AS/400 Toolbox for Java

AS/400 Toolbox for Java

Contents

Chapter 1. AS/400 Toolbox for Java

Warning: Temporary Level 2 Header

What it is:

AS/400 Toolbox for Java is a set of Java classes that allow you to access AS/400 data through a Java program. With these classes, you can write client/server applications and applets that work with data that resides on your AS/400. You also can run your Java applications that use the AS/400 Toolbox for Java on the AS/400 Java Virtual Machine (JVM).

How it works:

AS/400 Toolbox for Java uses the AS/400 servers as access points to the system. Each server runs in a separate job on the AS/400, and each server job sends and receives data streams on a socket connection.

How to use these pages:

1. ▲ Use What's new for a summary of V4R4 new and changed AS/400 Toolbox for Java functions.
2. Use setting up for how to install and configure AS/400 Toolbox for Java. ▼
3. Use AS/400 Toolbox for Java access classes to access and manage resources on your AS/400.
4. Use AS/400 Toolbox for Java graphical user interface (GUI) classes to visually present and manipulate data.
5. ▲ Use the AS/400 Toolbox for Java utility class to do administrative tasks, such as using the AS400ToolBoxInstaller class.
6. Use AS/400 Toolbox for Java JavaBeans as reusable software components in your applications.
7. Use the Program Call Markup Language (PCML) to call AS/400 programs by writing less Java code.
8. Use the Graphical Toolbox to create your own GUI panels.
9. Use the Secure sockets layer to set up a secure connection with your AS/400. ▼
10. Follow our tips for programming with the AS/400 Toolbox for Java.

How to get additional information:

1. ▲ Use the tutorial to see code examples with detailed descriptions of what is happening in the example.
2. Use the examples section to navigate to the code examples that are provided throughout this document.
3. Use the Javadoc section for detailed information about each of the classes. ▼
4. See our related links section for more Java information.
5. ▲ Use download to download a PDF or zip file of the HTML files that comprise the AS/400 Toolbox for Java documentation. ▼

How to see what's new or changed:

To help you see where technical changes have been made, this information uses:

- The  image to mark where new or changed information begins.
- The  image to mark where new or changed information ends.

[Legal | AS/400 Glossary]

What's new for V4R4

The AS/400 Toolbox for Java Version 4 Release 4 (V4R4) licensed program is supported on V4R2 and later systems.

New visual classes

For V4R4 of the AS/400 Toolbox for Java, even more graphical user interface (GUI) classes have been added to the vaccess package.

AS/400 Toolbox for Java now provides GUI classes for the following resources:

- Jobs
- Spooled file viewer
- Permission
- System status
- System values
- Users and groups

Additional access classes

AS/400 Toolbox for Java V4R4 also features new classes in the access package. These new classes provide access to the following AS/400 resources:

- Data area
- JDBC 2.0 classes
- Message file (page 52)
- Permission
- System status
- System values

Additional new functions

AS/400 Toolbox for Java V4R4 also features the following new functions:

- Graphical Toolbox - contains tools that let you create custom GUI panels for your application
- Program Call Markup Language (PCML) - provides a way to define API parameters that let you call AS/400 programs with significantly less Java code
- JarMaker - allows you to customize JAR files
- Secure sockets layer - provides secure connections by encrypting the data exchanged between a client and an AS/400 server session

Performance enhancements

AS/400 Toolbox for Java V4R4 also features some performance enhancements (page).

Documentation enhancements

The documentation for AS/400 Toolbox for Java includes the following enhancements:

- Download - provides PDF and HTML packages that you can download to your PC
- Tutorial - provides code examples with explanations of important lines
- Examples - provides a collection of the code examples used throughout the information

Compatibility

If you intend to run a Java program that uses the AS/400 Toolbox for Java classes on the AS/400 Java Virtual Machine (JVM), you must run the AS/400 Toolbox for Java at the same version and release level as the Operating System/400 program that is running on your system. Shipped with OS/400 are parts of the AS/400 Toolbox for Java that are needed to improve performance when your application is running on the AS/400 JVM. To ensure compatibility, the level of the AS/400 Toolbox for Java must match the level of OS/400.

Level of OS/400	Compatible Level of AS/400 Toolbox for Java
V4R2	V3R2M0
V4R3	V3R2M1
V4R4	V4R2M0

How to see what's new or changed

To help you see where technical changes have been made, this information uses:

- The  image to mark where new or changed information begins.
- The image to mark where new or changed information ends.

[Legal | AS/400 Glossary]

Chapter 2. Setting up AS/400 Toolbox for Java

The AS/400 Toolbox for Java classes allow you to access AS/400 resources, data, and programs through Java applets and applications.

You must do the following tasks to install the AS/400 Toolbox for Java:

1. "Workstation requirements for AS/400 Toolbox for Java"
2. "OS/400 requirements for running AS/400 Toolbox for Java" on page 6
3. "Installing AS/400 Toolbox for Java on the AS/400" on page 8

You also need to consider the following:

- "Configuring an HTTP server for use with AS/400 Toolbox for Java" on page 8 if you want to use applets from an AS/400 that uses AS/400 Toolbox for Java classes served from the same AS/400.
- "Performance considerations related to installation location" on page 8 to understand when **significant performance impacts** may occur because of where and how you install the class files.
- "Copying the AS/400 Toolbox for Java class files on your workstation" on page 9 for information on copying files to your workstation.

Additional information on AS/400 Toolbox for Java:

All of the V4R4 Java information is provided on the *AS/400e Information Center*. This CD-ROM was shipped with your AS/400 system. The AS/400e series Information Center is also available at the following URL:

- http://publib.boulder.ibm.com/html/as400/infocenter.html 

Workstation requirements for AS/400 Toolbox for Java

To run AS/400 Toolbox for Java, your workstation must have the following:

- **For Java applications:**
  1. A Java Virtual Machine that fully supports JDK 1.1.6 or later. The following environments have been tested:
     - Windows 98
     - Windows 95
     - Windows NT Workstation 4.0
     - AIX Version 4.1.4.0
     - Sun Solaris Version 2.5
     - AS/400 Version 4 Release 4
     - OS/2 Warp Version 4.0
  2. TCP installed.
- **For Java applets:**
  1. A browser that fully supports JDK 1.1.6 or later. The following environments have been tested:
     - JavaSoft HotJava browser
     - Netscape Communicator 4.04 with the JDK 1.1 patch from http://developer.netscape.com

  - Netscape Communicator 4.05 with the JDK 1.1 patch built-in
  - Microsoft Internet Explorer 4.0.

2. TCP/IP installed.

- Java programs that use only the *access* classes of the AS/400 Toolbox for Java need only a Java Virtual Machine. Java programs that use the graphical user interface classes of AS/400 Toolbox for Java also need Sun Microsystems Java Swing 1.0.3 (Java Foundataion Classes (JFC)1.1). Download JFC 1.1 from http://java.sun.com/products/jfc/index.html . 

OS/400 requirements for running AS/400 Toolbox for Java

To run AS/400 Toolbox for Java, the AS/400 system to which you are connecting must have the following:

1. An AS/400 running one of the following:
   - OS/400 Version 4 Release 4
   - OS/400 Version 4 Release 3
   - OS/400 Version 4 Release 2

   If you intend to run a Java program that uses the AS/400 Toolbox for Java classes on the AS/400 Java Virtual Machine (JVM), you must run the AS/400 Toolbox for Java at the same version and release level as the Operating System/400 program that is running on your system. Shipped with OS/400 are the parts of the AS/400 Toolbox for Java needed to improve performance when your application is running on the AS/400 JVM. To ensure compatibility, the level of the AS/400 Toolbox for Java must match the level of OS/400.

Level of OS/400	Compatible Level of AS/400 Toolbox for Java
V4R2	V3R2M0
V4R3	V3R2M1
V4R4	V4R2M0

2. If you are going to use the spooled file viewer functions (SpooledFileViewer class) of the AS/400 Toolbox for Java, you must ensure that host option 8 (AFP Compatibility Fonts) is installed on your AS/400.

Note: SpooledFileViewer, PrintObjectPageInputStream, and PrintObjectTransformedInputStream classes work only when connecting to V4R4 or later systems.

3. Host Servers option of OS/400 installed and started on the AS/400.
   - The print support in AS/400 Toolbox for Java requires additional function in the OS/400 print server. You must have the appropriate PTF from the following list:
     - For V4R3, 5769SS1: PTF SF48498
     - For V4R2, 5769SS1: PTF SF46476
   - The JDBC driver requires a database server PTF. You must have the appropriate PTF from the following list:
     - For V4R2, 5769SS1: PTF SF46460

- The process and accuracy of retrieving sign-on server CCSIDs have been improved. These PTRs are not required, but they do improve performance:
  - For V4R3, 5769SS1: PTF SF1257
  - For V4R2, 5769SS1: PTF SF1256
- Ensure that the QUSER profile is enabled and has a valid password. To do this, enter **DSPUSRPRF USRPRF(QUSER)** on an AS/400 command line. The resulting display shows the status for QUSER.
- Start the OS/400 host servers by running two commands from an AS/400 command line:
  - **STRHOSTSVR** (Start Host Server)
  - **STRTCPSVR SERVER(*DDM)** (Start TCP/IP Server command with *DDM specified for the Server paramenter.
- For more information on host server options, see the TCP/IP topic in the *AS/400e Information Center*, which is also available at http://publib.boulder.ibm.com/html/as400/infocenter.html .

4. The TCP/IP Connectivity Utilities for AS/400 licensed program, 5769-TC1, is installed on the AS/400. For more information on TCP/IP, see the OS/400 TCP/IP Configuration and Reference, SC41-5420 .
5. If you are going to use the secure sockets layer (SSL), you need to have the following installed:
   - IBM HTTP Server licensed program, 5769-DG1
   - OS/400 Option 34 (Digital Certificate Manager)
   - One of the Cryptographic Access Provider licensed programs:
     - Cryptographic Access Provider (40-bit), 5769-AC1
     - Cryptographic Access Provider (56-bit), 5769-AC2
     - Cryptographic Access Provider (128-bit), 5769-AC3
   - One of the client encryption licensed programs:
     - AS/400 Client Encryption (40-bit), 5769-CE1
     - AS/400 Client Encryption (56-bit), 5769-CE2
     - AS/400 Client Encryption (128-bit), 5769-CE3

Note: You must install the same level of the Cryptogarphic Access Provider licensed program and AS/400 Client Encryption licensed program. In other words, 5769-AC1 and 5769-CE1 are a pair, 5769-AC2 and 5769-CE2 are a pair, and so on.

SSL connections perform slower than connections without encryption and can be only invoked from an SSL capable server, V4R4 or later.

For more information on SSL, see Secure sockets layer in the AS/400 Toolbox for Java topic of the *AS/400e Information Center*.

Note: Like the SpooledFileViewer, PrintObjectPageInputStream, and PrintObjectTransformedInputStream classes mentioned above, full Blob and Clob (JDBC) support and SSL are available only when connecting to V4R4 and later AS/400 systems.

Installing AS/400 Toolbox for Java on the AS/400

To install the AS/400 Toolbox for Java licensed program:

1. On the AS/400 command line, enter **GO LICPGM**.
2. Select **11. Install licensed program**.
3. Select **5769JC1 AS/400 Toolbox for Java**.

For more information on installing licensed programs, see the Software Installation book, SC41-5120. 

Configuring an HTTP server for use with AS/400 Toolbox for Java

If you want to use applets or the AS400ToolboxInstaller class, you must set up an HTTP server and install the class files on the AS/400 system. For more information on the IBM HTTP Server, see the IBM HTTP Server for AS/400 Webmaster's Guide, GC41-5434, at the following URL: http://www.as400.ibm.com/http . 

From this URL, take the Documentation link to a short list of books available on the IBM HTTP Server.

For information on the Digital Certificate Manager and how to create and work with digital certificates using the IBM HTTP Server, see the Getting started with IBM Digital Certificate Manager topic in the Internet section of the *AS/400e Information Center*.

Performance considerations related to installation location

You can install the AS/400 Toolbox for Java classes on your workstation or on the AS/400:

- In some cases, serving the classes from the workstation is a better solution than serving from the AS/400:
  - If a low-speed communication link connects the AS/400 and the workstation, the performance of loading the classes from the AS/400 to the workstation may be unacceptable.
  - If your Java application accesses classes via the CLASSPATH environment variable, you do not need a method of file redirection when the classes are on your workstation. If the classes are on the AS/400, you need a method of file redirection, such as Client Access for AS/400, to access the files on the AS/400.
- Choosing to install the AS/400 Toolbox for Java classes on your AS/400 gives you a centralized administration point for maintaining the classes.

Important performance tip regarding the CRTJVAPGM command:

If your Java program runs on the AS/400 Java Virtual Machine (JVM), you can **significantly improve performance** if you create an AS/400 Java program from the AS/400 Toolbox for Java zip file or jar file. Enter **CRTJVAPGM** on an AS/400 command line to create the program. You must run the **CRTJVAPGM** command at Level 30 to ensure proper protection of your program resources. (See the online help information for the **CRTJVAPGM** command for more information.) By using the CRTJVAPGM command, you save the AS/400 Java program that is created (and that contains the AS/400 Toolbox for Java classes) when your Java program starts.

Saving the AS/400 Java program that is created allows you to save startup processing time. You save startup processing time because the AS/400 Java program does not have to be re-created each time your Java program is started.

If you are using the V4R2 or V4R3 version of the AS/400 Toolbox for Java, you cannot run the **CRTJVAPGM** command against the jt400.zip file because it is too big. At V4R4, the AS/400 Toolbox for Java licensed program includes an additional file, jt400access.zip. jt400access.zip contains only the access classes, not the visual classes. If your Java program will be running on the AS/400, you should use jt400access.zip because you probably only need the access classes. The **CRTJVAPGM** command has already been run against jt400access.zip.

Copying the AS/400 Toolbox for Java class files on your workstation

Copying the class files to your workstation allows you to serve the files from your workstation. You can use the AS400ToolBoxInstaller class or rely on existing mechanisms for obtaining server updates on your workstation.

You can use either the jt400.zip file or the jt400.jar file on your workstation. (The jt400.jar file is smaller, but some tools and Java Virtual Machines (JVMs) accept only zip files. Use the file that works best for you.) The following instructions use the jt400.zip file, but these instructions also work for the jt400.jar file. To copy the files from the AS/400 to your workstation:

1. Decide what method you would like to use to copy files to your workstation. You can use the AS400ToolboxInstaller class or manually copy either the zip or jar file.
   - The AS/400 Toolbox for Java information fully documents the AS400ToolboxInstaller class. In the AS/400 Toolbox for Java information in the *AS/400e Information Center*, look under ″Tips for Programming″ and then ″Install and update.″ Or if you are viewing this information through the Information Center, see Client installation and update classes .
   - Find the file named jt400.zip. It should reside in the **/QIBM/ProdData/HTTP/Public/jt400/lib** directory. Copy jt400.zip from the AS/400 to your workstation. This can be accomplished in a variety of ways. The easiest way is to use Client Access/400 to map a network drive on your workstation to the AS/400. Another method is to use file transfer protocol (FTP) to send the file to your workstation (ensure that you transfer the file in binary mode).
2. Update the CLASSPATH environment variable of your workstation by adding the location where you put the program files. For example, on a personal computer (PC) that is using the Windows 95 operating system, if jt400.zip resides in **C:\jt400\lib\jt400.zip**, add **;C:\jt400\lib\jt400.zip** to the CLASSPATH variable.

Chapter 3. AS/400 Toolbox for Java access classes

The AS/400 Toolbox for Java access classes represent AS/400 data and resources. The classes work with AS/400 servers to provide an internet-enabled interface to access and update AS/400 data and resources.

The following classes provide access to AS/400 resources:

- AS400 - manages sign-on information, creates and maintains socket connections, sends and receives data
- Command call - runs AS/400 batch commands
- Data area - creates, accesses, and deletes data areas
- Data conversion and description - converts and handles data, and describes the record format of a buffer of data
- Data queues - creates, accesses, changes, and deletes data queues
- Digital certificates - manages digital certificates on AS/400
- Exceptions - throws errors when, for example, device errors or programming errors occur
- Integrated file system - accesses files, opens files, opens input and output streams, and lists the contents of directories
- Java Database Connectivity (JDBC) - accesses DB2 for AS/400 data
- Jobs - accesses AS/400 jobs and job logs
- Messages - accesses messages and message queues on the AS/400
- Network print - manipulates AS/400 print resources
- Permission - displays and changes authorities in AS/400 objects
- Program call - calls any AS/400 program
- QSYS object path name - represents objects in the AS/400 integrated file system
- Record-level access - creates, reads, updates, and deletes AS/400 files and members
- System status - displays system status information and allows access to system pool information
- System values - retrieves and changes system values and network attributes
- Trace (serviceability) - logs trace points and diagnostic messages
- Users and groups - accesses AS/400 users and groups
- User space - accesses an AS/400 user space

[Legal | AS/400 Glossary]

AS400 class

The AS400 class manages the following:

- A set of socket connections to the server jobs on the AS/400.
- Sign-on behavior for the AS/400. This includes prompting the user for sign-on information, password caching, and default user management.

The Java program must provide an AS400 object when the Java program uses an instance of a class that accesses the AS/400. For example, the CommandCall object requires an AS400 object before it can send commands to the AS/400.

The AS400 object handles connections, user IDs, and passwords differently when it is running in the AS/400 Java Virtual Machine. For more information, see AS/400 Java Virtual Machine.

See managing connections for information on managing connections to the AS/400 through the AS400 object.

AS400 class provides the following sign-on functions:

- Manage default user IDs
- Cache passwords
- Prompt for user ID
- Change a password
- Get the version and release of the AS/400

[Legal | AS/400 Glossary]

Managing default user IDs

To minimize the number of times a user has to sign on, use a default user ID. The Java program uses the default user ID when a the program does not provide a user ID. The default user ID can be set either by the Java program or through the user interface. If the default user ID has not been established, the Sign-On dialog allows the user to set the default user ID. Once the default user ID is established for a given AS/400, the Sign-On dialog does not allow the default user ID to be changed. When an AS400 object is constructed, the Java program can supply the user ID and password. When a program supplies the user ID to the AS400 object, the default user ID is not affected. The program must explicitly set the default user ID (setUseDefaultUser()) if the program wants to set or change the default user ID. See Prompting, default user ID, and password caching summary for more information.

The AS400 object has methods to get, set, and remove the default user ID. The Java program can also disable default user ID processing through the setUseDefaultUser() method. If default user ID processing is disabled and the Java application does not supply a user ID, the AS400 object prompts for user ID every time a connection is made to the AS/400 system.

All AS400 objects that represent the same AS/400 system within a Java Virtual Machine use the same default user ID.

In the following example, two connections to the AS/400 are created by using two AS400 objects. If the user checked the Default User ID box when signing on, the user is not prompted for a user ID when the second connection is made.

```
                    // Create two AS400 objects to the
                    // same AS/400.
AS400 sys1 = new AS400("mySystem.myCompany.com");
AS400 sys2 = new AS400("mySystem.myCompany.com");
                    // Start a connection to the command
                    // call service. The user is prompted
                    // for user ID and password.
sys1.connectService(AS400.COMMAND);
```

```
                    // Start another connection to the
                    // command call service. The user is
                    // not prompted.
sys2.connectService(AS400.COMMAND);
```

The default user ID information is discarded when the last AS400 object for an AS/400 system is garbage collected.

[Legal | AS/400 Glossary]

Using a password cache

The password cache allows the AS/400 Toolbox for Java to save password and user ID information so that it does not prompt the user for that information everytime a connection is made. Use the methods provided by the AS400 object to do the following:

- Clear the password cache and disable the password cache
- Minimize the number of times a user must type sign-on information

The password cache applies to all AS400 objects that represent an AS/400 system within a Java Virtual Machine. Java does not allow sharing information between virtual machines, so a cached password in one Java Virtual Machine is not visible to another virtual machine. The cache is discarded when the last AS400 object is garbage collected. The Sign-On dialog has a checkbox that gives the user the option to cache the password. When an AS400 object is constructed, the Java program has the option to supply the user ID and password. Passwords supplied on constructors are not cached.

The AS400 object provides methods to clear the password cache and disable the password cache . See Prompting, default user ID, and password caching summary for more information.

[Legal | AS/400 Glossary]

Prompting for user IDs and passwords

Prompting for user ID and password:

- May occur when connecting to the AS/400 system
- Can be turned off by your Java program

Java programs can turn off user ID and password prompting and message windows displayed by the AS400 object. An example of when this may be needed is when an application is running on a gateway on behalf of many clients. If prompts and messages are displayed on the gateway machine, the user has no way of interacting with the prompts. These types of applications can turn off all prompting by using the setGuiAvailable() method on the AS400 object.

See Prompting, default user ID, and password caching summary for more information.

[Legal | AS/400 Glossary]

Secure AS/400 Class

You can setup a secure AS/400 connection by creating an instance of a SecureAS400() object with or without prompting the user for sign-on information as indicated below:

- SecureAS400(SecureAS400(String systemName, String userID) prompts you for sign-on information
- SecureAS400(String systemName, String userID, String password) does not prompt you for sign-on information

The SecureAS400 class is a subclass of the AS400 class.

The following example shows you how to use CommandCall to send commands to the AS/400 system using a secure connection:

```
 // Create a secure AS400 object.  This is the only statement that changes
 // from the non-SSL case.
SecureAS400 sys = new SecureAS400("mySystem.myCompany.com");
// Create a command call object
CommandCall cmd = new CommandCall(sys, "myCommand");
// Run the commands.  A secure connection is made when the
// command is run.  All the information that passes between the
// client and server is encrypted.
cmd.run();
```

For more information, see secure sockets layer. 

[Legal | AS/400 Glossary]

Command call

The CommandCall class allows a Java program to call a non-interactive AS/400 command. Results of the command are available in a list of AS400 Message objects.

Input to CommandCall is as follows:

- The command string to run
- The AS400 object that represents the AS/400 system that will run the command

The command string can be set on the constructor, through the setCommand() method, or on the run() method. After the command is run, the Java program can use the getMessageList() method to retrieve any AS/400 messages resulting from the command.

Using the CommandCall class causes the AS400 object to connect to the AS/400.

The following example shows how to use the CommandCall class run a command on an AS/400 system:

```
                    // Create an AS400 object.
AS400 sys = new AS400("mySystem.myCompany.com");
                    // Create a command call object. This
                    // program sets the command to run
                    // later. It could set it here on the
                    // constructor.
CommandCall cmd = new CommandCall(sys);
                    // Run the CRTLIB command
cmd.run("CRTLIB MYLIB");
```

```
                    // Get the message list which
                    // contains the result of the
                    // command.
    AS400Message[] messageList = cmd.getMessageList();
                    // ... process the message list.
                    // Disconnect since I am done sending
                    // commands to the AS/400
    sys.disconnectService(AS400.COMMAND);
```

Using the CommandCall class causes the AS400 object to connect to the AS/400. See managing connections for information on managing connections.

Example

Run a command that is specified by the user.

[Legal | AS/400 Glossary]

Data area

The DataArea class is an abstract base class that represents an AS/400 data area object. This base class has four subclasses that support the following: character data, decimal data, logical data, and local data areas that contain character data.

Using the DataArea class, you can do the following:

- Get the size of the data area
- Get the name of the data area
- Return the AS/400 system object for the data area
- Refresh the attributes of the data area
- Set the system where the data area exists

Using the DataArea class causes the AS400 object to connect to the AS/400. See managing connections for information on managing connections.

CharacterDataArea

The CharacterDataArea class represents a data area on the AS/400 that contains character data. Character data areas do not have a facility for tagging the data with the proper CCSID; therefore, the data area object assumes that the data is in the user's CCSID. When writing, the data area object converts from a string (Unicode) to the user's CCSID before writing the data to the AS/400. When reading, the data area object assumes that the data is the CCSID of the user and converts from that CCSID to Unicode before returning the string to the program. When reading data from the data area, the amount of data read is by number of characters, not by the number of bytes.

Using the CharacterDataArea class, you can do the following:

- Clear the data area so that it contains all blanks.
- Create a character data area on the AS/400 system using default property values
- Create a character data area with specific attributes
- Delete the data area from the AS/400 system where the data area exists
- Return the integrated file system path name of the object represented by the data area.

- Read all of the data that is contained in the data area
- Read a specified amount of data from the data area starting at offset 0 or the offset that you specified
- Set the fully qualified integrated file system path name of the data area
- Write data to the beginning of the data area
- Write a specified amount of data to the data area starting at offset 0 or the offset that you specified

DecimalDataArea

The DecimalDataArea class represents a data area on the AS/400 that contains decimal data.

Using the DecimalDataArea class, you can do the following:

- Clear the data area so that it contains 0.0
- Create a decimal data area on the AS/400 system using default property values
- Create a decimal data area with specified attributes
- Delete the data area from the AS/400 system where the data area exists
- Return the number of digits to the right of the decimal point in the data area
- Return the integrated file system path name of the object represented by the data area.
- Read all of the data that is contained in the data area
- Set the fully qualified integrated file system path name of the data area
- Write data to the beginning of the data area

The following example shows how to create and to write to a decimal data area:

```
   // Establish a connection to the AS/400 "My400".
AS400 system = new AS400("My400");
   // Create a DecimalDataArea object.
QSYSObjectPathName path = new QSYSObjectPathName("MYLIB", "MYDATA", "DTAARA");
DecimalDataArea dataArea = new DecimalDataArea(system, path.getPath());
   // Create the decimal data area on the AS/400 using default values.
dataArea.create();
   // Clear the data area.
dataArea.clear();
   // Write to the data area.
dataArea.write(new BigDecimal("1.2"));
   // Read from the data area.
BigDecimal data = dataArea.read();
   // Delete the data area from the AS/400.
dataArea.delete();
```

LocalDataArea

The LocalDataArea class represents a local data area on the AS/400. A local data area exists as a character data area on the AS/400, but the local data area does have some restrictions of which you should be aware.

The local data area is associated with a server job and cannot be accessed from another job. Therefore, you cannot create or delete the local data area. When the server job ends, the local data area associated with that server job is automatically deleted, and the LocalDataArea object that is referring to the job is no longer valid. You should also note that local data areas are a fixed size of 1024 characters on the AS/400 system.

Using the LocalDataArea class, you can do the following:

- Clear the data area so that it contains all blanks
- Read all of the data that is contained in the data area
- Read a specified amount of data from the data area starting at offset that you specified
- Write data to the beginning of the data area
- Write a specified amount of data to the data area where the first character is written to offset

LogicalDataArea

The LogicalDataArea class represents a data area on the AS/400 that contains logical data.

Using the LogicalDataArea class, you can do the following:

- Clear the data area so that it contains false
- Create a character data area on the AS/400 system using default property values
- Create a character data area with specified attributes
- Delete the data area from the AS/400 system where the data area exists
- Return the integrated file system path name of the object represented by the data area.
- Read all of the data that is contained in the data area
- Set the fully qualified integrated file system path name of the data area
- Write data to the beginning of the data area

DataAreaEvent

The DataAreaEvent class represents a data area event.

You can use the DataAreaEvent class with any of the DataArea classes. Using the DataAreaEvent class, you can do the following:

- Get the identifier for the event

DataAreaListener

The DataAreaListener class provides an interface for receiving data area events.

You can use the the DataAreaListener class with any of the DataArea classes. You can invoke the DataAreaListener class when any of the following are performed:

- Clear
- Create
- Delete
- Read
- Write

[Legal | AS/400 Glossary]

Data conversion and description

The **data conversion** classes provide the capability to convert numeric and character data between AS/400 and Java formats. Conversion may be needed when accessing AS/400 data from a Java program. The data conversion classes support conversion of various numeric formats and between various EBCDIC code pages and unicode.

The **data description** classes build on the data conversion classes to convert all fields in a record with a single method call. The RecordFormat class allows the program to describe data that makes up a DataQueueEntry, ProgramCall parameter, a record in a database file accessed through record-level access classes, or any buffer of AS/400 data. The Record class allows the program to convert the contents of the record and access the data by field name or index.

Data types

The AS400DataType is an interface that defines the methods required for data conversion. A Java program uses data types when individual pieces of data need to be converted. Conversion classes exist for the following types of data:

- Numeric
- Text (character)
- Composite (numeric and text)

Conversion specifying a record format

The AS/400 Toolbox for Java provides classes for building on the data types classes to handle converting data one record at a time instead of one field at a time. For example, suppose a Java program reads data off a data queue. The data queue object returns a byte array of AS/400 data to the Java program. This array can potentially contain many types of AS/400 data. The application can convert one field at a time out of the byte array by using the data types classes, or the program can create a record format that describes the fields in the byte array. That record then does the conversion.

Record format conversion can be useful when you are working with data from the program call, data queue, and record-level access classes. The input and output from these classes are byte arrays that can contain many fields of various types. Record format converters can make it easier to convert this data between AS/400 format and Java format.

Conversion through record format uses three classes:

1. Field description classes identify a field or parameter with a data type and a name.
2. A record format class describes a group of fields.
3. A record class joins the description of a record (in the record format class) with the actual data.

Example

The data queue example shows how RecordFormat and Record can be used with the data queue classes.

Numeric conversion

Conversion classes for numeric data simply convert numeric data from AS/400 format to Java format. Supported types are shown in the following table:

Numeric Type	Description
AS400Bin2	Converts between a signed two-byte AS/400 number and a Java Short object.
AS400Bin4	Converts between a signed four-byte AS/400 number and a Java Integer object.
AS400ByteArray	Converts between two byte arrays. This is useful because the converter correctly zero-fills and pads the target buffer.
AS400Float4	Converts between a signed four-byte floating point AS/400 number and a Java Float object.
AS400Float8	Converts between a signed eight-byte floating point AS/400 number and a Java Double object.
AS400PackedDecimal	Converts between a packed-decimal AS/400 number and a Java BigDecimal object.
AS400UnsignedBin2	Converts between an unsigned two-byte AS/400 number and a Java Integer object.
AS400UnsignedBin4	Converts between an unsigned four-byte AS/400 number and a Java Long object.
AS400ZonedDecimal	Converts between a zoned-decimal AS/400 number and a Java BigDecimal object.

The following example shows conversion from an AS/400 numeric type to a Java int:

```
                    // Create a buffer to hold the AS/400
                    // type. Assume the buffer is filled
                    // with numeric AS/400 data by data
                    // queues, program call, etc.
byte[] data = new byte[100];
                    // Create a converter for this
                    // AS/400 data type.
AS400Bin4 bin4Converter = new AS400Bin4();
                    // Convert from AS/400 type to Java
                    // object. The number starts at the
                    // beginning of the buffer.
Integer intObject = (Integer) bin4Converter.toObject(data,0);
                    // Extract the simple Java type from
                    // the Java object.
int i = intObject.intValue();
```

The following example shows conversion from a Java int to an AS/400 numeric data type:

```
                    // Create a Java object that contains
                    // the value to convert.
Integer intObject = new Integer(22);
                    // Create a converter for the AS/400
                    // data type.
AS400Bin4 bin4Converter = new AS400Bin4();
                    // Convert from Java object to
                    // AS/400 type.
byte[] data = bin4Converter.toBytes(intObject);
                    // Find out how many bytes of the
```

```
                    // buffer were filled with the
                    // AS/400 value.
int length = bin4Converter.getByteLength();
```

[Legal | AS/400 Glossary]

Text conversion

Character data is converted through the AS400Text class. This class converts character data between an EBCDIC code page and character set (CCSID), and unicode. When the AS400Text object is constructed, the Java program specifies the length of the string to be converted and the AS/400 CCSID or encoding. The CCSID of the Java program is assumed to be unicode. The toBytes() method converts from Java form to byte array in AS/400 format. The toObject() method converts from a byte array in AS/400 format to Java format.

For example, assume that a DataQueueEntry object returns AS/400 text in EBCDIC. The following example converts this data to unicode so that the Java program can use it:

```
                    // ... Assume the data queues work
                    // has already been done to retrieve
                    // the text from the AS/400 and the
                    // data has been put in the
                    // following buffer.
int textLength = 100;
byte[] data = new byte[textLength];
                    // Create a converter for the AS/400
                    // data type. Note a default
                    // converter is being built.  This
                    // converter assumes the AS/400
                    // EBCDIC code page matches the
                    // client's locale.  If this is not
                    // true the Java program can
                    // explicitly specify the EBCDIC
                    // ccsid to use.
AS400Text textConverter = new AS400Text(textLength);
                    // Convert the data from EBCDIC to
                    // unicode.
String javaText = (String) textConverter.toObject(data);
```

[Legal | AS/400 Glossary]

Composite types

Conversion classes for composite types are as follows:

- AS400Array - Allows the Java program to work with an array of data types.
- AS400Structure - Allows the Java program to work with a structure whose elements are data types.

The following example shows conversion from a Java structure to a byte array and back again. The example assumes that the same data format is used for both sending and receiving data.

```
                    // Create a structure of data types
                    // that corresponds to a structure
                    // that contains:
                    //    - a four-byte number
                    //    - four bytes of pad
                    //    - an eight-byte number
                    //    - 40 characters
AS400DataType[] myStruct =
{
```

```
   new AS400Bin4(),
   new AS400ByteArray(4),
   new AS400Float8(),
   new AS400Text(40)
};
                  // Create a conversion object using
                  // the structure.
AS400Structure myConverter = new AS400Structure(myStruct);
                  // Create the Java object that holds
                  // the data to send to the AS/400.
Object[] myData =
{
     new Integer(88),        // the four-byte number
     new byte[0],            // the pad (let the conversion object 0 pad)
     new Double(23.45),      // the eight-byte floating point number
     "This is my structure"  // the character string
};
                  // Convert from Java object to byte array.
byte[] myAS400Data = myConverter.toBytes(myData);
                  // ... send the byte array to the
                  // AS/400. Get data back from the
                  // AS/400.  The returned data will
                  // also be a byte array.
                  // Convert the returned data from
                  // AS/400 to Java format.
Object[] myRoundTripData =
            (Object[])myConverter.toObject(myAS400Data,0);
                  // Pull the third object out of the
                  // structure. This is the double.
Double doubleObject = (Double) myRoundTripData[2];
                  // Extract the simple Java type from
                  // the Java object.
double d = doubleObject.doubleValue();
```

[Legal | AS/400 Glossary]

Field descriptions

The field description classes allow the Java program to describe the contents of a field or parameter with a data type and a string containing the name of the field. If the program is working with data from record-level access, it can also specify any AS/400 data definition specification (DDS) keywords that further describe the field.

The field description classes are as follows:

- BinaryFieldDescription
- CharacterFieldDescription
- DateFieldDescription
- DBCSEitherFieldDescription
- DBCSGraphicFieldDescription
- DBCSOnlyFieldDescription
- DBCSOpenFieldDescription
- FloatFieldDescription
- HexFieldDescription
- PackedDecimalFieldDescription
- TimeFieldDescription
- TimestampFieldDescription
- ZonedDecimalFieldDescription

For example, assume that the entries on a data queue have the same format. Each entry has a message number (AS400Bin4), a time stamp (8 characters), and message text (50 characters). These can be described with field descriptions as follows:

```
                    // Create a field description for
                    // the numeric data. Note it uses
                    // the AS400Bin4 data type. It also
                    // names the field so it can be
                    // accessed by name in the record
                    // class.
    BinaryFieldDescription bfd = new BinaryFieldDescription(new AS400Bin4(),
                                                            "msgNumber");
                    // Create a field description for
                    // the character data. Note it uses
                    // the AS400Text data type. It also
                    // names the field so it can be
                    // accessed by name by the record
                    // class.
    CharacterFieldDescription cfd1 = new CharacterFieldDescription(new AS400Text(8),
                                                            "msgTime");
                    // Create a field description for
                    // the character data. Note it uses
                    // the AS400Text data type. It also
                    // names the field so it can be
                    // accessed by name by the record
                    // class.
    CharacterFieldDescription cfd2 = new CharacterFieldDescription(new AS400Text(50),
                                                            "msgText");
```

The field descriptions are now ready to be grouped in a record format class. The example continues in the record format section.

[Legal | AS/400 Glossary]

Record format

The record format class allows the Java program to describe a group of fields or parameters. A record object contains data described by a record format object. If the program is using record-level access classes, the record format class also allows the program to specify descriptions for key fields.

A record format object contains a set of field descriptions. The field description can be accessed by index or by name. Methods exist on the record format class to do the following:

- Add field descriptions to the record format.
- Add key field descriptions to the record format.
- Retrieve field descriptions from the record format by index or by name.
- Retrieve key field descriptions from the record format by index or by name.
- Retrieve the names of the fields that make up the record format.
- Retrieve the names of the key fields that make up the record format.
- Retrieve the number of fields in the record format.
- Retrieve the number of key fields in the record format.
- Create a Record object based on this record format.

For example, to add the field descriptions created in the field description example to a record format:

```
                    // Create a record format object,
                    // then fill it with field
                    // descriptions.
    RecordFormat rf = new RecordFormat();
    rf.addFieldDescription(bfd);
    rf.addFieldDescription(cfd1);
    rf.addFieldDescription(cfd2);
```

The program is now ready to create a record from the record format. The example continues in the record section.

[Legal | AS/400 Glossary]

Record

The record class allows the Java program to process data described by the record format class. Data is converted between byte arrays containing the AS/400 data and Java objects. Methods are provided in the record class to do the following:

- Retrieve the contents of a field, by index or by name, as a Java object.
- Retrieve the number of fields in the record.
- Set the contents of a field, by index or by name, with a Java object.
- Retrieve the contents of the record as AS/400 data into a byte array or output stream.
- Set the contents of the record from a byte array or an input stream.
- Convert the contents of the record to a String.

For example, to use the record format created in the record format example:

```
                    // Assume data queue setup work has
                    // already been done. Now read a
                    // record from the data queue.
    DataQueueEntry dqe = dq.read();
                    // The data from the data queue is
                    // now in a data queue entry. Get
                    // the data out of the data queue
                    // entry and put it in the record.
                    // We obtain a default record from
                    // the record format object and
                    // initialize it with the data from the
                    // data queue entry.
    Record dqRecord = rf.getNewRecord(dqe.getData());
                    // Now that the data is in the
                    // record, pull the data out one
                    // field at a time, converting the
                    // data as it is removed. The result
                    // is data in a Java object that the
                    // program can now process.
    Integer msgNumber = (Integer) dqRecord.getField("msgNumber");
    String  msgTime   = (String)  dqRecord.getField("msgTime");
    String  msgText   = (String)  dqRecord.getField("msgText");
```

[Legal | AS/400 Glossary]

Retrieving the contents of a field

Retrieve the contents of a Record object by having your Java program either get one field at a time or get all the the fields at once. Use the getField() method to retrieve a single field by name or by index. Use the getFields() method to retrieve all of the fields as an Object[].

The Java program must cast the Object (or element of the Object[]) returned to the appropriate Java object for the retrieved field. The following table shows the appropriate Java object to cast based on the field type.

Field Type (DDS)	Field Type (FieldDescription)	Java Object
BINARY (B), length <=4	BinaryFieldDescription	Short
BINARY (B), length >=5	BinaryFieldDescription	Integer
CHARACTER (A)	CharacterFieldDescription	String
DBCS Either (E)	DBCSEitherFieldDescription	String
DBCS Graphic (G)	DBCSGraphicFieldDescription	String
DBCS Only (J)	DBCSOnlyFieldDescription	String
DBCS Open (O)	DBCSOpenFieldDescription	String
DATE (L)	DateFieldDescription	String
FLOAT (F), single precision	FloatFieldDescription	Float
FLOAT (F), double precision	FloatFieldDescription	Double
HEXADECIMAL (H)	HexFieldDescription	byte[]
PACKED DECIMAL (P)	PackedDecimalFieldDescription	BigDecimal
TIME (T)	TimeDecimalFieldDescription	String
TIMESTAMP (Z)	TimestampDecimalFieldDescription	String
ZONED DECIMAL (P)	ZonedDecimalFieldDescription	BigDecimal

[Legal | AS/400 Glossary]

Setting the contents of a field

Set the contents of a Record object by using the setField() method in your Java program. The Java program must specify the appropriate Java object for the field being set. The following table shows the appropriate Java object for each possible field type.

Field Type (DDS)	Field Type (FieldDescription)	Java Object
BINARY (B), length <=4	BinaryFieldDescription	Short
BINARY (B), length >=5	BinaryFieldDescription	Integer
CHARACTER (A)	CharacterFieldDescription	String
DBCS Either (E)	DBCSEitherFieldDescription	String
DBCS Graphic (G)	DBCSGraphicFieldDescription	String
DBCS Only (J)	DBCSOnlyFieldDescription	String
DBCS Open (O)	DBCSOpenFieldDescription	String
DATE (L)	DateFieldDescription	String
FLOAT (F), single precision	FloatFieldDescription	Float
FLOAT (F), double precision	FloatFieldDescription	Double
HEXADECIMAL (H)	HexFieldDescription	byte[]
PACKED DECIMAL (P)	PackedDecimalFieldDescription	BigDecimal
TIME (T)	TimeDecimalFieldDescription	String
TIMESTAMP (Z)	TimestampDecimalFieldDescription	String
ZONED DECIMAL (P)	ZonedDecimalFieldDescription	BigDecimal

Data queues

The DataQueue classes allow the Java program to interact with AS/400 data queues. AS/400 data queues have the following characteristics:

- The data queue allows for fast communications between jobs. Therefore, it is an excellent way to synchronize and pass data between jobs.
- Many jobs can simultaneously access the data queues.
- Messages on a data queue are free format. Fields are not required as they are in database files.
- The data queue can be used for either synchronous or asynchronous processing.
- The messages on a data queue can be ordered in one the following ways:
  - Last-in first-out (LIFO). The last (newest) message that is placed on the data queue is the first message that is taken off the queue.
  - First-in first-out (FIFO). The first (oldest) message that is placed on the data queue is the first message that is taken off the queue.
  - Keyed. Each message on the data queue has a key associated with it. A message can be taken off the queue only by specifying the key that is associated with it.

The data queue classes provide a complete set of interfaces for accessing AS/400 data queues from your Java program. It is an excellent way to communicate between Java programs and AS/400 programs that are written in any programming language.

A required parameter of each data queue object is the AS400 object that represents the AS/400 system that has the data queue or where the data queue is to be created.

Using the data queue classes causes the AS400 object to connect to the AS/400. See managing connections for information about managing connections.

Each data queue object requires the integrated file system path name of the data queue. The type for the data queue is DTAQ. See integrated file system path names for more information.

Sequential and keyed data queues

The data queue classes support the following data queues:

- Sequential data queues
- Keyed data queues

Methods common to both types of queues are in the BaseDataQueue class. The DataQueue class extends the BaseDataQueue class in order to complete the implementation of sequential data queues. The BaseDataQueue class is extended by the KeyedDataQueue class to complete the implementation of keyed data queues.

When data is read from a data queue, the data is placed in a DataQueueEntry object. This object holds the data for both keyed and sequential data queues.

Additional data available when reading from a keyed data queue is placed in a KeyedDataQueueEntry object that extends the DataQueueEntry class.

The data queue classes do not alter data that is written to or is read from the AS/400 data queue. The Java program must correctly format the data. The data conversion classes provide methods for converting data.

The following example creates a DataQueue object, reads data from the DataQueueEntry object, and then disconnects from the system.

```
                    // Create an AS400 object
AS400 sys = new AS400("mySystem.myCompany.com");
                    // Create the DataQueue object
DataQueue dq = new DataQueue(sys, "/QSYS.LIB/MYLIB.LIB/MYQUEUE.DTAQ");
                    // read data from the queue
DataQueueEntry dqData = dq.read();
                    // get the data out of the DataQueueEntry object.
byte[] data = dqData.getData();
                    // ... process the data
                    // Disconnect since I am done using data queues
sys.disconnectService(AS400.DATAQUEUE);
```

[Legal | AS/400 Glossary]

Sequential data queues

Entries on a sequential AS/400 data queue are removed in first-in first-out (FIFO) or last-in first-out (LIFO) sequence. The BaseDataQueue and DataQueue classes provide the following methods for working with sequential data queues:

- Create a data queue on the AS/400. The Java program must specify the maximum size of an entry on the data queue. The Java program can optionally specify additional data queue parameters (FIFO vs LIFO, save sender information, specify authority information, force to disk, and provide a queue description) when the queue is created.
- Peek at an entry on the data queue without removing it from the queue. The Java program can wait or return immediately if no entry is currently on the queue.
- Read an entry off the queue. The Java program can wait or return immediately if no entry is available on the queue.
- Write an entry to the queue.
- Clear all entries from the queue.
- Delete the queue.

The BaseDataQueue class provides additional methods for retrieving the attributes of the data queue.

Examples

Sequential data queue examples, in which the producer puts items on a data queue, and the consumer takes the items off the queue and processes them:

- Sequential data queue producer example.
- Sequential data queue consumer example.

[Legal | AS/400 Glossary]

Keyed data queues

The BaseDataQueue and KeyedDataQueue classes provide the following methods for working with keyed data queues:

- Create a keyed data queue on the AS/400. The Java program must specify key length and maximum size of an entry on the queue. The Java program can optionally specify authority information, save sender information, force to disk, and provide a queue description.
- Peek at an entry based on the specified key without removing it from the queue. The Java program can wait or return immediately if no entry is currently on the queue that matches the key criteria.
- Read an entry off the queue based on the specified key. The Java program can wait or return immediately if no entry is available on the queue that matches the key criteria.
- Write a keyed entry to the queue.
- Clear all entries or all entries that match a specified key.
- Delete the queue.

The BaseDataQueue and KeyedDataQueue classes also provide additional methods for retrieving the attributes of the data queue.

Examples

In the following keyed data queue examples, the producer puts items on a data queue, and the consumer takes the items off the queue and processes them:

- Keyed data queue producer example
- Keyed data queue consumer example

[Legal | AS/400 Glossary]

Digital certificates

Digital certificates are digitally-signed statements used for secured transactions over the internet. (Digital certificates can be used on AS/400 systems running on Version 4 Release 3 (V4R3) and later.) To make a secure connection using the secure sockets layer (SSL), a digital certificate is required.

Digital certificates comprise the following:

- The public encryption key of the user
- The name and address of the user
- The digital signature of a third-party certification authority (CA). The authority's signature means that the user is a trusted entity.
- The issue date of the certificate
- The expiration date of the certificate

As an administrator of a secured server, you can add a certification authority's "trusted root key" to the server. This means that your server will trust anyone who is certified through that particular certification authority.

Digital certificates also offer encryption, ensuring a secure transfer of data through a private encryption key.

You can create digital certificates through the javakey tool. (For more information about javakey and Java security, see the Sun Microsystems, Inc., Java Security page at http://java.sun.com/security/index.html  .) The AS/400 Toolbox for Java licensed program has classes that administer digital certificates on an AS/400.

The AS/400 Digital Certificate classes provide methods to manage X.509 ASN.1 encoded certificates. Classes are provided to do the following:

- Get and set certificate data.
- List certificates by validation list or user profile.
- Manage certificates, for example, add a certificate to a user profile or delete a certificate from a validation list.

Using a certificate class causes the AS400 object to connect to the AS/400. See managing connections for information about managing connections.

On the AS/400, certificates belong to a validation list or to a user profile.

- The AS400CertificateUserProfileUtil class has methods for managing certificates on a user profile.
- The AS400CertificateVldlUtil class has methods for managing certificates in a validation list.

These two classes extend AS400CertificateUtil, which is an abstract base classes that defines methods common to both subclasses.

The AS400Certificate class provides methods to read and write certificate data. Data is accessed as an array of bytes. The Java.Security package in JVM 1.2 provides classes that can be used to get and set individual fields of the certificate.

Listing certificates

To get a list of certificates, the Java program must do the following:

1. Create an AS400 object.
2. Construct the correct certificate object. Different objects are used for listing certificates on a user profile (AS400CertificateUserProfileUtil) versus listing certificates in a validation list (AS400CertificateVldlUtil).
3. Create selection criteria based on certificate attributes. The AS400CertificateAttribute class contains attributes used as selection criteria. One or more attribute objects define the criteria that must be met before a certificate is added to the list. For example, a list might contain only certificates for a certain user or organization.
4. Create a user space on the AS/400 and put the certificate into the user space. Large amounts of data can be generated by a list operation. The data is put into a user space before it can be retrieved by the Java program. Use the listCertificates() method to put the certificates into the user space.
5. Use the getCertificates() method to retrieve certificates from the user space.

The following example lists certificates in a validation list. It lists only those certificates belonging to a certain person.

```
                    // Create an AS400 object.  The
                    // certificates are on this system.
AS400 sys = new AS400("mySystem.myCompany.com");
                    // Create the certificate object.
AS400CertificateVldlUtil certificateList =
```

```
            new AS400CertificateVldlUtil(sys, "/QSYS.LIB/MYLIB.LIB/CERTLIST.VLDL");
                    // Create the certificate attribute
                    // list. We only want certificates
                    // for a single person so the list
                    // consists of only one element.
    AS400CertificateAttribute[] attributeList = new AS400CertificateAttribute[1];
    attributeList[0] = new AS400CertificateAttribute(AS400CertificateAttribute.SUBJECT_COMMON_NA
                    // Retrieve the list that matches
                    // the criteria. User space "myspace"
                    // in library "mylib" will be used
                    // for storage of the certificates.
                    // The user space must exist before
                    // calling this API.
    int count = certificateList.listCertificates(attributeList,
                                                 "/QSYS.LIB/MYLIB.LIB/MYSPACE.USRSPC");
                    // Retrieve the certificates from
                    // the user space.
    AS400Certificates[] certificates = certificateList.getCertificates("/QSYS.LIB/MYLIB.LIB/MYSP
                    // ... process the certificates
```

[Legal | AS/400 Glossary]

Exceptions

The AS/400 Toolbox for Java access classes throw exceptions when device errors, physical limitations, programming errors, or user input errors occur. The exception classes are based upon the type of error that occurs instead of the location where the error originates.

Each exception contains three pieces of information:

- **Error type** - The exception object that is thrown indicates the type of error that occurred. Errors of the same type are grouped together in an exception class. See the list of exceptions for more information about error types.
- **Error details** - The exception contains a return code to further identify the cause of the error that occurred. The return code values are constants within the exception class.
- **Error text** - The exception contains a text string that describes the error that occurred. The string is translated in the locale of the client Java Virtual Machine (JVM).

The following example shows how to catch a thrown exception, retrieve the return code, and display the exception text:

```
                    // ... all the setup work to delete
                    // a file on the AS/400 through the
                    // IFSFile class is done. Now try
                    // deleting the file.
    try
    {
       aFile.delete();
    }
                    // The delete failed.
    catch (ExtendedIOException e)
    {
                    // Display the translated string
                    // containing the reason that the
                    // delete failed.
       System.out.println(e);
                    // Get the return code out of the
                    // exception and display additional
                    // information based on the return
```

```
                    // code.
        int rc = e.getReturnCode()
        switch (rc)
        {
           case ExtendedIOException.FILE_IN_USE:
              System.out.println("Delete failed, file is in use "):
              break;
           case ExtendedIOException.PATH_NOT_FOUND:
              System.out.println("Delete failed, path not found ");
              break;
                       // ... for every specific error you
                       // want to track
           default:
              System.out.println("Delete failed, rc = ");
              System.out.println(rc);
        }
     }
```

See exceptions inheritance structure for more information about exceptions.

[Legal | AS/400 Glossary]

Exceptions thrown by the AS/400 Toolbox for Java access classes

The following table describes when various exceptions are thrown.

Exception	Description
AS400Exception.java	Thrown if the AS/400 system returns an error message.
AS400SecurityException.java	Thrown if a security or authority error occurs.
ConnectionDroppedException.java	Thrown if the connection is dropped unexpectedly.
ErrorCompletingRequestException.java	Thrown if an error occurs before the request is completed.
ExtendedIOException.java	Thrown if an error occurs while communicating with the AS/400.
ExtendedIllegalArgumentException.java	Thrown if an argument is not valid.
ExtendedIllegalStateException.java	Thrown if the AS/400 object is not in the proper state to perform the operation.
IllegalObjectTypeException.java	Thrown if the AS/400 object is not of the required type.
IllegalPathNameException.java	Thrown if an integrated file system path name is not valid.
InternalErrorException.java	Thrown if an internal problem occurs. When this type of exception is thrown, contact your service representative to report the problem.
ObjectAlreadyExistsException.java	Thrown if the AS/400 object already exists.
ObjectDoesNotExistException.java	Thrown if the AS/400 object does not exist.
RequestNotSupportedException.java	Thrown if the requested function is not supported because the AS/400 system is not at the correct level.
ReturnCodeException.java	An **interface** for exceptions that contain a return code. The return code is used to further identify the cause of an error.
ServerStartupException.java	Thrown if the AS/400 server cannot be started.

See Inheritance structure for exceptions for more information about exceptions thrown by the AS/400 Toolbox for Java.

[Legal | AS/400 Glossary]

Inheritance structure for exceptions

The exceptions that are thrown by AS/400 Toolbox for Java access classes inherit from Exception, IOException, or RuntimeException:

- class java.lang.Exception
  - AS400SecurityException
  - ErrorCompletingRequestException
    - AS400Exception
  - IllegalObjectTypeException
  - ObjectAlreadyExistsException
  - ObjectDoesNotExistException
  - RequestNotSupportedException
  - class java.io.IOException
    - ConnectionDroppedException
    - ExtendedIOException
    - ServerStartupException
  - class java.lang.RuntimeException
    - class java.io.IllegalArgumentException
      - ExtendedIllegalArgumentException
    - IllegalPathNameException
    - InternalErrorException
    - class java.lang.IllegalStateException
      - ExtendedIllegalStateException
  - class java.sql.SQLException

[Legal | AS/400 Glossary]

Integrated file system

The integrated file system classes allow a Java program to access files in the AS/400 integrated file system as a stream of bytes or a stream of characters. The integrated file system classes were created because the java.io package does not provide file redirection and other AS/400 functionality.

The function that is provided by the IFSFile classes is a superset of the function provided by the file IO classes in the java.io package. All methods in java.io FileInputStream, FileOutputStream, and RandomAccessFile are in the integrated file system classes.

In addition to these methods, the classes contain methods to do the following:

- Specify a file sharing mode to deny access to the file while it is in use
- Specify a file creation mode to open, create, or replace the file
- Lock a section of the file and deny access to that part of the file while it is in use
- List the contents of a directory more efficiently
- Determine the number of bytes available on the AS/400 file system
- Allow a Java applet to access files in the AS/400 file system

- Read and write data as text instead of as binary data

Through the integrated file system classes, the Java program can directly access stream files on the AS/400. The Java program can still use the java.io package, but the client operating system must then provide a method of redirection. For example, if the Java program is running on a Windows 95 or Windows NT operating system, the Network Drives function of AS/400 Client Access for 32-bit Windows is required to redirect java.io calls to the AS/400. With the integrated file system classes, you do not need Client Access for AS/400.

A required parameter of the integrated file system classes is the AS400 object that represents the AS/400 system that contains the file. Using the integrated file system classes causes the AS400 object to connect to the AS/400. See managing connections for information about managing connections.

The integrated file system classes require the hierarchical name of the object in the integrated file system. Use the forward slash as the path separator character. The following example shows how to access FILE1 in directory path DIR1/DIR2:

```
/DIR1/DIR2/FILE1
```

The integrated file system classes are as follows.

Class	Description
IFSFile	Represents an object in the integrated file system
 IFSJavaFile	Represents a file in the integrated file system
IFSFileInputStream	Represents an input stream for reading data from an AS/400 file
IFSTextFileInputStream	Represents a stream of character data read from a file
IFSFileOutputStream	Represents an output stream for writing data to an AS/400 file
IFSTextFileOutputStream	Represents a stream of character data being written to a file
IFSRandomAccessFile	Represents a file on the AS/400 for reading and writing data
IFSFileDialog	Allows the user to move within the file system and to select a file within the file system

Examples

The IFSCopyFile example shows how to use the integrated file system classes to copy a file from one directory to another on the AS/400.

The File List Example shows how to use the integrated file system classes to list the contents of a directory on the AS/400.

[Legal | AS/400 Glossary]

IFSFile

The IFSFile class represents an object in the AS/400 integrated file system. The methods on IFSFile represent operations that are done on the object as a whole.

You can use IFSFileInputStream, IFSFileOutputStream, and IFSRandomAccessFile to read and write to the file. The IFSFile class allows the Java program to do the following:

- Determine if the object exists and is a directory or a file
- Determine if the Java program can read from or write to a file
- Determine the length of a file
- Determine the permissions of an object and set the permissions of an object.
- Create a directory
- Delete a file or directory
- Rename a file or directory
- Get or set the last modification date of a file
- List the contents of a directory
- Determine the amount of space available on the AS/400 system

The following examples show how to use the IFSFile class.

Example 1: To create a directory:

```
                    // Create an AS400 object.  This new
                    // directory will be created on this
                    // AS/400.
AS400 sys = new AS400("mySystem.myCompany.com");
                    // Create a file object that
                    // represents the directory.
IFSFile aDirectory = new IFSFile(sys, "/mydir1/mydir2/newdir");
                    // Create the directory.
if (aDirectory.mkdir())
   System.out.println("Create directory was successful");
                    // Else the create directory failed.
else
{
                    // If the object already exists,
                    // find out if it is a directory or
                    // file, then display a message.
   if (aDirectory.exists())
   {
      if (aDirectory.isDirectory())
         System.out.println("Directory already exists");
      else
         System.out.println("File with this name already exists");
   }
   else
      System.out.println("Create directory failed");
}
                    // Disconnect since I am done
                    // accessing files.
sys.disconnectService(AS400.FILE);
```

Example 2: When an error occurs, the IFSFile class throws the ExtendedIOException exception. This exception contains a return code that indicates the cause of the failure. The IFSFile class throws the exception even when the java.io class that IFSFile duplicates does not. For example, the delete method from java.io.File returns a boolean to indicate success or failure. The delete method in IFSFile returns a boolean, but if the delete fails, an ExtendedIOException is thrown. The ExtendedIOException provides the Java program with detailed information about why the delete failed.

```
                    // Create an AS400 object.
AS400 sys = new AS400("mySystem.myCompany.com");
                    // Create a file object that
                    // represents the file.
IFSFile aFile = new IFSFile(sys, "/mydir1/mydir2/myfile");
                    // Delete the file.
try
{
   aFile.delete();
                    // The delete was successful.
   System.out.println("Delete successful ");
}
                    // The delete failed. Get the return
                    // code out of the exception and
                    // display why the delete failed.
catch (ExtendedIOException e)
{
   int rc = e.getReturnCode();
   switch (rc)
   {
      case ExtendedIOException.FILE_IN_USE:
         System.out.println("Delete failed, file is in use ");
         break;
      case ExtendedIOException.PATH_NOT_FOUND:
         System.out.println("Delete failed, path not found ");
         break;
                     // ... for every specific error
                     // you want to track.
      default:
         System.out.println("Delete failed, rc = ");
         System.out.println(rc);
   }
}
```

Example 3: The following example shows how to list files on the AS/400. A filter object is supplied so that only directories are listed.

```
                    // Create the AS400 object.
AS400 system = new AS400("mySystem.myCompany.com");
                    // Create the file object.
IFSFile directory = new IFSFile(system, "/");
                    // Generate a list of all
                    // subdirectories in the directory.
                    // It uses the filter defined below.
String[] DirNames = directory.list(new DirectoryFilter());
                    // Display the results.
if (subDirNames != null)
  for (int i = 0; i < subDirNames.length; i++)
    System.out.println(subDirNames[i]);
else
  System.out.println("No subdirectories.");
                    // Here is the filter. It keeps
                    // directories and discards files.
                    // The accept method is called for
                    // every directory entry in the list.
                    // If the element is a directory,
                    // 'true' is returned so the
                    // directory is returned. The results
                    // are returned in the string array
                    // returned to the list() method
                    // above.
class DirectoryFilter implements IFSFileFilter
{
   public boolean accept(IFSFile file)
```

```
    {
      return file.isDirectory();
    }
  }
```

Example 4: The Java program can optionally specify match criteria when listing files in the directory. Match criteria reduce the number of files that are returned by AS/400 to the IFSFile object, which improves performance. The following example shows how to list files with extension .txt:

```
                    // Create the AS400 object.
AS400 system = new AS400("mySystem.myCompany.com");
                    // Create the file object.
IFSFile directory = new IFSFile(system, "/");
                    // Generate a list of all files with
                    // extension .txt
String[] names = directory.list("*.txt");
                    // Display the names.
if (names != null)
  for (int i = 0; i < names.length; i++)
    System.out.println(names[i]);
else
  System.out.println("No .txt files");
```

[Legal | AS/400 Glossary]

IFSJavaFile

The IFSJavaFile class represents a file in the AS/400 integrated file system and extends the java.io.File class. IFSJavaFile allows you to write files for the java.io.File interface that access AS/400 integrated file systems.

IFSJavaFile makes portable interfaces that are compatible with java.io.File and uses only the errors and exceptions that java.io.File uses. IFSJavaFile uses the security manager features from java.io.File, but unlike java.io.File, IFSJavaFile uses security features continuously.

You use IFSJavaFile with IFSFileInputStream and IFSFileOutputStream. It does not support java.io.FileInputStream and java.io.FileOutputStream.

IFSJavaFile is based on IFSFile; however, its interface is more like java.io.File than IFSFile. IFSFile is an alternative to the IFSJavaFile class.

An example of how to use the IFSJavaFile class is given below.

```
// Work with /Dir/File.txt on the system flash.
AS400 as400 = new AS400("flash");
IFSJavaFile file = new IFSJavaFile(as400, "/Dir/File.txt");
// Determine the parent directory of the file.
String directory = file.getParent();
// Determine the name of the file.
String name = file.getName();
// Determine the file size.
long length = file.length();
// Determine when the file was last modified.
Date date = new Date(file.lastModified());
// Delete the file.
```

```
  if (file.delete() == false)
  {
    // Display the error code.
    System.err.println("Unable to delete file.");
  }
  try
  {
    IFSFileOutputStream os = new IFSFileOutputStream(file.getSystem(),
file,
IFSFileOutputStream.SHARE_ALL,
false);
    byte[] data = new byte[256];
    int i = 0;
    for (; i < data.length; i++)
    {
      data[i] = (byte) i;
      os.write(data[i]);
    }
    os.close();
  }
  catch (Exception e)
  {
    System.err.println ("Exception: " + e.getMessage());
  }
```

[Legal | AS/400 Glossary]

IFSFileInputStream

The IFSFileInputStream class represents an input stream for reading data from a file on the AS/400. As in the IFSFile class, methods exist in IFSFileInputStream that duplicate the methods in FileInputStream from the java.io package. In addition to these methods, IFSFileInputStream has additional methods specific to the AS/400. The IFSFileInputStream class allows a Java program to do the following:

- Open a file for reading. The file must exist since this class does not create files on the AS/400.
- Open a file for reading and specify the file sharing mode.
- Determine the number of bytes in the stream.
- Read bytes from the stream.
- Skip bytes in the stream.
- Lock or unlock bytes in the stream.
- Close the file.

As in FileInputStream in java.io, this class allows a Java program to read a stream of bytes from the file. The Java program reads the bytes sequentially with only the additional option of skipping bytes in the stream.

The following example shows how to use the IFSFileInputStream class.

```
                    // Create an AS400 object.
    AS400 sys = new AS400("mySystem.myCompany.com");
                    // Open a file object that
                    // represents the file.
```

```
IFSFileInputStream aFile =
              new IFSFileInputStream(sys,"/mydir1/mydir2/myfile");
                  // Determine the number of bytes in
                  // the file.
int available = aFile.available();
                  // Allocate a buffer to hold the data
byte[] data = new byte[10240];
                  // Read the entire file 10K at a time
for (int i = 0; i < available; i += 10240)
{
    aFile.read(data);
}
                  // Close the file.
aFile.close();
```

In addition to the methods in FileInputStream, IFSFileInputStream gives the Java program the following options:

- Locking and unlocking bytes in the stream. See IFSKey for more information.
- Specifying a sharing mode when the file is opened. See sharing modes for more information.

[Legal | AS/400 Glossary]

IFSKey

If the Java program allows other programs access to a file at the same time, the Java program can lock bytes in the file for a period of time. During that time, the program has exclusive use of that section of the file. When a lock is successful, the integrated file system classes return an IFSKey object. This object is supplied to the unlock() method to indicate which bytes to unlock. When the file is closed, the system unlocks all locks that are still on the file (the system does an unlock for every lock that the program did not unlock).

The following example shows how to use the IFSKey class.

```
                  // Create an AS400 object.
AS400 sys = new AS400("mySystem.myCompany.com");
                  // Open an input stream. This
                  // constructor opens with share_all
                  // so other programs can open this
                  // file.
IFSFileInputStream aFile =
              new IFSFileInputStream(sys,"/mydir1/mydir2/myfile");
                  // Lock the first 1K bytes in the
                  // file. Now no other instance can
                  // read these bytes.
IFSKey key = aFile.lock(1024);
                  // Read the first 1K of the file.
byte data[] = new byte[1024];
aFile.read(data);
                  // Unlock the bytes of the file.
aFile.unlock(key);
                  // Close the file.
aFile.close();
```

[Legal | AS/400 Glossary]

File sharing mode

The Java program can specify a sharing mode when a file is opened. The program either allows other programs to open the file at the same time or has exclusive access to the file.

The following example shows how to specify a file sharing mode.

```
                    // Create an AS400 object.
    AS400 sys = new AS400("mySystem.myCompany.com");
                    // Open a file object that
                    // represents the file. Since this
                    // program specifies share-none, all
                    // other open attempts fail until
                    // this instance is closed.
    IFSFileOutputStream aFile =
                 new IFSFileOutputStream(sys,
                                         "/mydir1/mydir2/myfile",
                                         IFSFileOutputStream.SHARE_NONE,
                                         false);
                    // ... perform operations on the
                    // file.
                    // Close the file. Now other open
                    // requests succeed.
    aFile.close();
```

[Legal | AS/400 Glossary]

IFSTextFileInputStream

The IFSTextFileInputStream class represents a stream of character data read from a file. The data read from the IFSTextFileInputStream object is supplied to the Java program in a Java String object, so it is always unicode. When the file is opened, the IFSTextFileInputStream object determines the CCSID of the data in the file. If the data is stored in an encoding other than unicode, the IFSTextFileInputStream object converts the data from the file's encoding to unicode before giving the data to the Java program. If the data cannot be converted, an UnsupportedEncodingException is thrown.

The following example shows how to use the IFSTextFileInputStream:

```
                    // Work with /File on the system
                    // mySystem.
    AS400 as400 = new AS400("mySystem");
    IFSTextFileInputStream file = new IFSTextFileInputStream(as400, "/File");
                    // Read the first four characters of
                    // the file.
    String s = file.read(4);
                    // Display the characters read. Read
                    // the first four characters of the
                    // file. If necessary, the data is
                    // converted to unicode by the
                    // IFSTextFileInputStream object.
    System.out.println(s);
                    // Close the file.
    file.close();
```

[Legal | AS/400 Glossary]

IFSFileOutputStream

The IFSFileOutputStream class represents an output stream for writing data to a file on the AS/400. As in the IFSFile class, methods exist in IFSFileOutputStream that duplicate the methods in FileOutputStream from the java.io package. IFSFileOutputStream also has additional methods specific to the AS/400. The IFSFileOutputStream class allows a Java program to do the following:

- Open a file for writing. If the file already exists, it is replaced. Also available is a constructor that takes a boolean argument that specifies whether the contents of an existing file have been appended.
- Open a file for writing and specifying the file sharing mode.
- Write bytes to the stream.
- Commit to disk the bytes that are written to the stream.
- Lock or unlock bytes in the stream.
- Close the file.

As in FileOutputStream in java.io, this class allows a Java program to sequentially write a stream of bytes to the file.

The following example shows how to use the IFSFileOutputStream class.

```
                    // Create an AS400 object
AS400 sys = new AS400("mySystem.myCompany.com");
                    // Open a file object that
                    // represents the file.
IFSFileOutputStream aFile =
              new IFSFileOutputStream(sys,"/mydir1/mydir2/myfile");
                    // Write to the file
byte i = 123;
aFile.write(i);
                    // Close the file.
aFile.close();
```

In addition to the methods in FileOutputStream, IFSFileOutputStream gives the Java program the following options:

- Locking and unlocking bytes in the stream. See IFSKey for more information.
- Specifying a sharing mode when the file is opened. See sharing modes for more information.

[Legal | AS/400 Glossary]

IFSTextFileOutputStream

The IFSTextFileOutputStream class represents a stream of character data being written to a file. The data supplied to the IFSTextFileOutputStream object is in a Java String object so the input is always unicode. The IFSTextFileOutputStream object can convert the data to another CCSID as it is written to the file, however. The default behavior is to write unicode characters to the file, but the Java program can set the target CCSID before the file is opened. In this case, the IFSTextFileOutputStream object converts the characters from unicode to the specified CCSID before writing them to the file. If the data cannot be converted, an UnsupportedEncodingException is thrown.

The following example shows how to use IFSTextFileOutputStream:

```
                    // Work with /File on the system
                    // mySystem.
AS400 as400 = new AS400("mySystem");
```

```
IFSTextFileOutputStream file = new IFSTextFileOutputStream(as400, "/File");
                    // Write a String to the file.
                    // Because no CCSID was specified
                    // before writing to the file,
                    // unicode characters will be
                    // written to the file. The file
                    // will be tagged as having unicode
                    // data.
file.write("Hello world");
                    // Close the file.
file.close();
```

[Legal | AS/400 Glossary]

IFSRandomAccessFile

The IFSRandomAccessFile class represents a file on the AS/400 for reading and writing data. The Java program can read and write data sequentially or randomly. As in IFSFile, methods exist in IFSRandomAccessFile that duplicate the methods in RandomAccessFile from the java.io package. In addition to these methods, IFSRandomAccessFile has additional methods specific to the AS/400. Through IFSRandomAccessFile, a Java program can do the following:

- Open a file for read, write, or read/write access. The Java program can optionally specify the file sharing mode and the existence option.
- Read data at the current offset from the file.
- Write data at the current offset to the file.
- Get or set the current offset of the file.
- Close the file.

The following example shows how to use the IFSRandomAccessFile class to write four bytes at 1K intervals to a file.

```
                    // Create an AS400 object.
AS400 sys = new AS400("mySystem.myCompany.com");
                    // Open a file object that represents
                    // the file.
IFSRandomAccessFile aFile =
              new IFSRandomAccessFile(sys,"/mydir1/myfile", "rw");
                    // Establish the data to write.
byte i = 123;
                    // Write to the file 10 times at 1K
                    // intervals.
for (int j=0; j<10; j++)
{
                    // Move the current offset.
   aFile.seek(j * 1024);
                    // Write to the file. The current
                    // offset advances by the size of
                    // the write.
   aFile.write(i);
}
                    // Close the file.
aFile.close();
```

In addition to the methods in java.io RandomAccessFile, IFSRandomAccessFile gives the Java program the following options:

- Committing to disk bytes written.
- Locking or unlocking bytes in the file.
- Locking and unlocking bytes in the stream. See IFSKey for more information.
- Specifying a sharing mode when the file is opened. See sharing modes for more information.
- Specify the existence option when a file is opened. The Java program can choose one of the following:
  - Open the file if it exists; create the file if it does not.
  - Replace the file if it exists; create the file if it does not.
  - Fail the open if the file exists; create the file if it does not.
  - Open the file if it exists; fail the open if it does not.
  - Replace the file if it exists; fail the open if it does not.

[Legal | AS/400 Glossary]

IFSFileDialog

The IFSFileDialog class allows you to traverse the file system and select a file. This class uses the IFSFile class to traverse the list of directories and files in the integrated file system on the AS/400. Methods on the class allow a Java program to set the text on the push buttons of the dialog and to set filters. Note that an IFSFileDialog class based on Swing 1.0.3 is also available.

You can set filters through the FileFilter class. If the user selects a file in the dialog, the getFileName() method can be used to get the name of the file that was selected. The getAbsolutePath() method can be used to get the path and name of the file that was selected.

The following example shows how to set up a dialog with two filters and to set the text on the push buttons of the dialog.

```
                    // Create an AS400 object.
AS400 sys = new AS400("mySystem.myCompany.com");
                    // Create a dialog object setting
                    // the text of the dialog's title
                    // bar and the AS/400 to traverse.
IFSFileDialog dialog = new IFSFileDialog(this, "Title Bar Text", sys);
                    // Create a list of filters then set
                    // the filters in the dialog. The
                    // first filter will be used when
                    // the dialog is first displayed.
FileFilter[] filterList = {new FileFilter("All files (*.*)", "*.*"),
                           new FileFilter("HTML files (*.HTML", "*.HTM")};
dialog.setFileFilter(filterList, 0);
                    // Set the text on the buttons of
                    // the dialog.
dialog.setOkButtonText("Open");
dialog.setCancelButtonText("Cancel");
                    // Show the dialog. If the user
                    // selected a file by pressing the
                    // Open button, get the file the
                    // user selected and display it.
if (dialog.showDialog() == IFSFileDialog.OK)
   System.out.println(dialog.getAbsolutePath());
```

[Legal | AS/400 Glossary]

JDBC

The AS/400 Toolbox for Java JDBC (Java Database Connectivity) driver allows Java programs to access AS/400 database files using standard JDBC interfaces. Use these standard JDBC interfaces to issue SQL statements and process results. JDBC is a standard part of Java and is included in JDK 1.1.

▲ The AS/400 Toolbox for Java supports JDBC 2.0. If you want to use any of the following JDBC 2.0 enhancements, you also need to use JDK 1.2:

- Blob interface
- Clob interface
- Scrollable result set
- Updatable result set
- Batch update capability with Statement, PreparedStatement, and CallableStatement objects ▼

JDBC defines the following Java interfaces:

- The Driver interface creates the connection and returns information about the driver version.
- The Connection interface represents a connection to a specific database.
- The Statement interface runs SQL statements and obtains the results.
- The PreparedStatement interface runs compiled SQL statements.
- The CallableStatement interface runs SQL stored procedures.
- The ResultSet interface provides access to a table of data that is generated by running a SQL query or DatabaseMetaData catalog method.
- The DatabaseMetaData interface provides information about the database as a whole.
- ▲ The Blob interface provides access to binary large objects (BLOBs).
- The Clob interface provides access to character large objects (CLOBs). ▼

Examples

Using the JDBC driver to create and populate a table.

Using the JDBC driver to query a table and output its contents.

[Legal | AS/400 Glossary]

Registering the JDBC driver

Before using JDBC to access data in an AS/400 database file, you need to register the JDBC driver for the AS/400 Toolbox for Java licensed program with the DriverManager. You can register the driver either by using a Java system property or by having the Java program register the driver:

- Register by using a system property

  Each virtual machine has its own method of setting system properties. For example, the Java command from the JDK uses the -D option to set system properties. To set the driver using system properties, specify the following:

  ```
  "-Djdbc.drivers=com.ibm.as400.access.AS400JDBCDriver"
  ```

- Register by using the Java program

  To explicitly load the driver, add the following to the Java program before the first JDBC call:

  ```
  java.sql.DriverManager.registerDriver (new com.ibm.as400.access.AS400JDBCDriver ());
  ```

The AS/400 Toolbox for Java JDBC driver does not require an AS400 object as an input parameter like the other AS/400 Toolbox for Java classes that get data from an AS/400. An AS400 object is used internally, however, to manage default user and password caching. When a connection is first made to the AS/400, the user may be prompted for user ID and password. The user has the option to save the user ID as the default user ID and add the password to the password cache. As in the other AS/400 Toolbox for Java functions, if the user ID and password are supplied by the Java program, the default user is not set and the password is not cached. See managing connections for information on managing connections.

[Legal | AS/400 Glossary]

Using the JDBC driver to connect to an AS/400 database

You can use the DriverManager.getConnection() method to connect to the AS/400 database. DriverManager.getConnection() takes a uniform resource locator (URL) string as an argument. The JDBC driver manager attempts to locate a driver that can connect to the database that is represented by the URL. When using the AS/400 Toolbox for Java driver, use the following syntax for the URL:

```
"jdbc:as400://systemName/defaultSchema;listOfProperties"
```

Note: Either systemName or defaultSchema can be omitted from the URL.

Examples: Using the JDBC driver to connect to an AS/400

Example 1: Using a URL in which a system name is not specified. This will result in the user being prompted to type in the name of the system to which the user wants to connect.

```
"jdbc:as400:"
```

Example 2: Connecting to the AS/400 database; no default schema or properties specified.

```
                    // Connect to system 'mySystem'. No
                    // default schema or properties are
                    // specified.
Connection c  = DriverManager.getConnection("jdbc:as400://mySystem");
```

Example 3: Connecting to the AS/400 database; default schema specified.

```
                    // Connect to system 'mySys2'. The
                    // default schema 'myschema' is
                    // specified.
Connection c2 = DriverManager.getConnection("jdbc:as400://mySys2/mySchema");
```

Example 4: Connecting to the AS/400 database; properties are specified using java.util.Properties. The Java program can specify a set of JDBC properties either by using the java.util.Properties interface or by specifying the properties as part of the URL. See JDBC properties for a list of supported properties.

For example, to specify properties using the Properties interface, use the following code as an example:

```
                    // Create a properties object.
Properties p = new Properties();
                    // Set the properties for the
                    // connection.
p.put("naming", "sql");
p.put("errors", "full");
                    // Connect using the properties
                    // object.
Connection c = DriverManager.getConnection("jdbc:as400://mySystem",p);
```

Example 5: Connecting to the AS/400 database; properties are specified using a uniform resource locator (URL).

```
                    // Connect using properties. The
                    // properties are set on the URL
                    // instead of through a properties
                    // object.
Connection c = DriverManager.getConnection(
                     "jdbc:as400://mySystem;naming=sql;errors=full");
```

Example 6: Connecting to the AS/400 database; user ID and password are specified.

```
                    // Connect using properties on the
                    // URL and specifying a user ID and
                    // password
Connection c = DriverManager.getConnection(
                     "jdbc:as400://mySystem;naming=sql;errors=full",
                     "auser",
                     "apassword");
```

Example 7: Disconnecting from the database. Use the close() method on the Connecting object to disconnect from the AS/400. To close the connection that is created in the above example, use the following statement:

```
c.close();
```

[Legal | AS/400 Glossary]

Running SQL statements with Statement objects

Use a Statement object to run an SQL statement and optionally obtain the ResultSet produced by it.

PreparedStatement inherits from Statement, and CallableStatement inherits from PreparedStatement. Use the following Statement objects to run different SQL statements:

- “Statement interface” on page 45 - to run a simple SQL statement that has no parameters.
- PreparedStatement - to run a precompiled SQL statement that may or may not have IN parameters.
- CallableStatement - to run a call to a database stored procedure. A CallableStatement may or may not have IN, OUT, and INOUT parameters.

The Statement object allows you to submit multiple update commands as a single group to a database through the use of a batch update facility. Through the use of the batch update facility, you may get better performance because it is usually faster to process a group of update operations than to process one update operation at a time. If you want to use the batch update facility, you need JDBC 2.0 and JDK 1.2.

When using batch updates, usually you should turn off auto-commit. Turning off auto-commit allows your program to determine whether to commit the transaction if an error occurs and not all of the commands have executed. In JDBC 2.0, a Statement object can keep track of a list of commands that can be successfully submitted and executed together in a group. When this list of batch commands is executed by the executeBatch() method, the commands are executed in the order in which they were added to the list.

Statement interface

Use Connection.createStatement() to create new Statement objects.

The following example shows how to use a Statement object.

```
                    // Connect to the AS/400.
Connection c = DriverManager.getConnection("jdbc:as400://mySystem");
                    // Create a Statement object.
Statement s = c.createStatement();
                    // Run an SQL statement that creates
                    // a table in the database.
s.executeUpdate("CREATE TABLE MYLIBRARY.MYTABLE (NAME VARCHAR(20), ID INTEGER)");
                    // Run an SQL statement that inserts
                    // a record into the table.
s.executeUpdate("INSERT INTO MYLIBRARY.MYTABLE (NAME, ID) VALUES ('DAVE', 123)");
                    // Run an SQL statement that inserts
                    // a record into the table.
s.executeUpdate("INSERT INTO MYLIBRARY.MYTABLE (NAME, ID) VALUES ('CINDY', 456)");
                    // Run an SQL query on the table.
ResultSet rs = s.executeQuery("SELECT * FROM MYLIBRARY.MYTABLE");
                    // Close the Statement and the
                    // Connection.
s.close();
c.close();
```

[Legal | AS/400 Glossary]

PreparedStatement interface

You can use a PreparedStatement object when an SQL statement is going to be run many times. An SQL statement can be precompiled. A “prepared.” statement is an SQL statement that has been precompiled. This approach is more efficient than running the same statement multiple times using a Statement object, which compiles the statement each time it is run. In addition, the SQL statement contained in a PreparedStatement object may have one or more IN parameters. Use Connection.prepareStatement() to create PreparedStatement objects.

 You can use a batch update facility to associate a single PreparedStatement object with multiple sets of input parameter values. This unit then can be sent to the database for processing as a single entity. You may get better performance with batch updates because it is usually faster to process a group of update operations than one update operation at a time. If you want to use the batch update facility, you need JDBC 2.0 and JDK 1.2.

The following example shows how to use the PreparedStatement interface.

```
                    // Connect to the AS/400.
Connection c = DriverManager.getConnection("jdbc:as400://mySystem");
                    // Create the PreparedStatement
                    // object. It precompiles the
                    // specified SQL statement. The
```

```
                    // question marks indicate where
                    // parameters must be set before the
                    // statement is run.
PreparedStatement ps = c.prepareStatement("INSERT INTO MYLIBRARY.MYTABLE (NAME, ID) VALUES (?,
                    // Set parameters and run the
                    // statement.
ps.setString(1, "JOSH");
ps.setInt(2, 789);
ps.executeUpdate();
                    // Set parameters and run the
                    // statement again.
ps.setString(1, "DAVE");
ps.setInt(2, 456);
ps.executeUpdate();
                    // Close PreparedStatement and the
                    // Connection.
ps.close();
c.close();
```

[Legal | AS/400 Glossary]

CallableStatement interface

You can use a CallableStatement object to run SQL stored procedures. The stored procedure being called must already be stored in the database. CallableStatement does not contain the stored procedure, it only calls the stored procedure.

A stored procedure can return one or more ResultSet objects and can use IN parameters, OUT parameters, and INOUT parameters. Use Connection.prepareCall() to create new CallableStatement objects.

 You can use a batch update facility to associate a single CallableStatement object with multiple sets of input parameter values. This unit then can be sent to the database for processing as a single entity. You may get better performance with batch updates because it is usually faster to process a group of update operations than one update operation at a time. If you want to use the batch update facility, you need JDBC 2.0 and JDK 1.2.

The following example shows how to use the CallableStatement interface.

```
                    // Connect to the AS/400.
Connection c = DriverManager.getConnection("jdbc:as400://mySystem");
                    // Create the CallableStatement
                    // object. It precompiles the
                    // specified call to a stored
                    // procedure. The question marks
                    // indicate where input parameters
                    // must be set and where output
                    // parameters can be retrieved.
                    // The first two parameters are
                    // input parameters, and the third
                    // parameter is an output parameter.
CallableStatement cs = c.prepareCall("CALL MYLIBRARY.ADD (?, ?, ?)");
                    // Set input parameters.
cs.setInt (1, 123);
cs.setInt (2, 234);
                    // Register the type of the output
                    // parameter.
cs.registerOutParameter (3, Types.INTEGER);
                    // Run the stored procedure.
cs.execute ();
                    // Get the value of the output
```

```
                    // parameter.
int sum = cs.getInt (3);
                    // Close the CallableStatement and
                    // the Connection.
cs.close();
c.close();
```

[Legal | AS/400 Glossary]

DatabaseMetaData interface

You can use a DatabaseMetaData object to obtain information about the database as a whole as well as catalog information.

The following example shows how to return a list of tables, which is a catalog function:

```
                    // Connect to the AS/400.
Connection c = DriverManager.getConnection("jdbc:as400://mySystem");
                    // Get the database metadata from
                    // the connection.
DatabaseMetaData dbMeta = c.getMetaData();
                    // Get a list of tables matching the
                    // following criteria.
String catalog = "myCatalog";
String schema  = "mySchema";
String table   = "myTable%"; // % indicates search pattern
String types[]  = {"TABLE", "VIEW", "SYSTEM TABLE"};
ResultSet rs = dbMeta.getTables(catalog, schema, table, types);
                    // ... iterate through the ResultSet
                    // to get the values
                    // Close the Connection.
c.close();
```

[Legal | AS/400 Glossary]

AS400JDBCBlob interface

You can use a AS400JDBCBlob object to access binary large objects (BLOBs), such as sound byte (.wav) files or image (.gif) files.

The key difference between the AS400JDBCBlob class and the AS400JDBCBlobLocator class is where the blob is stored. With the AS400JDBCBlob class, the blob is stored in the database, which inflates the size of the database file. The AS400JDBCBlobLocator class stores a locator (think of it as a pointer) in the database file that points to where the blob is located.

With the AS400JDBCBlob class, the lob threshold property can be used. This property specifies the maximum large object (LOB) size (in kilobytes) that can be retrieved as part of a result set. LOBs that are larger than this threshold are retrieved in pieces using extra communication to the server. Larger LOB thresholds reduce the frequency of communication to the server, but they download more LOB data, even if it is not used. Smaller lob thresholds may increase frequency of communication to the server, but they only download LOB data as it is needed. See JDBC properties for information on additional properties that are available.

Using the AS400JDBCBlob interface, you can do the following:

- Return the entire blob as a stream of uninterpreted bytes
- Return part of the contents of the blob

- Return the length of the blob

The following example shows how to use the AS400JDBCBlob interface:

```
Blob blob = resultSet.getBlob (1);
long length = blob.length ();
byte[] bytes = blob.getBytes (0, (int) length);
```

AS400JDBCBlobLocator interface

You can use a AS400JDBCBlobLocator object to access a binary large objects.

Using the AS400JDBCBlobLocator interface, you can do the following:

- Return the entire blob as a stream of uninterpreted bytes
- Return part of the contents of the blob
- Return the length of the blob

[Legal | AS/400 Glossary]

AS400JDBCClob interface

You can use a AS400JDBCClob object to access character large objects (CLOBs), such as large documents.

The key difference between the AS400JDBCClob class and the AS400JDBCClobLocator class is where the blob is stored. With the AS400JDBCClob class, the blob is stored in the database, which inflates the size of the database file. The AS400JDBCClobLocator class stores a locator (think of it as a pointer) in the database file that points to where the blob is located.

With the AS400JDBCClob class, the lob threshold property can be used. This property specifies the maximum large object (LOB) size (in kilobytes) that can be retrieved as part of a result set. LOBs that are larger than this threshold are retrieved in pieces using extra communication to the server. Larger LOB thresholds reduce the frequency of communication to the server, but they download more LOB data, even if it is not used. Smaller lob thresholds may increase frequency of communication to the server, but they only download LOB data as it is needed. See JDBC properties for information on additional properties that are available.

Using the AS400JDBCClob interface, you can do the following:

- Return the entire clob as a stream of ASCII characters
- Return the contents of the clob as a character stream
- Return a part of the contents of the clob
- Return the length of the clob

The following example shows how to use the AS400JDBCClob interface:

```
Clob clob = rs.getClob (1);
int length = clob.getLength ();
String s = clob.getSubString (0, (int) length);
```

AS400JDBCClobLocator interface

You can use a AS400JDBCClobLocator object to access character large objects (CLOBs).

Using the AS400JDBCClobLocator interface, you can do the following:

- Return the entire clob as a stream of ASCII characters
- Return the entire clob as a character stream
- Return a part of the contents of the clob
- Return the length of the clob

[Legal | AS/400 Glossary]

Jobs

The AS/400 Toolbox for Java Jobs classes allow a Java program to retrieve and
change the following type of job information:

- Date and Time Information
- Job Queue
- Language Identifiers
- Message Logging
- Output Queue
- Printer Information

The job classes are as follows:

- Job - retrieves and changes AS/400 job information
- JobList - retrieves a list of AS/400 jobs
- JobLog - represents the job log of an AS/400

Examples

List the jobs belonging to a specific user and list jobs with job status information.

Display the messages in a job log.

Use a cache when setting a value and getting a value:

```
try {
    // Creates AS400 object.
    AS400 as400 = new AS400("systemName");
    // Constructs a Job object
    Job job = new Job(as400,"QDEV002");
    // Gets job information
    System.out.println("User of this job :" + job.getUser());
    System.out.println("CPU used :" + job.getCPUUsed();
    System.out.println("Job enter system date : " + job.getJobEnterSystemDate())
    // Sets cache mode
    job.setCacheChanges(true);
    // Changes will be store in the cache.
    job.setRunPriority(66);
    job.setDateFormat("*YMD");
    // Commit changes. This will change the value on the AS/400.
    job.commitChanges();
    // Set job information to system directly(without cache).
```

```
        job.setCacheChanges(false);
        job.setRunPriority(60);
    } catch (Exception e)
    {
        System.out.println(quot;error :" + e)
    }
```

[Legal | AS/400 Glossary]

Job

The job class allows a java program to retrieve and change AS/400 jobs information.

The following type of job information can be retrieved and changed with the Job class:

- Job queues
- Output queues
- Message logging
- Printer device
- Country identifier
- Date format

The job class also allows the ability to change a single value at a time, or cache several changes using the setCacheChanges(true) method and committing the changes using the commitChanges() method. If caching is not turned on, you do not need to do a commit.

Use this example for how to set and get values to and from the cache in order to set the run priority with the setRunPriority() method and set the date format with the setDateFormat() method. 

[Legal | AS/400 Glossary]

JobList

You can use JobList class to list AS/400 jobs. With the JobList class, you can retrieve the following:

- All jobs
- Jobs by name, job number, or user

Use the getJobs() method to return a list of AS/400 jobs  or getLength() method to return the number of jobs retrieved with the last getJobs(). 

The following example lists all active jobs on the system:

```
                    // Create an AS400 object. List the
                    // jobs on this AS/400.
AS400 sys = new AS400("mySystem.myCompany.com");
                    // Create the job list object.
JobList jobList = new JobList(sys);
```

```
                    // Get the list of active jobs.
Enumeration list = jobList.getJobs();
                    // For each active job on the system
                    // print job information.
while (list.hasMoreElements())
{
    Job j = (Job) list.nextElement();
    System.out.println(j.getName() + "." +
                       j.getUser() + "." +
                       j.getNumber());
}
```

[Legal | AS/400 Glossary]

JobLog

The JobLog class retrieves messages in the job log of an AS/400 job by calling getMessages().

The following example prints all messages in the job log for the specified user:

```
                    // ... Setup work to create an AS400
                    // object and a jobList object has
                    // already been done
                    // Get the list of active jobs on
                    // the AS/400
Enumeration list = jobList.getJobs();
                    // Look through the list to find a
                    // job for the specified user.
while (list.hasMoreElements())
{
   Job j = (Job) list.nextElement();
   if (j.getUser().trim().equalsIgnoreCase(userID))
   {
                    // A job matching the current user
                    // was found. Create a job log
                    // object for this job.
      JobLog jlog = new JobLog(system,
                               j.getName(),
                               j.getUser(),
                               j.getNumber());
                    // Enumerate the messages in the job
                    // log then print them.
      Enumeration messageList = jlog.getMessages();
      while (messageList.hasMoreElements())
      {
          AS400Message message = (AS400Message) messageList.nextElement();
          System.out.println(message.getText());
      }
   }
}
```

[Legal | AS/400 Glossary]

Messages

AS400Message

AS400 Message object allows the Java program to retrieve an AS/400 message that is generated from a previous operation (for example, from a command call). From a message object, the Java program can retrieve the following:

- The AS/400 library and message file that contain the message
- The message ID
- The message type
- The message severity
- The message text
- The message help text

The following example shows how to use the AS/400 Message object:

```
                    // Create a command call object.
CommandCall cmd = new CommandCall(sys, "myCommand");
                    // Run the command
cmd.run();
                    // Get the list of messages that are
                    // the result of the command that I
                    // just ran
AS400Message[] messageList = cmd.getMessageList();
                   // Iterate through the list
                   // displaying the messages
for (int i = 0; i < messageList.length; i++)
{
   System.out.println(messageList[i].getText());
}
```

Examples

The CommandCall example shows how a message list is used with CommandCall.

The ProgramCall example shows how a message list is used with ProgramCall.

QueuedMessage

The QueuedMessage class extends the AS400Message class. The QueuedMessage class accesses information about a message on an AS/400 message queue. With this class, a Java program can retrieve:

- Information about where a message originated, such as program, job name, job number, and user
- The message queue
- The message key
- The message reply status

The following example prints all messages in the message queue of the current (signed-on) user:

```
                    // The message queue is on this as400.
AS400 sys = new AS400(mySystem.myCompany.com);
                    // Create the message queue object.
                    // This object will represent the
                    // queue for the current user.
MessageQueue queue = new MessageQueue(sys, MessageQueue.CURRENT);
                    // Get the list of messages currently
                    // in this user's queue.
Enumeration e = queue.getMessages();
                    // Print each message in the queue.
while (e.hasMoreElements())
{
   QueuedMessage msg = e.getNextElement();
   System.out.println(msg.getText());
}
```

MessageFile

The MessageFile class allows you to receive a message from an AS/400 message file. The MessageFile class returns an AS400Message object that contains the message. Using the MessageFile class, you can do the following:

- Return a message object that contains the message
- Return a message object that contains substitution text in the message

The following example shows how to retrieve and print a message:

```
AS400 system = new AS400("mysystem.mycompany.com");
MessageFile messageFile = new MessageFile(system);
messageFile.setPath("/QSYS.LIB/QCPFMSG.MSGF");
AS400Message message = messageFile.getMessage("CPD0170");
System.out.println(message.getText());
```

MessageQueue

The MessageQueue class allows a Java program to interact with an AS/400 message queue. It acts as a container for the QueuedMessage class. The getMessages() method, in particular, returns a list of QueuedMessage objects. The MessageQueue class can do the following:

- Set message queue attributes
- Get information about a message queue
- Receive messages from a message queue
- Send messages to a message queue
- Reply to messages

The following example lists messages in the message queue for the current user:

```
                // The message queue is on this as400.
AS400 sys = new AS400(mySystem.myCompany.com);
                // Create the message queue object.
                // This object will represent the
                // queue for the current user.
MessageQueue queue = new MessageQueue(sys, MessageQueue.CURRENT);
                // Get the list of messages currently
                // in this user's queue.
Enumeration e = queue.getMessages();
                // Print each message in the queue.
while (e.hasMoreElements())
{
   QueuedMessage msg = e.getNextElement();
   System.out.println(msg.getText());
}
```

[Legal | AS/400 Glossary]

Network print

Print objects include spooled files, output queues, printers, printer files, writer jobs, and Advanced Function Printing (AFP) resources, which include fonts, form definitions, overlays, page definitions, and page segments. AFP resources are accessible only on Version 3 Release 7 (V3R7) and later AS/400 systems. (Trying

to open an AFPResourceList to a system that is running an earlier version than V3R7 generates a RequestNotSupportedException exception.)

The AS/400 Toolbox for Java classes for print objects are organized on a base class, PrintObject, and on a subclass for each of the six types of print objects. The base class contains the methods and attributes common to all AS/400 print objects. The subclasses contain methods and attributes specific to each subtype.

Use the network print classes for the following:

- Working with AS/400 print objects:
  - PrintObjectList class - use for listing and working with AS/400 print objects. (Print objects include spooled files, output queues, printers, Advanced Function Printing (AFP) resources, printer files, and writer jobs.)
  - PrintObject base class - use this base class and its subclasses for working with print objects.
- Retrieving PrintObject attributes
- Creating new AS/400 spooled files using the SpooledFileOutputStream class (use for EBCDIC-based printer data)
- Generating SNA Character Stream (SCS) printer data streams
- Reading spooled files and AFP resources using the PrintObjectInputStream
- Reading spooled files using PrintObjectPageInputStream and PrintObjectTransformedInputStream
- Viewing Advanced Function Printing (AFP) and SNA Character Stream (SCS) spooled files

Examples

- The Create Spooled File Example shows how to create a spooled file on an AS/400 from an input stream.
- The Create SCS Spooled File Example shows how to generate a SCS data stream using the SCS3812Writer class, and how to write the stream to a spooled file on the AS/400.
- The Read Spooled File Example shows how to read an existing AS/400 spooled file.
- The first Asynchronous List Example shows how to asynchronously list all spooled files on a system and how to use the PrintObjectListListener interface to get feedback as the list is being built.
- The second Asynchronous List Example shows how to asynchronously list all spooled files on a system *without* using the PrintObjectListListener interface
- The Synchronous List Example shows how to synchronously list all spooled files on a system.

[Legal | AS/400 Glossary]

Listing Print objects

You can use the PrintObjectList class and its subclasses to work with lists of print objects. Each subclass has methods that allow filtering of the list based on what makes sense for that particular type of print object. For example, SpooledFileList allows you to filter a list of spooled files based on the user who created the spooled files, the output queue that the spooled files are on, the form type, or user data of

the spooled files. Only those spooled files that match the filter criteria are listed. If no filters are set, a default for each of the filters are used.

To actually retrieve the list of print objects from the AS/400, the openSynchronously() or openAsynchronously() methods are used. The openSynchronously() method does not return until all objects in the list have been retrieved from the AS/400 system. The openAsynchronously() method returns immediately, and the caller can do other things in the foreground while waiting for the list to build. The asynchronously opened list also allows the caller to start displaying the objects to the user as the objects come back. Because the user can see the objects as they come back, the response time may seem faster to the user. In fact, the response time may actually take longer overall due to the extra processing being done on each object in the list.

If the list is opened asynchronously, the caller may get feedback on the building of the list. Methods, such as isCompleted() and size(), indicate whether the list has finished being built or return the current size of the list. Other methods, waitForListToComplete() and waitForItem(), allow the caller to wait for the list to complete or for a particular item. In addition to calling these PrintObjectList methods, the caller may register with the list as a listener. In this situation, the caller is notified of events that happen to the list. To register or unregister for the events, the caller uses PrintObjectListListener(), and then calls addPrintObjectListListener() to register or removePrintObjectListListener() to unregister. The following table shows the events that are delivered from a PrintObjectList.

Event	When Delivered
listClosed	When the list is closed.
listCompleted	When the list completes.
listErrorOccurred	If any exception is thrown while the list is being retrieved.
listOpened	When the list is opened.
listObjectAdded	When an object is added to the list.

After the list has been opened and the objects in the list processed, close the list using the close() method. This frees up any resources allocated to the garbage collector during the open. After a list has been closed, its filters can be modified, and the list can be opened again.

When print objects are listed, attributes about each print object listed are sent from the AS/400 and stored with the print object. These attributes can be updated using the update() method in the PrintObject class. Which attributes are sent back from the AS/400 depends on the type of print object being listed. A default list of attributes for each type of print object that can be overridden by using the setAttributesToRetrieve() method in PrintObjectList exists. See the Retrieving PrintObject attributes section for a list of the attributes each type of print object supports.

Listing AFP Resources is allowed only on V3R7 and later release of AS/400. Opening an AFPResourceList to an system older than V3R7 generates a RequestNotSupportedException exception.

Examples

Asynchronous List Example 1

Asynchronous List Example 2

Synchronous List Example

[Legal | AS/400 Glossary]

Working with Print objects

PrintObject is an abstract class. (An abstract class does not allow you to create an instance of the PrintObject class. Instead, you must create an instance of one of its subclasses.) Create objects of the subclasses in any of the following ways:

- If you know the system and the identifying attributes of the object, construct the object explicitly by calling its public constructor.
- You can use a PrintObjectList subclass to build a list of the objects and then get at the individual objects through the list.
- An object may be created and returned to you as a result of a method or set methods being called. For example, the static method start() in the WriterJob class returns a WriterJob object.

Use the base class, PrintObject, and its subclasses to work with AS/400 print objects:

- OutputQueue
- Printer
- PrinterFile
- SpooledFile
- WriterJob

[Legal | AS/400 Glossary]

Retrieving PrintObject attributes

You can retrieve print object attributes by using the attribute ID and one of these methods from the base PrintObject class:

- Use getIntegerAttribute(int attributeID) to retrieve an integer type attribute.
- Use getFloatAttribute(int attributeID) to retrieve a floating point type attribute.
- Use getStringAttribute(int attributeID) to retrieve a string type attribute.

The attributeID parameter is an integer that identifies which attribute to retrieve. All of the IDs are defined as public constants in the base PrintObject class. The PrintAttributes file contains an entry of each attribute ID. The entry includes a description of the attribute and its type (integer, floating point, or string). For a list of which attributes may be retrieved using these methods, select the following links:

- AFPResourceAttrs for AFP Resources
- OutputQueueAttrs for output queues
- PrinterAttrs for printers
- PrinterFileAttrs for printer files
- SpooledFileAttrs for spooled files
- WriterJobAttrs for writer jobs

To achieve acceptable performance, these attributes are copied to the client. These attributes are copied either when the objects are listed, or the first time they are needed if the object was created implicitly. This keeps the object from going to the host every time the application needs to retrieve an attribute. This also makes it

possible for the Java print object instance to contain out-of-date information about the object on the AS/400. The user of the object can refresh all of the attributes by calling the update() method on the object. In addition, if the application calls any methods on the object that would cause the object's attributes to change, the attributes are automatically updated. For example, if an output queue has a status attribute of RELEASED (getStringAttribute(ATTR_OUTQSTS); returns a string of "RELEASED"), and the hold() method is called on the output queue, getting the status attribute after that returns HELD.

setAttributes method

You can use the setAttributes method to change the attributes of spooled files and printer file objects. Select the following links for a list or which attributes may be set:

- PrinterFileAttrs file for printer files
- SpooledFileAttrs for spooled files

The setAttributes method takes a PrintParameterList parameter, which is a class that is used to hold a collection of attributes IDs and their values. The list starts out empty, and the caller can add attributes to the list by using the various setParameter() methods on it.

PrintParameterList class

You can use the PrintParameterList class to pass a group of attributes to a method that takes any of a number of attributes as parameters. For example, you can send a spooled file using TCP (LPR) by using the SpooledFile method, sendTCP(). The PrintParameterList object contains the required parameters for the send command, such as the remote system and queue, plus any optional parameters desired, such as whether to delete the spooled file after it is sent. In these cases, the method documentation gives a list of required and optional attributes. The PrintParameterList setParameter() method does not check which attributes you are setting and the values that you set them to. The PrintParameterList setParameter() method simply contains the values to pass along to the method. In general, extra attributes in the PrintParameterList are ignored, and illegal values on the attributes that are used are diagnosed on the AS/400.

[Legal | AS/400 Glossary]

Creating new spooled files

You can use the SpooledFileOutputStream class to create new AS/400 spooled files. The class derives from the standard JDK java.io.OutputStream class; after its construction, it can be used anywhere an OutputStream is used.

When creating a new SpooledFileOutputStream, the caller may specify the following:

- Which printer file to use
- Which output queue to put the spooled file on
- A PrintParameterList object that may contain parameters to override fields in the printer file

These parameters are all optional (the caller may pass null of any or all of them). If a printer file is not specified, the network print server uses the default network print printer file, QPNPSPRTF. The output queue parameter is there as a convenience; it

also can be specified in the PrintParameterList. If the output queue parameter is specified in both places, the PrintParameterList field overrides the output queue parameter. See the documentation of the SpooledFileOutputStream constructor for a complete list of which attributes may be set in the PrintParameterList for creating new spooled files.

Use one of the write() methods to write data into the spooled file. The SpooledFileOutputStream object buffers the data and sends it when either the output stream is closed or the buffer is full. Buffering is done for two reasons:

- It allows the automatic data typing (see **Data Stream Types In Spooled Files** below) to analyze a full-buffer of data to determine the data type
- It makes the output stream work faster because not every write request is communicated to the AS/400.

Use the flush() method to force the data to be written to the AS/400.

When the caller is finished writing data to the new spooled file, the close() method is called to close the spooled file. Once the spooled file has been closed, no more data can be written to it. By calling the getSpooledFile() method once the spooled file has been closed, the caller can get a reference to a SpooledFile object that represents the spooled file.

Data stream types in spooled files

Use the Printer Data Type attribute of the spooled file to set the type of data to be put into the spooled file. If the caller does not specify a printer data type, the default is to use automatic data typing. This method looks at the first few thousand bytes of the spooled file data, determines if it fits either SNA Character Stream (SCS) or Advanced Function Printing data stream (AFPDS) data stream architectures, and then sets the attribute appropriately. If the bytes of spooled file data do not match either of these architectures, the data is tagged as *USERASCII. Automatic data typing works most of the time. The caller generally should use it unless the caller has a specific case in which automatic data typing does not work. In those cases, the caller can set the Printer Data Type attribute to a specific value (for example, *SCS). If the caller wants to use the printer data that is in the printer file, the caller must use the special value *PRTF. If the caller overrides the default data type when creating a spooled file, caution must be used to ensure that the data put into the spooled file matches the data type attribute. Putting non-SCS data into a spooled file that is marked to receive SCS data triggers an error message from the host and the loss of the spooled file.

Generally, this attribute can have three values:

- ***SCS** - an EBCDIC, text-based printer data stream.
- ***AFPDS** (Advanced Function Presentation Data Stream) - another data stream supported on the AS/400. *AFPDS can contain text, image, and graphics, and can use external resources such as page overlays and external images in page segments.
- ***USERASCII** - any non-SCS and non-AFPDS printer data that the AS/400 handles by just passing it through. Postscript and HP-PCL data streams are examples data streams that would be in a *USERASCII spooled file.

Examples

Create Spooled File Example

Create SCS Spooled File Example

[Legal | AS/400 Glossary]

Generating an SCS data stream

To generate spooled files that will print on certain printers attached to AS/400, an SNA Character Stream (SCS) data stream may have to be created. (SCS is a text-based, EBCDIC data stream that can be printed on SCS printers, IPDS printers, or to PC printers.) SCS can be printed by converting it using an emulator or the host print transform on the AS/400.

You can use the SCS writer classes to generate such an SCS data stream. The SCS writer classes convert Java unicode characters and formatting options into an SCS data stream. Five SCS writer classes generate varying levels of SCS data streams. The caller should choose the writer that matches the final printer destination to which the caller or end user will be printing.

Use the following SCS writer classes to generate an SCS printer data stream:

SCS5256Writer	The simplest SCS writer class. Supports text, carriage return, line feed, new line, form feed, absolute horizontal and vertical positioning, relative horizontal and vertical positioning, and set vertical format.
SCS5224Writer	Extends the 5256 writer and adds methods to set character per inch (CPI) and lines per inch (LPI).
SCS5219Writer	Extends the 5224 writer and adds support for left margin, underline, form type (paper or envelope), form size, print quality, code page, character set, source drawer number, and destination drawer number.
SCS5553Writer	Extends the 5219 writer and adds support for adds support for character rotation, grid lines, and font scaling. The 5553 is a double-byte character set (DBCS) data stream.
SCS3812Writer	Extends the 5219 writer and adds support for bold, duplex, text orientation, and fonts.

To construct an SCS writer, the caller needs an output stream and, optionally, an encoding. The data stream is written to the output stream. To create an SCS spooled file, the caller first constructs a SpooledFileOutputStream, and then uses that to construct an SCS writer object. The encoding parameter gives a target EBCDIC coded character set identifier (CCSID) to convert the characters to.

Once the writer is constructed, use the write() methods to output text. Use the carriageReturn(), lineFeed(), and newLine() methods to position the write cursor on the page. Use the endPage() method to end the current page and start a new page.

When all of the data has been written, use the close() method to end the data stream and close the output stream.

Reading spooled files and AFP resources

You can use the PrintObjectInputStream class to read the raw contents of a spooled file or Advanced Function Printing (AFP) resource from the AS/400. The class extends the standard JDK java.io.InputStream class so that it can be used anywhere an InputStream is used.

Obtain a PrintObjectInputStream object by calling either the getInputStream() method on an instance of the SpooledFile class or the getInputStream() method on an instance of the AFPResource class. Getting an input stream for a spooled file is supported for Version 3 Release 2 (V3R2), V3R7, and later versions of the OS/400 program. Getting input streams for AFP resources is supported for V3R7 and later.

Use one of the read() methods for reading from the input stream. These methods all return the number of bytes actually read, or -1 if no bytes were read and the end of file was reached.

Use the available() method of PrintObjectInputStream to return the total number of bytes in the spooled file or AFP resource. The PrintObjectInputStream class supports marking the input stream, so PrintObjectInputStream always returns true from the markSupported() method. The caller can use the mark() and reset() methods to move the current read position backward in the input stream. Use the skip() method to move the read position forward in the input stream without reading the data.

Example

Read Spooled File Example

Reading spooled files using PrintObjectPageInputStream and PrintObjectTransformedInputStream

You can use the PrintObjectPageInputStream class to read the data out of an AS/400 AFP and SCS spooled file one page at a time.

You can obtain a PrintObjectPageInputStream object with the getPageInputStream method.

Use one of the read() methods for reading from the input stream. All these methods return the number of bytes actually read, or -1 if no bytes were read and the end of page was reached.

Use the available() method of PrintObjectPageInputStream to return the total number of bytes in the current page. The PrintObjectPageInputStream class supports marking the input stream, so PrintObjectPageInputStream always returns true from the markSupported() method. The caller can use the mark() and reset() methods to move the current read position backward in the input stream so that subsequent reads reread the same bytes. The caller can use the skip() method to move the read position forward in the input stream without reading the data.

However, when transforming an entire spooled file data stream is desired, use the PrintObjectTransformedInputStream class.

Example

Read Spooled File Example

[Legal | AS/400 Glossary]

SpooledFileViewer class

The SpooledFileViewer class creates a window for viewing Advanced Function Printing (AFP) and Systems Network Architecture character string (SCS) files that have been spooled for printing. The class essentially adds a “print preview” function to your spooled files, common to most word processing programs. See Figure 1 below.

The spooled file viewer is especially helpful when viewing the accuracy of the data layout of the files is more important than printing the files, when viewing the data is more economical than printing them, or when a printer is not available.

Note: SS1 Option 8 (AFP Compatibility Fonts) must be installed on the host AS/400 system.

Using the SpooledFileViewer class

Three constructor methods are available to create an instance of the SpooledFileViewer class. The SpooledFileViewer() constructor can be used to create a viewer without a spooled file associated with it. If this constructor is used, a spooled file will need to be set later using setSpooledFile(SpooledFile). The SpooledFileViewer(SpooledFile) constructor can be used to create a viewer for the given spooled file, with page one as the initial view. Finally, the SpooledFileViewer(spooledFile, int) constructor can be used to create a viewer for the given spooled file with the specified page as the initial view. No matter which constructor is used, once a viewer is created, a call to load() must be performed in order to actually retrieve the spooled file data.

Then, your program can traverse the individual pages of the spooled file by using the following methods:

- load FlashPage()
- load Page()
- pageBack()
- pageForward()

If, however, you need to examine particular sections of the document more closely, you can magnify or reduce the image of a page of the document by altering the ratio proportions of each page with the following:

- fitHeight()
- fitPage()
- fitWidth()
- actualSize()

Your program would conclude with calling the close() method that closes the input stream and releases any resource associations with the stream.

Using the SpooledFileViewer

An instance of the SpooledFileViewer class is actually a graphical representation of a viewer capable of displaying and navigating through an AFP or SCS spooled file. For example, the following code creates the spooled file viewer in Figure 1 to display a spooled file previously created on the AS/400.

Note: Select each button on the image in Figure 1 below for an explanation of its function. If your browser is not JavaScript enabled, use the button link for a description of each button on the image instead.

```
// Assume splf is the spooled file.
// Create the spooled file viewer
SpooledFileViewer splfv = new SpooledFileViewer(splf, 1);
splfv.load();
// Add the spooled file viewer to a frame
JFrame frame = new JFrame("My Window");
frame.getContentPane().add(splfv);
frame.pack();
frame.show();
```

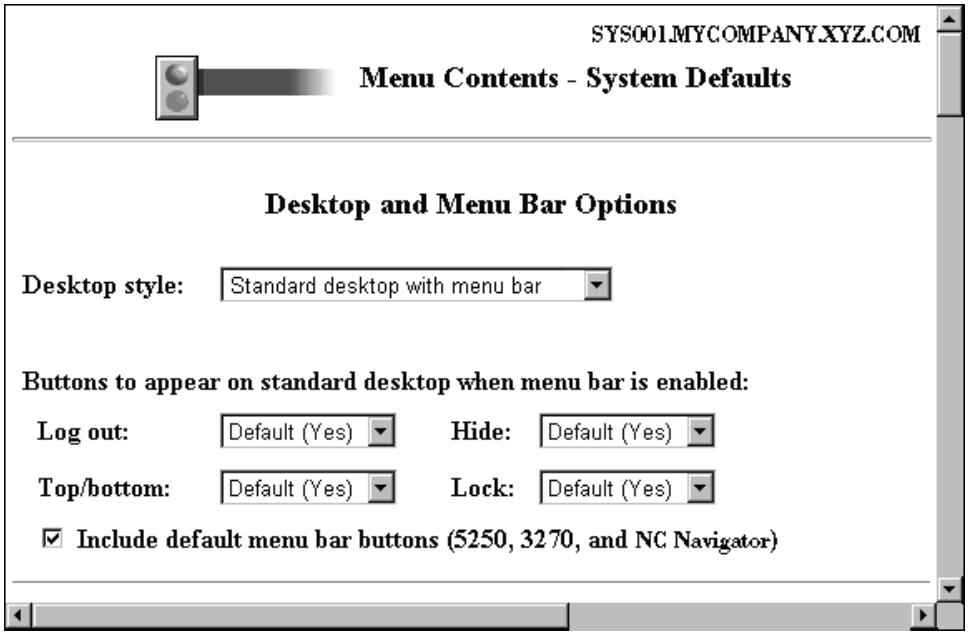
Figure 1: SpooledFileViewer..

[Legal | AS/400 Glossary]

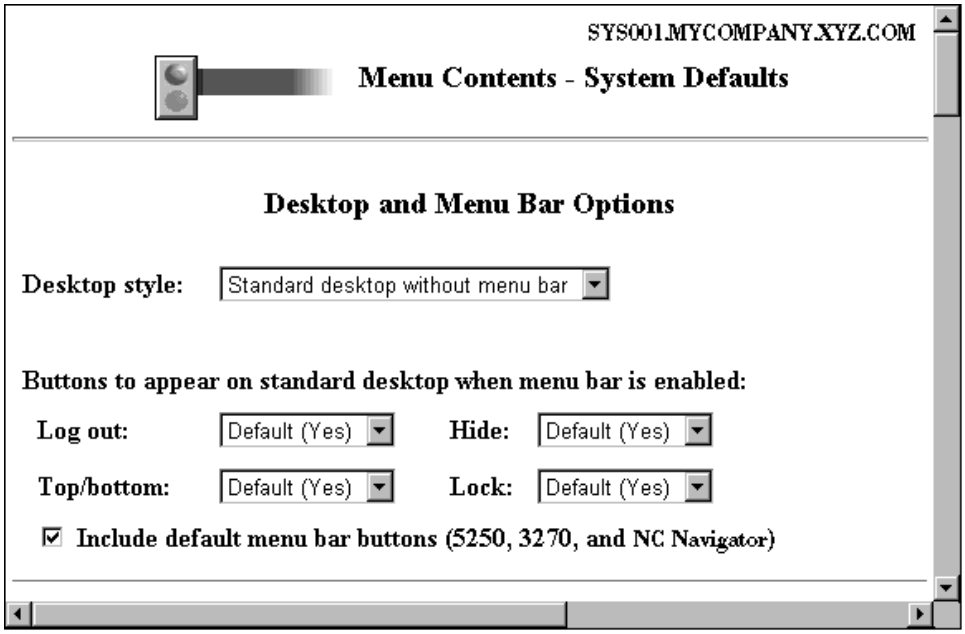
SpooledFileViewer toolbar explanation

SpooledFileViewer

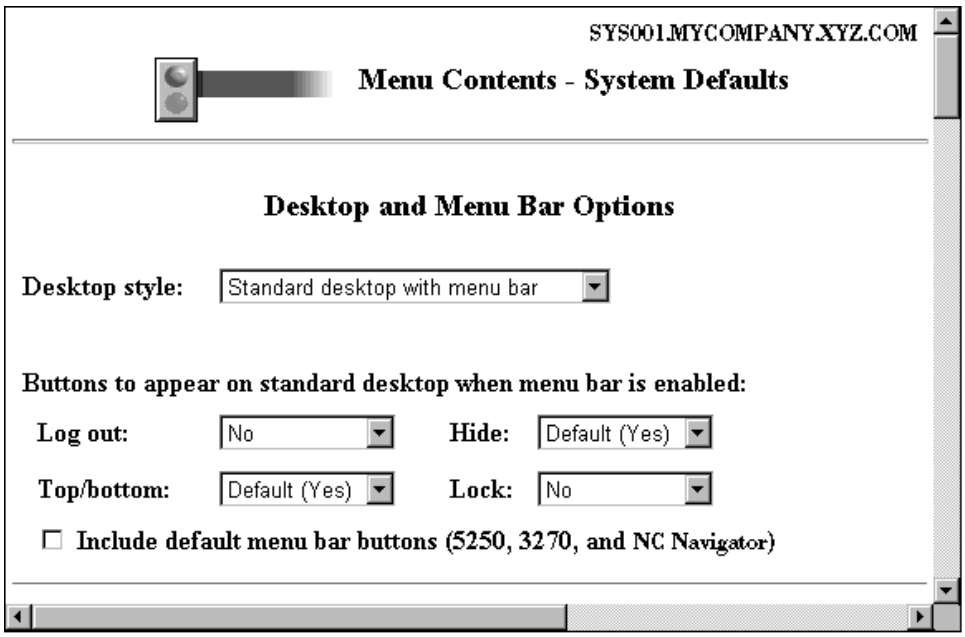
The actual size button returns the spooled file page image to its original size by using the actualSize() method.



The fit width button stretches the spooled file page image to the left and right edges of the viewer's frame by using the fitWidth() method.



The fit page button stretches the spooled file page image vertically and horizontally to fit within the spooled file viewer's frame by using the fitPage() method.



The zoom button allows you to increase or decrease the size of the spooled file page image by selecting one of the preset percentages or entering your own percent in a text field that appears in a dialog box after selecting the zoom button.

The go to page button allows you to go to a specific page within the spooled file when selected.

The first page button takes you to the first page of the spooled file when selected and indicates that you are on the first page when deactivated.

The previous page button takes you to the page immediately before the page you are viewing when selected.

The next page button advances you to the page immediately after the page you are viewing when selected.

The last page button advances you to the last page of the spooled file when selected and indicates that you are on the last page when deactivated.

The load flash page button loads the previously viewed page by using the loadFlashPage() method when selected.

The set paper size button allows you to set the paper size when selected.

The set viewing fidelity button allows you to set the viewing fidelity when selected.

[Legal | AS/400 Glossary]

Permission classes

The permission classes allow you to get and set object authority information. Object authority information is also known as permission. The Permission class represents a collection of many users' authority to a specific object. The UserPermission class represents a single user's authority to a specific object.

Permission class

The Permission class allows you to retrieve and change object authority information. It includes a collection of many users who are authorized to the object. The Permission object allows the Java program to cache authority changes until the commit() method is called. Once the commit() method is called, all changes made up to that point are sent to the AS/400. Some of the functions provided by the Permission class include:

- addAuthorizedUser(): Adds an authorized user.
- commit(): Commits the permission changes to the AS/400.
- getAuthorizationList(): Returns the authorization list of the object.
- getAuthorizedUsers(): Returns an enumeration of authorized users.
- getOwner(): Returns the name of the object owner.
- getSensitivityLevel(): Returns the sensitivity level of the object.
- getType(): Returns the object authority type (QDLO, QSYS, or Root).
- getUserPermission(): Returns the permission of a specific user to the object.
- getUserPermissions(): Returns an enumeration of permissions of the users to the object.
- setAuthorizationList(): Sets the authorization list of the object.
- setSensitivityLevel(): Sets the sensitivity level of the object.

Example

This example shows you how to create a permission and add an authorized user to an object.

```
// Create AS400 object
AS400 as400 = new AS400();

// Create Permission passing in the AS/400 and object
Permission myPermission = new Permission(as400, "QSYS.LIB/myLib.LIB");

// Add a user to be authorized to the object
myPermission.addAuthorizedUser("User1");
```

UserPermission class

The UserPermission class represents the authority of a single, specific user. UserPermission has three subclasses that handle the authority based on the object type:

Class	Description
DLOPermission	Represents a user's authority to Document Library Objects (DLOs), which are stored in QDLS.
QSYSPermission	Represents a users's authority to objects stored in QSYS.LIB and contained in the AS/400.
RootPermission	Represents a user's authority to objects contained in the root directory structure. RootPermissions objects are those objects not contained in QSYS.LIB or QDLS.

The UserPermission class allows you to do the following:

- Determine if the user profile is a group profile

- Return the user profile name
- Indicate whether the user has authority
- Set the authority of authorization list management

Example

This example shows you how to retrieve the users and groups that have permission on an object and print them out one at a time.

```
// Create a system object.
AS400 sys = new AS400("MYAS400", "USERID", "PASSWORD");
// Represent the permissions to an object on the system, such as a library.
Permission objectInQSYS = new Permission(sys, "/QSYS.LIB/FRED.LIB");
// Retrieve the various users/groups that have permissions set on that object.
Enumeration enum = objectInQSYS.getUserPermissions();
while (enum.hasMoreElements())
{
  // Print out the user/group profile names one at a time.
  UserPermission userPerm = (UserPermission)enum.nextElement();
  System.out.println(userPerm.getUserID());
}
```

[Legal | AS/400 Glossary]

DLOPermission

DLOPermission is a subclass of UserPermission. DLOPermission allows you to display and set the permission a user has for a document library object (DLO).

One of the following authority values is assigned to each user.

Value	Description
*ALL	The user can perform all operations except those operations that are controlled by authorization list management.
*AUTL	The authorization list is used to determine the authority for the document.
*CHANGE	The user can change and perform basic functions on the object.
*EXCLUDE	The user cannot access the object.
*USE	The user has object operational authority, read authority, and execute authority.

You must use one of the following methods to change or determine the user's authority:

- Use getDataAuthority() to display the authority value of the user
- Use setDataAuthority() to set the authority value of the user

To send the changes to the AS/400, use commit from the Permission class.

Warning: Temporary Level 3 Header

Example

This example shows you how to retrieve and print the dlo permissions, including the user profiles for each permission.

```
// Create a system object.
AS400 sys = new AS400("MYAS400", "USERID", "PASSWORD");
// Represent the permissions to a DLO object.
Permission objectInQDLS = new Permission(sys, "/QDLS/MyFolder");
// Print the object pathname and retrieve its permissions.
System.out.println("Permissions on "+objectInQDLS.getObjectPath()+" are as follows:");
Enumeration enum = objectInQDLS.getUserPermissions();
while (enum.hasMoreElements())
{
  // For each of the permissions, print out the user profile name
  // and that user's authorities to the object.
  DLOPermission dloPerm = (DLOPermission)enum.nextElement();
  System.out.println(dloPerm.getUserID()+": "+dloPerm.getDataAuthority());
}
```

[Legal | AS/400 Glossary]

RootPermission

RootPermission is a subclass of UserPermission. The RootPermission class allows you to display and set the permissions for the user of an object contained in the root directory structure.

An object on the root directory structure can set the data authority or the object authority. You can set the data authority to the values listed below. Use the getDataAuthority() method to to display the current values and the setDataAuthority() method to set the data authority.

Value	Description
*none	The user has no authority to the object.
*RWX	The user has read, add, update, delete, and execute authorities.
*RW	The user has read, add, and delete authorities.
*RX	The user has read and execute authorities.
*WX	The user has add, update, delete, and execute authorities.
*R	The user has read authority.
*W	The user has add, update, and delete authorities.
*X	The user has execute authority.
*EXCLUDE	The user cannot access the object.
*AUTL	The public authorities on this object come from the authorization list.

The object authority can be set to one or more of the following values: alter, existence, management, or reference. You can use the setAlter(), setExistence(), setManagement(), or setReference() methods to set the values on or off.

After setting either the data authority or the object authority of an object, it is important that you use the commit() method from the Permissions class to send the changes to the AS/400.

Warning: Temporary Level 3 Header

Example

This example shows you how to retrieve and print the permissions for a root object.

```
// Create a system object.
AS400 sys = new AS400("MYAS400", "USERID", "PASSWORD");
// Represent the permissions to an object in the root file system.
Permission objectInRoot = new Permission(sys, "/fred");
// Print the object pathname and retrieve its permissions.
System.out.println("Permissions on "+objectInRoot.getObjectPath()+" are as follows:
Enumeration enum = objectInRoot.getUserPermissions();
while (enum.hasMoreElements())
{
  // For each of the permissions, print out the user profile name
  // and that user's authorities to the object.
  RootPermission rootPerm = (RootPermission)enum.nextElement();
  System.out.println(rootPerm.getUserID()+": "+rootPerm.getDataAuthority());
}
```

[Legal | AS/400 Glossary]

Program call

The ProgramCall class allows the Java program to call an AS/400 program. You can use the ProgramParameter class to specify input, output, and input/output parameters. If the program runs, the output and input/output parameters contain the data that is returned by the AS/400 program. If the AS/400 program fails to run successfully, the Java program can retrieve any resulting AS/400 messages as a list of AS400Message objects.

Required parameters are as follows:

- The program and parameters to run
- The AS400 object that represents the AS/400 system that has the program.

The program name and parameter list can be set on the constructor, through the setProgram() method, or on the run() method The run() method calls the program.

The ProgramCall object class causes the AS400 object to connect to the AS/400.

The following example shows how to use the ProgramCall class:

```
                  // Create an AS400 object.
AS400 sys = new AS400("mySystem.myCompany.com");
                  // Create a program object. I choose
                  // to set the program to run later.
ProgramCall pgm = new ProgramCall(sys);
                  // Set the name of the program.
                  // Because the program does not take
```

```
                    // any parameters, pass null for the
                    // ProgramParameter[] argument.
pgm.setProgram(QSYSObjectPathName.toPath("MYLIB",
                                         "MYPROG",
                                         "PGM"),null;
                    // Run the program. My program has
                    // no parms. If it fails, the failure
                    // is returned as a set of messages
                    // in the message list.
if (pgm.run() != true)
{
                    // If you get here, the program
                    // failed to run. Get the list of
                    // messages to determine why the
                    // program didn't run.
   AS400Message[] messageList = pgm.getMessageList();
                    // ... Process the message list.
}
                    // Disconnect since I am done
                    // running programs
sys.disconnectService(AS400.COMMAND);
```

The ProgramCall object requires the integrated file system path name of the program.

Using the ProgramCall class causes the AS400 object to connect to the AS/400. See managing connections for information about managing connections.

Using ProgramParameter objects

You can use the ProgramParameter objects to pass parameter data between the Java program and the AS/400 program. Set the input data with the setInputData() method. After the program is run, retrieve the output data with the getOutputData() method. Each parameter is a byte array. The Java program must convert the byte array between Java and AS/400 formats. The data conversion classes provide methods for converting data. Parameters are added to the ProgramCall object as a list.

The following example shows how to use the ProgramParameter object to pass parameter data.

```
                    // Create an AS400 object
AS400 sys = new AS400("mySystem.myCompany.com");
                    // My program has two parameters.
                    // Create a list to hold these
                    // parameters.
ProgramParameter[] parmList = new ProgramParameter[2];
                    // First parameter is an input
                    // parameter
byte[] name = {1, 2, 3};
parmList[0] = new ProgramParameter(key);
                    // Second parameter is an output
                    // parameter. A four-byte number
                    // is returned.
parmList[1] = new ProgramParameter(4);
                    // Create a program object
                    // specifying the name of the
                    // program and the parameter list.
ProgramCall pgm = new ProgramCall(sys,
                                  "/QSYS.LIB/MYLIB.LIB/MYPROG.PGM",
                                  parmList);
                    // Run the program.
if (pgm.run() != true)
{
```

```
                    // If the AS/400 cannot run the
                    // program, look at the message list
                    // to find out why it didn't run.
   AS400Message[] messageList = pgm.getMessageList();
}
else
{
                    // Else the program ran. Process the
                    // second parameter, which contains
                    // the returned data.
                    // Create a converter for this
                    // AS/400 data type
   AS400Bin4 bin4Converter = new AS400Bin4();
                    // Convert from AS/400 type to Java
                    // object. The number starts at the
                    // beginning of the buffer.
   byte[] data = parmList[1].getOutputData();
   int i = bin4Converter.toInt(data);
}
                    // Disconnect since I am done
                    // running programs
sys.disconnectService(AS400.COMMAND);
```

Example

Call the AS/400 program that gets the status of the system.

[Legal | AS/400 Glossary]

QSYSObjectPathName class

You can use the QSYSObjectPathName class to represent an object in the integrated file system. Use this class to build an integrated file system name or to parse an integrated file system name into its components.

Several of the AS/400 Toolbox for Java classes require an integrated file system path name in order to be used. Use a QSYSObjectPathName object to build the name.

The following examples show how to use the QSYSObjectPathName class:

Example 1: The ProgramCall object requires the integrated file system name of the AS/400 program to call. A QSYSObjectPathName object is used to build the name. To call program PRINT_IT in library REPORTS using a QSYSObjectPathName:

```
                    // Create an AS400 object.
AS400 sys = new AS400("mySystem.myCompany.com");
                    // Create a program call object.
ProgramCall pgm = new ProgramCall(sys);
                    // Create a path name object that
                    // represents program PRINT_IT in
                    // library REPORTS.
QSYSObjectPathName pgmName = new QSYSObjectPathName("REPORTS",
                                                    "PRINT_IT",
                                                    "PGM");
                    // Use the path name object to set
                    // the name on the program call
                    // object.
pgm.setProgram(pgmName.getPath());
                    // ... run the program, process the
                    // results
```

Example 2: If the name of the AS/400 object is used just once, the Java program can use the toPath() method to build the name. This method is more efficient than creating a QSYSObjectPathName object.

```
                    // Create an AS400 object.
AS400 sys = new AS400("mySystem.myCompany.com");
                    // Create a program call object.
ProgramCall pgm = new ProgramCall(sys);
                    // Use the toPath method to create
                    // the name that represents program
                    // PRINT_IT in library REPORTS.
pgm.setProgram(QSYSObjectPathName.toPath("REPORTS",
                                         "PRINT_IT",
                                         "PGM"));
                    // ... run the program, process the
                    // results
```

Example 3: In this example, a Java program was given an integrated file system path. The QSYSObjectPathName class can be used to parse this name into its components:

```
                    // Create a path name object from
                    // the fully qualified integrated
                    // file system name.
QSYSObjectPathName ifsName = new QSYSObjectPathName(pathName);
                    // Use the path name object to get
                    // the library, name and type of
                    // AS/400 object.
String library = ifsName.getLibraryName();
String name    = ifsName.getObjectName();
String type    = ifsName.getObjectType();
```

[Legal | AS/400 Glossary]

Record-level access

Record-level access supports both Java programs and Java applets when the programs and applets are running on an AS/400 system that is at Version 4 Release 2 (V4R2) or later.

The record-level access classes provide the ability to do the following:

- Create an AS/400 physical file specifying one of the following:
  - The record length
  - An existing data description specifications (DDS) source file
  - A RecordFormat object
- Retrieve the record format from an AS/400 physical or logical file, or the record formats from an AS/400 multiple format logical file.

  Note: The record format of the file is not retrieved in its entirety. The record formats retrieved are meant to be used when setting the record format for an AS400File object. Only enough information is retrieved to describe the contents of a record of the file. Record format information, such as column headings and aliases, is not retrieved.
- Access the records in an AS/400 file sequentially, by record number, or by key.
- Write records to an AS/400 file.
- Update records in an AS/400 file sequentially, by record number, or by key.
- Delete records in an AS/400 file sequentially, by record number, or by key.
- Lock an AS/400 file for different types of access.

- Use commitment control to allow a Java program to do the following:
  - Start commitment control for the connection.
  - Specify different commitment control lock levels for different files.
  - Commit and rollback transactions.
- Delete AS/400 files.
- Delete a member from an AS/400 file.

Note: The record-level access classes do not support logical join files or null key fields.

The following classes perform these functions:

- The AS400File class is the abstract base class for the record-level access classes. It provides the methods for sequential record access, creation and deletion of files and members, and commitment control activities.
- The KeyedFile class represents an AS/400 file whose access is by key.
- The SequentialFile class represents an AS/400 file whose access is by record number.
- The AS400FileRecordDescription class provides the methods for retrieving the record format of an AS/400 file.

The record-level access classes require an AS400 object that represents the AS/400 system that has the database files. Using the record-level access classes causes the AS400 object to connect to the AS/400. See managing connections for information about managing connections.

The record-level access classes require the integrated file system path name of the data base file. See integrated file system path names for more information.

The record-level access classes use the following:

- The RecordFormat class to describe a record of the database file
- The Record class to provide access to the records of the database file

These classes are described in the data conversion section.

Examples

- The sequential access example shows how to access an AS/400 file sequentially.
- The read file example shows how to use the record-level access classes to read an AS/400 file.
- The keyed file example shows to to use the record-level access classes to read records by key from an AS/400 file.

[Legal | AS/400 Glossary]

AS400File

The AS400File class provides the methods for the following:

- Creating and deleting AS/400 physical files and members
- Reading and writing records in AS/400 files
- Locking files for different types of access
- Using record blocking to improve performance
- Setting the cursor position within an open AS/400 file
- Managing commitment control activities

Creating and deleting files and members

AS/400 physical files are created by specifying a record length, an existing AS/400 data description specifications (DDS) source file, or a RecordFormat object.

When you create a file and specify a record length, a data file or a source file can be created. The method sets the record format for the object. Do not call the setRecordFormat() method for the object.

A data file has one field. The field name is the name of the file, the field type is of type character, and the field length is the length that is specified on the create method.

A source file has three fields:

- Field SRCSEQ is ZONED DECIMAL (6,2)
- Field SRCDAT is ZONED DECIMAL (6,0)
- SRCDTA is a character field with a length that is the length specified on the create method minus 12

The following examples show how to create files and members.

Example 1: To create a data file with a 128-byte record:

```
                    // Create an AS400 object, the file
                    // will be created on this AS/400.
AS400 sys = new AS400("mySystem.myCompany.com");
                    // Create a file object that represents the file
SequentialFile newFile = new SequentialFile(sys, "/QSYS.LIB/MYLIB.LIB/MYFILE.FILE/%FILE%.MBI
                    // Create the file
newFile.create(128, "*DATA", "Data file with a 128 byte record");
                    // Open the file for writing only.
                    // Note: The record format for the file
                    // has already been set by create()
newFile.open(AS400File.WRITE_ONLY, 0, AS400File.COMMIT_LOCK_LEVEL_NONE);
                    // Write a record to the file.  Because the record
                    // format was set on the create(), getRecordFormat()
                    // can be called to get a record properly formatted
                    // for this file.
Record writeRec = newFile.getRecordFormat().getNewRecord();
writeRec.setField(0, "Record one");
newFile.write(writeRec);
                  ....
                    // Close the file since I am done using it
newFile.close();
                    // Disconnect since I am done using
                    // record-level access
sys.disconnectService(AS400.RECORDACCESS);
```

Example 2: When creating a file specifying an existing DDS source file, the DDS source file is specified on the create() method. The record format for the file must be set using the setRecordFormat() method before the file can be opened. For example:

```
                    // Create an AS400 object, the
                    // file will be created on this AS/400.
AS400 sys = new AS400("mySystem.myCompany.com");
                    // Create QSYSObjectPathName objects for
                    // both the new file and the DDS file.
QSYSObjectPathName file    = new QSYSObjectPathName("MYLIB", "MYFILE", "FILE", "MBR");
QSYSObjectPathName ddsFile = new QSYSObjectPathName("MYLIB", "DDSFILE", "FILE", "MBR");
```

```
                    // Create a file object that represents the file
SequentialFile newFile = new SequentialFile(sys, file);
                    // Create the file
newFile.create(ddsFile,  "File created using DDSFile description");
                    // Set the record format for the file
                    // by retrieving it from the AS/400.
newFile.setRecordFormat(new AS400FileRecordDescription(sys,
newFile.getPath()).retrieveRecordFormat()[0]);
                    // Open the file for writing
newFile.open(AS400File.WRITE_ONLY, 0, AS400File.COMMIT_LOCK_LEVEL_NONE);
                    // Write a record to the file.  The getRecordFormat()
                    // method followed by the getNewRecord() method is used to get
  // a default record for the file.
Record writeRec = newFile.getRecordFormat().getNewRecord();
newFile.write(writeRec);
                  ....
                    // Close the file since I am done using it
newFile.close();
                    // Disconnect since I am done using
                    // record-level access
sys.disconnectService(AS400.RECORDACCESS);
```

Example 3: When creating a file specifying a RecordFormat object, the RecordFormat object is specified on the create() method. The method sets the record format for the object. The setRecordFormat() method should not be called for the object.

```
                    // Create an AS400 object, the file will be created
                    // on this AS/400.
AS400 sys = new AS400("mySystem.myCompany.com");
                    // Create a file object that represents the file
SequentialFile newFile = new SequentialFile(sys, "/QSYS.LIB/MYLIB.LIB/MYFILE.FILE/%FILE%.MBR")
                   // Retrieve the record format from an existing file
RecordFormat recordFormat = new AS400FileRecordDescription(sys,
"/QSYS.LIB/MYLIB.LIB/EXISTING.FILE/MBR1.MBR").retrieveRecordFormat()[0];
                    // Create the file
newFile.create(recordFormat,  "File created using record format object");
                    // Open the file for writing only.
                    // Note: The record format for the file
                    // has already been set by create()
newFile.open(AS400File.WRITE_ONLY, 0, AS400File.COMMIT_LOCK_LEVEL_NONE);
                    // Write a record to the file.  The recordFormat
                    // object is used to get a default record
                    // properly formatted for the file.
Record writeRec = recordFormat.getNewRecord();
newFile.write(writeRec);
                  ....
                    // Close the file since I am done using it
newFile.close();
                    // Disconnect since I am done using
                    // record-level access
sys.disconnectService(AS400.RECORDACCESS);
```

When deleting files and members, use these methods:

- Use the delete() method to delete AS/400 files and all of their members.
- Use the deleteMember() method to delete just one member of a file.

Use the addPhysicalFileMember() method to add members to a file.

[Legal | AS/400 Glossary]

Reading and writing records

You can use the AS400File class to read, write, update, and delete records in AS/400 files. The record is accessed through the Record class, which is described by a RecordFormat class. The record format must be set through the setRecordFormat() method before the file is opened, unless the file was just created (without an intervening close()) by one of the create() methods, which sets the record format for the object.

Use the read() methods to read a record from the file. Methods are provided to do the following:

- read() - read the record at the current cursor position
- readFirst() - read the first record of the file
- readLast() - read the last record of the file
- readNext() - read the next record in the file
- readPrevious() - read the previous record in the file

The following example shows how to use the readNext() method:

```
                    // Create an AS400 object, the file exists on this
                    // AS/400.
AS400 sys = new AS400("mySystem.myCompany.com");
                    // Create a file object that represents the file
SequentialFile myFile = new SequentialFile(sys, "/QSYS.LIB/MYLIB.LIB/MYFILE.FILE/%FILE%.MBR"
                    // Assume that the AS400FileRecordDescription class
                    // was used to generate the code for a subclass of
                    // RecordFormat that represents the record format
                    // of file MYFILE in library MYLIB.  The code was
                    // compiled and is available for use by the Java
                    // program.
RecordFormat recordFormat = new MYFILEFormat();
                    // Set the record format for myFile.  This must
                    // be done prior to invoking open()
myFile.setRecordFormat(recordFormat);
                    // Open the file.
myFile.open(AS400File.READ_ONLY, 0, AS400File.COMMIT_LOCK_LEVEL_NONE);
                    // Read each record in the file writing field
                    // CUSTNAME to System.out
System.out.println("             CUSTOMER LIST");
System.out.println("________________________________________________");
Record record = myFile.readNext();
while(record != null)
{
  System.out.println(record.getField("CUSTNAME"));
  record = myFile.readNext();
}
                  ....
                    // Close the file since I am done using it
myFile.close();
                    // Disconnect since I am done using
                    // record-level access.
sys.disconnectService(AS400.RECORDACCESS);
```

Use the update() method to update the record at the cursor position.

For example:

```
                    // Create an AS400 object, the file exists on this
                    // AS/400.
AS400 sys = new AS400("mySystem.myCompany.com");
                    // Create a file object that represents the file
SequentialFile myFile = new SequentialFile(sys, "/QSYS.LIB/MYLIB.LIB/MYFILE.FILE/%FILE%.MBR"
                    // Assume that the AS400FileRecordDescription class
```

```
                    // was used to generate the code for a subclass of
                    // RecordFormat that represents the record format
                    // of file MYFILE in library MYLIB.  The code was
                    // compiled and is available for use by the Java program.
RecordFormat recordFormat = new MYFILEFormat();
                    // Set the record format for myFile.  This must
                    // be done prior to invoking open()
myFile.setRecordFormat(recordFormat);
                    // Open the file for updating
myFile.open(AS400File.READ_WRITE, 0, AS400File.COMMIT_LOCK_LEVEL_NONE);
                    // Update the first record in the file.  Assume
                    // that newName is a String with the new name for
                    // CUSTNAME
Record updateRec = myFile.readFirst();
updateRec.setField("CUSTNAME", newName);
myFile.update(updateRec);
                  ....
                    // Close the file since I am done using it
myFile.close();
                    // Disconnect since I am done using record-level access
sys.disconnectService(AS400.RECORDACCESS);
```

Use the write() method to append records to the end of a file. A single record or an array of records can be appended to the file.

Use the deleteCurrentRecord() method to delete the record at the cursor position.

[Legal | AS/400 Glossary]

Locking files

The Java program can lock a file to prevent other users from accessing the file while the first Java program is using the file. Lock types are as follows:

- Read/Exclusive Lock - The current Java program reads records, and no other program can access the file.
- Read/Allow shared read Lock - The current Java program reads records, and other programs can read records from the file.
- Read/Allow shared write Lock - The current Java program reads records, and other programs can change the file.
- Write/Exclusive Lock - The current Java program changes the file, and no other program can access the file.
- Write/Allow shared read Lock - The current Java program changes the file, and other programs can read records from the file.
- Write/Allow shared write Lock - The current Java program changes the file, and other programs can change the file.

To give up the locks obtained through the lock() method, the Java program should invoke the releaseExplicitLocks() method.

[Legal | AS/400 Glossary]

Using record blocking

The AS400File class uses record blocking to improve performance:

- If the file is opened for read-only access, a block of records is read when the Java program reads a record. Blocking improves performance because subsequent read requests may be be handled without accessing the server. Little performance difference exists between reading a single record and reading

several records. Performance improves significantly if records can be served out of the block of records cached on the client.

The number of records to read in each block can be set when the file is opened. For example:

```
                    // Create an AS400 object, the file exists on this
                    // AS/400.
  AS400 sys = new AS400("mySystem.myCompany.com");
                    // Create a file object that represents the file
  SequentialFile myFile = new SequentialFile(sys, "/QSYS.LIB/MYLIB.LIB/MYFILE.FILE/%FILE%.MBR"
                    // Assume that the AS400FileRecordDescription class
                    // was used to generate the code for a subclass of
                    // RecordFormat that represents the record format
                    // of file MYFILE in library MYLIB.  The code was
                    // compiled and is available for use by the Java
                    // program.
  RecordFormat recordFormat = new MYFILEFormat();
                    // Set the record format for myFile.  This must
                    // be done prior to invoking open()
  myFile.setRecordFormat(recordFormat);
                    // Open the file.  Specify a blocking factor of 50.
  int blockingFactor = 50;
  myFile.open(AS400File.READ_ONLY, blockingFactor, AS400File.COMMIT_LOCK_LEVEL_NONE);
                    // Read the first record of the file.  Because
                    // a blocking factor was specified, 50 records
                    // are retrieved during this read() invocation.
  Record record = myFile.readFirst();
  for (int i = 1; i < 50 && record != null; i++)
  {
    // The records read in this loop will be served out of the block of
    // records cached on the client.
    record = myFile.readNext();
  }
                  ....
                    // Close the file since I am done using it
  myFile.close();
                    // Disconnect since I am done using
                    // record-level access
  sys.disconnectService(AS400.RECORDACCESS);
```

- If the file is opened for write-only access, the blocking factor indicates how many records are written to the file at one time when the write(Record[]) method is invoked.

  For example:

```
                    // Create an AS400 object, the file exists on this
                    // AS/400.
  AS400 sys = new AS400("mySystem.myCompany.com");
                    // Create a file object that represents the file
  SequentialFile myFile = new SequentialFile(sys, "/QSYS.LIB/MYLIB.LIB/MYFILE.FILE/%FILE%.MBR"
                    // Assume that the AS400FileRecordDescription class
                    // was used to generate the code for a subclass of
                    // RecordFormat that represents the record format
                    // of file MYFILE in library MYLIB.  The code was
                    // compiled and is available for use by the Java
                    // program.
  RecordFormat recordFormat = new MYFILEFormat();
                    // Set the record format for myFile.  This must
                    // be done prior to invoking open()
  myFile.setRecordFormat(recordFormat);
                    // Open the file.  Specify a blocking factor of 50.
  int blockingFactor = 50;
  myFile.open(AS400File.WRITE_ONLY, blockingFactor, AS400File.COMMIT_LOCK_LEVEL_NONE);
                    // Create an array of records to write to the file
  Record[] records = new Record[100];
  for (int i = 0; i < 100; i++)
  {
```

```
                    // Assume the file has two fields,
                    // CUSTNAME and CUSTNUM
  records[i] = recordFormat.getNewRecord();
  records[i].setField("CUSTNAME", "Customer " + String.valueOf(i));
  records[i].setField("CUSTNUM", new Integer(i));
}
                    // Write the records to the file.  Because the
                    // blocking factor is 50, only two trips to the
                    // AS/400 are made with each trip writing 50 records
myFile.write(records);
                  ....
                    // Close the file since I am done using it
myFile.close();
                    // Disconnect since I am done using
                    // record-level access
sys.disconnectService(AS400.RECORDACCESS);
```

- If the file is opened for read-write access, no blocking is done. Any blocking factor specified on open() is ignored.

[Legal | AS/400 Glossary]

Setting the cursor position

An open file has a cursor. The cursor points to the record to be read, updated, or deleted. When a file is first opened the cursor points to the beginning of the file. The beginning of the file is before the first record. Use the following methods to set the cursor position:

- positionCursorAfterLast() - Set cursor to after the last record. This method exists so Java programs can use the readPrevious() method to access records in the file.
- positionCursorBeforeFirst() - Set cursor to before the first record. This method exists so Java programs can use the readNext() method to access records in the file.
- positionCursorToFirst() - Set the cursor to the first record.
- positionCursorToLast() - Set the cursor to the last record.
- positionCursorToNext() - Move the cursor to the next record.
- positionCursorToPrevious() - Move the cursor to the previous record.

The following example shows how to use the positionCursorToFirst() method to position the cursor.

```
                    // Create an AS400 object, the file exists on this
                    // AS/400.
AS400 sys = new AS400("mySystem.myCompany.com");
                    // Create a file object that represents the file
SequentialFile myFile = new SequentialFile(sys, "/QSYS.LIB/MYLIB.LIB/MYFILE.FILE/%FILE%.MBR");
                    // Assume that the AS400FileRecordDescription class
                    // was used to generate the code for a subclass of
                    // RecordFormat that represents the record format
                    // of file MYFILE in library MYLIB.  The code was
                    // compiled and is available for use by the Java
                    // program.
RecordFormat recordFormat = new MYFILEFormat();
                    // Set the record format for myFile.  This must
                    // be done prior to invoking open()
myFile.setRecordFormat(recordFormat);
                    // Open the file.
myFile.open(AS400File.READ_WRITE, 1, AS400File.COMMIT_LOCK_LEVEL_NONE);
                    // I want to delete the first record of the file.
myFile.positionCursorToFirst();
myFile.deleteCurrentRecord();
```

```
                ....
                  // Close the file since I am done using it
myFile.close();
                  // Disconnect since I am done using
                  // record-level access
sys.disconnectService(AS400.RECORDACCESS);
```

[Legal | AS/400 Glossary]

Commitment control

Through commitment control, your Java program has another level of control over changing a file. With commitment control turned on, transactions to a file are pending until they are either committed or rolled back. If committed, all changes are put to the file. If rolled back, all changes are discarded. The transaction can be changing an existing record, adding a record, deleting a record, or even reading a record depending on the commitment control lock level specified on the open().

The levels of commitment control are as follows:

- All - Every record accessed in the file is locked until the transaction is committed or rolled back.
- Change - Updated, added, and deleted records in the file are locked until the transaction is committed or rolled back.
- Cursor Stability - Updated, added, and deleted records in the file are locked until the transaction is committed or rolled back. Records that are accessed but not changed are locked only until another record is accessed.
- None - There is no commitment control on the file. Changes are immediately put to the file and cannot be rolled back.

You can use the startCommitmentControl() method to start commitment control. Commitment control applies to the AS400 **connection**. Once commitment control is started for a connection, it applies to all files opened under that connection from the time that commitment control was started. Files opened before commitment control is started are not under commitment control. The level of commitment control for individual files is specified on the open() method. You should specify COMMIT_LOCK_LEVEL_DEFAULT to use the same level of commitment control as was specified on the startCommitmentControl() method.

For example:

```
                  // Create an AS400 object, the files exist on this
                  // AS/400.
AS400 sys = new AS400("mySystem.myCompany.com");
                  // Create three file objects
SequentialFile myFile = new SequentialFile(sys, "/QSYS.LIB/MYLIB.LIB/MYFILE.FILE/%FILE%.MBR"
SequentialFile yourFile = new SequentialFile(sys, "/QSYS.LIB/YOURLIB.LIB/YOURFILE.FILE/%FIL
SequentialFile ourFile = new SequentialFile(sys, "/QSYS.LIB/OURLIB.LIB/OURFILE.FILE/%FILE%.M
                  // Open yourFile before starting commitment control
                  // No commitment control applies to this file.  The
                  // commit lock level parameter is ignored because
                  // commitment control is not started for the connection.
yourFile.setRecordFormat(new YOURFILEFormat());
yourFile.open(AS400File.READ_WRITE, 0, AS400File.COMMIT_LOCK_LEVEL_DEFAULT);
                  // Start commitment control for the connection.
                  // Note: Any of the three files could be used for
                  // this call to startCommitmentControl().
myFile.startCommitmentControl(AS400File.COMMIT_LOCK_LEVEL_CHANGE);
                  // Open myFile and ourFile
myFile.setRecordFormat(new MYFILEFormat());
                  // Use the same commit lock level as specified
```

```
                    // when commitment control was started
myFile.open(AS400File.WRITE_ONLY, 0, COMMIT_LOCK_LEVEL_DEFAULT);
ourFile.setRecordFormat(new OURFILEFormat());
                    // Specify a different commit lock level than
                    // when commitment control was started
ourFile.open(AS400File.READ_WRITE, 0, COMMIT_LOCK_LEVEL_CURSOR_STABILITY);
                    // write and update records in all three files
                  ....
                    // Commit the changes for files myFile and ourFile.
                    // Note that the commit commits all changes for the connection
  // even though it is invoked on only one AS400File object.
myFile.commit();
                    // Close the files
myFile.close();
yourFile.close();
ourFile.close();
                    // End commitment control
  // This ends commitment control for the connection.
ourFile.endCommitmentControl();
                    // Disconnect since I am done using record-level access
sys.disconnectService(AS400.RECORDACCESS);
```

The commit() method commits all transactions since the last commit boundary for the **connection**. The rollback() method discards all transactions since the last commit boundary for the **connection**. Commitment control for a connection is ended through the endCommitmentControl() method. If a file is closed prior to invoking the commit() or rollback() method, all uncommitted transactions are rolled back. All files opened under commitment control should be closed before the endCommitmentControl() method is called.

The following examples shows how to start commitment control, commit or roll back functions, and then end commitment control:

```
                    // ... assume the AS400 object and file have been
                    // instantiated.
                    // Start commitment control for *CHANGE
aFile.startCommitmentControl(AS400File.COMMIT_LOCK_LEVEL_CHANGE);
                    // ... open the file and do several changes.  For
                    // example, update, add or delete records.
                    // Based on a flag either save or discard the
                    // transactions.
if (saveChanges)
   aFile.commit();
else
   aFile.rollback();
                    // Close the file
aFile.close();
                    // End commitment control for the connection.
aFile.endCommitmentControl();
```

[Legal | AS/400 Glossary]

KeyedFile

The KeyedFile class gives a Java program keyed access to an AS/400 file. Keyed access means that the Java program can access the records of a file by specifying a key. Methods exist to position the cursor, read, update, and delete records by key.

To position the cursor, use the following methods:

- positionCursor(Object[]) - set cursor to the first record with the specified key.
- positionCursorAfter(Object[]) - set cursor to the record after the first record with the specified key.

- positionCursorBefore(Object[]) - set cursor to the record before the first record with the specified key.

To delete a record, use the following method :

- deleteRecord(Object[]) - delete the first record with the specified key.

The read methods are:

- read(Object[]) - read the first record with the specified key.
- readAfter(Object[]) - read the record after the first record with the specified key.
- readBefore(Object[]) - read the record before the first record with the specified key.
- readNextEqual() - read the next record whose key matches the specified key. Searching starts from the record after the current cursor position.
- readPreviousEqual() - read the previous record whose key matches the specified key. Searching starts from the record before the current cursor position.

To update a record, use the following method:

- update(Object[]) - update the record with the specified key.

Methods are also provided for specifying a search criteria when positioning, reading, and updating by key. Valid search criteria values are as follows:

- Equal - find the first record whose key matches the specified key.
- Less than - find the last record whose key comes before the specified key in the key order of the file.
- Less than or equal - find the first record whose key matches the specified key. If no record matches the specified key, find the last record whose key comes before the specified key in the key order of the file.
- Greater than - find the first record whose key comes after the specified key in the key order of the file.
- Greater than or equal - find the first record whose key matches the specified key. If no record matches the specified key, find the first record whose key comes after the specified key in the key order of the file.

KeyedFile is a subclass of AS400File; all methods in AS400File are available to KeyedFile.

Specifying the key

The key for a KeyedFile object is represented by an array of Java Objects whose types and order correspond to the types and order of the key fields as specified by the RecordFormat object for the file.

The following example shows how to specify the key for the KeyedFile object.

```
                    // Specify the key for a file whose key fields, in order,
                    // are:
                    //    CUSTNAME    CHAR(10)
                    //    CUSTNUM     BINARY(9)
                    //    CUSTADDR    CHAR(100)VARLEN()
                    // Note that the last field is a variable-length field.
Object[] theKey = new Object[3];
theKey[0] = "John Doe";
theKey[1] = new Integer(445123);
theKey[2] = "2227 John Doe Lane, ANYTOWN, NY 11199";
```

A KeyedFile object accepts partial keys as well as complete keys. However, the key field values that are specified must be in order.

For example:

```
                    // Specify a partial key for a file whose key fields,
                    // in order, are:
                    //    CUSTNAME    CHAR(10)
                    //    CUSTNUM     BINARY(9)
                    //    CUSTADDR    CHAR(100)VARLEN()
Object[] partialKey = new Object[2];
partialKey[0] = "John Doe";
partialKey[1] = new Integer(445123);
                    // Example of an INVALID partial key
Object[] INVALIDPartialKey = new Object[2];
INVALIDPartialKey[0] = new Integer(445123);
INVALIDPartialKey[1] = "2227 John Doe Lane, ANYTOWN, NY 11199";
```

Null keys and null key fields are not supported.

The key field values for a record can be obtained from the Record object for a file through the getKeyFields() method.

The following example shows how to read from a file by key:

```
                    // Create an AS400 object, the file exists on this
                    // AS/400.
AS400 sys = new AS400("mySystem.myCompany.com");
                    // Create a file object that represents the file
KeyedFile myFile = new KeyedFile(sys, "/QSYS.LIB/MYLIB.LIB/MYFILE.FILE/%FILE%.MBR");
                    // Assume that the AS400FileRecordDescription class
                    // was used to generate the code for a subclass of
                    // RecordFormat that represents the record format
                    // of file MYFILE in library MYLIB.  The code was
                    // compiled and is available for use by the Java program.
RecordFormat recordFormat = new MYKEYEDFILEFormat();
                    // Set the record format for myFile.  This must
                    // be done prior to invoking open()
myFile.setRecordFormat(recordFormat);
                    // Open the file.
myFile.open(AS400File.READ_WRITE, 0, AS400File.COMMIT_LOCK_LEVEL_NONE);
                    // The record format for the file contains
                    // four key fields, CUSTNUM, CUSTNAME, PARTNUM
                    // and ORDNUM in that order.
                    // The partialKey will contain 2 key field
                    // values.  Because the key field values must be
                    // in order, the partialKey will consist of values for
                    // CUSTNUM and CUSTNAME.
Object[] partialKey = new Object[2];
partialKey[0] = new Integer(1);
partialKey[1] = "John Doe";
                    // Read the first record matching partialKey
Record keyedRecord = myFile.read(partialKey);
                    // If the record was not found, null is returned.
if (keyedRecord != null)
{ // Found the record for John Doe, print out the info.
  System.out.println("Information for customer " + (String)partialKey[1] + ":");
  System.out.println(keyedRecord);
}
                  ....
                    // Close the file since I am done using it
myFile.close();
                    // Disconnect since I am done using record-level access
sys.disconnectService(AS400.RECORDACCESS);
```

[Legal | AS/400 Glossary]

SequentialFile

The SequentialFile class gives a Java program access to an AS/400 file by record number. Methods exist to position the cursor, read, update, and delete records by record number.

To position the cursor, use the following methods:

- positionCursor(int) - set cursor to the record with the specified record number.
- positionCursorAfter(int) - set cursor to the record after the specified record number.
- positionCursorBefore(int) - set cursor to the record before the specified record number.

To delete a record, use the following method:

- deleteRecord(int) - delete the record with the specified record number.

To read a record, use the following methods:

- read(int) - read the record with the specified record number.
- readAfter(int) - read the record after the specified record number.
- readBefore(int) - read the record before the specified record number.

To update a record, use the following method:

- update(int) - update the record with the specified record number.

SequentialFile is a subclass of AS400File; all methods in AS400File are available to SequentialFile.

The following example shows how to use the SequentialFile class:

```
                    // Create an AS400 object, the file exists on this
                    // AS/400.
AS400 sys = new AS400("mySystem.myCompany.com");
                    // Create a file object that represents the file
SequentialFile myFile = new SequentialFile(sys, "/QSYS.LIB/MYLIB.LIB/MYFILE.FILE/%FILE%.MBR"
                    // Assume that the AS400FileRecordDescription class
                    // was used to generate the code for a subclass of
                    // RecordFormat that represents the record format
                    // of file MYFILE in library MYLIB.  The code was
                    // compiled and is available for use by the Java program.
RecordFormat recordFormat = new MYFILEFormat();
                    // Set the record format for myFile.  This must
                    // be done prior to invoking open()
myFile.setRecordFormat(recordFormat);
                    // Open the file.
myFile.open(AS400File.READ_WRITE, 0, AS400File.COMMIT_LOCK_LEVEL_NONE);
                    // Delete record number 2.
myFile.delete(2);
                    // Read record number 5 and update it
Record updateRec = myFile.read(5);
updateRec.setField("CUSTNAME", newName);
                    // Use the base class' update() method since I am
                    // already positioned on the record.
myFile.update(updateRec);
                    // Update record number 7
updateRec.setField("CUSTNAME", nextNewName);
updateRec.setField("CUSTNUM", new Integer(7));
myFile.update(7, updateRec);
                  ....
                    // Close the file since I am done using it
```

```
myFile.close();
                    // Disconnect since I am done using record-level access
sys.disconnectService(AS400.RECORDACCESS);
```

[Legal | AS/400 Glossary]

AS400FileRecordDescription

The AS400FileRecordDescription class provides the methods for retrieving the record format of an AS/400 file. This class provides methods for creating Java source code for subclasses of RecordFormat and for returning RecordFormat objects, which describe the record formats of user-specified AS/400 physical or logical files. The output of these methods can be used as input to an AS400File object when setting the record format.

It is recommended that the AS400FileRecordDescription class always be used to generate the RecordFormat object when the AS/400 file already exists on the AS/400 system.

Note: The AS400FileRecordDescription class does not retrieve the entire record format of a file. Only enough information is retrieved to describe the contents of the records that make up the file. Information such as column headings, aliases, and reference fields is not retrieved. Therefore, the record formats retrieved cannot necessarily be used to create a file whose record format is identical to the file from which the format was retrieved.

Creating Java source code for subclasses of RecordFormat to represent the record format of AS/400 files

The createRecordFormatSource() method creates Java source files for subclasses of the RecordFormat class. The files can be compiled and used by an application or applet as input to the AS400File.setRecordFormat() method.

The createRecordFormatSource() method should be used as a development time tool to retrieve the record formats of existing AS/400 files. This method allows the source for the subclass of the RecordFormat class to be created once, modified if necessary, compiled, and then used by many Java programs accessing the same AS/400 files. Because this method creates files on the local system, it can be used only by Java applications. The output (the Java source code), however, can be compiled and then used by Java applications and applets alike.

Note: This method overwrites files with the same names as the Java source files being created.

Example 1: The following example shows how to use the createRecordFormatSource() method:

```
                    // Create an AS400 object, the file exists on this
                    // AS/400.
AS400 sys = new AS400("mySystem.myCompany.com");
                    // Create an AS400FileRecordDescription object that represents the file
AS400FileRecordDescription myFile = new AS400FileRecordDescription(sys, "/QSYS.LIB/MYLIB.LIB/M
                    // Create the Java source file in the current working directory.
                    // Specify "package com.myCompany.myProduct;" for the
                    // package statement in the source since I will ship the class
                    // as part of my product.
myFile.createRecordFormatSource(null, "com.myCompany.myProduct");
                    // Assuming that the format name for file MYFILE is FILE1, the
```

```
                    // file FILE1Format.java will be created in the current working directory
                    // It will overwrite any file by the same name.  The name of the class
                    // will be FILE1Format.  The class will extend from RecordFormat.
```

Example 2: Compile the file you created above, FILE1Format.java, and use it as follows:

```
                    // Create an AS400 object, the file exists on this
                    // AS/400.
AS400 sys = new AS400("mySystem.myCompany.com");
                    // Create an AS400File object that represents the file
SequentialFile myFile = new SequentialFile(sys, "/QSYS.LIB/MYLIB.LIB/MYFILE.FILE");
                    // Set the record format
                    // This assumes that import.com.myCompany.myProduct.FILE1Format;
                    // has been done.
myFile.setRecordFormat(new FILE1Format());
                    // Open the file and read from it
                    ....
                    // Close the file since I am done using it
myFile.close();
                    // Disconnect since I am done using record-level access
sys.disconnectService(AS400.RECORDACCESS);
```

Creating RecordFormat objects to represent the record format of AS/400 files

The retrieveRecordFormat() method returns an array of RecordFormat objects that represent the record formats of an existing AS/400 file. Typically, only one RecordFormat object is returned in the array. When the file for which the record format is being retrieved is a multiple format logical file, more than one RecordFormat object is returned. Use this method to dynamically retrieve the record format of an existing AS/400 file during runtime. The RecordFormat object then can be used as input to the AS400File.setRecordFormat() method.

The following example shows how to use the retrieveRecordFormat() method:

```
                    // Create an AS400 object, the file exists on this
                    // AS/400.
AS400 sys = new AS400("mySystem.myCompany.com");
                    // Create an AS400FileRecordDescription object that represents the file
AS400FileRecordDescription myFile = new AS400FileRecordDescription(sys, "/QSYS.LIB/MYLIB.LI
                    // Retrieve the record format for the file
RecordFormat[] format = myFile.retrieveRecordFormat();
                    // Create an AS400File object that represents the file
SequentialFile myFile = new SequentialFile(sys, "/QSYS.LIB/MYLIB.LIB/MYFILE.FILE");
                    // Set the record format
myFile.setRecordFormat(format[0]);
                    // Open the file and read from it
                    ....
                    // Close the file since I am done using it
myFile.close();
                    // Disconnect since I am done using record-level access
sys.disconnectService(AS400.RECORDACCESS);
```

[Legal | AS/400 Glossary]

System Status

The SystemStatus classes allow you to retrieve system status information and to retrieve and change system pool information. The SystemStatus object allows you to retreive system status information including the following:

- getUsersCurrentSignedOn(): Returns the number of users currently signed on the system
- getUsersTemporarilySignedOff(): Returns the number of interactive jobs that are disconnected
- getDateAndTimeStatusGathered(): Returns the date and time when the system status information was gathered
- getJobsInSystem(): Returns the total number of user and system jobs that are currently running
- getBatchJobsRunning(): Returns the number of batch jobs currently running on the system
- getBatchJobsEnding(): Returns the number of batch jobs that are in the process of ending
- getSystemPools(): Returns an enumeration containing a SystemPool object for each system pool

In addition to the methods within the SystemStatus class, you also can access SystemPool through SystemStatus. SystemPool allows you to get information about system pools and change system pool information.

Example

This example shows you how to get the number of users that are temporarily signed off:

```
SystemStatus status = new SystemStatus(new AS400());
System.out.println(" There are " + status.getUsersTemporarilySignedOff()
+ " Users temporarily signed off.");
```

[Legal | AS/400 Glossary]

SystemPool

The SystemPool class allows you to retrieve and change system pool information including the following:

- The getPoolSize() method returns the size of the pool, and the setPoolSize() method sets the size of the pool.
- The getPoolName() method retrieves the name of the pool, and the setPoolName() method sets the pool's name.
- The getReservedSize() method returns the amount of storage in the pool that is reserved for system use.
- The getDescription() method returns the description of the system pool.
- The getMaximumActiveThreads() method returns the maximum number of threads that can be active in the pool at any one time.
- The setMaximumFaults() method sets the maximum faults-per-second guideline to use for this system pool.
- The setPriority() method sets the priority of this system pool relative to the priority of the other system pools.

Example

```
//Create AS400 object.
AS400 as400 = new AS400("system name");

//Construct a system pool object.
SystemPool systemPool = new SystemPool(as400,"*SPOOL");

//Get system pool paging option
System.out.println("Paging option : "+systemPool.getPagingOption());
```

[Legal | AS/400 Glossary]

System values

The system value classes allow a Java program to retrieve and change system values and network attributes.

This includes the ability to display the following:

- Description
- Name
- Release

Using the SystemValue class, a single system value can also be retrieved using the getValue() method and changed using the setValue() method. However, to retrieve a group of system values with the getGroup() method, SystemValueList should be used.

System value list

SystemValueList represents a list of system values on the specified AS/400 system. The list is divided into several groups that allow the Java program to access a portion of the system values at a time.

The following is a list of the groups:

- All values
- Allocation system values
- Date and time system values
- Editing system values
- Library list system values
- Message system values
- Network attributes
- Security system values
- Storage system values
- System control system values

Whenever the value of a system value is retrieved for the first time, the value is retrieved from the AS/400 and cached. On subsequent retrievals, the cached value is returned. If the current AS/400 value is desired instead of the cached value, a clear() must be done to clear the current cache.

Examples of using the SystemValue and SystemValueList classes

The following example shows how to create and retrieve a system value:

```
//Create an AS400 object AS400 sys = new AS400("mySystem.myCompany.com");
//Create a system value representing the current second on the system.
SystemValue sysval = new SystemValue(sys, "QSECOND"); //Retrieve the value.
String second = (String)sysval.getValue(); //At this point QSECOND is cached. Clear
the cache to retrieve the most //up-to-date value from the system. sysval.clear();
second = (String)sysval.getValue(); //Create a system value list. SystemValueList list
= new SystemValueList(sys); //Retrieve all the of the date/time system values.
Vector vec = list.getGroup(SystemValueList.GROUP_DATTIM); //Disconnect from
the system. sys.disconnectAllServices();
```

[Legal | AS/400 Glossary]

Trace (Serviceability)

The Trace object allows the Java program to log trace points and diagnostic messages. This information helps reproduce and diagnose problems.

The Trace class logs the following categories of information:

Conversion	**Logs character set conversions between Unicode and native code pages. This category should be used only by the AS/400 Toolbox for Java classes.**
Information	Traces the flow through a program.
Warning	Logs information about errors the program was able to recover from.
Error	Logs additional errors that cause an exception.
Diagnostic	Logs state information.
Data stream	Logs the data that flows between the AS/400 and the Java program. This category should be used only by the AS/400 Toolbox for Java classes.

The AS/400 Toolbox for Java classes also use the trace categories. When a Java program enables logging, AS/400 Toolbox for Java information is included with the information that is recorded by the application.

You can enable the trace for a single category or a set of categories. Once the categories are selected, use the setTraceOn method to turn tracing on and off. Data is written to the log using the log method.

Excessive logging can impact performance. Use the isTraceOn method to query the current state of the trace. Your Java program can use this method to determine whether it should build the trace record before it calls the log method. Calling the log method when logging is off is not an error, but it takes more time.

The default is to write log information to standard out. To redirect the log to a file, call the setFileName() method from your Java application. In general, this works only for Java applications because most browsers do not give applets access to write to the local file system.

Logging is off by default. Java programs should provide a way for the user to turn on logging so that it is easy to enable logging. For example, the application can parse for a command line parameter that indicates which category of data should be logged. The user can set this parameter when log information is needed.

The following examples show how to use the Trace class.

Example 1: The following is an example of how to use the setTraceOn method, and how to write data to a log by using the log method.

```
                    // Enable diagnostic, information, and warning logging.
Trace.setTraceDiagnosticOn(true);
Trace.setTraceInformationOn(true);
Trace.setTraceWarningOn(true);
                    // Turning tracing on.
Trace.setTraceOn(true);
                    // ... At this point in the Java program,
                    // write to the log.
Trace.log(Trace.INFORMATION, "Just entered class xxx, method xxx");
                    // Turning tracing off.
Trace.setTraceOn(false);
```

Example 2: The following examples show how to use trace. Method 2 is the preferable way to write code that uses trace.

```
                    // Method 1 - build a trace record
                    // then call the log method and let
                    // the trace class determine if the
                    // data should be logged. This will
                    // work but will be slower than the
                    // following code.
String traceData = new String("just entered class xxx, data = ");
       traceData = traceData + data + "state = " + state;
Trace.log(Trace.INFORMATION, traceData);
                    // Method 2 - check the log status
                    // before building the information to
                    // log. This is faster when tracing
                    // is not active.
if (Trace.isTraceOn() && Trace.isTraceInformationOn())
{
   String traceData = new String("just entered class xxx, data = ");
   traceData = traceData + data + "state = " + state;
   Trace.log(Trace.INFORMATION, traceData);
}
```

[Legal | AS/400 Glossary]

Users and groups

The user and group classes allow you to get a list of users and user groups on the AS/400 system as well as information about each user through a Java program. Some of the user information you can retrieve includes previous sign-on date, status, date the password was last changed, date the password expires, and user class. When you access the User object, you should use the setSystem() method to set the system name and the setName() method to set the user name. After those steps, you use the loadUserInformation() method to get the information from the AS/400.

The UserGroup object represents a special user whose user profile is a group profile. Using the getMembers() method, a list of users that are members of the group can be returned.

The Java program can iterate through the list using an enumeration. All elements in the enumeration are User objects; for example:

```
// Create an AS400 object.
AS400 system = new AS400 ("mySystem.myCompany.com");
// Create the UserList object.
UserList userList = new UserList (system);
// Get the list of all users and groups.
Enumeration enum = userList.getUsers ();
// Iterate through the list.
while (enum.hasMoreElements ())
{
User u = (User) enum.nextElement ();
System.out.println  (u);
}
```

Retrieving information about users and groups

You use a UserList to get a list of the following:

- All users and groups
- Only groups
- All users who are members of groups
- All users who are not members of groups

The only property of the UserList object that must be set is the AS400 object that represents the AS/400 system from which the list of users is to be retrieved.

By default, all users are returned. Use a combination of setUserInfo() and setGroupInfo() to specify exactly which users should be returned.

Example

Use a UserList to list all of the users in a given group. 

[Legal | AS/400 Glossary]

User space

The UserSpace class represents a user space on the AS/400. Required parameters are the name of the user space and the AS400 object that represents the AS/400 system that has the user space. Methods exist in user space class to do the following:

- Create a user space.
- Delete a user space.
- Read from a user space.
- Write to user space.
- Get the attributes of a user space. A Java program can get the initial value, length value, and automatic extendible attributes of a user space.
- Set the attributes of a user space. A Java program can set the initial value, length value, and automatic extendible attributes of a user space.

The UserSpace object requires the integrated file system path name of the program. See integrated file system path names for more information.

Using the UserSpace class causes the AS400 object to connect to the AS/400. See managing connections for information about managing connections.

The following example creates a user space, then writes data to it.

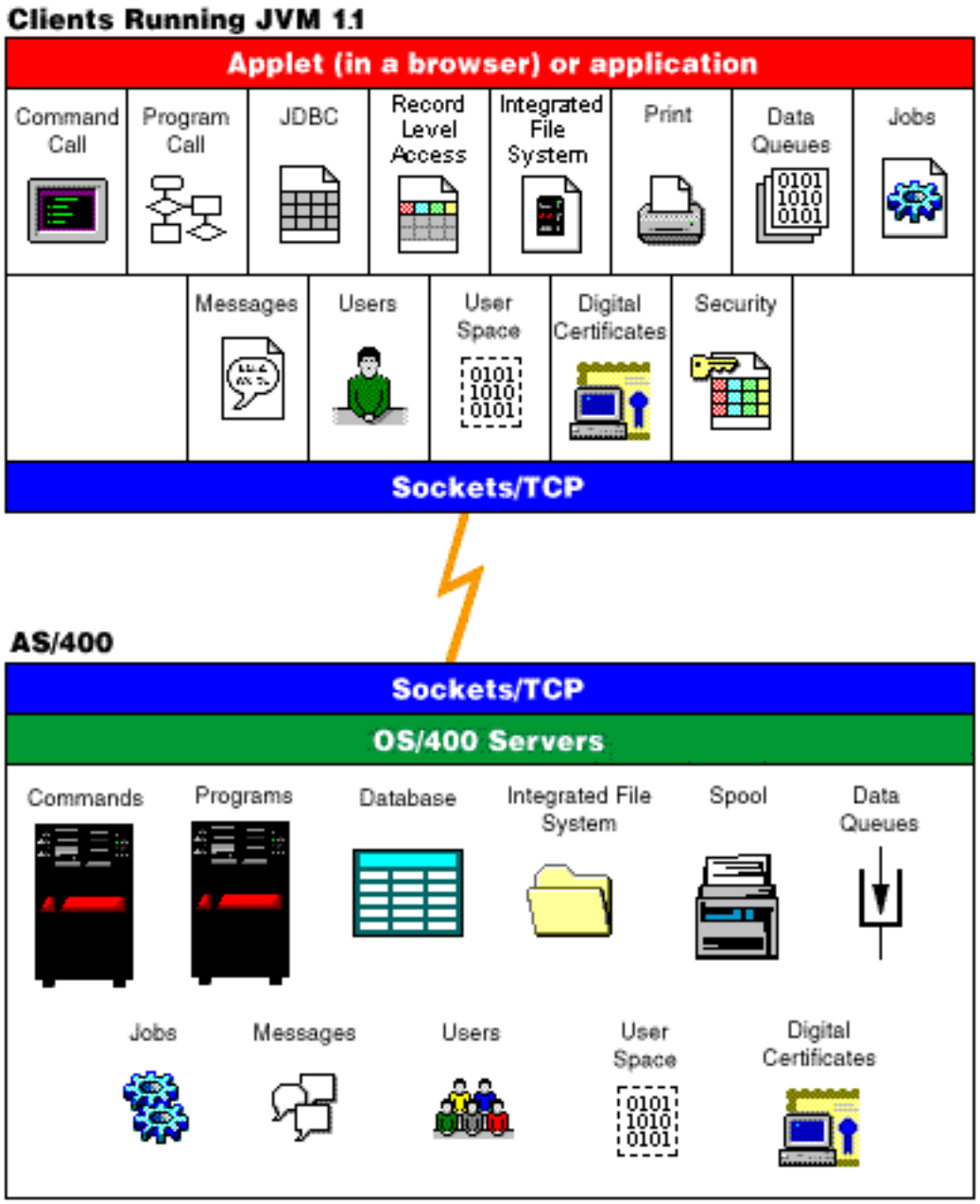
```
                    // Create an AS400 object.
AS400 sys = new AS400("mySystem.myCompany.com");
                    // Create a user space object.
UserSpace US = new UserSpace(sys,
          "/QSYS.LIB/MYLIB.LIB/MYSPACE.USRSPC");
                    // Use the create method to create the user space on
                    // the AS/400.
US.create(10240,                          // The initial size is 10K
          true,                           // Replace if the user space already exists
          " ",                            // No extended attribute
          (byte) 0x00,                    // The initial value is a null
          "Created by a Java program",    // The description of the user space
          "*USE");                        // Public has use authority to the user space
                    // Use the write method to write bytes to the user space.
US.write("Write this string to the user space.", 0);
```

[Legal | AS/400 Glossary]

AS/400 server access points

The AS/400 Toolbox for Java access classes provide functionality that is similar to using Client Access for AS/400 APIs. However, Client Access for AS/400 is not a requirement for using the classes.

The access classes use the existing AS/400 servers as the access points to the AS/400 system. Each server runs in a separate job on the AS/400 and sends and receives data streams on a socket connection.



[Legal | AS/400 Glossary]

Chapter 4. Graphical user interface classes

AS/400 Toolbox for Java provides a set of graphical user interface (GUI) classes in the vaccess package. These classes use the access classes to retrieve data and to present the data to the user.

Java programs that use the AS/400 Toolbox for Java GUI (graphical user interface) classes need Sun Microsystems, Inc., JDK 1.1.2 plus Java Swing 1.0.3 or later, in addition to AS/400 Toolbox for Java. Swing is available with Sun's JFC (Java Foundation Classes) 1.1. See http://www.javasoft.com/products/jfc/index.html 
for more information about Swing.

For more information about the relationships between the AS/400 Toolbox for Java GUI classes, the Access classes, and Java Swing, see the Graphical user interface classes diagram.

Use the AS400 panes classes to display AS/400 data.

APIs are available to access the following AS/400 resources and their tools:

- Command call
- Data queues
- Error events*
- Integrated file system
- Java database connectivity (JDBC)
- Jobs *
- Messages*
- Network print* including the spooled file viewer
- Permission
- Program call
- Record-level access
- System status
- System values
- Users and Groups

Note: AS400 panes are used with other vaccess classes (see items marked above with an asterisk) to present and allow manipulation of AS/400 resources.

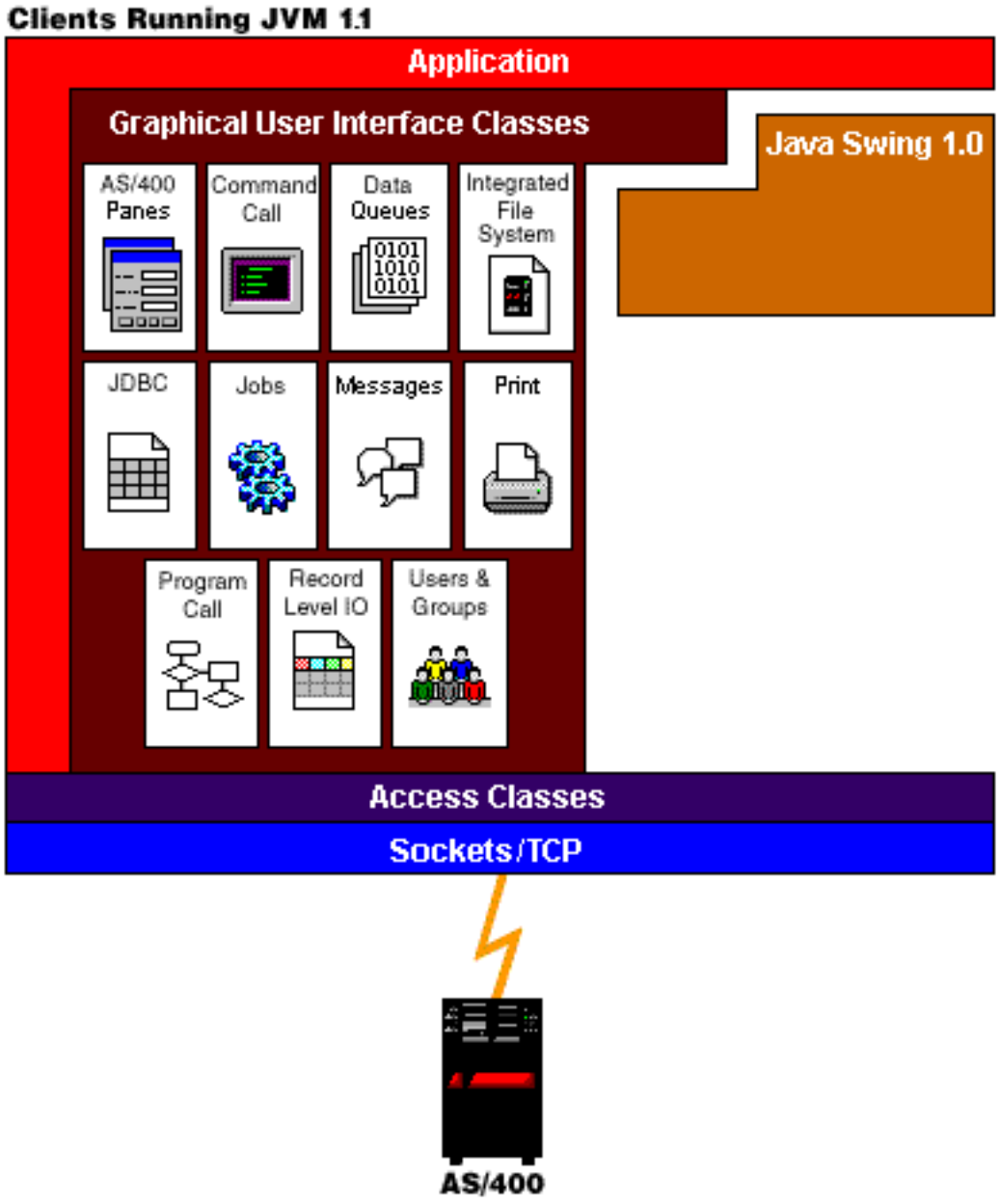
When programming with the AS/400 Toolbox for Java graphical user interface components, use the Error events classes to report and handle error events to the user.

See Access classes for more information about accessing AS/400 data.

[Legal | AS/400 Glossary]

Graphical user interface classes

AS/400 Toolbox for Java provides graphical user interface (GUI) classes to retrieve and display, and in some cases manipulate, AS/400 data. These classes use the Java Swing 1.0 (JFC 1.1) framework. The following diagram shows the relationship between these classes.



[Legal | AS/400 Glossary]

AS/400 Panes

AS/400 panes are graphical user interface components that present and allow manipulation of one or more AS/400 resource. The behavior of each AS/400 resource varies depending on the type of resource.

All panes extend the Java Component class. As a result, they can be added to any AWT Frame, Window, or Container.

The following AS/400 panes are available:

- AS400ListPane presents a list of AS/400 resources and allows selection of one or more resources.
- AS400DetailsPane presents a list of AS/400 resources in a table where each row displays various details about a single resource. The table allows selection of one or more resources.
- AS400TreePane presents a tree hierarchy of AS/400 resources and allows selection of one or more resources.
- AS400ExplorerPane combines an AS400TreePane and AS400DetailsPane so that the resource selected in the tree is presented in the details.

AS/400 resources

AS/400 resources are represented in the graphical user interface with an icon and text. AS/400 resources are defined with hierarchical relationships where a resource might have a parent and zero or more children. These are predefined relationships and are used to specify what resources are displayed in an AS/400 pane. For example, VJobList is the parent to zero or more VJobs, and this hierarchical relationship is represented graphically in an AS/400 pane.

The AS/400 Toolbox for Java provides access to the following AS/400 resources:

- VIFSDirectory represents a directory in the integrated file system.
- VJob represents a job.
- VJobList represents a list of jobs.
- VMessageList represents a list of messages returned from a CommandCall or ProgramCall.
- VMessageQueue represents a message queue.
- VPrinters represents a list of printers.
- VPrinter represents a printer.
- VPrinterOutput represents a list of spooled files.
- VUserList represents a list of users.

All resources are implementations of the VNode interface.

Setting the root

To specify which AS/400 resources are presented in an AS/400 pane, set the root using the constructor or setRoot() method. The root defines the top level object and is used differently based on the pane:

- AS400ListPane presents all of the root's children in its list.
- AS400DetailsPane presents all of the root's children in its table.
- AS400TreePane uses the root as the root of its tree.

- AS400ExplorerPane uses the root as the root of its tree.

Any combination of panes and roots is possible.

The following example creates an AS400DetailsPane to present the list of users defined on the system:

```
                    // Create the AS/400 resource
                    // representing a list of users.
                    // Assume that "system" is an AS400
                    // object created and initialized
                    // elsewhere.
VUserList userList = new VUserList (system);
                    // Create the AS400DetailsPane object
                    // and set its root to be the user
                    // list.
AS400DetailsPane detailsPane = new AS400DetailsPane ();
detailsPane.setRoot (userList);
                    // Add the details pane to a frame.
                    // Assume that "frame" is a JFrame
                    // created elsewhere.
frame.getContentPane ().add (detailsPane);
```

Loading the contents

When AS/400 pane objects and AS/400 resource objects are created, they are initialized to a default state. The relevant information that makes up the contents of the pane is not loaded at creation time.

To load the contents, the application must explicitly call the load() method. In most cases, this initiates communication to the AS/400 system to gather the relevant information. Because it can sometimes take a while to gather this information, the application can control exactly when it happens. For example, you can:

- Load the contents before adding the pane to a frame. The frame does not appear until all information is loaded.
- Load the contents after adding the pane to a frame and displaying that frame. The frame appears, but it does not contain much information. A “wait cursor” appears and the information is filled in as it is loaded.

The following example loads the contents of a details pane before adding it to a frame:

```
                    // Load the contents of the details
                    // pane. Assume that the detailsPane
                    // was created and initialized
                    // elsewhere.
detailsPane.load ();
                    // Add the details pane to a frame.
                    // Assume that "frame" is a JFrame
                    // created elsewhere.
frame.getContentPane ().add (detailsPane);
```

Actions and properties panes

At run time, the user can select a pop-up menu on any AS/400 resource. The pop-up menu presents a list of relevant actions that are available for the resource. When the user selects an action from the pop-up menu, that action is performed. Each resource has different actions defined.

In some cases, the pop-up menu also presents an item that allows the user to view a properties pane. A properties pane shows various details about the resource and may allow the user to change those details. The application can control whether actions and properties panes are available by using the setAllowActions() method on the pane.

Models

The AS/400 panes are implemented using the model-view-controller paradigm, in which the data and the user interface are separated into different classes. The AS/400 panes integrate AS/400 Toolbox for Java models with Java graphical user interface components. The models manage AS/400 resources and the graphical user interface components display them graphically and handle user interaction.

The AS/400 panes provide enough functionality for most requirements. However, if an application needs more control of the JFC component, then the application can access an AS/400 model directly and provide customized integration with a different graphical user interface component.

The following models are available:

- AS400ListModel implements the JFC ListModel interface as a list of AS/400 resources. This can be used with a JFC JList object.
- AS400DetailsModel implements the JFC TableModel interface as a table of AS/400 resources where each row contains various details about a single resource. This can be used with a JFC JTable object.
- AS400TreeModel implements the JFC TreeModel interface as a tree hierarchy of AS/400 resources. This can be used with a JFC JTree object.

Examples

- Present a list of users on the system using an AS400ListPane with a VUserList object.

  The following image shows the finished product:

- Present the list of messages generated by a command call using an AS400DetailsPane with a VMessageList object.

  The following image shows the finished product:

- Present an integrated file system directory hierarchy using an AS400TreePane with a VIFSDirectory object.

  The following image shows the finished product:

- Present network print resources using an AS400ExplorerPane with a VPrinters object.

  The following image shows the finished product:

[Legal | AS/400 Glossary]

Command Call

The command call graphical user interface components allow a Java program to present a button or menu item that calls a non-interactive AS/400 command.

A CommandCallButton object represents a button that calls an AS/400 command when pressed. The CommandCallButton class extends the Java Foundation Classes (JFC) JButton class so that all buttons have a consistent appearance and behavior.

Similarly, a CommandCallMenuItem object represents a menu item that calls an AS/400 command when selected. The CommandCallMenuItem class extends the JFC JMenuItem class so that all menu items also have a consistent appearance and behavior.

To use a command call graphical user interface component, set both the system and command properties. These properties can be set using a constructor or through the setSystem() and setCommand() methods.

The following example creates a CommandCallButton. At run time, when the button is pressed, it creates a library called “FRED”:

```
                    // Create the CommandCallButton
                    // object. Assume that "system" is
                    // an AS400 object created and
                    // initialized elsewhere.  The button
                    // text says "Press Me", and there is
                    // no icon.
CommandCallButton button = new CommandCallButton ("Press Me", null, system);
                    // Set the command that the button will run.
button.setCommand ("CRTLIB FRED");
                    // Add the button to a frame. Assume
                    // that "frame" is a JFrame created
                    // elsewhere.
frame.getContentPane ().add (button);
```

When an AS/400 command runs, it may return zero or more AS/400 messages. To detect when the AS/400 command runs, add an ActionCompletedListener to the button or menu item using the addActionCompletedListener() method. When the command runs, it fires an ActionCompletedEvent to all such listeners. A listener can use the getMessageList() method to retrieve any AS/400 messages that the command generated.

This example adds an ActionCompletedListener that processes all AS/400 messages that the command generated:

```
                    // Add an ActionCompletedListener that
                    // is implemented using an anonymous
                    // inner class. This is a convenient
                    // way to specify simple event
                    // listeners.
    button.addActionCompletedListener (new ActionCompletedListener ()
    {
         public void actionCompleted (ActionCompletedEvent event)
         {
                    // Cast the source of the event to a
                    // CommandCallButton.
              CommandCallButton sourceButton = (CommandCallButton) event.getSource ();
                    // Get the list of AS/400 messages
                    // that the command generated.
              AS400Message[] messageList = sourceButton.getMessageList ();
                    // ... Process the message list.
         }
    });
```

Examples

This example shows how to use a CommandCallMenuItem in an application.

The image below shows the CommandCall graphical user interface component:

[Legal | AS/400 Glossary]

Data queues

The data queue graphical components allow a Java program to use any Java Foundation Classes (JFC) graphical text component to read or write to an AS/400 data queue.

The DataQueueDocument and KeyedDataQueueDocument classes are implementations of the JFC Document interface. These classes can be used directly with any JFC graphical text component. Several text components, such as single line fields (JTextField) and multiple line text areas (JTextArea), are available in JFC.

Data queue documents associate the contents of a text component with an AS/400 data queue. (A text component is a graphical component used to display text that the user can optionally edit.) The Java program can read and write between the text component and data queue at any time. Use DataQueueDocument for **sequential** data queues and KeyedDataQueueDocument for **keyed** data queues.

To use a DataQueueDocument, set both the system and path properties. These properties can be set using a constructor or through the setSystem() and setPath() methods. The DataQueueDocument object is then “plugged” into the text component, usually using the text component’s constructor or setDocument() method. KeyedDataQueueDocuments work the same way.

The following example creates a DataQueueDocument whose contents are associated with a data queue:

```
                    // Create the DataQueueDocument
                    // object. Assume that "system" is
                    // an AS400 object created and
                    // initialized elsewhere.
DataQueueDocument dqDocument = new DataQueueDocument (system, "/QSYS.LIB/MYLIB.LIB/MYQUEUE.
                    // Create a text area to present the
                    // document.
JTextArea textArea = new JTextArea (dqDocument);
                    // Add the text area to a frame.
                    // Assume that "frame" is a JFrame
                    // created elsewhere.
frame.getContentPane ().add (textArea);
```

Initially, the contents of the text component are empty. Use read() or peek() to fill the contents with the next entry on the queue. Use write() to write the contents of the text component to the data queue. Note that these documents only work with String data queue entries.

Examples

Example of using a DataQueueDocument in an application.

The following image shows the DataQueueDocument graphical user interface component being used in a JTextField. A button has been added to provide a GUI interface for the user to write the contents of the test field to the data queue.

[Legal | AS/400 Glossary]

Error events

In most cases, the AS/400 Toolbox for Java graphical user interface (GUI) components fire error events instead of throw exceptions.

An error event is a wrapper around an exception that is thrown by an internal component.

You can provide an error listener that handles all error events that are fired by a particular graphical user interface component. Whenever an exception is thrown, the listener is called, and it can provide appropriate error reporting. By default, no action takes place when error events are fired.

The AS/400 Toolbox for Java provides a graphical user interface component called ErrorDialogAdapter, which automatically displays a dialog to the user whenever an error event is fired.

The following example shows how you can handle error events by displaying a dialog:

```
                    // ... all the setup work to lay out
                    // a graphical user interface
                    // component is done. Now add an
                    // ErrorDialogAdapter as a listener
                    // to the component. This will report
                    // all error events fired by that
                    // component through displaying a
                    // dialog.
ErrorDialogAdapter errorHandler = new ErrorDialogAdapter (parentFrame);
component.addErrorListener (errorHandler);
```

You can write a custom error listener to handle errors in a different way. Use the ErrorListener interface to accomplish this.

The following example shows how to define an simple error listener that only prints errors to System.out:

```
class MyErrorHandler
implements ErrorListener
{
                    // This method is invoked whenever
                    // an error event is fired.
        public void errorOccurred(ErrorEvent event)
        {
             Exception e = event.getException ();
             System.out.println ("Error: " + e.getMessage ());
        }
}
```

The following example shows how to handle error events for a graphical user interface component using this customized handler:

```
MyErrorHandler errorHandler = new MyErrorHandler ();
component.addErrorListener (errorHandler);
```

Integrated file system

The integrated file system graphical user interface components allow a Java program to present directories and files in the AS/400 integrated file system in a graphical user interface.

The following components are available:

- IFSFileDialog presents a dialog that allows the user to choose a directory and select a file by navigating through the directory hierarchy.
- VIFSDirectory is a resource that represents a directory in the integrated file system for use in AS/400 panes.
- IFSTextFileDocument represents a text file for use in any Java Foundation Classes (JFC) graphical text component.

To use the integrated file system graphical user interface components, set both the system and the path or directory properties. These properties can be set using a constructor or through the setDirectory() (for IFSFileDialog) or setSystem() and setPath() methods (for VIFSDirectory and IFSTextFileDocument).

You should set the path to something other than “/QSYS.LIB” because this directory is usually large, and downloading its contents can take a long time.

File dialogs

The IFSFileDialog class is a dialog that allows the user to traverse the directories of the AS/400 integrated file system and select a file. The caller can set the text on the buttons on the dialog. In addition, the caller can use FileFilter objects, which allow the user to limit the choices to certain files.

If the user selects a file in the dialog, use the getFileName() method to get the name of the selected file. Use the getAbsolutePath() method to get the full path name of the selected file.

The following example sets up an integrated file system file dialog with two file filters:

```
                    // Create a IFSFileDialog object
                    // setting the text of the title bar.
                    // Assume that "system" is an AS400
                    // object and "frame" is a JFrame
                    // created and initialized elsewhere.
    IFSFileDialog dialog = new IFSFileDialog (frame, "Select a file", system);
                    // Set a list of filters for the dialog.
                    // The first filter will be used
                    // when the dialog is first displayed.
    FileFilter[] filterList = {new FileFilter ("All files (*.*)", "*.*"),
                               new FileFilter ("HTML files (*.HTML", "*.HTM")};
     // Then, set the filters in the dialog.
    dialog.setFileFilter (filterList, 0);
                    // Set the text on the buttons.
    dialog.setOkButtonText ("Open");
    dialog.setCancelButtonText ("Cancel");
                    // Show the dialog. If the user
                    // selected a file by pressing the
```

```
                    // "Open" button, then print the path
                    // name of the selected file.
if (dialog.showDialog () == IFSFileDialog.OK)
     System.out.println (dialog.getAbsolutePath ());
```

Example

Present an IFSFileDialog and print the selection, if any.

The following image shows the IFSFileDialog graphical user interface component:

[Legal | AS/400 Glossary]

Directories in AS/400 panes

AS/400 panes are graphical user interface components that present and allow manipulation of one or more AS/400 resources. A VIFSDirectory object is a resource that represents a directory in the integrated file system for use in AS/400 panes. AS/400 panes and VIFSDirectory objects can be used together to present many views of the integrated file system, and to allow the user to navigate, manipulate, and select directories and files.

To use a VIFSDirectory, set both the system and path properties. You set these properties using a constructor or through the setSystem() and setPath() methods. You then plug the VIFSDirectory object into the AS/400 pane as the root, using the pane's constructor or setRoot() method.

VIFSDirectory has some other useful properties for defining the set of directories and files that are presented in AS/400 panes. Use setInclude() to specify whether directories, files, or both appear. Use setPattern() to set a filter on the items that are shown by specifying a pattern that the file name must match. You can use wildcard characters, such as "*" and "?", in the patterns. Similarly, use setFilter() to set a filter with an IFSFileFilter object.

When AS/400 pane objects and VIFSDirectory objects are created, they are initialized to a default state. The subdirectories and the files that make up the contents of the root directory have not been loaded. To load the contents, the caller must explicitly call the load() method on either object to initiate communication to the AS/400 system to gather the contents of the directory.

At run-time, a user can perform actions on any directory or file through the pop-up menu.

The following actions are available for directories:

- Create file - creates a file in the directory. This will give the file a default name.
- Create directory - creates a subdirectory with a default name.
- Rename - renames a directory.
- Delete - deletes a directory.
- Properties - displays properties such as the location, number of files and subdirectories, and modification date.

The following actions are available for files:

- Edit - edits a text file in a different window.

- View - views a text file in a different window.
- Rename - renames a file.
- Delete - deletes a file.
- Properties - displays properties such as the location, size, modification date, and attributes.

Users can only read or write directories and files to which they are authorized. In addition, the caller can prevent the user from performing actions by using the setAllowActions() method on the pane.

The following example creates a VIFSDirectory and presents it in an AS400ExplorerPane:

```
                    // Create the VIFSDirectory object.
                    // Assume that "system" in an AS400
                    // object created and initialized
                    // elsewhere.
VIFSDirectory root = new VIFSDirectory (system, "/DirectoryA/DirectoryB");
                    // Create and load an AS400ExplorerPane object.
AS400ExplorerPane explorerPane = new AS400ExplorerPane (root);
explorerPane.load ();
                    // Add the explorer pane to a frame.
                    // Assume that "frame" is a JFrame
                    // created elsewhere.
frame.getContentPane ().add (explorerPane);
```

Example

Present an integrated file system directory hierarchy using an AS400TreePane with a VIFSDirectory object.

The following image shows the VIFSDirectory graphical user interface component:

[Legal | AS/400 Glossary]

Text file documents

Text file documents allow a Java program to use any Java Foundation Classes (JFC) graphical text component to edit or view text files in AS/400 integrated file system. (A text component is a graphical component used to display text that the user can optionally edit.)

The IFSTextFileDocument class is an implementation of the JFC Document interface. It can be used directly with any JFC graphical text component. Several text components, such as single line fields (JTextField) and multiple line text areas (JTextArea), are available in JFC.

Text file documents associate the contents of a text component with a text file. The Java program can load and save between the text component and the text file at any time.

To use an IFSTextFileDocument, set both the system and path properties. These properties can be set using a constructor or through the setSystem() and setPath() methods. The IFSTextFileDocument object is then “plugged” into the text component, usually using the text component’s constructor or setDocument() method.

Initially, the contents of the text component are empty. Use load() to load the contents from the text file. Use save() to save the contents of the text component to the text file.

The following example creates and loads an IFSTextFileDocument:

```
                    // Create and load the
                    // IFSTextFileDocument object. Assume
                    // that "system" is an AS400 object
                    // created and initialized elsewhere.
IFSTextFileDocument ifsDocument = new IFSTextFileDocument (system, "/DirectoryA/MyFile.txt");
ifsDocument.load ();
                    // Create a text area to present the
                    // document.
JTextArea textArea = new JTextArea (ifsDocument);
                    // Add the text area to a frame.
                    // Assume that "frame" is a JFrame
                    // created elsewhere.
frame.getContentPane ().add (textArea);
```

Example

Present an IFSTextFileDocument in a JTextPane.

The following image shows the IFSTextFileDocument graphical user interface component:

[Legal | AS/400 Glossary]

JDBC

The Java Database Connectivity (JDBC) graphical user interface components allow a Java program to present various views and controls for accessing a database using SQL (Structured Query Language) statements and queries.

The following components are available:

- SQLStatementButton is a button that issues an SQL statement when clicked.
- SQLStatementMenuItem is a menu item that issues an SQL statement when selected.
- SQLStatementDocument is a document that can be used with any Java Foundation Classes (JFC) graphical text component to issue an SQL statement.
- SQLResultSetFormPane presents the results of an SQL query in a form.
- SQLResultSetTablePane presents the results of an SQL query in a table.
- SQLResultSetTableModel manages the results of an SQL query in a table.
- SQLQueryBuilderPane presents an interactive tool for dynamically building SQL queries.

All JDBC graphical user interface components communicate with the database using a JDBC driver. The JDBC driver must be registered with the JDBC driver manager in order for any of these components to work. The following example registers the AS/400 Toolbox for Java JDBC driver:

```
                    // Register the JDBC driver.
DriverManager.registerDriver (new com.ibm.as400.access.AS400JDBCDriver ());
```

SQL connections

An SQLConnection object represents a connection to a database using JDBC. **The SQLConnection object is used with all of the JDBC graphical user interface components.**

To use an SQLConnection, set the URL property using the constructor or setURL(). This identifies the database to which the connection is made. Other optional properties can be set:

- Use setProperties() to specify a set of JDBC connection properties.
- Use setUserName() to specify the user name for the connection.
- Use setPassword() to specify the password for the connection.

The actual connection to the database is not made when the SQLConnection object is created. Instead, it is made when getConnection() is called. This method is normally called automatically by the JDBC graphical user interface components, but it can be called at any time in order to control when the connection is made.

The following example creates and initializes an SQLConnection object:

```
                    // Create an SQLConnection object.
SQLConnection connection = new SQLConnection ();
                    // Set the URL and user name properties of the connection.
connection.setURL ("jdbc:as400://MySystem");
connection.setUserName ("Lisa");
```

An SQLConnection object can be used for more than one JDBC graphical user interface component. All such components will use the same connection, which can improve performance and resource usage. Alternately, each JDBC graphical user interface component can use a different SQL object. It is sometimes necessary to use separate connections, so that SQL statements are issued in different transactions.

When the connection is no longer needed, close the SQLConnection object using close(). This frees up JDBC resources on both the client and server.

[Legal | AS/400 Glossary]

Buttons and menu items

An SQLStatementButton object represents a button that issues an SQL (Structured Query Language) statement when pressed. The SQLStatementButton class extends the Java Foundation Classes (JFC) JButton class so that all buttons have a consistent appearance and behavior.

Similarly, an SQLStatementMenuItem object represents a menu item that issues an SQL statement when selected. The SQLStatementMenuItem class extends the JFC JMenuItem class so that all menu items have a consistent appearance and behavior.

To use either of these classes, set both the connection and the SQLStatement properties. These properties can be set using a constructor or the setConnection() and setSQLStatement() methods.

The following example creates an SQLStatementButton. When the button is pressed at run time, it deletes all records in a table:

```
                    // Create an SQLStatementButton object.
                    // The button text says "Delete All",
                    // and there is no icon.
SQLStatementButton button = new SQLStatementButton ("Delete All");
                    // Set the connection and SQLStatement
                    // properties.  Assume that "connection"
                    // is an SQLConnection object that is
                    // created and initialized elsewhere.
button.setConnection (connection);
button.setSQLStatement ("DELETE FROM MYTABLE");
                    // Add the button to a frame. Assume
                    // that "frame" is a JFrame created
                    // elsewhere.
frame.getContentPane ().add (button);
```

After the SQL statement is issued, use getResultSet(), getMoreResults(), getUpdateCount(), or getWarnings() to retrieve the results.

[Legal | AS/400 Glossary]

Documents

The SQLStatementDocument class is an implementation of the Java Foundation Classes (JFC) Document interface. It can be used directly with any JFC graphical text component. Several text components, such as single line fields (JTextField) and multiple line text areas (JTextArea), are available in JFC. SQLStatementDocument objects associate the contents of text components with SQLConnection objects. The Java program can run the SQL statement contained in the document contents at any time and then process the results, if any.

To use an SQLStatementDocument, you must set the connection property. Set this property by using the constructor or the setConnection() method. The SQLStatementDocument object is then “plugged” into the text component, usually using the text component’s constructor or setDocument() method. Use execute() at any time to run the SQL statement contained in the document.

The following example creates an SQLStatementDocument in a JTextField:

```
                    // Create an SQLStatementDocument
                    // object. Assume that "connection"
                    // is an SQLConnection object that is
                    // created and initialized elsewhere.
                    // The text of the document is
                    // initialized to a generic query.
SQLStatementDocument document = new SQLStatementDocument (connection, "SELECT * FROM QIWS.QCUS
                    // Create a text field to present the
                    // document.
JTextField textField = new JTextField ();
textField.setDocument (document);
                    // Add the text field to a frame.
                    // Assume that "frame" is a JFrame
                    // created elsewhere.
frame.getContentPane ().add (textField);
                    // Run the SQL statement that is in
                    // the text field.
document.execute ();
```

After the SQL statement is issued, use getResultSet(), getMoreResults(), getUpdateCount(), or getWarnings() to retrieve the results.

[Legal | AS/400 Glossary]

Result set form panes

An SQLResultSetFormPane presents the results of an SQL (Structed Query Language) query in a form. The form displays one record at a time and provides buttons that allow the user to scroll forward, backward, to the first or last record, or refresh the view of the results.

To use an SQLResultSetFormPane, set the connection and query properties. Set these properties by using the constructor or the setConnection() and setQuery() methods. Use load() to execute the query and present the first record in the result set. When the results are no longer needed, call close() to ensure that the result set is closed.

The following example creates an SQLResultSetFormPane object and adds it to a frame:

```
                    // Create an SQLResultSetFormPane
                    // object. Assume that "connection"
                    // is an SQLConnection object that is
                    // created and initialized elsewhere.
SQLResultSetFormPane formPane = new SQLResultSetFormPane (connection, "SELECT * FROM QIWS.Q
                    // Load the results.
formPane.load ();
                    // Add the form pane to a frame.
                    // Assume that "frame" is a JFrame
                    // created elsewhere.
frame.getContentPane ().add (formPane);
```

[Legal | AS/400 Glossary]

Result set table panes

An SQLResultSetTablePane presents the results of an SQL (Structured Query Language) query in a table. Each row in the table displays a record from the result set and each column displays a field.

To use an SQLResultSetTablePane, set the connection and query properties. Set properties by using the constructor or the setConnection() and setQuery() methods. Use load() to execute the query and present the results in the table. When the results are no longer needed, call close() to ensure that the result set is closed.

The following example creates an SQLResultSetTablePane object and adds it to a frame:

```
                    // Create an SQLResultSetTablePane
                    // object. Assume that "connection"
                    // is an SQLConnection object that is
                    // created and initialized elsewhere.
SQLResultSetTablePane tablePane = new SQLResultSetTablePane (connection, "SELECT * FROM QIWS
                    // Load the results.
tablePane.load ();
                    // Add the table pane to a frame.
                    // Assume that "frame" is a JFrame
                    // created elsewhere.
frame.getContentPane ().add (tablePane);
```

Example

Present an SQLResultSetTablePane that displays the contents of a table. This example uses an SQLStatementDocument (denoted in the following image by the

text, “Enter a SQL statement here”) that allows the user to type in any SQL statement, and an SQLStatementButton (denoted by the text, “Delete all rows”) that allows the user to delete all rows from the table.

The following image shows the SQLResultSetTablePane graphical user interface component.

[Legal | AS/400 Glossary]

Result set table models

SQLResultSetTablePane is implemented using the model-view-controller paradigm, in which the data and the user interface are separated into different classes. The implementation integrates SQLResultSetTableModel with the Java Foundation Classes’ (JFC) JTable. The SQLResultSetTableModel class manages the results of the query and JTable displays the results graphically and handles user interaction.

SQLResultSetTablePane provides enough functionality for most requirements. However, if a caller needs more control of the JFC component, then the caller can use SQLResultSetTableModel directly and provide customized integration with a different graphical user interface component.

To use an SQLResultSetTableModel, set the connection and query properties. Set these properties by using the constructor or the setConnection() and setQuery() methods. Use load() to execute the query and load the results. When the results are no longer needed, call close() to ensure that the result set is closed.

The following example creates an SQLResultSetTableModel object and presents it with a JTable:

```
                    // Create an SQLResultSetTableModel
                    // object. Assume that "connection"
                    // is an SQLConnection object that is
                    // created and initialized elsewhere.
SQLResultSetTableModel tableModel = new SQLResultSetTableModel (connection, "SELECT * FROM QIW
                    // Load the results.
tableModel.load ();
                    // Create a JTable for the model.
JTable table = new JTable (tableModel);
                    // Add the table to a frame.  Assume
                    // that "frame" is a JFrame created
                    // elsewhere.
frame.getContentPane ().add (table);
```

[Legal | AS/400 Glossary]

SQL query builders

An SQLQueryBuilderPane presents an interactive tool for dynamically building SQL queries.

To use an SQLQueryPane, set the connection property. This property can be set using the constructor or the setConnection() method. Use load() to load data needed for the query builder graphical user interface. Use getQuery() to get the SQL query that the user has built.

The following example creates an SQLQueryBuilderPane object and adds it to a frame:

```
                    // Create an SQLQueryBuilderPane
                    // object. Assume that "connection"
                    // is an SQLConnection object that is
                    // created and initialized elsewhere.
SQLQueryBuilderPane queryBuilder = new SQLQueryBuilderPane (connection);
                    // Load the data needed for the query
                    // builder.
queryBuilder.load ();
                    // Add the query builder pane to a
                    // frame. Assume that "frame" is a
                    // JFrame created elsewhere.
frame.getContentPane ().add (queryBuilder);
```

Example

Present an SQLQueryBuilderPane and a button. When the button is clicked, present the results of the query in an SQLResultSetFormPane in another frame.

The following image shows the SQLQueryBuilderPane graphical user component:

[Legal | AS/400 Glossary]

Jobs

The jobs graphical user interface components allow a Java program to present lists of AS/400 jobs and job log messages in a graphical user interface.

The following components are available:

- A VJobList object is a resource that represents a list of AS/400 jobs for use in AS/400 panes.
- A VJob object is a resource that represents the list of messages in a job log for use in AS/400 panes.

You can use AS/400 panes, VJobList objects, and VJob objects together to present many views of a job list or job log.

To use a VJobList, set the system, name, number, and user properties. Set these properties by using a constructor or through the setSystem(), setName(), setNumber(), and setUser() properties.

To use a VJob, set the system property. Set this property by using a constructor or through the setSystem() method.

Either the VJobList or VJob object is then “plugged” into the AS/400 pane as the root, using the pane’s constructor or setRoot() method.

VJobList has some other useful properties for defining the set of jobs that are presented in AS/400 panes. Use setName() to specify that only jobs with a certain name should appear. Use setNumber() to specify that only jobs with a certain number should appear. Similarly, use setUser() to specify that only jobs for a certain user should appear.

When AS/400 pane, VJobList, and VJob objects are created, they are initialized to a default state. The list of jobs or job log messages are not loaded at creation time. To load the contents, the caller must explicitly call the load() method on either object. This will initiate communication to the AS/400 system to gather the contents of the list.

At run-time, a user can perform actions on any job list, job, or job log message through the pop-up menu.

The following actions are available for jobs:

- Properties - displays many properties such as the type and status.
- Modify - changes properties

The following menu item is available for job lists:

- Properties - allows the user to set the name, number, and user properties. This may be used to change the contents of the list.

The following action is available for job log messages:

- Properties - displays many properties such as the full text, severity, and time sent.

Users can only access jobs to which they are authorized. In addition, the Java program can prevent the user from performing actions by using the setAllowActions() method on the pane.

The following example creates a VJobList and presents it in an AS400ExplorerPane:

```
                    // Create the VJobList object. Assume
                    // that "system" is an AS400 object
                    // created and initialized elsewhere.
VJobList root = new VJobList (system);
                    // Create and load an
                    // AS400ExplorerPane object.
AS400ExplorerPane explorerPane = new AS400ExplorerPane (root);
explorerPane.load ();
                    // Add the explorer pane to a frame.
                    // Assume that "frame" is a JFrame
                    // created elsewhere.
frame.getContentPane ().add (explorerPane);
```

Examples

This VJobList example presents an AS400ExplorerPane filled with a list of jobs. The list shows jobs on the system that have the same job name.

The following image shows the VJobList graphical user interface component:

[Legal | AS/400 Glossary]

Messages

The messages graphical user interface components allow a Java program to present lists of AS/400 messages in a graphical user interface.

The following components are available:

- A VMessageList object is a resource that represents a list of messages for use in AS/400 panes. This is for message lists generated by command or program calls.
- A VMessageQueue object is a resource that represents the messages in an AS/400 message queue for use in AS/400 panes.

AS/400 panes are graphical user interface components that present and allow manipulation of one or more AS/400 resources. VMessageList and VMessageQueue objects are resources that represent lists of AS/400 messages in AS/400 panes.

You can use AS/400 pane, VMessageList, and VMessageQueue objects together to present many views of a message list and to allow the user to select and perform operations on messages.

[Legal | AS/400 Glossary]

Message lists

A VMessageList object is a resource that represents a list of messages for use in AS/400 panes. This is for message lists generated by command or program calls. The following methods return message lists:

- CommandCall.getMessageList()
- CommandCallButton.getMessageList()
- CommandCallMenuItem.getMessageList()
- ProgramCall.getMessageList()
- ProgramCallButton.getMessageList()
- ProgramCallMenuItem.getMessageList()

To use a VMessageList, set the messageList property. Set this property by using a constructor or through the setMessageList() method. The VMessageList object is then “plugged” into the AS/400 pane as the root, using the pane’s constructor or setRoot() method.

When AS/400 pane and VMessageList objects are created, they are initialized to a default state. The list of messages is not loaded at creation time. To load the contents, the caller must explicitly call the load() method on either object.

At run-time, a user can perform actions on any message through the pop-up menu. The following action is available for messages:

- Properties - displays properties such as the severity, type, and date.

The caller can prevent the user from performing actions by using the setAllowActions() method on the pane.

The following example creates a VMessageList for the messages generated by a command call and presents it in an AS400DetailsPane:

```
                    // Create the VMessageList object.
                    // Assume that "command" is a
                    // CommandCall object created and run
                    // elsewhere.
VMessageList root = new VMessageList (command.getMessageList ());
                    // Create and load an AS400DetailsPane
                    // object.
AS400DetailsPane detailsPane = new AS400DetailsPane (root);
detailsPane.load ();
                    // Add the details pane to a frame.
                    // Assume that "frame" is a JFrame
                    // created elsewhere.
frame.getContentPane ().add (detailsPane);
```

Example

Present the list of messages generated by a command call using an AS400DetailsPane with a VMessageList object.

The following image shows the VMessageList graphical user interface component:

[Legal | AS/400 Glossary]

Message queues

A VMessageQueue object is a resource that represents the messages in an AS/400 message queue for use in AS/400 panes.

To use a VMessageQueue, set the system and path properties. These properties can be set using a constructor or through the setSystem() and setPath() methods. The VMessageQueue object is then “plugged” into the AS/400 pane as the root, using the pane’s constructor or setRoot() method.

VMessageQueue has some other useful properties for defining the set of messages that are presented in AS/400 panes. Use setSeverity() to specify the severity of messages that should appear. Use setSelection() to specify the type of messages that should appear.

When AS/400 pane and VMessageQueue objects are created, they are initialized to a default state. The list of messages is not loaded at creation time. To load the contents, the caller must explicitly call the load() method on either object. This will initiate communication to the AS/400 system to gather the contents of the list.

At run-time, a user can perform actions on any message queue or message through the pop-up menu. The following actions are available for message queues:

- Clear - clears the message queue.
- Properties - allows the user to set the severity and selection properties. This may be used to change the contents of the list.

The following action is available for messages on a message queue:

- Remove - removes the message from the message queue.
- Reply - replies to an inquiry message.
- Properties - displays properties such as the severity, type, and date.

Of course, users can only access message queues to which they are authorized. In addition, the caller can prevent the user from performing actions by using the setAllowActions() method on the pane.

The following example creates a VMessageQueue and presents it in an AS400ExplorerPane:

```
                    // Create the VMessageQueue object.
                    // Assume that "system" is an AS400
                    // object created and initialized
                    // elsewhere.
VMessageQueue root = new VMessageQueue (system, "/QSYS.LIB/MYLIB.LIB/MYMSGQ.MSGQ");
                    // Create and load an
                    // AS400ExplorerPane object.
AS400ExplorerPane explorerPane = new AS400ExplorerPane (root);
explorerPane.load ();
                    // Add the explorer pane to a frame.
                    // Assume that "frame" is a JFrame
                    // created elsewhere.
frame.getContentPane ().add (explorerPane);
```

Example

Present the list of messages in a message queue using an AS400ExplorerPane with a VMessageQueue object.

The following image shows the VMessageQueue graphical user interface component:

[Legal | AS/400 Glossary]

Network Print

The network print graphical user interface components allow a Java program to present lists of AS/400 network print resources in a graphical user interface.

The following components are available:

- A VPrinters object is a resource that represents a list of printers for use in AS/400 panes.
- A VPrinter object is a resource that represents a printer and its spooled files for use in AS/400 panes.
- A VPrinterOutput object is a resource that represents a list of spooled files for use in AS/400 panes.
- A SpooledFileViewer object is a resource that visually represents spooled files.

AS/400 panes are graphical user interface components that present and allow manipulation of one or more AS/400 resources. VPrinters, VPrinter, and VPrinterOutput objects are resources that represent lists of AS/400 network print resources in AS/400 panes.

AS/400 pane, VPrinters, VPrinter, and VPrinterOutput objects can be used together to present many views of network print resources and to allow the user to select and perform operations on them.

VPrinters

A VPrinters object is a resource that represents a list of printers for use in AS/400 panes.

To use a VPrinters object, set the system property. Set this property by using a constructor or through the setSystem() method. The VPrinters object is then “plugged” into the AS/400 pane as the root, using the pane’s constructor or setRoot() method.

A VPrinters object has another useful property for defining the set of printers that is presented in AS/400 panes. Use setPrinterFilter() to specify a filter that defines which printers should appear.

When AS/400 pane and VPrinters objects are created, they are initialized to a default state. The list of printers has not been loaded. To load the contents, the caller must explicitly call the load() method on either object.

At run-time, a user can perform actions on any printer list or printer through the pop-up menu. The following action is available for printer lists:

- Properties - allows the user to set the printer filter property. This may be used to change the contents of the list.

The following actions are available for printers in a printer list:

- Hold - holds the printer.
- Release - releases the printer.
- Start - starts the printer.
- Stop - stops the printer.
- Make available - makes the printer available.
- Make unavailable - makes the printer unavailable.
- Properties - displays properties of the printer and allows the user to set filters.

Users can only access printers to which they are authorized. In addition, the caller can prevent the user from performing actions by using the setAllowActions() method on the pane.

The following example creates a VPrinters object and presents it in an AS400TreePane

```
                    // Create the VPrinters object.
                    // Assume that "system" is an AS400
                    // object created and initialized
                    // elsewhere.
    VPrinters root = new VPrinters (system);
                    // Create and load an AS400TreePane
                    // object.
    AS400TreePane treePane = new AS400TreePane (root);
    treePane.load ();
                    // Add the tree pane to a frame.
                    // Assume that "frame" is a JFrame
                    // created elsewhere.
    frame.getContentPane ().add (treePane);
```

Example

Present network print resources using an AS400ExplorerPane with a VPrinters object.

The following image shows the VPrinters graphical user interface component:

[Legal | AS/400 Glossary]

VPrinters Example

```
////////////////////////////////////////////////////////////////////////
//
// VPrinters example.  This program presents various network
// print resources with an explorer pane.
//
// Command syntax:
//    VPrintersExample system
//
////////////////////////////////////////////////////////////////////////
//
// This source is an example of AS/400 Toolbox for Java "VPrinters".
// IBM grants you a nonexclusive license to use this as an example
// from which you can generate similar function tailored to
// your own specific needs.
//
// This sample code is provided by IBM for illustrative purposes
// only. These examples have not been thoroughly tested under all
// conditions. IBM, therefore, cannot guarantee or imply
// reliability, serviceability, or function of these programs.
//
// All programs contained herein are provided to you "AS IS"
// without any warranties of any kind.  The implied warranties of
// merchantablility and fitness for a particular purpose are
// expressly disclaimed.
//
// AS/400 Toolbox for Java
// (C) Copyright IBM Corp. 1997, 1998
// All rights reserved.
// US Government Users Restricted Rights -
// Use, duplication, or disclosure restricted
// by GSA ADP Schedule Contract with IBM Corp.
//
////////////////////////////////////////////////////////////////////////
import com.ibm.as400.access.*;
import com.ibm.as400.vaccess.*;
import com.sun.java.swing.*;
import java.awt.*;
import java.awt.event.*;
public class VPrintersExample
{
    public static void main (String[] args)
    {
        // If a system was not specified, then display help text and
        // exit.
        if (args.length != 1)
        {
            System.out.println("Usage:  VPrintersExample system");
            return;
        }
        try
        {
            // Create an AS400 object.  The system name was passed
```

```
            // as the first command line argument.
            AS400 system = new AS400 (args[0]);
            // Create a VPrinters object which represents the list
            // of printers attached to the system.
            VPrinters printers = new VPrinters (system);
            // Create a frame.
            JFrame f = new JFrame ("VPrinters example");
            // Create an error dialog adapter.  This will display
            // any errors to the user.
            ErrorDialogAdapter errorHandler = new ErrorDialogAdapter (f);
            // Create an explorer pane to present the network print resources.
            // Use load to load the information from the system.
            AS400ExplorerPane explorerPane = new AS400ExplorerPane (printers);
            explorerPane.addErrorListener (errorHandler);
    explorerPane.load ();
            // When the frame closes, exit.
            f.addWindowListener (new WindowAdapter () {
                public void windowClosing (WindowEvent event)
                {
                    System.exit (0);
                }
            });
            // Layout the frame with the explorer pane.
            f.getContentPane ().setLayout (new BorderLayout ());
            f.getContentPane ().add ("Center", explorerPane);
            f.pack ();
            f.show ();
        }
        catch (Exception e)
        {
           System.out.println ("Error: " + e.getMessage ());
           System.exit (0);
        }
    }
}
```

VPrinter

A VPrinter object is a resource that represents an AS/400 printer and its spooled files for use in AS/400 panes.

To use a VPrinter, set the printer property. Set this property by using a constructor or through the setPrinter() method. The VPrinter object is then “plugged” into the AS/400 pane as the root, using the pane’s constructor or setRoot() method.

When AS/400 pane and VPrinter objects are created, they are initialized to a default state. The printer’s attributes and list of spooled files are not loaded at creation time.

To load the contents, the caller must explicitly call the load() method on either object. This will initiate communication to the AS/400 system to gather the contents of the list.

At run-time, a user can perform actions on any printer or spooled file through the pop-up menu. The following actions are available for printers:

- Hold - holds the printer.
- Release - releases the printer.
- Start - starts the printer.
- Stop - stops the printer.
- Make available - makes the printer available.

- Make unavailable - makes the printer unavailable.
- Properties - displays properties of the printer and allows the user to set filters.

The following actions are available for spooled files listed for a printer:

- Reply - replies to the spooled file.
- Hold - holds the spooled file.
- Release - releases the spooled file.
- Print next - prints the next spooled file.
- Send - sends the spooled file.
- Move - moves the spooled file.
- Delete - deletes the spooled file.
- Properties - displays many properties of the spooled file and allows the user to change some of them.

Users can only access printers and spooled files to which they are authorized. In addition, the caller can prevent the user from performing actions by using the setAllowActions() method on the pane.

The following example creates a VPrinter and presents it in an AS400ExplorerPane:

```
                    // Create the VPrinter object.
                    // Assume that "system" is an AS400
                    // object created and initialized
                    // elsewhere.
    VPrinter root = new VPrinter (new Printer (system, "MYPRINTER"));
                    // Create and load an
                    // AS400ExplorerPane object.
    AS400ExplorerPane explorerPane = new AS400ExplorerPane (root);
    explorerPane.load ();
                    // Add the explorer pane to a frame.
                    // Assume that "frame" is a JFrame
                    // created elsewhere.
    frame.getContentPane ().add (explorerPane);
```

Example

Present network print resources using an AS400ExplorerPane with a VPrinter object.

The following image shows the VPrinter graphical user interface component:

[Legal | AS/400 Glossary]

VPrinter Example

```
////////////////////////////////////////////////////////////////////////
//
// VPrinter example.  This program presents a printer and its spooled
// files in an explorer pane.
//
// Command syntax:
//    VPrinterExample system
//
////////////////////////////////////////////////////////////////////////
//
// This source is an example of AS/400 Toolbox for Java "VPrinter".
// IBM grants you a nonexclusive license to use this as an example
```

```
// from which you can generate similar function tailored to
// your own specific needs.
//
// This sample code is provided by IBM for illustrative purposes
// only. These examples have not been thoroughly tested under all
// conditions. IBM, therefore, cannot guarantee or imply
// reliability, serviceability, or function of these programs.
//
// All programs contained herein are provided to you "AS IS"
// without any warranties of any kind.  The implied warranties of
// merchantablility and fitness for a particular purpose are
// expressly disclaimed.
//
// AS/400 Toolbox for Java
// (C) Copyright IBM Corp. 1997, 1998
// All rights reserved.
// US Government Users Restricted Rights -
// Use, duplication, or disclosure restricted
// by GSA ADP Schedule Contract with IBM Corp.
//
////////////////////////////////////////////////////////////////////////
import com.ibm.as400.access.*;
import com.ibm.as400.vaccess.*;
import com.sun.java.swing.*;
import java.awt.*;
import java.awt.event.*;
public class VPrinterExample
{
    public static void main (String[] args)
    {
        // If the user does not supply a printer name then show printer information
        // for a printer called OS2VPRT;
        String printerName = "OS2VPRT";
        // If a system was not specified, then display help text and
        // exit.
        if (args.length == 0)
        {
            System.out.println("Usage:  VPrinterExample system printer");
            return;
        }
        // If the user specified a name, use it instead of the default.
        if (args.length > 1)
           printerName = args[1];
        try
        {
            // Create an AS400 object.  The system name was passed
            // as the first command line argument.
            AS400 system = new AS400 (args[0]);
            // Create a Printer object (from the Toolbox access package)
            // which represents the printer, then create a VPrinter
            // object to graphically show the spooled files on the printer.
            Printer   printer = new Printer(system, printerName);
            VPrinter vprinter = new VPrinter(printer);
            // Create a frame to hold our window.
            JFrame f = new JFrame ("VPrinter Example");
            // Create an error dialog adapter.  This will display
            // any errors to the user.
            ErrorDialogAdapter errorHandler = new ErrorDialogAdapter (f);
            // Create an explorer pane to present the printer and its spooled
            // files.  Use load to load the information from the system.
            AS400ExplorerPane explorerPane = new AS400ExplorerPane (vprinter);
            explorerPane.addErrorListener (errorHandler);
                    explorerPane.load ();
            // When the frame closes, exit.
            f.addWindowListener (new WindowAdapter () {
                public void windowClosing (WindowEvent event)
                {
```

```
                    System.exit (0);
                }
            });
            // Layout the frame with the explorer pane.
            f.getContentPane ().setLayout (new BorderLayout ());
            f.getContentPane ().add ("Center", explorerPane);
            f.pack ();
            f.show ();
        }
        catch (Exception e)
        {
           System.out.println ("Error: " + e.getMessage ());
           System.exit (0);
        }
    }
}
```

Printer output

A VPrinterOutput object is a resource that represents a list of spooled files on an AS/400 for use in AS/400 panes.

To use a VPrinterOutput object, set the system property. This property can be set using a constructor or through the setSystem() method. The VPrinterOutput object is then “plugged” into the AS/400 pane as the root, using the pane’s constructor or setRoot() method.

A VPrinterOutput object has other useful properties for defining the set of spooled files that is presented in AS/400 panes. Use setFormTypeFilter() to specify which types of forms should appear. Use setUserDataFilter() to specify which user data should appear. Finally, use setUserFilter() to specify which users spooled files should appear.

When AS/400 pane and VPrinterOutput objects are created, they are initialized to a default state. The list of spooled files is not loaded at creation time. To load the contents, the caller must explicitly call the load() method on either object. This will initiate communication to the AS/400 system to gather the contents of the list.

At run-time, a user can perform actions on any spooled file or the spooled file list through the pop-up menu. The following action is available for spooled file lists:

- Properties - Allows the user to set the filter properties. This may be used to change the contents of the list.

The following actions are available for spooled files:

- Reply - replies to the spooled file.
- Hold - holds the spooled file.
- Release - releases the spooled file.
- Print next - prints the next spooled file.
- Send - sends the spooled file.
- Move - moves the spooled file.
- Delete - deletes the spooled file.
- Properties - displays many properties of the spooled file and allows the user to change some of them.

Of course, users can only access spooled files to which they are authorized. In addition, the caller can prevent the user from performing actions by using the setAllowActions() method on the pane.

The following example creates a VPrinterOutput and presents it in an AS400ListPane:

```
                    // Create the VPrinterOutput object.
                    // Assume that "system" is an AS400
                    // object created and initialized
                    // elsewhere.
    VPrinterOutput root = new VPrinterOutput (system);
                    // Create and load an AS400ListPane
                    // object.
    AS400ListPane listPane = new AS400ListPane (root);
    listPane.load ();
                    // Add the list pane to a frame.
                    // Assume that "frame" is a JFrame
                    // created elsewhere.
    frame.getContentPane ().add (listPane);
```

Example

Present a list of spooled files by using the network print resource, VPrinterOutput object.

The following image shows the VPrinterOutput graphical user interface component:

[Legal | AS/400 Glossary]

VPrinterOutput Example

```
///////////////////////////////////////////////////////////////////////
//
// VPrinterOutput example.  This program presents a list of spooled
// files on the AS/400.  All spooled files, or spooled files for
// a specific user can be displayed.
//
// Command syntax:
//    VPrinterOutputExample system <user>
//
//    (User is optional, if not specified all spooled files on the system
//    will be displayed.  Caution - listing all spooled files on the system
//    and take a long time)
//
///////////////////////////////////////////////////////////////////////
//
// This source is an example of AS/400 Toolbox for Java "VPrinterOutput".
// IBM grants you a nonexclusive license to use this as an example
// from which you can generate similar function tailored to
// your own specific needs.
//
// This sample code is provided by IBM for illustrative purposes
// only. These examples have not been thoroughly tested under all
// conditions. IBM, therefore, cannot guarantee or imply
// reliability, serviceability, or function of these programs.
//
// All programs contained herein are provided to you "AS IS"
// without any warranties of any kind.  The implied warranties of
// merchantablility and fitness for a particular purpose are
// expressly disclaimed.
//
```

```
// AS/400 Toolbox for Java
// (C) Copyright IBM Corp. 1997, 1998
// All rights reserved.
// US Government Users Restricted Rights -
// Use, duplication, or disclosure restricted
// by GSA ADP Schedule Contract with IBM Corp.
//
////////////////////////////////////////////////////////////////////////
import com.ibm.as400.access.*;
import com.ibm.as400.vaccess.*;
import com.sun.java.swing.*;
import java.awt.*;
import java.awt.event.*;
public class VPrinterOutputExample
{
    public static void main (String[] args)
    {
        // If a system was not specified, display help text and exit.
        if (args.length == 0)
        {
            System.out.println("Usage:  VPrinterOutputExample system <user>");
            return;
        }
        try
        {
            // Create an AS400 object.  The system name was passed
            // as the first command line argument.
            AS400 system = new AS400 (args[0]);
            system.connectService(AS400.PRINT);
            // Create the VPrinterOutput object.
            VPrinterOutput printerOutput = new VPrinterOutput(system);
            // If a user was specified as a command line parameter, tell
            // the printerObject to get spooled files only for that user.
            if (args.length > 1)
               printerOutput.setUserFilter(args[1]);
            // Create a frame to hold our window.
            JFrame f = new JFrame ("VPrinterOutput Example");
            // Create an error dialog adapter.  This will display
            // any errors to the user.
            ErrorDialogAdapter errorHandler = new ErrorDialogAdapter (f);
            // Create an details pane to present the list of spooled files.
            // Use load to load the information from the system.
            AS400DetailsPane detailsPane = new AS400DetailsPane (printerOutput);
            detailsPane.addErrorListener (errorHandler);
            detailsPane.load ();
            // When the frame closes, exit.
            f.addWindowListener (new WindowAdapter () {
                public void windowClosing (WindowEvent event)
                {
                    System.exit (0);
                }
            });
            // Layout the frame with the details pane.
            f.getContentPane ().setLayout (new BorderLayout ());
            f.getContentPane ().add ("Center", detailsPane);
            f.pack ();
            f.show ();
        }
        catch (Exception e)
        {
           System.out.println ("Error: " + e.getMessage ());
           System.exit (0);
        }
    }
}
```

Permission

The Permission information can be used in a graphical user interface (GUI) through the VIFSFile and VIFSDirectory classes. Permission has been added as an action in each of these classes.

The following example shows how to use Permission with the VIFSDirectory class:

```
// Create AS400 object
AS400 as400 = new AS400();

// Create an IFSDirectory using the system name
// and the full path of a QSYS object
VIFSDirectory directory = new VIFSDirectory(as400,
                                "/QSYS.LID/testlib1.lib");

// Create as explorer Pane
AS400ExplorerPane pane = new AS400ExplorerPane((VNode)directory);

// Load the information
pane.load();
```

[Legal | AS/400 Glossary]

Program call

The program call graphical user interface components allow a Java program to present a button or menu item that calls an AS/400 program. Input, output, and input/output parameters can be specified using ProgramParameter objects. When the program runs, the output and input/output parameters contain data returned by the AS/400 program.

A ProgramCallButton object represents a button that calls an AS/400 program when pressed. The ProgramCallButton class extends the Java Foundation Classes (JFC) JButton class so that all buttons have a consistent appearance and behavior.

Similarly, a ProgramCallMenuItem object represents a menu item that calls an AS/400 program when selected. The ProgramCallMenuItem class extends the JFC JMenuItem class so that all menu items also have a consistent appearance and behavior.

To use a program call graphical user interface component, set both the system and program properties. Set these properties by using a constructor or through the setSystem() and setProgram() methods.

The following example creates a ProgramCallMenuItem. At run time, when the menu item is selected, it calls a program:

```
// Create the ProgramCallMenuItem
// object. Assume that "system" is
// an AS400 object created and
// initialized elsewhere. The menu
// item text says "Select Me", and
// there is no icon.
```

```
ProgramCallMenuItem menuItem = new ProgramCallMenuItem ("Select Me", null, system);
                  // Create a path name object that
                  // represents program MYPROG in
                  // library MYLIB
QSYSObjectPathName programName = new QSYSObjectPathName("MYLIB", "MYPROG", "PGM");
                  // Set the name of the program.
menuItem.setProgram (programName.getPath());
                  // Add the menu item to a menu.
                  // Assume that the menu was created
                  // elsewhere.
menu.add (menuItem);
```

When an AS/400 program runs, it may return zero or more AS/400 messages. To detect when the AS/400 program runs, add an ActionCompletedListener to the button or menu item using the addActionCompletedListener() method. When the program runs, it fires an ActionCompletedEvent to all such listeners. A listener can use the getMessageList() method to retrieve any AS/400 messages that the program generated.

This example adds an ActionCompletedListener that processes all AS/400 messages that the program generated:

```
                  // Add an ActionCompletedListener
                  // that is implemented by using an
                  // anonymous inner class. This is a
                  // convenient way to specify simple
                  // event listeners.
menuItem.addActionCompletedListener (new ActionCompletedListener ()
{
     public void actionCompleted (ActionCompletedEvent event)
     {
                  // Cast the source of the event to a
                  // ProgramCallMenuItem.
          ProgramCallMenuItem sourceMenuItem = (ProgramCallMenuItem) event.getSource ();
                  // Get the list of AS/400 messages
                  // that the program generated.
          AS400Message[] messageList = sourceMenuItem.getMessageList ();
                  // ... Process the message list.
     }
});
```

Parameters

ProgramParameter objects are used to pass parameter data between the Java program and the AS/400 program. Input data is set with the setInputData() method. After the program is run, output data is retrieved with the getOutputData() method.

Each parameter is a byte array. It is up to the Java program to convert the byte array between Java and AS/400 formats. The data conversion classes provide methods for converting data.

You can add parameters to a program call graphical user interface component one at a time using the addParameter() method or all at once using the setParameterList() method.

For more information about using ProgramParameter objects, see the ProgramCall access class.

The following example adds two parameters:

```
                    // The first parameter is a String
                    // name of up to 100 characters.
                    // This is an input parameter.
                    // Assume that "name" is a String
                    // created and initialized elsewhere.
AS400Text parm1Converter = new AS400Text (100, system.getCcsid (), system);
ProgramParameter parm1 = new ProgramParameter (parm1Converter.toBytes (name));
menuItem.addParameter (parm1);
                    // The second parameter is an Integer
                    // output parameter.
AS400Bin4 parm2Converter = new AS400Bin4 ();
ProgramParameter parm2 = new ProgramParameter (parm2Converter.getByteLength ());
menuItem.addParameter (parm2);
                    // ... after the program is called,
                    // get the value returned as the
                    // second parameter.
int result = parm2Converter.toInt (parm2.getOutputData ());
```

Examples

Example of using a ProgramCallButton in an application.

The following image shows how the ProgramCallButton looks:

[Legal | AS/400 Glossary]

Record-Level Access

The record-level access graphical user interface components allow a Java program to present various views of AS/400 files.

The following components are available:

- RecordListFormPane presents a list of records from an AS/400 file in a form.
- RecordListTablePane presents a list of records from an AS/400 file in a table.
- RecordListTableModel manages the list of records from an AS/400 file for a table.

Keyed access

You can use the record-level access graphical user interface components with keyed access to an AS/400 file. Keyed access means that the Java program can access the records of a file by specifying a key.

Keyed access works the same for each record-level access graphical user interface component. Use setKeyed() to specify keyed access instead of sequential access. Specify a key using the constructor or the setKey() method. See Specifying the key (page) for more information about how to specify the key.

By default, only records whose keys are equal to the specified key are displayed. To change this, specify the searchType property using the constructor or setSearchType() method. Possible choices are as follows:

- KEY_EQ - Display records whose keys are equal to the specified key.
- KEY_GE - Display records whose keys are greater than or equal to the specified key.
- KEY_GT - Display records whose keys are greater than the specified key.

- KEY_LE - Display records whose keys are less than or equal to the specified key.
- KEY_LT - Display records whose keys are less than the specified key.

The following example creates a RecordListTablePane object to display all records less than or equal to a key.

```
                    // Create a key that contains a
                    // single element, the Integer 5.
Object[] key = new Object[1];
key[0] = new Integer (5);
                    // Create a RecordListTablePane
                    // object. Assume that "system" is an
                    // AS400 object that is created and
                    // initialized elsewhere. Specify
                    // the key and search type.
RecordListTablePane tablePane = new RecordListTablePane (system,
     "/QSYS.LIB/QGPL.LIB/PARTS.FILE", key, RecordListTablePane.KEY_LE);
                    // Load the file contents.
tablePane.load ();
                    // Add the table pane to a frame.
                    // Assume that "frame" is a JFrame
                    // created elsewhere.
frame.getContentPane ().add (tablePane);
```

[Legal | AS/400 Glossary]

Record list form panes

A RecordListFormPane presents the contents of an AS/400 file in a form. The form displays one record at a time and provides buttons that allow the user to scroll forward, backward, to the first or last record, or refresh the view of the file contents.

To use a RecordListFormPane, set the system and fileName properties. Set these properties by using the constructor or the setSystem() and setFileName() methods. Use load() to retrieve the file contents and present the first record. When the file contents are no longer needed, call close() to ensure that the file is closed.

The following example creates a RecordListFormPane object and adds it to a frame:

```
                    // Create a RecordListFormPane
                    // object. Assume that "system" is
                    // an AS400 object that is created
                    // and initialized elsewhere.
RecordListFormPane formPane = new RecordListFormPane (system, "/QSYS.LIB/QIWS.LIB/QCUSTCDT.
                    // Load the file contents.
formPane.load ();
                    // Add the form pane to a frame.
                    // Assume that "frame" is a JFrame
                    // created elsewhere.
frame.getContentPane ().add (formPane);
```

Example

Present an RecordListFormPane which displays the contents of a file.

The following image shows the RecordListFormPane graphical user interface component:

Record list table panes

A RecordListTablePane presents the contents of an AS/400 file in a table. Each row in the table displays a record from the file and each column displays a field.

To use a RecordListTablePane, set the system and fileName properties. Set these properties by using the constructor or the setSystem() and setFileName() methods. Use load() to retrieve the file contents and present the records in the table. When the file contents are no longer needed, call close() to ensure that the file is closed.

The following example creates a RecordListTablePane object and adds it to a frame:

```
                    // Create an RecordListTablePane
                    // object. Assume that "system" is
                    // an AS400 object that is created
                    // and initialized elsewhere.
RecordListTablePane tablePane = new RecordListTablePane (system, "/QSYS.LIB/QIWS.LIB/QCUSTCDT.
                    // Load the file contents.
tablePane.load ();
                    // Add the table pane to a frame.
                    // Assume that "frame" is a JFrame
                    // created elsewhere.
frame.getContentPane ().add (tablePane);
```


Record list table models

RecordListTablePane is implemented using the model-view-controller paradigm, in which the data and the user interface are separated into different classes. The implementation integrates RecordListTableModel with Java Foundation Classes' (JFC) JTable. The RecordListTableModel class retrieves and manages the contents of the file and JTable displays the file contents graphically and handles user interaction.

RecordListTablePane provides enough functionality for most requirements. However, if a caller needs more control of the JFC component, then the caller can use RecordListTableModel directly and provide customized integration with a different graphical user interface component.

To use a RecordListTableModel, set the system and fileName properties. Set these properties by using the constructor or the setSystem() and setFileName() methods. Use load() to retrieve the file contents. When the file contents are no longer needed, call close() to ensure that the file is closed.

The following example creates a RecordListTableModel object and presents it with a JTable:

```
                    // Create a RecordListTableModel
                    // object. Assume that "system" is
                    // an AS400 object that is created
                    // and initialized elsewhere.
RecordListTableModel tableModel = new RecordListTableModel (system, "/QSYS.LIB/QIWS.LIB/QCUSTC
                    // Load the file contents.
tableModel.load ();
                    // Create a JTable for the model.
JTable table = new JTable (tableModel);
```

```
                    // Add the table to a frame. Assume
                    // that "frame" is a JFrame
                    // created elsewhere.
frame.getContentPane ().add (table);
```

[Legal | AS/400 Glossary]

System status

The System Status graphical user interface (GUI) components allow you to create GUIs by using the exisiting AS/400 Panes. You also have the option to create your own GUIs using the Java Foundation Classes (JFC). The VSystemStatus object represents a system status on the AS/400. The VSystemPool object represents a system pool on the AS/400. The VSystemStatusPane represents a visual pane that displays the system status information.

The VSystemStatus class allows you to get information about the status of an AS/400 session within a graphical user interface (GUI) environment. Some of the possible pieces of information you can get are listed below:

- The getSystem() method returns the AS/400 system where the system status information is contained
- The getText() method returns the description text
- The setSystem() method sets the AS/400 where the system status information is located

In addition to the methods mentioned above, you can also access and change system pool information in a graphic interface.

You use VSystemStatus with VSystemStatusPane. VSystemPane is the visual display pane where information is shown for both system status and system pool.

[Legal | AS/400 Glossary]

System pool

The VSystemPool class allows you to retrieve and set system pool information from an AS/400 using a graphical user interface design. VSystemPool works with various panes in the vaccess package including the VSystemStatusPane.

Some of the methods that are available to use in VSystemPool are listed below:

- The getActions() method returns a list of actions that you can perform
- The getSystem() method returns the AS/400 system where the system pool information is found
- The setSystemPool() method sets the system pool object

[Legal | AS/400 Glossary]

System status pane

The VSystemStatusPane class allows a Java program to display system status and system pool information.

VSystemStatusPane includes the following methods:

- getVSystemStatus(): Returns the VSystemStatus information in a VSystemStatusPane.
- setAllowModifyAllPools(): Sets the value to determine if system pool information can be modified.

The following example shows you how to use the VSystemStatusPane class:

```
// Create an as400 object.
AS400 mySystem = new AS400("mySystem.myCompany.com");

// Create a VSystemStatusPane
VSystemStatusPane myPane = new VSystemStatusPane(mySystem);

// Set the value to allow pools to be modified
myPane.setAllowModifyAllPools(true);

//Load the information
myPane.load();
```

[Legal | AS/400 Glossary]

System values

The system value graphical user interface (GUI) components allow a Java program to create GUIs by using the existing AS400 Panes or by creating your own panes using the Java Foundation Classes(JFC). The VSystemValueList object represents a system value list on the AS/400.

To use the System Value GUI component, set the system name with a constructor or through the setSystem() method.

Example

The following example creates a system value GUI using the AS400Explorer Pane:

```
//Create an AS400 object
AS400 mySystem = newAS400("mySystem.myCompany.com");
VSystemValueList mySystemValueList = new VSystemValueList(mySystem);
as400Panel=new AS400ExplorerPane((VNode)mySystemValueList);
//Create and load an AS400ExplorerPane object
as400Panel.load();
```

[Legal | AS/400 Glossary]

Users and groups

The users and groups graphical user interface components allow you to present lists of AS/400 users and groups through the VUser class.

The following components are available:

- AS/400 panes are graphical user interface components that present and allow manipulation of one or more AS/400 resources.
- A VUserList object is a resource that represents a list of AS/400 users and groups for use in AS/400 panes.
- A VUserAndGroup object is a resource for use in AS/400 panes that represents groups of AS/400 users. It allows a Java program to list all users, list all groups, or list users who are not in groups.

AS/400 panes and VUserList objects can be used together to present many views of the list. They can also be used to allow the user to select users and groups.

To use a VUserList, you must first set the system property. Set this property by using a constructor or through the setSystem() method. The VUserList object is then “plugged” into the AS/400 pane as the root, using the pane’s constructor or setRoot() method.

VUserList has some other useful properties for defining the set of users and groups that are presented in AS/400 panes:

- Use the setUserInfo() method to specify the types of users that should appear.
- Use the setGroupInfo() method to specify a group name.

You can use the VUserAndGroup object to get information about the Users and Groups on the system. Before you can get information about a particular object, you need to load the information so that it can be accessed. You can display the AS/400 system in which the information is found by using the getSystem method.

When AS/400 pane objects and VUserList or VUserAndGroup objects are created, they are initialized to a default state. The list of users and groups has not been loaded. To load the contents, the Java program must explicitly call the load() method on either object to initiate communication to the AS/400 system to gather the contents of the list.

At run-time, you can perform actions on any user list, user, or group through the pop-up menu.

The following action is available for users:

- Properties - displays a list of user information including the description, user class, status, job description, output information, message information, international information, security information, and group information.

The following menu item is available for user lists:

- Properties - allows the user to set the user information and group information properties. This may be used to change the contents of the list.

The following action is available for users and groups:

- Properties - displays properties such as the user name and description.

Users can only access users and groups to which they are authorized. In addition, the Java program can prevent the user from performing actions by using the setAllowActions() method on the pane.

The following example creates a VUserList and presents it in an AS400DetailsPane:

```
// Create the VUserList object.
// Assume that "system" is an AS400
// object created and initialized
// elsewhere.
VUserList root = new VUserList (system);
// Create and load an
// AS400DetailsPane object.
AS400DetailsPane detailsPane = new AS400DetailsPane (root);
detailsPane.load ();
// Add the details pane to a frame.
// Assume that "frame" is a JFrame
// created elsewhere.
frame.getContentPane ().add (detailsPane);
```

The following example shows how to use the VUserAndGroup object:

```
// Create the VUserAndGroup object.
// Assume that "system" is an AS/400 object created and initialized elsewhere.
VUserAndGroup root = new VUserAndGroup(system);
// Create and Load an AS/400ExplorerPane
AS400ExplorerPane explorerPane = new AS400ExplorerPane(root);
explorerPane.load();

// Add the explorer pane to a frame
// Assume that "frame" is a JFrame created elsewhere
frame.getContentPane().add(explorerPane);
```

Other Examples

Present a list of users on the system using an AS400ListPane with a VUserList object.

The following image shows the VUserList graphical user interface component:

[Legal | AS/400 Glossary]

Chapter 5. Utility classes

Utility classes are classes that help you do administrative tasks. For V4R4, AS/400 Toolbox for Java offers the following utilities:

- AS400ToolboxInstaller (page 131)
- JarMaker

[Legal | AS/400 Glossary]

Client installation and update classes

The AS/400 Toolbox for Java classes can be referenced at their location in the integrated file system on the AS/400. Because program temporary fixes (PTFs) are applied to this location, Java programs that access these classes directly on the AS/400 system automatically receive these updates. Accessing the classes from the AS/400 does not work for every situation, specifically those listed below:

- If a low-speed communication link connects AS/400 and the client, the performance of loading the classes from the AS/400 may be unacceptable.
- If Java applications use the CLASSPATH environment variable to access the classes on the client file system, you need AS/400 Client Access to redirect file system calls to the AS/400. It may not be possible for AS/400 Client Access to reside on the client.

In these cases, installing the classes on the client is a better solution. The AS400ToolboxInstaller class provides client installation and update functions to manage AS/400 Toolbox for Java classes when they reside on a client.

Install and update using the AS400ToolboxInstaller class

The AS400ToolboxInstaller class provides the application programming interfaces (APIs) that are necessary to install and update AS/400 Toolbox for Java classes on the client. The AS400ToolboxInstaller class is not a stand-alone application, but a class that is intended to be a part of a Java program.

If your Java program uses AS/400 Toolbox for Java functions, you can include the AS400ToolboxInstaller class as a part of the program. When the Java program is installed or first run, it can use the AS400ToolboxInstaller class to install the AS/400 Toolbox for Java classes on the client. When the Java program is restarted, it can use the AS400ToolboxInstaller to update the classes on the client.

Note: If you are using the V3R2 or V3R2M1 version of the AS/400 Toolbox for Java and you want to upgrade to V4R2 or a later version, you must use a V4R2 or later level of the AS400ToolboxInstaller class. You must use this level to ensure that machine readable information (MRI) stored in .property files in earlier releases of the AS/400 Toolbox for Java is properly replaced by .class files used in later releases.

The AS400ToolboxInstaller class copies files to the client's local file system. This class may not work in an applet; many browsers do not allow a Java program to write to the local file system.

Use the install() method to install or update the AS/400 Toolbox for Java classes. To install or update, provide the source and target path, and the name of the package of classes in your Java program. The source URL points to the location of the control files on the server. The directory structure is copied from the server to the client.

The install() method only copies files; it **does not** update the CLASSPATH environment variable. If the install() method is successful, the Java program can call the getClasspathAdditions() method to determine what must be added to the CLASSPATH environment variable.

The following example shows how to use the AS400ToolboxInstaller class to install files from an AS/400 called “mySystem” to directory “jt400” on drive d:, and then how to determine what must be added to the CLASSPATH environment variable:

```
                    // Install the AS/400 Toolbox for Java
                    // classes on the client.
URL sourceURL = new URL("http://mySystem.myCompany.com/QIBM/ProdData/HTTP/Public/jt400/");
if (AS400ToolboxInstaller.install(
        "ACCESS",
        "d:\\jt400",
        sourceURL))
{
                    // If the AS/400 Toolbox for Java classes were installed
                    // or updated, find out what must be added to the
                    // CLASSPATH environment variable.
   Vector additions = AS400ToolboxInstaller.getClasspathAdditions();
                    // If updates must be made to CLASSPATH
   if (additions.size() > 0)
   {
                    // ... Process each classpath addition
   }
}
                    // ... Else no updates were needed.
```

Use the isInstalled() method to determine if the AS/400 Toolbox for Java classes are already installed on the client. Using the isInstalled() method allows you to determine if you want to complete the install now or postpone it to a more convenient time.

The install() method both installs and updates files on the client. A Java program can call the isUpdateNeeded() method to determine if an update is needed before calling install().

Example

The Install/Update example shows how to use the AS400ToolboxInstaller class to install and update the AS/400 Toolbox for Java package on a client workstation.

Uninstall

Use the unInstall() method to remove the AS/400 Toolbox for Java classes from the client. The unInstall method only removes files; the CLASSPATH environment variable is not changed. Call the getClasspathRemovals() method to determine what can be removed from the CLASSPATH environment variable.

[Legal | AS/400 Glossary]

JarMaker

While the JAR file format was designed to speed up the downloading of Java program files, the JarMaker class generates an even faster loading Java Toolbox JAR file through its ability to create a smaller JAR file from a larger one.

Also, the JarMaker class can unzip a JAR file for you to gain access to the individual content files for basic use.

Flexibility of JarMaker

All of the JarMaker functions are performed with the JarMaker class and the AS400ToolboxJarMaker subclass:

- The JarMaker splits a jar file or reduces the size of a jar file by removing classes that are not used.
- The AS400ToolboxJarMaker customizes JarMaker functions for easier use on jt400.jars.

According to your needs, you can invoke the JarMaker methods from within your own Java program or as a standalone program (**java utilities.JarMaker [options]**). For a complete set of options available to run at a command line prompt, see the following:

- Options for the JarMaker class
- Extended options for the AS/400 ToolboxJarMaker subclass

Using JarMaker

Uncompressing a JAR file

Suppose you wanted to uncompress just one file bundled within a JAR file. JarMaker allows you to remove the file and place it into one of the following:

- Current directory (extract(jarFile sourceJARfile))
- Another directory (extract(jarFile, outputDirectory))

For example, with the following code, you are extracting AS400.class and all of its dependent classes from jt400.jar:

```
java utilities.JarMaker -source jt400.jar
-extract outputDir
-requiredFile com/ibm/as400/access/AS400.class
```

Breaking a single JAR file into multiple, smaller JAR files

Suppose you wanted to break up a large JAR file into smaller JAR files, according to your preferences for content and size. JarMaker, accordingly, provides you with the split(jarFile sourceJARFile | int splitSizeKbytes) function.

In the following code, jt400.jar is split into a set of 300K files:

```
java utilities.AS400ToolboxJarMaker -split 300
```

Removing unused files from a JAR file

With the AS/400ToolboxJarMaker subclass of JarMaker, you can exclude any unnecessary files by selecting only the AS/400 Toolbox for Java components, languages, and CCSIDs that you need to make your application run. This extension of JarMaker also provides you with the option of including only the JavaBeans files that are associated with the components that you have chosen.

In the following command, for example, a JAR file is created containing only those Toolbox classes needed to make the command call and program call functions of the Toolbox work:

```
java utilities.AS400ToolboxJarMaker -component CommandCall,ProgramCall
```

Additionally, because it is not necessary to convert text strings between Unicode and the double byte character set (DBCS) conversion tables, you can create a 400K byte smaller JAR file by omitting the conversion tables with the -ccsid paramater:

```
java utilities.AS400ToolboxJarMaker -component CommandCAll,ProgramCall -ccsid
```

Notes:

1. The conversion table for CCSID 61952 must be included via the -ccsid parameter when including the integrated file system classes in the jar file.
2. Conversion classes are not included with the program call classes. When including program call classes the conversion classes used by your program must also be explicitly included via the -component parameter.

Tips for frequently using JarMaker

Finally, JarMaker's convenience can, also, be found in its ability to reset itself for its next invocation. That is, you can include the reset() method after each use to have the system automatically revert to the default settings upon completion, thus, enabling the reuse of the JarMaker object. 

[Legal | AS/400 Glossary]

Chapter 6. JavaBeans

JavaBeans are reuseable software components that are written in Java. The component is a piece of program code that provides a well-defined, functional unit, which can be as small as a label for a button on a window or as large as an entire application.

JavaBeans can be either visual or nonvisual components. Non-visual JavaBeans still have a visual representation, such as an icon or a name, to allow visual manipulation.

▲ All AS/400 Toolbox for Java public classes are also JavaBeans. ▼ These classes were built to Javasoft JavaBean standards; they function as reuseable components. The properties and methods for an AS/400 Toolbox for Java JavaBean are the same as the properties and methods of the class.

JavaBeans can be used within an application program or they can be visually manipulated in builder tools, such as the IBM VisualAge for Java product.

Examples

- See JavaBeans code example as an example of how to use JavaBeans in your program.
- ▲ See Visual bean builder code example as an example of how to create a program from JavaBeans by using a visual bean builder such as IBM Visual Age for Java. ▼

[Legal | AS/400 Glossary]

JavaBeans code example

The following example creates an AS400 object and a CommandCall object, and then registers listeners on the objects. The listeners on the objects print a comment when the AS/400 system connects or disconnects and when the CommandCall object completes the running of a command.

```
////////////////////////////////////////////////////////////////////////
//
// Beans example.  This program uses the JavaBeans support in the
// AS/400 Toolbox for Java classes.
//
// Command syntax:
//    BeanExample
//
////////////////////////////////////////////////////////////////////////
//
// This source is an example of JavaBeans in the AS/400 Toolbox for Java.
// IBM grants you a nonexclusive license to use this as an example from
// which you can generate a similar function tailored to your own
// specific needs.
//
// This sample code is provided by IBM for illustrative purposes only.
// These examples have not been thoroughly tested under all conditions.
// IBM, therefore, cannot guarantee or imply reliability, serviceability,
// or function of these programs.
//
// All programs contained herein are provided to you "AS IS" without any
// warranties of any kind. The implied warranties of merchantability and
```

```
// fitness for a particular purpose are expressly disclaimed.
//
// AS/400 Toolbox for Java
// (C) Copyright IBM Corp. 1997
// All rights reserved.
// US Government Users Restricted Rights -
// Use, duplication, or disclosure restricted
// by GSA ADP Schedule Contract with IBM Corp.
//
////////////////////////////////////////////////////////////////////////
import com.ibm.as400.access.AS400;
import com.ibm.as400.access.CommandCall;
import com.ibm.as400.access.ConnectionListener;
import com.ibm.as400.access.ConnectionEvent;
import com.ibm.as400.access.ActionCompletedListener;
import com.ibm.as400.access.ActionCompletedEvent;
class BeanExample
{
    AS400       as400_  = new AS400();
    CommandCall cmd_    = new CommandCall( as400_ );
    BeanExample()
    {
        // Whenever the system is connected or disconnected print a
        // comment. Do this by adding a listener to the AS400 object.
        // When a system is connected or disconnected, the AS400 object
        // will call this code.
        as400_.addConnectionListener
        (new ConnectionListener()
          {
             public void connected(ConnectionEvent event)
             {
                 System.out.println( "System connected." );
             }
             public void disconnected(ConnectionEvent event)
             {
                 System.out.println( "System disconnected." );
             }
          }
        );
        // Whenever a command runs to completion print a comment. Do this
        // by adding a listener to the commandCall object. The commandCall
        // object will call this code when it runs a command.
        cmd_.addActionCompletedListener(
            new ActionCompletedListener()
            {
                public void actionCompleted(ActionCompletedEvent event)
                {
                    System.out.println( "Command completed." );
                }
            }
        );
    }
    void runCommand()
    {
        try
        {
            // Run a command. The listeners will print comments when the
            // system is connected and when the command has run to
            // completion.
            cmd_.run( "TESTCMD PARMS" );
        }
        catch (Exception ex)
        {
            System.out.println( ex );
        }
    }
    public static void main(String[] parameters)
```

```
    {
       BeanExample be = new BeanExample();
       be.runCommand();
       System.exit(0);
    }
}
```

[Legal | AS/400 Glossary]

Visual bean builder code example

This example uses the IBM VisualAge for Java Enterprise Edition V2.0 Composition Editor, but other visual bean builders are similar. This example creates an applet for a button that, when pressed, runs a command on an AS/400.

1. Drag and drop a Button (Button1 in Figure 1) on the applet. (The Button can be found in the bean builder on the left side of the Visual Composition tab in Figure 1.)
2. Drop a CommandCall bean and an AS400 bean outside the applet. (The beans can be found in the bean builder on the left side of the Visual Composition tab in Figure 1.)

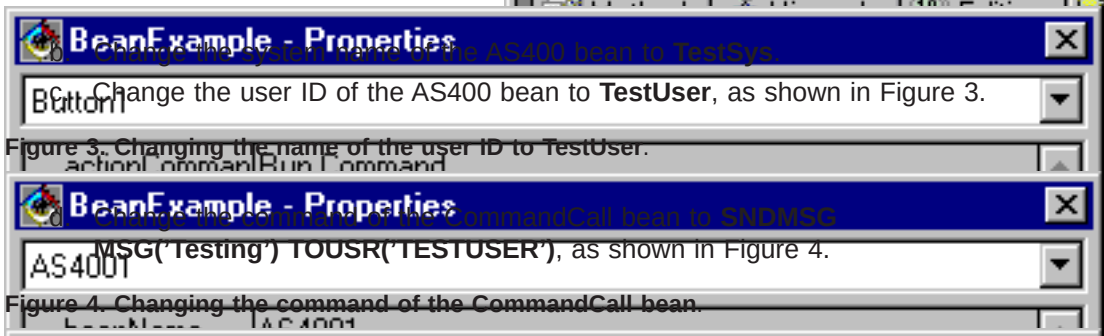
Figure 1. VisualAge Visual Composition Editor window - gui.BeanExample.

1. Edit the bean properties. (To edit, select the bean and then right-click to display a pop-up window, which has Properties as an option.)
   a. Change the label of the Button to **Run command**, as shown in Figure 2.

Figure 2. Changing the label of the button to Run command.

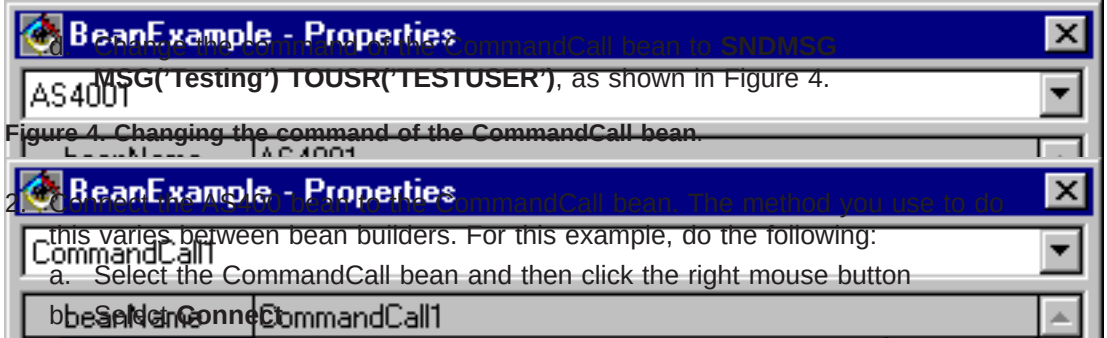
   b. Change the system name of the AS400 bean to TestSys.
   c. Change the user ID of the AS400 bean to **TestUser**, as shown in Figure 3.

Figure 3. Changing the name of the user ID to TestUser.



   d. Change the command of the CommandCall bean to **SNDMSG MSG('Testing') TOUSR('TESTUSER')**, as shown in Figure 4.

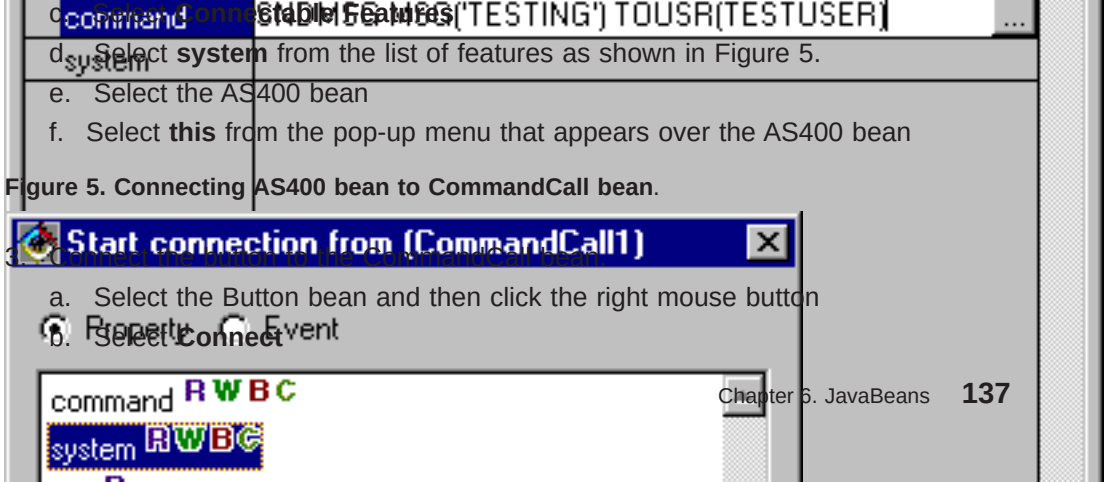
Figure 4. Changing the command of the CommandCall bean.



2. Connect the AS400 bean to the CommandCall bean. The method you use to do this varies between bean builders. For this example, do the following:
   a. Select the CommandCall bean and then click the right mouse button
   b. Select **Connect**
   c. Select **Connectable Features**
   d. Select **system** from the list of features as shown in Figure 5.
   e. Select the AS400 bean
   f. Select **this** from the pop-up menu that appears over the AS400 bean

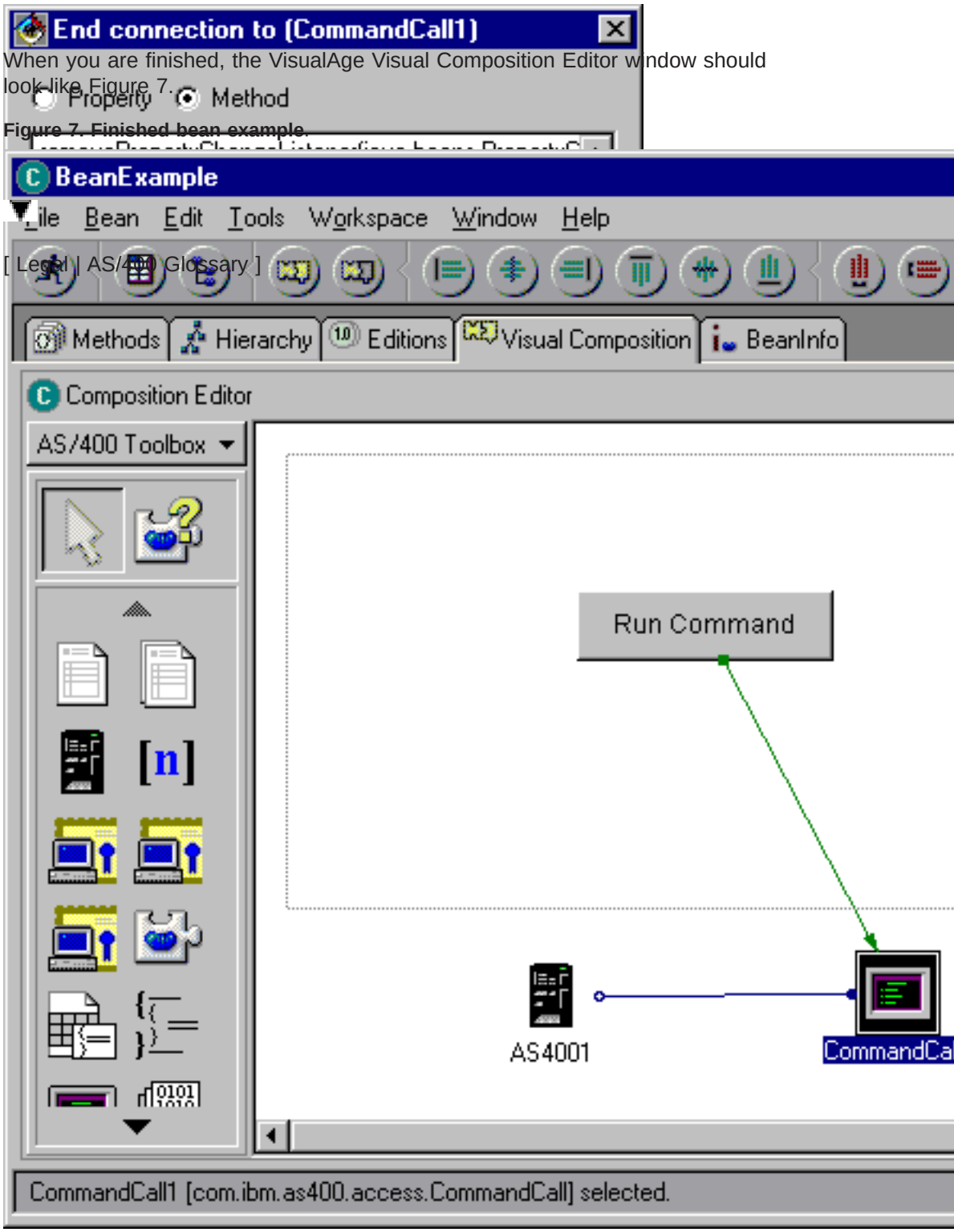
Figure 5. Connecting AS400 bean to CommandCall bean.

3. Connect the button to the CommandCall bean.
   a. Select the Button bean and then click the right mouse button
   b. Select **Connect**



c. Select **actionPerformed**
d. Select the CommandCall bean
e. Select **Connectable Features** from the pop-up menu that appears
f. Select **run()** from the list of methods as shown in Figure 6.

Figure 6. Connecting a method to a button.



When you are finished, the VisualAge Visual Composition Editor window should look like Figure 7.

Figure 7. Finished bean example.

[Legal | AS/400 Glossary]

Chapter 7. Secure Sockets Layer

Secure Sockets Layer (SSL) provides secure connections by encrypting the data exchanged between a client and an AS/400 server session and by performing server authentication. There is an increased cost in performance with SSL because SSL connections perform slower than connections without encryption. SSL can be used only with an SSL capable AS/400 running OS/400, V4R4 or later. You use SSL connections when the sensitivity of the data transferred merits the increased cost in performance, for example, credit card or bank statement information.

Chapter 8. SSL versions

AS/400 Toolbox for Java does not contain the algorithms needed to encrypt and decrypt data. These algorithms are shipped with AS/400 licensed programs 5769-CE1, 5769-CE2, and 5769-CE3. You need to order one of the 5769-CEx product versions of SSL depending on the country in which you live. Contact your IBM representative for more information or to order:

- AS/400 Client Encryption (40-bit), 5769-CE1, is used in France.
- AS/400 Client Encryption (56-bit), 5769-CE2, is used in countries other than the US, Canada or France. Note that the 5769-CE2 client encryption is only 40-bit within AS/400 Toolbox for Java.
- AS/400 Client Encryption (128-bit), 5769-CE3, is used only in the United States and Canada.

Before you begin using SSL with AS/400 Toolbox for Java:

- You must understand your legal responsibilities.
- You must meet some prerequisites.
- You must download the class files containing the SSL algorithms and point to them within the CLASSPATH. The zip file that must be put in your CLASSPATH is below, listed by SSL version:
  - For 5769-CE1 and 5769-CE2, download sslightx.zip
  - For 5769-CE3, download sslightu.zip
- You must install a Cryptographic Access Provider licensed program (5769-AC1, 5769-AC2, or 5769-AC3). The 5769-CE products provide encryption capabilities on the client side. You also need encryption on the AS/400 side, which is provided by the 5769-AC products. Contact your IBM representative for more information.

Chapter 9. Using SSL certificates

Once you point to SSL in your CLASSPATH, the server certificate authenticates the connection with the AS/400. Without a certificate, SSL will not work. You can use two types of certificates: certificates from a trusted authority or certificates that you build.

If you are using certificates issued by a trusted authority, you need to do a few steps. Afterward, the certificate keyring is set up for you, the connection is secure, and SSL is working for you.

AS/400 Toolbox for Java supports certificates issued by the following trusted authorities:

- VeriSign, Inc
- Integrion Financial Network
- IBM World Registry
- Thawte Consulting
- RSA Data Security, Inc.

If you choose not to use a certificate from a trusted authority, you can also build your own certificate. Some reasons for building your own certificate are:

- You do not have to pay for it
- You have more control over it
- You are putting together a local intranet of systems

[Legal | AS/400 Glossary]

SSL legal responsibilities

IBM AS/400 Client Encryption products provide SSL Version 3.0 encryption support using nonexportable 128-bit (designated U.S. and Canada use only) and exportable 40-bit encryption algorithms for international use.

In customer configurations where client encryption products might be downloaded across national boundaries, the customer is responsible to assure that the nonexportable client encryption products are not made available outside the U.S. and Canada. Both the non-exportable and exportable Client Encryption products can be used in combination to allow the appropriate Client Encryption product to be downloaded based on different URLs.

You and your users must comply with other country's import/export laws.

SSL requirements

SSL prerequisites

Before you can use SSL with AS/400 Toolbox for Java, you must follow the steps outlined below:

1. Install the Cryptographic Access Provider licensed program for AS/400 (5769-AC1, 5769-AC2, or 5769-AC3) on your AS/400

2. Install the AS/400 Client Encryption licensed program (5769-CE1, 5769-CE2, or 5769-CE3) on your AS/400
3. You should control authorization of the users to the files. To help you to meet the SSL legal responsibilities, you must change the authority of the directory that contains the SSL files to control user access to the files. In order to change the authority, you must follow the steps below:
   - Enter the command: wrklnk '/QIBM/ProdData/HTTP/Public/jt400/*'
   - Select option 9 in the directory (SSL40, SSL56, or SSL128)
   - Ensure *PUBLIC has *EXCLUDE authority.
   - Give users who need access to the SSL files *RX authority to the directory. You can authorize individual users or groups of users.

   Note: Users with *ALLOBJ special authority cannot be denied access to the SSL files.
4. Get and configure the server certificate. To do this, you ned to do the following:
   a. Install the following products:
      - IBM HTTP Server for AS/400 (5769-DG1) licensed program
      - Base operating system option 34 (Digital Certificate Manager)
   b. Get a server certificate:
      - From a trusted authority
      - Build your own
5. Apply the certificate to the following AS/400 servers that are used by AS/400 Toolbox for Java:
   - QIBM_OS400_QZBS_SVR_CENTRAL
   - QIBM_OS400_QZBS_SVR_DATABASE
   - QIBM_OS400_QZBS_SVR_DTAQ
   - QIBM_OS400_QZBS_SVR_NETPRT
   - QIBM_OS400_QZBS_SVR_RMTCMD
   - QIBM_OS400_QZBS_SVR_SIGNON
   - QIBM_OS400_QZBS_SVR_FILE
   - QIBM_OS400_QRW_SVR_DDM_DRDA

SSL requirements

After you are sure that your AS/400 meets the requirements for SSL, follow the steps outlined below to use SSL on your workstations.

1. Copy the proper SSL encryption algorithms: either sslightu.zip or sslightx.zip
2. Update the CLASSPATH
3. Download your certificate if you have built your own
4. Use the AS/400 Secure Class within your application

[Legal | AS/400 Glossary]

Using a certificate from a trusted authority

AS/400 Toolbox for Java ships a keyring file that supports server certificates from a set of trusted authorities. You can get a certificate from one of the following companies:

- VeriSign, Inc
- Integrion Financial Network

- IBM World Registry
- Thawte Consulting
- RSA Data Security, Inc.

If you get your certificate from one of these trusted authorities, you must do the following steps to use this certificate with SSL:

1. Install the certificate authority certificate on the AS/400
2. Apply the certificate to your host servers
3. Download the version of SSL that you want to use:
   - sslightx.zip (used with the licensed programs 5769-CE1 and 5769-CE2)
   - sslightu.zip (used with the licensed program 5769-CE3)

   Download the files from the following paths:
   - For 5769-CE1 from /QIBM/ProdData/HTTP/Public/jt400/SSL40
   - For 5769-CE2 from /QIBM/ProdData/HTTP/Public/jt400/SSL56
   - For 5769-CE3 from /QIBM/ProdData/HTTP/Public/jt400/SSL128
4. Add either sslightx.zip or sslightu.zip (depending on the country in which you live) to your CLASSPATH statement.

[Legal | AS/400 Glossary]

Building your own certificate

If you choose not to use a certificate from a trusted authority, you can build your own certificate to be used on an AS/400. The certificate is built using the digital certificate manager, and the steps that follow describe how to download and use the certificate with AS/400 Toolbox for Java:

1. Create the certificate authority on the AS/400
2. Apply the certificate to your host servers
3. Download the version of SSL that you want to use:
   - sslightx.zip (used with the licensed programs 5769-CE1 and 5769-CE2)
   - sslightu.zip (used with the licensed program 5769-CE3)

   Download the files from the following paths:
   - For 5769-CE1 from /QIBM/ProdData/HTTP/Public/jt400/SSL40
   - For 5769-CE2 from /QIBM/ProdData/HTTP/Public/jt400/SSL56
   - For 5769-CE3 from /QIBM/ProdData/HTTP/Public/jt400/SSL128
4. From the same directory that you downloaded either sslightx.zip or sslightu.zip, download SSLTools.zip
5. Add SSLTools.zip and either sslightx.zip or sslightu.zip to your CLASSPATH statement
6. Create a directory on your client named com/ibm/as400/access. This directory needs to be a subdirectory of your current directory.
7. Run the following command from a command prompt on your client:

   ```
   java com.ibm.sslight.nlstools.keyrng com.ibm.as400.access.KeyRing connect <systemname>:<port>
   ```

   The server port can be any of the host servers to which you have access. For example, you can use 9476, which is the default port for the secure sign-on server on the AS/400.

Notes: You **must** use com.ibm.as400.access.KeyRing because it is the only location that the AS/400 Toolbox for Java will look for your certificates.

When you are prompted to enter a password, you **must** enter *toolbox*. This is the only password that works.

The SSL tool then connects to the AS/400 and lists the certificates it finds.

8. Type the number of the Certificate Authority (CA) certificate that you want to add to your AS/400. Be sure to add the CA certificate and not the site certificate. A message is issued stating that the certificate is being added to com.ibm.as400.access.KeyRing.class. **Note:** For each certificate that you want to add, you must rerun the command:

   ```
   java com.ibm.sslight.nlstools.keyrng com.ibm.as400.access.KeyRing connect <systemname>:<port>
   ```

   You must download a certificate for each CA certificate you create. Each certificate is added to the KeyRing class. After adding one or more certificates, you must update your CLASSPATH statement. Your CLASSPATH must list the KeyRing.class file that resides in the directory that you created in Step 4 *before* jt400.zip. After this step, you are done using the sslight tool and can delete it.

Alternative method for building a certificate:

As an alternative to the method outlined above, use the following steps:

1. Extract the KeyRing.class file from jt400.zip
2. Run the following command to add the certificate to the KeyRing.class file that comes with the AS/400 Toolbox for Java:

   ```
   java com.ibm.sslight.nlstools.keyrng com.ibm.as400.access.KeyRing
   connect <systemname>:<port>
   ```

3. Put the KeyRing.class back into jt400.zip 

[Legal | AS/400 Glossary]

Chapter 10. Tips for programming

This section features pointers for programming with the AS/400 Toolbox for Java. Click on the links below to view the tips.

1. Find out how to properly shut down your Java program.
2. Use integrated file system path names in your programs. This section covers integrated file system path names, parameters, and special values.
3. Follow these tips on managing connections to an AS/400. See how to use the AS400 class to start and end socket connections.
4. Read about running AS/400 Toolbox for Java classes on the AS/400 Java Virtual Machine. This section covers how to best access AS/400 resources, how to run the classes, and what sign-on factors to consider.
5. When programming with the AS/400 Toolbox for Java access classes, use the exception classes to handle errors.
6. When programming with the graphical user interface (GUI) classes, use the error events classes to handle errors.
7. See how to use the Trace class in your programs.
8. Find out how to optimize your program for better performance.
9. Use the install and update section for information about the AS400ToolboxInstaller class and managing AS/400 Toolbox for Java classes on a client.
10. Read about AS/400 Toolbox for Java and Java national language support.
11. See these resources for AS/400 Toolbox for Java Service and support.

[Legal | AS/400 Glossary]

Shutting down your Java program

To ensure that your program shuts down properly, issue **System.exit(0)** as the last instruction before your Java program ends.

AS/400 Toolbox for Java connects to the AS/400 with user threads. Because of this, a failure to issue **System.exit(0)** may keep your Java program from properly shutting down.

[Legal | AS/400 Glossary]

Integrated file system path names for AS/400 objects

Your Java program must use integrated file system names to refer to AS/400 objects, such as programs, libraries, commands, or spooled files. The integrated file system name is the name of an AS/400 object as it would be accessed in the library file system of the integrated file system on the AS/400.

The path name may consist of the following pieces:

library	**The library in which the object resides. The library is a required portion of an integrated file system path name. The library name must be 10 or fewer characters and be followed by .lib.**
object	The name of the object that the integrated file system path name represents. The object is a **required** portion of an integrated file system path name. The object name must be 10 or fewer characters and be followed by **.type**, where **type** is the type of the object. Types can be found by prompting for the OBJTYPE parameter on commands, such as WRKOBJ.
type	The type of the object. The type of the object must be specified when specifying the **object**. (See **object** above.) The type name must be 6 or fewer characters.
member	The name of the member that this integrated file system path name represents. The member is an **optional** portion of an integrated file system path name. It can be specified only when the **object type** is **FILE**. The member name must be 10 or fewer characters and followed by **.mbr**.

Follow these conditions when determining and specifying the integrated file system name:

- The forward slash (/) is the path separator character.
- The root-level directory, called QSYS.LIB, contains the AS/400 library structure.
- Objects that reside in the AS/400 library QSYS have the following format:

  ```
  /QSYS.LIB/object.type
  ```

- Objects that reside in other libraries have the following format:

  ```
  /QSYS.LIB/library.LIB/object.type
  ```

- The object type extension is the AS/400 abbreviation used for that type of object.

  To see a list of these types, enter an AS/400 command that has object type as a parameter and press F4 (Prompt) for the type. For example, the AS/400 command **Work with Objects** (WRKOBJ) has an object type parameter.

  Below are some commonly used types:

Abbreviation	**Object**
.CMD	command
.DTAQ	data queue
.FILE	file
.FNTRSC	font resource
.FORMDF	form definition
.LIB	library
.MBR	member
.OVL	overlay
.PAGDFN	page definition
.PAGSET	page segment
.PGM	program
.OUTQ	output queue
.SPLF	spooled file

Use these examples to determine how to specify integrated file system path names:

Description	Integrated file system name
Program MY_PROG in library MY_LIB on the AS/400	/QSYS.LIB/MY_LIB.LIB/MY_PROG.PGM
Data queue MY_QUEUE in library MY_LIB on the AS/400	/QSYS.LIB/MY_LIB.LIB/MY_QUEUE.DTAQ
Member JULY in file MONTH in library YEAR1998 on the AS/400	/QSYS.LIB/YEAR1998.LIB/MONTH.FILE/JULY.MBR

Special values that the AS/400 Toolbox for Java recognizes in the integrated file system

In an integrated file system path name, special values that normally begin with an asterisk, such as ***ALL**, are depicted without the asterisk. Instead, use leading and trailing percent signs (**%ALL%**). In the integrated file system, an asterisk is a wildcard character.

The AS/400 Toolbox for Java classes recognize the following special values:

With	Use	(Instead Of)
Library name	%ALL%	(*ALL)
	%ALLUSR%	(*ALLUSR)
	%CURLIB%	(*CURLIB)
	%LIBL%	(*LIBL)
	%USRLIBL%	(*USRLIBL)
Object name	%ALL%	(*ALL)
Member name	%ALL%	(*ALL)
	%FILE%	(*FILE)
	%FIRST%	(*FIRST)
	%LAST%	(*LAST)

See the QSYSObjectPathName class for information about building and parsing integrated file system names.

[Legal | AS/400 Glossary]

Managing connections

Creating, starting, and ending a connection to an AS/400 are discussed below, and some code examples are provided as well.

To connect to an AS/400 system, your Java program must create an AS400 object. The AS400 object contains up to one socket connection for each AS/400 server type. A service corresponds to a job on the AS/400 and is the interface to the data on the AS/400.

Every connection to each server has its own job on the AS/400. A different server supports each of the following:

- JDBC
- Program call and command call
- Integrated file system

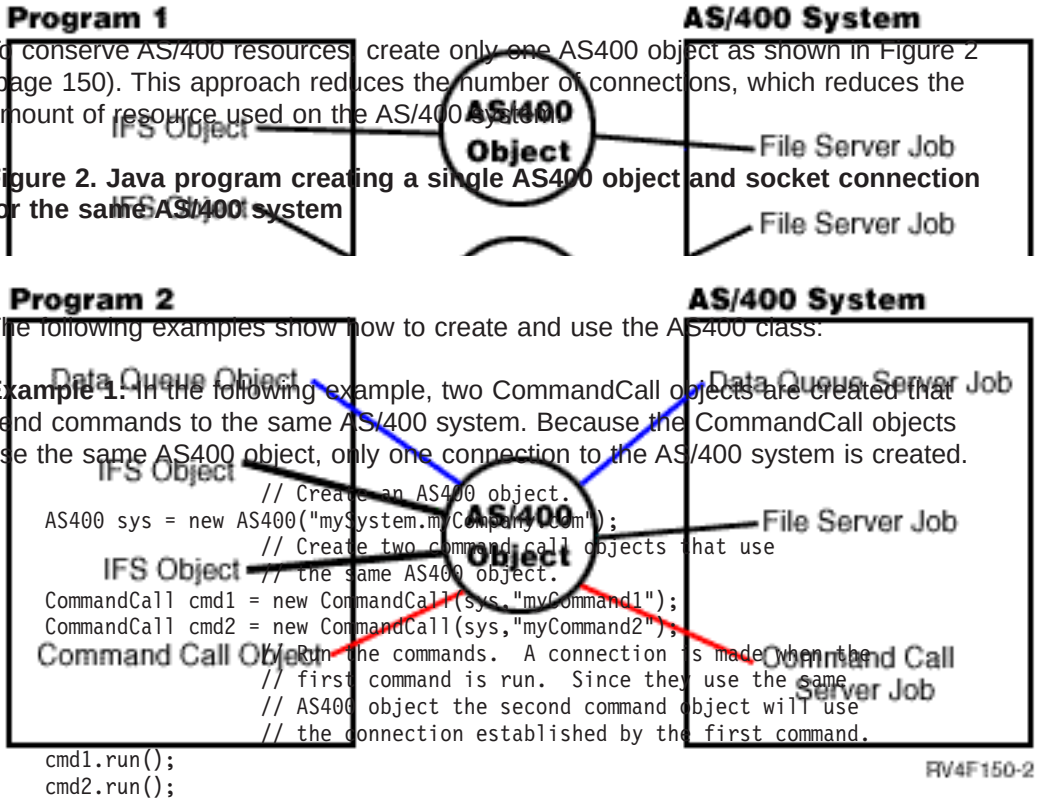
- Network print
- Data queue
- Record-level access

Note: The print classes use one socket connection per AS400 object if the application does not try to do two things that require the network print server at the same time.

A print class creates additional socket connections to the network print server if needed. The extra conversations are disconnected if they are not used for 5 minutes.

The Java program can control the number of connections to the AS/400. To optimize communications performance, a Java program can create multiple AS400 objects for the same AS/400 system as shown in Figure 1 (page 150). This creates multiple socket connections to the AS/400.

Figure 1. Java program creating multiple AS400 objects and socket connections for the same AS/400 system



To conserve AS/400 resources, create only one AS400 object as shown in Figure 2 (page 150). This approach reduces the number of connections, which reduces the amount of resource used on the AS/400 system.

Figure 2. Java program creating a single AS400 object and socket connection for the same AS/400 system

The following examples show how to create and use the AS400 class:

Example 1: In the following example, two CommandCall objects are created that send commands to the same AS/400 system. Because the CommandCall objects use the same AS400 object, only one connection to the AS/400 system is created.

```
                    // Create an AS400 object.
AS400 sys = new AS400("mySystem.myCompany.com");
                    // Create two command call objects that use
                    // the same AS400 object.
CommandCall cmd1 = new CommandCall(sys,"myCommand1");
CommandCall cmd2 = new CommandCall(sys,"myCommand2");
                    // Run the commands.  A connection is made when the
                    // first command is run.  Since they use the same
                    // AS400 object the second command object will use
                    // the connection established by the first command.
cmd1.run();
cmd2.run();
```

Example 2: In the following example, two CommandCall objects are created that send commands to the same AS/400 system. Because the CommandCall objects use different AS400 objects, two connections to the AS/400 system are created.

```
                    // Create two AS400 objects to the same AS/400 system.
AS400 sys1 = new AS400("mySystem.myCompany.com");
AS400 sys2 = new AS400("mySystem.myCompany.com");
                    // Create two command call objects.  They use
                    // different AS400 objects.
CommandCall cmd1 = new CommandCall(sys1,"myCommand1");
CommandCall cmd2 = new CommandCall(sys2,"myCommand2");
                    // Run the commands.  A connection is made when the
                    // first command is run.  Since the second command
```

```
                    // object uses a different AS400 object, a second
                    // connection is made when the second command is run.
cmd1.run();
cmd2.run();
```

Example 3: In the following example, a CommandCall object and an IFSFileInputStream object are created using the same AS400 object. Because the CommandCall object and the IFSFileInput Stream object use different services on the AS/400 system, two connections are created.

```
                    // Create an AS400 object.
AS400 sys = new AS400("mySystem.myCompany.com");
                    // Create a command call object.
CommandCall cmd = new CommandCall(sys,"myCommand1");
                    // Create the file object.  Creating it causes the
                    // AS400 object to connect to the file service.
IFSFileInputStream file = new IFSFileInputStream(sys,"/myfile");
                    // Run the command.  A connection is made to the
                    // command service when the command is run.
cmd.run();
```

Starting and ending connections

The Java program can control when a connection is started and ended. By default, a connection is started when information is needed from the AS/400. You can control exactly when the connection is made by preconnecting to the AS/400 by calling the connectService() method on the AS400 object.

The following examples show Java programs connecting and disconnecting to the AS/400.

Example 1: This example shows how to preconnect to the AS/400:

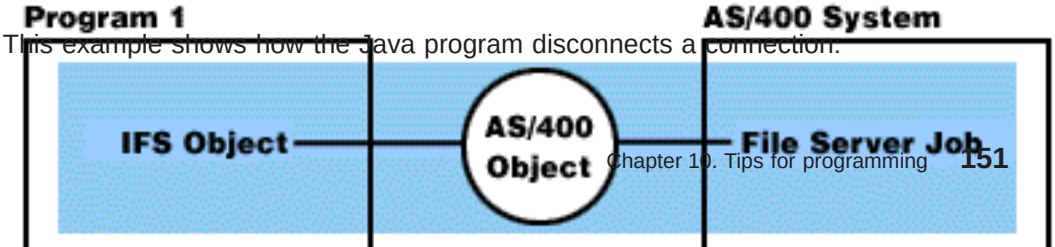
```
                    // Create an AS400 object.
AS400 system1 = new AS400("mySystem.myCompany.com");
                    // Connect to the command service.  Do it now
                    // instead of when data is first sent to the
                    // command service.  This is optional since the
                    // AS400 object will connect when necessary.
system1.connectService(AS400.COMMAND);
```

Example 2: Once a connection is started, the Java program is responsible for disconnecting, which is done either implicitly by the AS400 object, or explicitly by the Java program. A Java program disconnects by calling the disconnectService() method on the AS400 object. To improve performance, the Java program should disconnect only when the program is finished with a service. If the Java program disconnects before it is finished with a service, the AS400 object reconnects—if it is possible to reconnect—when data is needed from the service.

Figure 3 (page 151) shows how disconnecting the connection for the first integrated file system object connection ends only that single instance of the AS400 object connection, not all of the integrated file system object connections.

Figure 3. Single object using its own service for an instance of an AS400 object is disconnected

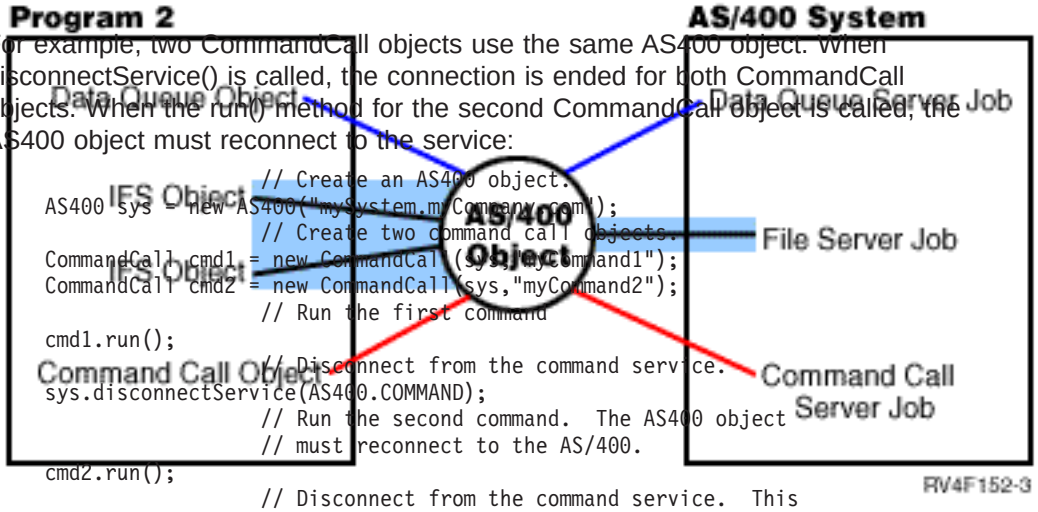
This example shows how the Java program disconnects a connection.



```
                    // Create an AS400 object.
AS400 system1 = new AS400("mySystem.myCompany.com");
                    // ... use command call to send several commands
                    // to the AS/400.  Since connectService() was not
                    // called, the AS400 object automatically
                    // connects when the first command is run.
                    // All done sending commands so disconnect the
                    // connection.
system1.disconnectService(AS400.COMMAND);
```

Example 3: Multiple objects that use the same service and share the same AS400 object share a connection. Disconnecting ends the connection for all objects that are using the same service for each instance of an AS400 object as is shown in Figure 4 (page 152).

Figure 4. All objects using the same service for an instance of an AS400 object are disconnected



For example, two CommandCall objects use the same AS400 object. When disconnectService() is called, the connection is ended for both CommandCall objects. When the run() method for the second CommandCall object is called, the AS400 object must reconnect to the service:

```
                    // Create an AS400 object.
AS400 sys = new AS400("mySystem.myCompany.com");
                    // Create two command call objects.
CommandCall cmd1 = new CommandCall(sys,"myCommand1");
CommandCall cmd2 = new CommandCall(sys,"myCommand2");
                    // Run the first command
cmd1.run();
                    // Disconnect from the command service.
sys.disconnectService(AS400.COMMAND);
                    // Run the second command.  The AS400 object
                    // must reconnect to the AS/400.
cmd2.run();
                    // Disconnect from the command service.  This
                    // is the correct place to disconnect.
sys.disconnectService(AS400.COMMAND);
```

Example 4: Not all AS/400 Toolbox for Java classes automatically reconnect. Some method calls in the integrated file system classes do not reconnect because the file may have changed. While the file was disconnected, some other process may have deleted the file or changed its contents. In the following example, two file objects use the same AS400 object. When disconnectService() is called, the connection is ended for both file objects. The read() for the second IFSFileInputStream object fails because it no longer has a connection to the AS/400.

```
                    // Create an AS400 object.
AS400 sys = new AS400("mySystem.myCompany.com");
                    // Create two file objects.  A connection to the
                    // AS/400 is created when the first object is
                    // created.  The second object uses the connection
                    // created by the first object.
IFSFileInputStream file1 = new IFSFileInputStream(sys,"/file1");
IFSFileInputStream file2 = new IFSFileInputStream(sys,"/file2");
                    // Read from the first file, then close it.
int i1 = file1.read();
file1.close();
                    // Disconnect from the file service.
sys.disconnectService(AS400.FILE);
                    // Attempt to read from the second file.  This
                    // fails because the connection to the file service
```

```
                    // no longer exists.  The program must either
                    // disconnect later or have the second file use a
                    // different AS400 object (which causes it to
                    // have its own connection).
    int i2 = file2.read();
                    // Close the second file.
    file2.close();
                    // Disconnect from the file service.  This
                    // is the correct place to disconnect.
    sys.disconnectService(AS400.FILE);
```

[Legal | AS/400 Glossary]

AS/400 Java Virtual Machine (JVM)

Beginning with Version 4 Release 2 (V4R2), AS/400 has a Java Virtual Machine (JVM) that implements the JDK 1.1 specification.

Because the AS/400 Toolbox for Java classes run on any platform that supports JDK 1.1, the AS/400 Toolbox for Java classes run on the AS/400 JVM.

When you run AS/400 Toolbox for Java classes on the AS/400 JVM, do the following:

1. Choose whether to use the AS/400 Java Virtual Machine for the AS/400 Toolbox for Java classes to access AS/400 resources when running in the AS/400 JVM.
2. Check out Running AS/400 Toolbox for Java classes on the AS/400 Java Virtual Machine.
3. Read about setting system name, user ID, and password in the AS/400 Java Virtual Machine.

[Legal | AS/400 Glossary]

AS/400 Java Virtual Machine versus the AS/400 Toolbox for Java classes

You always have at least two ways to access an AS/400 resource when your Java program is running on the AS/400 Java Virtual Machine (JVM). You can use either of the following interfaces:

- Facilities built into Java
- An AS/400 Toolbox for Java class

When deciding which interface to use, consider the following factors:

1. **Location** - Where a program runs is the most important factor in deciding which interface set to use. Does the program do the following:
   - Run only on the client?
   - Run only on the server?
   - Run on both client and server, but in both cases the resource is an AS/400 resource?
   - Run on one AS/400 JVM and access resources on another AS/400?
   - Run on different kinds of servers?

If the program runs on both client and server (including the AS/400 as a client to a second AS/400) and accesses only AS/400 resources, it may be best to use the AS/400 Toolbox for Java interfaces.

If the program must access data on many types of servers, it may be best to use native Java interfaces.

2. **Consistency / Portability** - The ability to run AS/400 Toolbox for Java classes on AS/400 means that the same interfaces can be used for both client programs and server programs. When you have only one interface to learn for both client programs and server programs, you can be more productive.

   Writing to AS/400 Toolbox for Java interfaces makes your program less **server** portable, however.

   If your program must run to an AS/400 as well as other servers, you may find it better to use the facilities that are built into Java.

3. **Complexity** - The AS/400 Toolbox for Java interface is built especially for easy access to an AS/400 resource. Often, the only alternative to using the AS/400 Toolbox for Java interface is to write a native program that accesses the resource and communicates with that program through Java Native Interface (JNI).

   You must decide whether it is more important to have better Java neutrality and write a native program to access the resource, or to use the AS/400 Toolbox for Java interface, which is less portable.

4. **Function** - The AS/400 Toolbox for Java interface often provides more function than the Java interface. For example, the IFSFileOutputStream class of the AS/400 Toolbox for Java licensed program has more function than the FileOutputStream class of java.io. Using IFSFileOutputStream makes your program specific to the AS/400, however. You lose **server** portability by using the AS/400 Toolbox for Java class.

   You must decide whether portability is more important or whether you want to take advantage of the additional function.

5. **Resource** - When running on the AS/400 JVM, many of the AS/400 Toolbox for Java classes still make requests through the host servers. Therefore, a second job (the server job) carries out the request to access a resource.

   This request may take more resource than a native Java interface that runs under the job of the Java program.

6. **AS/400 as a client** - If your program runs on the AS/400 and accesses data on a second AS/400, your best choice may be to use AS/400 Toolbox for Java classes. These classes provide easy access to the resource on the second AS/400.

   An example of this is Data Queue access. The Data Queue interfaces of the AS/400 Toolbox for Java licensed program provide easy access to the data queue resource.

   Using the AS/400 Toolbox for Java also means your program works on both a client and server to access an AS/400 data queue. It also works when running on one AS/400 to access a data queue on another AS/400.

   The alternative is to write a separate program (in C, for example) that accesses the data queue. The Java program calls this native program when it needs to access the data queue.

   This method is more server-portable; you can have one Java program that handles data queue access and different versions of the native program for each server you support.

Running AS/400 Toolbox for Java classes on the AS/400 Java Virtual Machine

Below are special considerations for running the AS/400 Toolbox for Java classes on the AS/400 Java Virtual Machine (JVM):

Java Database Connectivity (JDBC)

Two IBM-supplied JDBC drivers are available to programs running on the AS/400 JVM:

- The AS/400 Toolbox for Java JDBC driver
- The native JDBC-DB2 for AS/400 driver

The AS/400 Toolbox for Java JDBC driver is best to use when the program is running in a client/server environment.

The native JDBC-DB2 for AS/400 driver is best to use when the program is running on AS/400.

If the same program runs on both the workstation and the AS/400, you should load the correct driver through a system property instead of coding the driver name into your program.

Program call

Two common ways to call a program are as follows:

- The ProgramCall class of the AS/400 Toolbox for Java
- Through a Java Native Interface (JNI) call

The ProgramCall class of the AS/400 Toolbox for Java licensed program has the advantage that it can call any AS/400 program.

You may not be able to call your AS/400 program through JNI. An advantage of JNI is that it is more portable across server platforms.

Command call

Two common ways to call a command are as follows:

- The CommandCall class of the AS/400 Toolbox for Java
- java.lang.runtime.exec()

The CommandCall class generates a list of messages that are available to the Java program once the command completes. This list of messages is not available through java.lang.runtime.exec().

java.lang.runtime.exec() is portable across many platforms, so if your program must access files on different types of servers, java.lang.runtime.exec() is a better solution.

Integrated file system

The two common ways to access a file in the integrated file system of the AS/400 are as follows:

- The IFSFile classes of the AS/400 Toolbox for Java licensed program
- The file classes that are a part of java.io

The AS/400 Toolbox for Java integrated file system classes have the advantage of providing more function than the java.io classes. The AS/400 Toolbox for Java classes also work in applets, and they do not need a method of redirection (such as Client Access for AS/400) to get from a workstation to the server.

The java.io classes are portable across many platforms, which is an advantage. If your program must access files on different types of servers, java.io is a better solution.

If you use java.io classes on a client, you need a method of redirection (such as Client Access/400) to get to the AS/400 file system.

[Legal | AS/400 Glossary]

Setting system name, user ID, and password with an AS400 object in the AS/400 Java Virtual Machine

The AS400 object allows special values for system name, user ID, and password when the Java program is running on the AS/400 Java Virtual Machine (JVM).

When you run a program on the AS/400 JVM, be aware of some special values and other considerations:

- If system name, user ID, or password is not set on the AS400 object, the AS400 object connects to the current AS/400 by using the user ID and password of the job that started the Java program. **A password must be supplied when using record-level access.**
- The special value, **localhost**, can be used as the system name. In this case, the AS400 object connects to the current AS/400.
- The special value, ***current[1] (page 156)**, can be used as the user ID or password on the AS400 object. In this case, the user ID or password (or both) of the job that started the Java program is used.
- The special value, ***current[1] (page 156)**, can be used as the user ID or password on the AS400 object when the Java program is running on the JVM of one AS/400, and the program is accessing resources on another AS/400. In this case, the user ID and password of the job that started the Java program on the source system are used when connecting to the target system.

Summary of User ID and Password Values on an AS400 Object

The following table summarizes how the user ID and password values on an AS400 object are handled by a Java program running on an AS/400 system versus a Java program running on a client:

Values on AS400 Object	Java Program Running on an AS/400	Java Program Running on a Client
System name, user ID, and password not set	Connect to the current AS/400 using the user ID and password of the job that started the program	Prompt for system, user ID, and password
System name = localhost System name = localhost User ID = *current System name = localhost User ID = *current Password ID = *current	Connect to the current AS/400 using the user ID and password of the job that started the program	**Error:** localhost is not valid when the Java program is running on a client
System name = “sys”	Connect to AS/400 “sys” using the user ID and password of the job that started the program. “sys” can be the current AS/400 or another AS/400	Prompt for user ID and password
System name = localhost User ID = “UID” Password ID = “PWD”	Connect to the current AS/400 using the user ID and password specified by the Java program instead of the user ID and password of the job that started the program	**Error:** localhost is not valid when the Java program is not running on a client

[Legal | AS/400 Glossary]

AS/400 optimization

The AS/400 Toolbox for Java licensed program is “Pure Java” so it runs on any platform that has a release 1.1.2 or later Java Virtual Machine. “Pure Java” also means the AS/400 Toolbox for Java classes function in the same way no matter where they run.

Additional classes come with OS/400 that alter the behavior of the AS/400 Toolbox for Java when it is running on the AS/400 Java Virtual Machine (JVM). Sign-on behavior and performance are improved when running on the AS/400 JVM and connecting to the same AS/400. The additional classes are part of OS/400 starting at Version 4 Release 3.

The classes that modify the behavior of the AS/400 Toolbox for Java are in directory **/QIBM/ProdData/Java400/com/ibm/as400/access** on the AS/400. If you want the Pure Java behavior of the AS/400 Toolbox for Java, delete the classes in this directory.

Sign-on considerations

With the additional classes provided by OS/400, Java programs have additional options for providing system name, user ID and password information to the AS/400 Toolbox for Java.

When accessing an AS/400 resource, the AS/400 Toolbox for Java classes must have a system name, user ID and password.

When running on a client, the system name, user ID and password are provided by the Java program, or the AS/400 Toolbox for Java retrieves these values from the user through a sign-on dialog.

Java national language support

Java supports a set of national languages, but it is a subset of the languages that the AS/400 system supports.

When a mismatch between languages occurs, for example, if you are running on a local workstation that is using a language that is not supported by Java, the AS/400 Toolbox for Java licensed program **may issue some error messages in English**.

[Legal | AS/400 Glossary]

Service and support for the AS/400 Toolbox for Java

Use the following resources for service and support:

- Use the online information provided at:
  http://as400service.rochester.ibm.com/as400/service.html 
- Use the online information provided at:
  http://www.as400.ibm.com/  under “Service/Support.”
- Use the trouble-shooting information found on the AS/400 Toolbox for Java page located off of the AS/400 home page at:
  http://www.as400.ibm.com/  .
- Use IBM Support Services for 5763-JC1, AS/400 Toolbox for Java, provided at:
  http://www.as400.ibm.com/toolbox  .

Support Services for the AS/400 Toolbox for Java, 5763-JC1, are provided under the usual terms and conditions for AS/400 software products. Support services include program services, voice support, and consulting services. Point your web browser to http://www.as400.ibm.com/  or contact your local IBM representative for more information.

Resolving AS/400 Toolbox for Java program defects is supported under program services and voice support, while resolving application programming and debugging issues is supported under consulting services.

AS/400 Toolbox for Java application program interface (API) calls are supported under consulting services unless any of the following are true:

- It is clearly a Java API defect, as demonstrated by re-creation in a relatively simple program.
- It is a question asking for documentation clarification.
- It is a question about the location of samples or documentation.

All programming assistance is supported under consulting services including those program samples provided in the AS/400 Toolbox for Java licensed program. Additional samples may be made available on the Internet at http://www.as400.ibm.com/  on an unsupported basis.

Problem solving information is provided with the AS/400 Toolbox for Java Licensed Program Product. If you believe there is a potential defect in the AS/400 Toolbox for Java API, a simple program that demonstrates the error will be required.

[Legal | AS/400 Glossary]

Chapter 11. Code Examples

The following table is the entry point for all of the **examples** used throughout the AS/400 Toolbox for Java information. The examples in the Tutorial section are not included below.

"Code examples from the access classes"	"Code examples using the GUI classes" on page 163
"Code examples from the utility classes" on page 165	JavaBeans
Graphical Toolbox	PCML
"Tips for Programming" on page 165	

[Legal | AS/400 Glossary]

Code examples from the access classes

This section lists the code examples that are provided throughout the documentation of the access classes.

Command call

- Example: Using the CommandCall class to run a command on AS/400 (page 14)
- Example: Using CommandCall to prompt for the name of the AS/400, command to run, and print the result

Data area

- Example: Creating and writing to a decimal data area (page 16)

Data conversion and description

- Example: How to use RecordFormat and Record with the data queue classes

Data queues

- Example: Create a DataQueue object, read data, and disconnect (page 26)

Digital certificate

- Example: List the digital certificates that belong to a user

Exceptions

- Example: Catching a thrown exception, retrieving the return code, and displaying the exception text (page 29)

Integrated file system

- Example: Using the integrated file system classes to copy a file from one directory to another on the AS/400
- Example: How to use IFSJavaFile instead of java.io.File (page 35)

- Example: Using the integrated file system classes to list the contents of a directory on the AS/400

JDBC

- Example: Using the JDBC driver to create and populate a table
- Example: Using the JDBC driver to query a table and output its contents

Jobs

- Example: Retrieving and changing job information using the cache (page 49)
- Example: Listing all active jobs (page 50)
- Example: Printing all of the messages in the job log for a specific user (page 51)
- Example: Listing the job identification information for a specific user
- Example: Getting a list of jobs on the AS/400 and output the job's status followed by a job identifier
- Example: Displaying messages in the job log for a job that belongs to the current user

Message queue

- Example: How to use the message queue object
- Example: Printing the contents of the message queue
- Example: How to retrieve and print a message
- Example: Listing the contents of the message queue

Network print

- Example: Creating a spooled file on an AS/400 from an input stream
- Example: Generating an SCS data stream using the SCS3812Writer class
- Example: Reading an existing AS/400 spooled file
- Example: Asynchronously listing all spooled files on a system and how to use the PrintObjectListListener interface to get feedback as the list is being built
- Example: Asynchronously listing all spooled files on a system *without* using the PrintObjectListListener interface
- Example: Synchronously listing all spooled files on a system

Permission

- Example: Set the authority of an AS/400 object (page 65)

Program call

- Example: Using the ProgramCall class (page 68)
- Example: Passing paramater data with a ProgramParamater object (page 69)

QSYSObjectPathName

- Example: Building an integrated file system name (page 70)
- Example: Using toPath() to build an AS/400 object name (page 71)
- Example: How to use the QSYSObjectPathName class to parse the integrated file system path name (page 71)

Record-level access

- Example: Accessing an AS/400 file sequentially
- Example: Using the record-level access classes to read an AS/400 file
- Example: Using the record-level access classes to read records by key from an AS/400 file

System Status

- Example: Determine the number of users temporarily signed off
- Example: Set the maximum faults size for a system pool

System Values

- “Examples of using the SystemValue and SystemValueList classes” on page 88

Trace

- Example: Using the setTraceOn() method (page 89)
- Example: Preferred way of using trace (page 89)

User Groups

- Example: Retrieving a list of users (page 89)
- Example: Listing all the users of a group

User Space

- Example: How to create a user space (page 90)

[Legal | AS/400 Glossary]

Code examples using the GUI classes

This section lists the code examples that are provided throughout the documentation of the graphical user interface (GUI) classes.

AS/400 panes

- Example: Creating an AS400DetailsPane to present the list of users defined on the systemAS400DetailsPane (page 96)
- Example: Loading the contents of a details pane before adding it to a frame (page 96)
- Example: Using an AS400 ListPane to present a list of users
- Example: Using an AS400DetailsPane to display messages returned from a command call
- Example: Using an AS400TreePane to display a tree view of a directory
- Example: Using an AS400ExplorerPane to present various network print resources

Command call

- Example: Creating a CommandCallButton (page 98)

- Example: Adding the ActionCompletedListener to process all AS/400 messages that a command generates (page 98)
- Example: Using the CommandCallMenuItem

Data queues

- Example: Creating a DataQueueDocument (page 99)
- Example: Using a DataQueueDocument

Error events

- Example: Handling error events (page 100)
- Example: Defining an error listener (page 100)
- Example: Using a customized handler to handle error events (page 100)

Jobs

- Example: Creating a VJobList and presenting the list in an AS400ExplorerPane (page 110)
- Example: Presenting a list of jobs in an explorer pane

JDBC

- Example: Using the JDBC driver to create and populate a table
- Example: Using the JDBC driver to query a table and output its contents

Program call

- Example: Creating a ProgramCallMenuItem (page 122)
- Example: Processing all program generated AS/400 messages (page 123)
- Example: Adding two parameters (page 123)
- Example: Using a ProgramCallButton in an application

Record-level access

- Example: Creating a RecordListTablePane object to display all records less than or equal to a key (page 125)

SpooledFileViewer

- Example: Creating a Spooled File Viewer to view a spooled file previously created on an AS/400 (page 62)

System Values

- “Example” on page 128

Users and groups

- Example: Creating a VUserList with the AS400DetailsPane (page 130)
- Example: Using an AS400ListPane to create a list of users for selection

[Legal | AS/400 Glossary]

Code examples from the utility classes

This section lists the code examples that are provided throughout the documentation of the utility classes.

AS/400 Toolbox Installer

- Example: Using the AS400ToolboxInstaller class (page 132)
- Example: Installing the AS/400Toolbox with the AS400ToolboxInstaller

JarMaker

- Example: Extracting AS400.class and all its dependent classes from jt400.jar (page 133)
- Example: Splitting jt400.jar into a set of 300K files (page 133)
- Example: Removing unused files from a JAR file (page 133)
- Example: Creating a 400K byte smaller JAR file by omitting the conversion tables with the -ccsid parameter (page 134) 

[Legal | AS/400 Glossary]

Code examples from the JavaBeans topics

This section lists the code examples that are provided throughout the documentation of the JavaBean topics.

- Example: Using listeners to print a comment when you connect and disconnect to the system and run commands
- Example: Using applets and IBM VisualAge for Java to create buttons that run commands

[Legal | AS/400 Glossary]

Tips for Programming

This section lists the code examples that are provided throughout the documentation of the managing connections topic.

Managing connections

- Example: Making a connection to the AS/400 with a CommandCall object (page 150)
- Example: Making two connections to the AS/400 with a CommandCall object (page 150)
- Example: Creating CommandCall and IFSFileInputStream objects with an AS400 object (page 151)
- Example: How a Java program preconnects to the AS/400 (page 151)
- Example: How a Java program disconnects from the AS/400 (page 151)

- Example: How a Java program disconnects and reconnects to the AS/400 with disconnectService() and run() (page 152)
- Example: How a Java program disconnects from the AS/400 and fails to reconnect (page 152)

Exceptions

- Example: Using exceptions (page 29)

Error events

- Example: Handling error events (page 100)
- Example: Defining an error listener (page 100)
- Example: Using a customized handler to handle error events (page 100)

Trace

- Example: Using trace (page 89)
- Example: Using setTraceOn() (page 89)

Optimization

- Example: How to create two AS400 objects (page)
- Example: How an AS400 object is used to represent a second AS/400 system (page)

Install and update

- Example: Using the AS400Toolbox Installer class (page 132)

[Legal | AS/400 Glossary]

Printed in U.S.A.