
vgj.ra.fileName.fileType
```

fileName
Name of the file specified in the message. This file name is a logical file name that is associated with an EGL record.

For an MQ record, the value is mq; for a serial record, the value is seqws. The source of the value is the generation-time resource associations part; specifically, the association element for the file, property **fileType**.

User Response

Do as follows:

- Change the run-time fileType property in the run-time properties file or deployment descriptor; or
- Reset the fileType value at generation time and regenerate the program:
  - In the file-name-specific association element of the resource associations part, change the property **fileType**
  - In the build descriptor used at generation, set the option **genProperties** to GLOBAL

  For other details, see the EGL help pages on the association element, on Java run-time properties, and on setting up the environment.

EGL Java run-time error code VGJ0754E

VGJ0754E: The record length item must contain a value that splits non-character data at item boundaries.

Explanation

The record has a variable length. When its data is written out, the record length item indicates how many bytes to write. The last byte of data must be the last byte of an item, unless the item is a char.

User Response

Change the program so that the record length item's value points to the last byte of an item, or falls within a char item.

EGL Java run-time error code VGJ0755E

VGJ0755E: The value in the occursItem or lengthItem is too big.

Explanation

The record has a variable length. An attempt has been made to write out more bytes than the record currently contains.

User Response

Change the program so that the value of the lengthItem or occursItem is within the size of the record.

EGL Java run-time error code VGJ0770E

VGJ0770E: An error occurred while creating the InitialContext or looking up the java:comp/env environment. The error was %1

Explanation

The exception was either thrown from the constructor of javax.naming.InitialContext, or from invoking the lookup method with the value ″java:comp/env″. The program needs to create the InitialContext object and look up ″java:comp/env″ in order to access the J2EE environment settings.

User Response

Use the text of the exception and the documentation of your J2EE environment to correct the problem.

EGL Java run-time error code VGJ0800E

VGJ0800E: The assignment of %1 to %2 is invalid.

Explanation

While using the debugger, you attempted to set a system variable to an invalid value.

User Response

Choose a valid value, as described in the help page for the system variable.

EGL Java run-time error code VGJ0801E

VGJ0801E: %1 cannot be modified or does not exist.

Explanation

When using the debugger, you attempted to set the value of a system variable that cannot be set or does not exist.

User Response

Review the help pages for a list of system variables and for a description of each.

EGL Java run-time error code VGJ0802E

VGJ0802E: Error debugging %1: %2

Explanation

An error occurred while attempting to debug a PageHandler.

User Response

Use the Error part of the message to diagnose and correct the problem.

EGL Java run-time error code VGJ0901E

VGJ0901E: The date/time span pattern (character string that declares the length and date/time components for a timeStamp/interval item) is invalid.

Explanation

The date/time span pattern (character string that declares the length and date/time components for a timeStamp/interval item) is invalid.

User Response

Adjust the date/time span pattern accordingly.

EGL Java run-time error code VGJ0902E

VGJ0902E: The precision of the date/time span pattern (character string that declares the length and date/time components for a timeStamp/interval item) is invalid.

Explanation

The precision date/time of the span pattern (character string that declares the length and date/time components for a timeStamp/interval item) is invalid.

User Response

Adjust the precision of the date/time span pattern accordingly.

EGL Java run-time error code VGJ0903E

VGJ0903E: The start code of the date/time span pattern (character string that declares the length and date/time components for a timeStamp/interval item) is invalid.

Explanation

The start code of the date/time span pattern (character string that declares the length and date/time components for a timeStamp/interval item) is invalid.

User Response

Adjust the start code of the date/time span pattern accordingly.

EGL Java run-time error code VGJ0904E

VGJ0904E: The end code of the date/time span pattern (character string that declares the length and date/time components for a timeStamp/interval item) is invalid.

Explanation

The end code of the date/time span pattern (character string that declares the length and date/time components for a timeStamp/interval item) is invalid.

User Response

Adjust the end code of the date/time span pattern accordingly.

EGL Java run-time error code VGJ0905E

VGJ0905E: Either the start code or the end code of the date/time span pattern (character string that declares the length and date/time components for a timeStamp/interval item) is invalid.

Explanation

Either the start code or the end code of the date/time span pattern (character string that declares the length and date/time components for a timeStamp/interval item) is invalid.

User Response

Adjust either the start code or the end code of the date/time span pattern accordingly.

EGL Java run-time error code VGJ0906E

VGJ0906E: The INTERVAL value is invalid.

Explanation

The INTERVAL value is invalid.

User Response

Adjust the INTERVAL value accordingly.

EGL Java run-time error code VGJ0907E

VGJ0907E: The TIMESTAMP value is invalid.

Explanation

The TIMESTAMP value is invalid.

User Response

Adjust the TIMESTAMP value accordingly.

EGL Java run-time error code VGJ0908E

VGJ0908E: The TIME value is invalid.

Explanation

The TIME value is invalid.

User Response

Adjust the TIME value accordingly.

EGL Java run-time error code VGJ0909E

VGJ0909E: The DATE value is invalid.

Explanation

The DATE value is invalid.

User Response

Adjust the DATE value accordingly.

EGL Java run-time error code VGJ0910E

VGJ0910E: The BLOB or CLOB is out of memory.

Explanation

The BLOB or CLOB is out of memory.

User Response

Adjust the BLOB or CLOB size accordingly or associate it to file.

EGL Java run-time error code VGJ0911E

VGJ0911E: An internal error occurred during the execution of loadTable. %1

Explanation

An internal error occurred during the execution of loadTable.

User Response

For cause of error, see the extend error message.

EGL Java run-time error code VGJ0912E

VGJ0912E: An SQL error occurred during the execution of loadTable. %1

Explanation

An SQL error occurred during the execution of loadTable.

User Response

For cause of error, see the extend error message.

EGL Java run-time error code VGJ0913E

VGJ0913E: An I/O error occured during the execution of loadTable. %1

Explanation

An I/O error occured during the execution of loadTable.

User Response

For cause of error, see the extend error message.

EGL Java run-time error code VGJ0914E

VGJ0914E: An error occurred during the loading of the VGJSystemCommandProcessing system library. %1

Explanation

An error occurred during the loading of the VGJSystemCommandProcessing system library.

User Response

For cause of error, see the extend error message.

EGL Java run-time error code VGJ0915E

VGJ0915E: System error occurred while executing the system command %1. Check your system's path whether the command exists, it is executable, etc.

Explanation

An error occurred while executing the system command.

User Response

Check your system's path whether the command exists, it is executable, etc.

EGL Java run-time error code VGJ0916E

VGJ0916E: An internal error occurred during the execution of loadTable. %1

Explanation

An internal error occurred during the execution of loadTable.

User Response

For cause of error, see the extended error message.

EGL Java run-time error code VGJ0917E

VGJ0917E: An SQL error occurred during the execution of unloadTable. %1

Explanation

An SQL error occurred during the execution of unloadTable.

User Response

For cause of error, see the extended error message.

EGL Java run-time error code VGJ0918E

VGJ0918E: An I/O error occured during the execution of unloadTable. %1

Explanation

An I/O error occured during the execution of unloadTable.

User Response

For cause of error, see the extend error message.

EGL Java run-time error code VGJ0920E

VGJ0920E: An error has been encountered while returning %1 from native C function.

Explanation

The error occured while returning a value from C to EGL.

User Response

This is an internal error.

EGL Java run-time error code VGJ0921E

VGJ0921E: An error has been encountered while passing %1 to native C function.

Explanation

The error occurred while passing a value from EGL to C.

User Response

This is an internal error.

EGL Java run-time error code VGJ0922E

VGJ0922E: An error has been encountered while assigning value returned by native C function to %1.

Explanation

The error occurred while returning a value from C to EGL.

User Response

This is an internal error.

EGL Java run-time error code VGJ0923E

VGJ0923E: Value too large to fit in %1.

Explanation

The number exceeds limits of smallint or int receiving variable.

User Response

To store numbers that are outside the range of smallint or int, redefine the variable to use int or decimal type.

EGL Java run-time error code VGJ0924E

VGJ0924E: It is not possible to convert between the specified types.

Explanation

During native function calls, EGL attempts any data conversion that makes sense. Some conversions, however, are not supported, such as interval to date, timestamp to money etc.

User Response

Check that you have specified the data types that you intended.

EGL Java run-time error code VGJ0925E

VGJ0925E: The argument stack is empty.

Explanation

An empty stack exception has been encountered while passing values to a native C function or while returning values from it.

User Response

Check that the number of receiving variables does not exceed the number of values passed or returned.

EGL Java run-time error code VGJ0926E

VGJ0926E: Memory allocation failed.

Explanation

Something in the current native function call required the allocation of memory, but the memory was not available.

User Response

Several things can cause this error. For example, your application is asking for more resources that the system is configured to allow, or a problem with the operating system requires that you reboot the system.

EGL Java run-time error code VGJ0927E

VGJ0927E: Invalid datetime or interval qualifier.

Explanation

An invalid qualifier has been used while receiving a timestamp or interval value in the native C function.

User Response

Check that you have specified the qualifier that you intended.

EGL Java run-time error code VGJ0928E

VGJ0928E: Character host variable is too short for the data.

Explanation

A character host variable that is not large enough has been used while receiving a character string in the native C function.

User Response

Check size of the variable.

EGL Java run-time error code VGJ0929E

VGJ0929E: The native C function, %1, was not found.

Explanation

The specified C function was not found in the function table.

User Response

Add an entry for this function in the function table and recreate the shared library.

EGL Java run-time error code VGJ0930E

VGJ0930E: A loc_t structure has been improperly modified in the native C code.

Explanation

A Clob or Blob data type has been passed to a native C function, but the loc_t C structure in which it was received has been improperly changed.

User Response

Check whether loc_loctype, loc_type or loc_fname in the loc_t structure have been changed in the native C code.

EGL Java run-time error code VGJ0931E

VGJ0931E: An error has occured while processing a large object.

Explanation

The error occurred while performing some internal operation on a Clob or Blob data type.

User Response

This is an internal error.

EGL Java run-time error code VGJ0932E

VGJ0932E: The native C function, %1, has not returned the correct number of values expected by the calling function.

Explanation

If the function was invoked as part of an expression, then it returned more than one value. Otherwise the number of returned variables was different from the number of receiving variables.

User Response

Check that the correct function was called. Review the logic of the native C function, especially the values returned by it, to ensure that it always returns the expected number of values.

EGL Java run-time error code VGJ0933E

VGJ0933E: Intervals are incompatible for the operation.

Explanation

Some combinations of interval values are meaningless and are not allowed.

User Response

Review the interval data types being passed or returned for compatibility.

EGL Java run-time error code VGJ1000E

VGJ1000E: %1 failed. Invoking a method or accessing a field called %2 resulted in an unhandled error. The error message is %3

Explanation

The error ocurred in a Java access function. Either an Exception was thrown and the function was not called within a try statement or VGVar.handleSysLibraryErrors is 0, or something other than an Exception was thrown, such as an Error.

User Response

Use information in the error message to correct the problem. If some kind of Exception was thrown, change the program logic to handle the error by calling the Java access function within a try statement, or by setting VGVar.handleSysLibraryErrors to 1 before invoking the Java access function.

EGL Java run-time error code VGJ1001E

VGJ1001E: %1 failed. %2 is not an identifier, or it is the identifier of a null object.

Explanation

The error ocurred in a Java access function. The identifier cannot be used because it does not refer to a non-null object.

User Response

Use an identifier of a non-null object.

EGL Java run-time error code VGJ1002E

VGJ1002E: %1 failed. A public method, field, or class named %2 does not exist or cannot be loaded, or the number or types of parameters are incorrect. The error message is %3

Explanation

The method, field, or class used by a Java access function could not be found.

User Response

Do as follows:

- Make sure the target is a public method, field, or class.
- Make sure the name of the method, field, or class is correct. Class names must be qualified with the name of their package.
- If the problem is a missing class and the name is correct, make sure the directory or archive containing the class is in the Java classpath.
- If the problem is a missing method and the name is correct, make sure the types and number of parameters are correct. Compare the values passed to the Java access function with the values expected by the method.

EGL Java run-time error code VGJ1003E

VGJ1003E: %1 failed. The type of a value in EGL does not match the type expected in Java for %2. The error message is %3

Explanation

The type of a value passed to a Java access function is not correct.

User Response

Values assigned to fields, and parameters passed to methods and constructors, must have the proper type. An exact match is not required as long as the conversion between the types is valid in Java. For example, a subclass may be used instead of its superclass, and a smaller primitive type, such as short, may be used instead of a larger one, such as int.

EGL Java run-time error code VGJ1004E

VGJ1004E: %1 failed. The target is a method that returned null, a method that does not return a value, or a field whose value is null.

Explanation

The Java access function expected the result of the operation to be a non-null object, but did not get one.

User Response

To call a method that may return null or does not return a value, either use javaStore; or use the java system function and do not assign the result to an item. To get the value of a field that may be null, use javaStoreField.

EGL Java run-time error code VGJ1005E

VGJ1005E: %1 failed. The returned value does not match the type of the return item.

Explanation

The value returned by the Java access function cannot be assigned to the return item because of a type mismatch.

User Response

Change the program logic to use a return item of an appropriate type.

EGL Java run-time error code VGJ1006E

VGJ1006E: %1 failed. The class %2 of an argument cast to null could not be loaded. The error message is %3

Explanation

The class of the argument passed to the Java access function could not be found.

User Response

Do as follows:

- Make sure the name of the class is correct. Class names must be qualified with the name of a package.
- If the name is correct, make sure the directory or archive containing the class is in the Java classpath.

EGL Java run-time error code VGJ1007E

VGJ1007E: %1 failed. Could not get information about the method or field named %2, or an attempt was made to set the value of a field declared final. The error message is %3

Explanation

A SecurityException or IllegalAccessException was thrown while trying to get information about the method or field, or an attempt was made to set the value of a field declared final. Fields declared final cannot be modified.

User Response

Do as follows:

- If the problem happened when setting a value, change the program logic so the code does not try to set the value of a field declared final; alternatively, change the declaration of the field.
- If the problem was access to information, ask a system administrator to update the security policy file of the Java Virtual Machine so that your program has the necessary permission. The administrator probably needs to grant the ReflectPermission ″suppressAccessChecks″.

EGL Java run-time error code VGJ1008E

VGJ1008E: %1 failed. %2 is an interface or abstract class, so the constructor cannot be called.

Explanation

The constructor of an interface or abstract class cannot be called.

User Response

Change the program logic to call the constructor of a class that is not abstract.

EGL Java run-time error code VGJ1009E

VGJ1009E: %1 failed. The method or field %2 is not static. An identifier must be used instead of a class name.

Explanation

When a method or field is not declared static, it exists only in a specific instance of a class, not the class itself. An identifier of the object must be used in this case.

User Response

Change the program logic to use an identifier instead of a class name.

EGL Java run-time error code VGJ1148E

VGJ1148E: Action field ″%1″ does not exist.

Explanation

The current OnEvent action refers to a field that cannot be found.

User Response

Verify that the field exists in the current form.

EGL Java run-time error code VGJ1149E

VGJ1149E: Cannot insert another row - the input array is full.

Explanation

The variable used to hold the array data does not have space for another row.

User Response

Increase the storage size of the EGL variable.

EGL Java run-time error code VGJ1150E

VGJ1150E: Array ″%1″ not found.

Explanation

The specified array could not be found in the ConsoleForm.

User Response

Verify the array is correctly defined in the ConsoleForm and EGL program.

EGL Java run-time error code VGJ1151E

VGJ1151E: Assignment to prompt result variable failed.

Explanation

Assignment to prompt result variable failed.

User Response

Verify the result variable can hold the result from the prompt action.

EGL Java run-time error code VGJ1152E

VGJ1152E: Screen Array Field ″%1″ size is incorrect.

Explanation

The specified screen array field size is not correct.

User Response

Verify the definition of the screen array, and its usage in the EGL program.

EGL Java run-time error code VGJ1153E

VGJ1153E: DrawBox parameters are out of range.

Explanation

DrawBox parameters do not fix inside the current screen/window dimensions

User Response

Verify the parameters to the drawbox function, and the current window dimensions.

EGL Java run-time error code VGJ1154E

VGJ1154E: Display coordinates are outside the window boundaries.

Explanation

Display coordinates are outside the window boundaries.

User Response

Verify that the coordinates being used are within the size of the window.

EGL Java run-time error code VGJ1155E

VGJ1155E: Malformed key name ''%1''.

Explanation

The specified key name does not follow the key name convention.

User Response

Rewrite the key name to follow the EGL key name conventions.

EGL Java run-time error code VGJ1156E

VGJ1156E: You cannot use this editing feature because a picture exists.

Explanation

The picture attribute restricts the editing features for this field.

User Response

Use alternate editing keys and actions to obtain the desired results.

EGL Java run-time error code VGJ1157E

VGJ1157E: Cannot find window ''%1''.

Explanation

The window could not be located.

User Response

Verify that the window is properly defined and used.

EGL Java run-time error code VGJ1158E

VGJ1158E: New window position/dimension values are invalid.

Explanation

The specified position/dimension values are not valid for the current display environment.

User Response

Verify that the window position/dimensions are valid for the current display environment.

EGL Java run-time error code VGJ1159E

VGJ1159E: The command stack is out of synch.

Explanation

The statements being executed in the OnEvent clauses are causing EGL to become out of sync.

User Response

Verify the usage of statements/function calls within the OnEvent block statements.

EGL Java run-time error code VGJ1160E

VGJ1160E: The Console UI library is not initialized.

Explanation

An attempt was made to use the Console UI library before it was initialized.

User Response

Verify that the Console UI statement sequence is valid.

EGL Java run-time error code VGJ1161E

VGJ1161E: Illegal field type for construct.

Explanation

The field type specified in the console field is invalid for a construct query operation.

User Response

Verify that the field type in the console field is valid for construct query operations..

EGL Java run-time error code VGJ1162E

VGJ1162E: ConstructQuery cannot be called with a variable list.

Explanation

A Construct Query operation was invoked with a variable list.

User Response

Verify that the construct query operation is being invoked properly.

EGL Java run-time error code VGJ1163E

VGJ1163E: Cannot disable an invisible menu item.

Explanation

Attempt to hide an invisible menu item is an invalid operation.

User Response

Verify that the correct menu item to be disabled is not an invisible menu item.

EGL Java run-time error code VGJ1164E

VGJ1164E: Edit action failed.

Explanation

The specified edit action failed to execute.

User Response

Verify that the consolefield is properly defined, and the edit actions being performed are valid operations.

EGL Java run-time error code VGJ1165E

VGJ1165E: Error occurred while executing hotkey action.

Explanation

The hotkey operation failed to executed.

User Response

Verify that the specified hotkey is valid, and the statement block also valid.

EGL Java run-time error code VGJ1166E

VGJ1166E: There is no active command to exit from.

Explanation

Attempt was make to exit the current command, which does not exist.

User Response

Verify that the exit command is being used in the correct context.

EGL Java run-time error code VGJ1167E

VGJ1167E: There is no active command to continue.

Explanation

Attempt was made to continue the current command.

User Response

Verify that the continue statement is being used in the correct context.

EGL Java run-time error code VGJ1168E

VGJ1168E: Fatal error: %1

Explanation

A Fatal runtime error occurred.

User Response

Verify the Console UI statements are used in a proper context and sequence.

EGL Java run-time error code VGJ1169E

VGJ1169E: Field ″%1″ does not exist.

Explanation

The specified console field does not exist.

User Response

Verify that the console field has been properly defined in the console form.

EGL Java run-time error code VGJ1170E

VGJ1170E: Screen array field ″%1″ is not an array.

Explanation

The referenced console field in the console form is not an array.

User Response

Verify that the console field is defined as an array; verify that the correct console field is being referenced.

EGL Java run-time error code VGJ1171E

VGJ1171E: Field ″%1″ not found.

Explanation

The specified console field could not be found.

User Response

Verify that the console field has been properly defined in the console form.

EGL Java run-time error code VGJ1172E

VGJ1172E: Cannot create ConsoleField without a window.

Explanation

An attempt was made to create a console field outside of a consoleform/window context.

User Response

Verify the correctness of the consoleform and consolefield definitions.

EGL Java run-time error code VGJ1173E

VGJ1173E: Array field count mismatch.

Explanation

The Console UI array field specified does not match the referenced EGL array.

User Response

Verify the ConsoleField and array definition; verify the correct EGL array variable is being used on the openui statement.

EGL Java run-time error code VGJ1174E

VGJ1174E: Form ''%1'' does not exist.

Explanation

The specified console form does not exist.

User Response

Verify that the specified console form is defined and used in the correct context.

EGL Java run-time error code VGJ1175E

VGJ1175E: Form ''%1'' does not fit in window ''%2''.

Explanation

The form has dimensions that make it unable to fit into the current window dimensions.

User Response

Alter the dimensions of either the form definition or the window defintion.

EGL Java run-time error code VGJ1176E

VGJ1176E: Field lists do not match.

Explanation

The specified field list do not contain the same number of items as the variable list that was supplied.

User Response

Alter the openUI statement to ensure that the same number of fields and variables are specified.

EGL Java run-time error code VGJ1177E

VGJ1177E: Form ″%1″ is busy.

Explanation

The form reference is currently already being used in another context.

User Response

Verify that the EGL program logic uses a form only once at a time.

EGL Java run-time error code VGJ1178E

VGJ1178E: Form name ″%1″ already used.

Explanation

The definition of the form resulted in a form name conflict.

User Response

Alter the Form definition to use a unique form name.

EGL Java run-time error code VGJ1179E

VGJ1179E: Form ″%1″ is not open.

Explanation

An attempt was made to reference a form object which is not defined.

User Response

Verify that the specified form is properly defined, and used in valid ConsoleUI statements.

EGL Java run-time error code VGJ1180E

VGJ1180E: Cannot create ConsoleForm without a window.

Explanation

An attempt was made to create ConsoleForm without a valid window reference.

User Response

Verify that the ConsoleForm is properly defined, and used within a valid ConsoleUI statement.

EGL Java run-time error code VGJ1181E

VGJ1181E: Cannot use KeyObject.getChar() for virtual keys.

Explanation

The consoleUI cannot use KeyObject.getChar() for virtual keys.

User Response

Alter the EGL program to construct Strings for virtual key definitions.

EGL Java run-time error code VGJ1182E

VGJ1182E: Cannot use KeyObject.getCookedChar() for virtual keys.

Explanation

The ConsoleUI cannot use KeyObject.getCookedChar() for virtual keys.

User Response

Alter the EGL program to use Strings to define virtual keys.

EGL Java run-time error code VGJ1183E

VGJ1183E: Retrieving prompt result string failed.

Explanation

Retrieving prompt result string failed.

User Response

EGL Java run-time error code VGJ1184E

VGJ1184E: Help message key ″%1″ not found in resource bundle ″%2″.

Explanation

The help message key could not be located within the specified message help file.

User Response

Verify that the correct help message key and help message file are being used.

EGL Java run-time error code VGJ1185E

VGJ1185E: Illegal array subscript.

Explanation

An attempt was made to reference an invalid array element.

User Response

Verify that the program logic is referencing array elements within the size of the defined array.

EGL Java run-time error code VGJ1186E

VGJ1186E: Cannot initialize Console UI library.

Explanation

At program startup, the Console UI library could not be initialized.

User Response

Verify that the program is being used in a supported display environment and platform.

EGL Java run-time error code VGJ1187E

VGJ1187E: INTERNAL ERROR

Explanation

A ConsoleUI INTERNAL ERROR was encountered.

User Response

EGL Java run-time error code VGJ1188E

VGJ1188E: An INTERRUPT signal was received.

Explanation

An INTERRUPT signal was received.

User Response

EGL Java run-time error code VGJ1189E

VGJ1189E: Cannot have invisible menu item with no accelerator.

Explanation

An attempt was made to create an invisible menu item with no accelerator key.

User Response

Alter the menu item definition to define an accelerator key for the invisible menu item.

EGL Java run-time error code VGJ1190E

VGJ1190E: Cannot create a ConsoleLabel without a window.

Explanation

During the creation of the ConsoleLabel, a valid window reference could not be located.

User Response

Verify that the console label is being correctly defined in the console form, and the console form correctly used in the EGL program.

EGL Java run-time error code VGJ1191E

VGJ1191E: Menu item %1 does not fit in window.

Explanation

The specified menu item is too large to fit into the current active window

User Response

Alter the menu item so that the name is smaller than the current active window width dimension.

EGL Java run-time error code VGJ1192E

VGJ1192E: Menu item ''%1'' does not exist.

Explanation

The specified menu item could not be found or does not exist.

User Response

Verify that the referenced menu item has been defined and added to the current menu instance.

EGL Java run-time error code VGJ1193E

VGJ1193E: Menu mnemonics conflict (key=%1).

Explanation

The current menu item definitions result in a mnemonic conflict.

User Response

Alter the menu items to ensure that the accelerator/OnEvent keys do not conflict.

EGL Java run-time error code VGJ1194E

VGJ1194E: There is no active form.

Explanation

The console UI does not have an active form reference.

User Response

Verify that a form has been defined an displayed.

EGL Java run-time error code VGJ1195E

VGJ1195E: Must have an active form for DISPLAY ARRAY.

Explanation

An attempt was made to display an array from the current active form which does not exist.

User Response

Verify that a form has been defined with an array before attempting to display the array.

EGL Java run-time error code VGJ1196E

VGJ1196E: Must have an active form for READ ARRAY.

Explanation

An attempt was made to read an array from the active form, which does not exist.

User Response

Verify that a form has been defined and made active before attempting to read an array from it.

EGL Java run-time error code VGJ1197E

VGJ1197E: Cannot start event loop with no current command.

Explanation

Cannot start event loop with no current command.

User Response

EGL Java run-time error code VGJ1198E

VGJ1198E: No blob editor was specified.

Explanation

An attempt was made to edit a blob, but no blob editor was specified.

User Response

Define an appropriate editor in the blob console field.

EGL Java run-time error code VGJ1199E

VGJ1199E: INTERNAL ERROR: No format object

Explanation

INTERNAL ERROR: No format object

User Response

EGL Java run-time error code VGJ1200E

VGJ1200E: No Help File was specified.

Explanation

A help request was received, but no help file was specified.

User Response

Define a valid help file in the EGL program.

EGL Java run-time error code VGJ1201E

VGJ1201E: No Help message was specified.

Explanation

A help request was received, but no help message was specified.

User Response

Alter the EGL program to supply help messages.

EGL Java run-time error code VGJ1202E

VGJ1202E: Menu is not laid out.

Explanation

An attempt was made to use Menu functions which has not been displayed.

User Response

Verify that the menu functions are used after the menu has been displayed.

EGL Java run-time error code VGJ1203E

VGJ1203E: There is no current screen array.

Explanation

A reference was made to use the current screen array, which does not exist.

User Response

Verify that the current active form contains a screen array.

EGL Java run-time error code VGJ1204E

VGJ1204E: No menu items are visible.

Explanation

During the construction of a menu, no menu items were found to be visible.

User Response

Alter the menu creation so that at least one menu item is visible and displayable.

EGL Java run-time error code VGJ1205E

VGJ1205E: The name for the new window was null.

Explanation

The declaration for the window was null.

User Response

Supply a window name when declaring a window.

EGL Java run-time error code VGJ1206E

VGJ1206E: Attempt to open a null window.

Explanation

An attempt was made to open a window which does not exist or is empty.

User Response

Verify the open window statement is using a valid window reference.

EGL Java run-time error code VGJ1207E

VGJ1207E: An exception occurred in the prompt.

Explanation

During the execution of a prompt, an exception occurred.

User Response

Verify the prompt OnEvent statement block is correct.

EGL Java run-time error code VGJ1208E

VGJ1208E: A QUIT signal was received.

Explanation

A QUIT signal was received.

User Response

EGL Java run-time error code VGJ1209E

VGJ1209E: There is no active screen array.

Explanation

The current active form does not contain a screen array.

User Response

Verify that the program logic is using a form that contains a screen array definition.

EGL Java run-time error code VGJ1210E

VGJ1210E: There is no active form.

Explanation

The current Console UI session does not contain an active form instance.

User Response

Verify that a form is being defined and displayed before being referenced.

EGL Java run-time error code VGJ1211E

VGJ1211E: Menu cannot scroll to current item.

Explanation

The attempt to move the menu cursor to a menu item failed.

User Response

Verify that the menu logic is correctly moving the menu cursor to the correct menu item. Verify that the menu item is not disabled.

EGL Java run-time error code VGJ1212E

VGJ1212E: Unknown attribute ″%1″

Explanation

The specified attribute was not recognized.

User Response

Verify that the attribute is correct for the current Console UI context.

EGL Java run-time error code VGJ1213E

VGJ1213E: Error in field ″%1″.

Explanation

The input into the field is incorrect.

User Response

Verify that the typed in data matches the data type or format properties of the field.

EGL Java run-time error code VGJ1214E

VGJ1214E: Not enough variables were supplied.

Explanation

The openUI statement was not supplied with enough variables to bind to the console form.

User Response

Alter the EGL program to list more variables for the openUI statement; alter the openUI statement to restrict the number of consolefields.

EGL Java run-time error code VGJ1215E

VGJ1215E: Window name ''%1'' is already used.

Explanation

The newly defined window name is already used by another window.

User Response

Alter the name of the window to not conflict with other window names.

EGL Java run-time error code VGJ1216E

VGJ1216E: Window size is too small for help screen.

Explanation

An attempt to display the help screen into a display environment which is too small.

User Response

Adjust the size of the display environment.

EGL Java run-time error code VGJ1217W

VGJ1217W: There are no more fields in the direction you are going.

Explanation

Attempt to move the cursor past the end of the field list.

User Response

EGL Java run-time error code VGJ1218E

VGJ1218E: Screen array '%1' contents are invalid.

Explanation

An on-screen arrayDictionary contains one entry for each column in the display. All entries must be the same type of object: either a ConsoleField or an array of ConsoleFields, and all arrays (if any) must have the same number of elements.

Examine and correct the declaration of the on-screen arrayDictionary.

EGL Java run-time error code VGJ1290E

VGJ1290E: %1 is not a valid parameter for the Blob/Clob function.

Explanation

An error occurred while processing a Blob/Clob function. The cause of the error is described in the message insert.

User Response

Take appropriate action based on the content of the message insert.

EGL Java run-time error code VGJ1301E

VGJ1301E: Report Fill Error %1.

Explanation

Report Fill Error. The data provided to the report is not correct. The reasons could be that the dynamic array record field names do not match the report field names, the connection does not exist or the SQL statement is invalid.

User Response

If you are using a dynamic array of records, please ensure that the field names defined in the report design match the elements in the record by name. If you are using a SQL statement, ensure that the SQL is valid. If you are using a connection, ensure that the connection has been established and the connection name is correct. In addition, make sure that the pathname specified for reportDesignFile is valid and that the file exists.

EGL Java run-time error code VGJ1302E

VGJ1302E: Report Export Error %1.

Explanation

Report Export Error. The report could not be exported to the specified format.

User Response

Ensure that the pathnames are correct. the filled report object exists in the specified location and is correctly assigned in the reportDestinationFile field.

EGL Java run-time error code VGJ1303E

VGJ1303E: Report Dynamic Access Error, content not found. %1

Explanation

The field name does not exist in the dynamic array of records.

User Response

Ensure that the field names match both in the report design and in the record that you are using in the EGL program.

EGL Java run-time error code VGJ1304E

VGJ1304E: Incorrect connection name

Explanation

The connection name is invalid.

User Response

Ensure that the connection is a valid EGL connection and the defineDatabaseAlias function has been used to assign a connection a name.

EGL Java run-time error code VGJ1305E

VGJ1305E: Connection with specified name %1 does not exist

Explanation

A connection with the connection name does not exist.

User Response

Ensure that the following statements arae present in the EGL program. A connect function with valid parameters and a defineDatabaseAlias giving the connection a name.

EGL Java run-time error code VGJ1306E

VGJ1306E: Incorrect EGL and Report type mapping. Check the mapping table.

Explanation

There is a type mismatch between the fields in the Report Design and the data types in the EGL program.

User Response

Ensure that the types are compatible as mentioned in the documentation. Some examples, for a EGL char type, the design file should have the class defined as java.lang.String, for an EGL int type, the design file should have the field class as java.lang.Integer.

EGL Java run-time error code VGJ1401E

VGJ1401E: Field ''%1'' at position(%2,%3) does not lie within the form.

Explanation

The specified field does not lie within the form at the given position.

User Response

Verify that the form and fields are correctly defined.

EGL Java run-time error code VGJ1402E

VGJ1402E: Field ''%1'' overlaps ''%2''.

Explanation

The size and position of the two fields causes them to overlap.

User Response

Adjust the size and position coordinates of the fields.

EGL Java run-time error code VGJ1403E

VGJ1403E: Internal error: Cannot determine form group.

Explanation

Internal error: Cannot determine form group.

User Response

Verify that the form and formgroup are correctly defined.

EGL Java run-time error code VGJ1404E

VGJ1404E: Form ''%1'' does not fit in any floating area.

Explanation

The form does not fit in any floating area.

User Response

Verify that the form can be properly displayed in a floating area.

EGL Java run-time error code VGJ1405E

VGJ1405E: Field ''%1'' coordinates are invalid.

Explanation

The field coordinates are invalid.

User Response

Verify that the specified field coordinates are valid for the form.

EGL Java run-time error code VGJ1406E

VGJ1406E: Cannot get print association.

Explanation

The attempt to setup an print association failed.

User Response

Verify that the printer association is setup correctly.

EGL Java run-time error code VGJ1407E

VGJ1407E: No suitable print device size exists.

Explanation

No suitable print device size exists.

User Response

EGL Java run-time error code VGJ1408E

VGJ1408E: Printer '%1' was not found.\nThese printers are available:\n%2

Explanation

The user attempted to print to a specific printer device, which was not found in the system.

User Response

Examine the printer configuration in the environment. Make sure the printer exists, or print to another printer.

EGL Java run-time error code VGJ1409E

VGJ1409E: No display device exists for forms.

Explanation

No display device exists for forms.

User Response

Verify that the EGL program is being executed on a supported platform and display environment.

EGL Java run-time error code VGJ1410E

VGJ1410E: No compatible device size exists for displayed forms.

Explanation

No compatible device size exists for displayed forms.

User Response

Verify that the EGL program is being executed on a supported platform and display environment.

EGL Java run-time error code VGJ1411E

VGJ1411E: Help form class ″%1″ does not exist.

Explanation

The attempt to reference the help form class, which does not exist.

User Response

Verify that the help form class is defined and referenced correctly.

EGL Java run-time error code VGJ1412E

VGJ1412E: Unknown attribute ″%1″.

Explanation

The specified attribute was not recognized.

User Response

Verify that the correct attribute name is being used.

EGL Java run-time error code VGJ9900E

VGJ9900E: An error has occurred. The error was %1. Unable to load the error description.

Explanation

The program either could not locate or load both the default message class file and the message class file for your locale. One or both of these message class files may be missing or corrupt.

Note: During run time, this message can only be displayed in U.S. English because of the problem in loading message files.

User Response

If you have extracted class files from the file fda6.jar, verify that the classes you have are at the same release or maintenance level as the classes in the most recent file. If you are using older classes, replace them with the correct version. Also, you can reinstall fda6.jar from EGL.

If the problem persists, do as follows:

1. Record the message number and the message text.

   Note: The error message includes the following important information:
   - Where the error occurred
   - The type of internal error
2. Record the situation in which this message occurs.
3. For further instructions on how to report possible defects to the IBM Support Center, refer to the *EGL Installation Guide*.

EGL Java run-time error code VGJ9901E

VGJ9901E: An error has occurred. The error was %1. The message text for %1 could not be found in the message class file %2. The message text for VGJ0002E also could not be found.

Explanation

The message class file does not contain the run-time message for the message ID or for message ID VGJ0002E. The message class file is either corrupt or from a previous release of EGL.

Note: During run time, this message can only be displayed in U.S. English because of the problem in loading message files.

User Response

If you have extracted class files from the file fda6.jar, verify that the classes you have are at the same release or maintenance level as the classes in that file. If you are using older classes, replace them with the correct version. Also, you can reinstall fda6.jar from EGL.

If the problem persists, do as follows:

1. Record the message number and the message text.

   Note: The error message includes the following important information:
   - Where the error occurred
   - The type of internal error
2. Record the situation in which this message occurs.
3. For further instructions on how to report possible defects to the IBM Support Center, refer to the *EGL Installation Guide*.

Appendix. Notices

Note to U.S. Government Users Restricted Rights - Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Lab Director
IBM Canada Ltd. Laboratory
8200 Warden Avenue
Markham, Ontario, Canada L6G 1C7

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to

IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Programming interface information

Programming interface information is intended to help you create application software using this program.

General-use programming interfaces allow you to write application software that obtain the services of this program's tools.

However, this information may also contain diagnosis, modification, and tuning information. Diagnosis, modification and tuning information is provided to help you debug your application software.

Warning: Do not use this diagnosis, modification, and tuning information as a programming interface because it is subject to change.

Trademarks and service marks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

- AIX
- AS/400®
- CICS
- CICS/ESA®
- ClearCase
- Cloudscape™
- DB2
- DB2 Connect™
- DB2 Universal Database™
- Everyplace®
- IBM
- ibm.com
- iSeries
- OS/390®
- IMS
- Informix
- MQSeries
- MVS™
- OS/400

- RACF®
- Rational
- Rational Unified Process®
- SP2
- Support Pac
- SystemView®
- VisualAge
- WebSphere
- z/OS

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Intel is a trademark of Intel Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product or service names, may be trademarks or service marks of others.

Index

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

Program Number: 5724-D46

Printed in USA

SC31-6838-01