Lab: IBM Toolbox for Java™

Student Exercises

Kim Button
John Eberhard

IBM Rochester
Session 41LA (403627)

Lab: IBM Toolbox for Java

Introduction

This lab provides an overview of the IBM Toolbox for Java™. You will build Java applications using the IBM Toolbox for Java.

Overview

The Java™ Language

Java is a simple, object-oriented, network-aware, portable, interpreted, robust, secure, architecture-neutral, high-performance, multithreaded, dynamic language. Java is object-oriented from the ground up. Java organizes code into a collection of classes. Each class is made up of methods and data. Classes can be grouped together and placed in packages.

- **Packages** are similar to ILE RPG service programs. They enable you to divide your pieces into easily reused units. Packages are Java language constructs. They may contain multiple classes.

- **Classes** are similar to ILE RPG modules. They enable you to divide your source code into functions ("methods" in Java, "procedures" and "subroutines" in RPG) and the variables those functions need. Classes are typically self-contained groupings. They normally contain multiple fields (variables) and methods.

- **Methods** are similar to ILE RPG procedures and subroutines. They contain all the actual code your program will run. In Java, unlike RPG, executable code can only exist in methods, and methods can only exist inside classes.

IBM Toolbox for Java

Introduction

Java programs can access i5/OS data and resources from any client platform (including i5/OS) using the IBM Toolbox for Java. The IBM Toolbox for Java contains the infrastructure that allows access to the following i5/OS data and resources:

- JDBC and record-level access to DB2/400 data
- print resources
- integrated file system
- data queues
- program calls
- command calls
- user lists
- job lists
- job logs
- message queues
- *and many others!*

The IBM Toolbox for Java provides Java Beans that can be used for visual application development. Developers can use the classes directly from Java code or in a visual application builder.

The V6R1 version of the Toolbox which ships with i5/OS Version 6 release 1 is the current version of the Toolbox. Alternatively, JTOpen 6.1 is the latest open source version of the Toolbox and it can be downloaded from the JTOpen website. The Toolbox runs on any JVM that supports JDK 1.1.8 or later. The GUI components of the Toolbox require Swing 1.1 (the new Swing names).

See http://www.ibm.com/systems/i/software/toolbox/ for more information.

The Toolbox source for almost all components is available via the JTOpen project at http://jt400.sourceforge.net/. This is part of IBM's open source development community. Developers can use the source as a debug tool, submit new function, modify it for their own use, and submit problem reports and bug fixes.

The Official JTOpen and Toolbox forum:

- Go to http://www-03.ibm.com/servers/eserver/support/iseries/index.html
- Select "Problem Solving", then "Forums"
- then "JTOpen/Toolbox for Java Forum"

An Example

The following is a simple program that uses the Toolbox to call an i5/OS command. Several lines in the program are numbered. These numbers correspond to explanations that follow the example.

```
import com.ibm.as400.access.*;                          // 1

public class CmdCall
{
   public static void main(String[] args)
   {
      try                                               // 2
      {
         AS400 system  = new AS400();                   // 3
         AS400 system2 = new AS400("myAS400", "myID", "myPW");

         CommandCall cc = new CommandCall(system);      // 4

         cc.run("CRTLIB MYLIB");                        // 5

         AS400Message[] ml = cc.getMessageList();       // 6

         for (int i=0; i<ml.length; i++)
         {
            System.out.println(ml[i].getText());        // 7
         }
      }
      catch (Exception e)
      {
         e.printStackTrace();
      }
   }
}
```

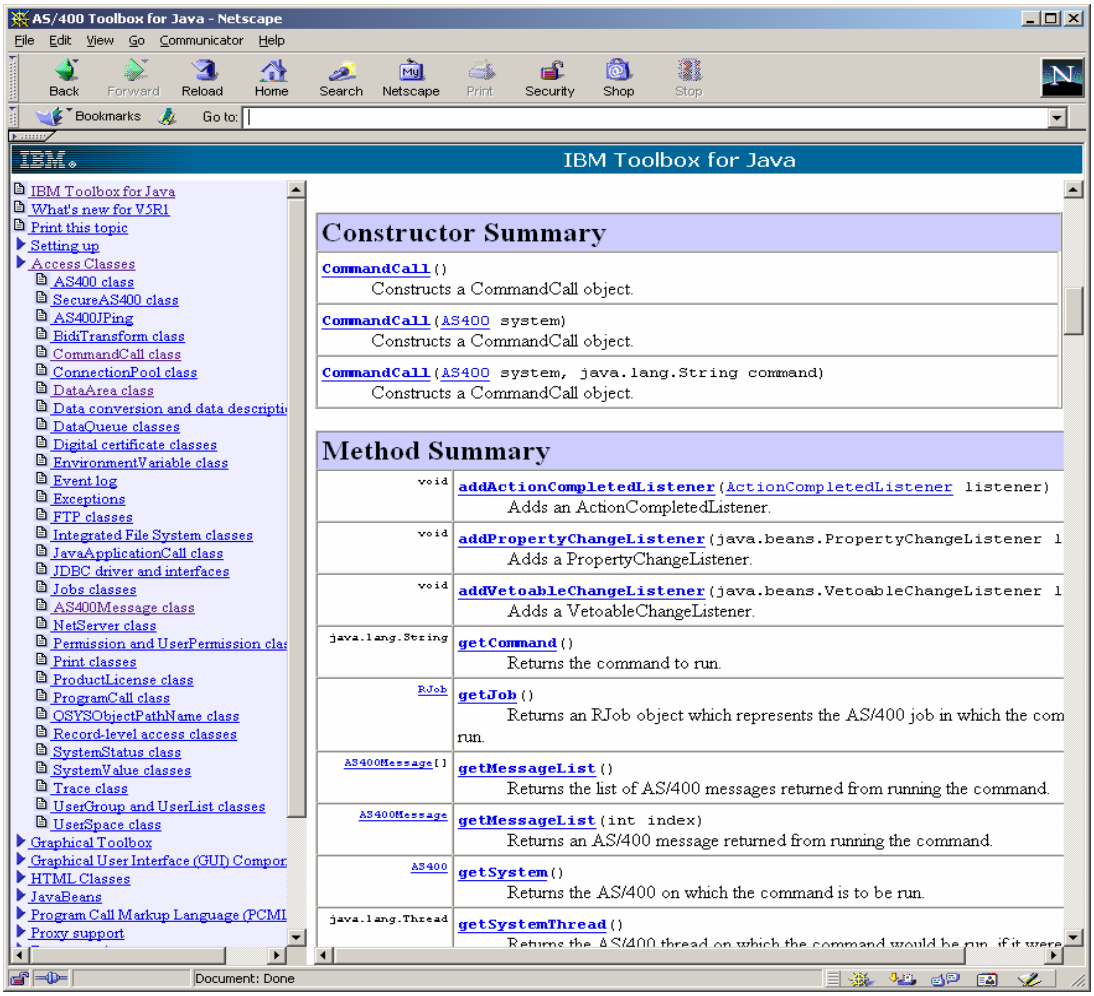
Comments

1. The access classes of the Toolbox are in the com.ibm.as400.access package. Import this package to use the Toolbox classes.
2. Like other Java objects, the Toolbox throws *exceptions* when something goes wrong. These must be caught by programs that use the Toolbox. Exceptions are caught in Java using a try/catch block. Any exceptions caused by the code in the try block will be handled in the catch block.
3. The Toolbox **AS400** object is the object used to identify the target (server). If you construct the AS400 object with no parameters, the Toolbox will prompt for system name, userid and password. Notice when we create the second AS400 object, we supply these values so the Toolbox will not prompt for them.
4. The Toolbox **CommandCall** object is used to send commands to the server. When we create the command call object, we pass it an AS400 object so it knows which server is the target of the command.
5. Use the run() method on the command call object to run a command.
6. The result of running a command is a list of i5/OS messages. The Toolbox represents these messages as **AS400Message** objects. When the command is complete, we get the resulting messages from the command call object.

7. Print the message text. Also available is the message ID, message severity and other information. In this program we are only printing the message text.

Javadoc

Like other Java classes, detailed API information can be displayed as Javadoc. Javadoc includes an overview of the class, public constants, and a detailed explanation of each public constructor and public method. As a component is developed, the developers include information in the Java source files. A tool that comes with the JDK pulls out this information and formats it as HTML. An example follows. The Toolbox Javadoc is installed on the lab PCs. To see it, start Netscape/Internet Explorer then open the file **C:\toolbox\doc\Toolbox.html**. Then select the Javadocs link on the left. Leave the Toolbox Javadoc open for the remainder of this lab. It will be a valuable resource to you.



The Lab

Overview

This lab consists of exercises that demonstrate various components of the IBM Toolbox for Java. In the exercises, you will create and run several Java applications. In most cases, you will start with an existing source file. For each exercise, your task is to complete the application by adding the IBM Toolbox for Java code. You will then compile and run each application.

You need the following hardware to complete this lab:

Server – IBM System i™ - i5/OS V5R3 or later

Client - PC - Windows 9x/NT/2000/XP; 32MB memory; 200MB free hard drive space

Actually, any computer with a Java Virtual Machine can be used (except for bonus exercise 1). However, the lab instructions explain the instructions using Windows terminology and tools.

You need the following software on the client to complete this lab:

- Java Developers Kit 1.1.8 or later
- Swing 1.1 (included in Java Developers Kit 1.2)
- Modification 5 of the IBM Toolbox for Java (or alternately JTOpen version 3.3) or later
  Note: It is suggested to get the latest release of JTOpen which is version 6.1.

Lab Flow

Most of the Java source is provided. What is missing is the code that demonstrates a particular feature of the Toolbox. You will fill in this code then compile and run the program. Hints are given in the lab instructions to help you write the code. If you get stuck, the lab solutions are at the end of this handout.

Remember

Java is case-sensitive. Enter all lines exactly in the case they appear in the prototypes that follow.

Lab Setup

During this lab you will need a server running i5/OS, a userid and a password. This information will be given to you by the instructor. Please fill in this information below so that you have it for reference during the lab.

My i5/OS **system** is: _____

My i5/OS **userid** is: _____

My i5/OS **password** is: _____

The PC **source code directory** is C:\toolbox\

You are now ready to begin the exercises.

Exercise 1: Command Call

Introduction

In this exercise, you will use the IBM Toolbox for Java to call an i5/OS command from a Java application.

The CommandCall object (part of the IBM Toolbox for Java) enables a Java program to call any non-interactive i5/OS command. The list of i5/OS messages that result from the command are available to the Java program when the i5/OS command completes.

In this exercise you will use **AS400**, **CommandCall**, and **AS400Message** objects to complete a Java application. Much of the application code has been provided for you. You will need to write Java code to connect to the server, execute a command, and display the results.

Goals of this exercise

At the end of this exercise, you should be able to:

1. Create an AS400 object.
2. Create a CommandCall object.
3. Run the command.
4. Retrieve AS400Message objects.

Part 1: Create an AS400 object

Note that all sections that need code written by you start with the comments:

```
// ------------------------------------------------------------
//             Lab Exercise #1 Part #x - Insert code here.
// ------------------------------------------------------------
```

and end with the comments:

```
// ------------------------------------------------------------
//                        End of code.
// ------------------------------------------------------------
```

Type the code for each exercise after the beginning comments of Exercise #1 Part #x and before the ending comments.

Setup

1. Start a DOS prompt (on Windows 2000 or XP) using the Start menu: select "Programs", "Accessories", "Command Prompt".
2. Change the current directory to the directory which contains the lab source code (C:\toolbox). To change the directory, enter:

   cd C:\toolbox

3. Run the batch file **`ToolboxSetup`**. This will set up the classpath environment variables to point to the Toolbox jar files. You must run this batch file any time you open a new MS-DOS prompt.
4. Edit the CommandCallExample.java source file. You can use any editor you like. For this lab, we will use Windows Notepad. In the DOS prompt, type:

   notepad CommandCallExample.java

5. Locate the section for Lab Exercise #1 Part #1.

Procedure

1. Create an AS400 object named *system* and specify your assigned i5/OS system name. (Hint: Java string literals are enclosed in double quotes). Some prototypes/examples that you may need are listed below. The complete set of prototypes is provided in the documentation (**Javadoc**) that is shipped in soft copy form with the IBM Toolbox for Java. This AS400 object represents the connection to the system running i5/OS. **Remember, Java is case-sensitive!**

Remember that at any time during this lab, if you get stuck, you can either ask a lab attendant for help or consult Appendix B for the solutions.

Prototypes

class AS400

- public AS400()
- public AS400(String systemName)
- public void setSystemName(String systemName)

Example code: AS400 system = new AS400();

Part 2: Create a CommandCall object

Setup

1. Continue editing the CommandCallExample.java source file.
2. Locate the section for Lab Exercise #1 Part #2.

Procedure

1. Create a CommandCall object named *command* and specify the AS400 object that was created in Part 1. Again, some prototypes that you may need are listed below. This CommandCall object represents a command call, although at this point, no command has been called.

Prototypes
class CommandCall

- public CommandCall()
- public CommandCall(AS400 system)
- public void setSystem(AS400 system)

Example Code: CommandCall command = new CommandCall();

Part 3: Run the command

Setup

1. Continue editing the CommandCallExample.java source file.
2. Locate the section for Lab Exercise #1 Part #3.

Procedure

1. The lab already has code that creates a String named *commandString*, which is made up of the command line arguments passed by the user. Add the code to run this command.
2. Notice that the run() method returns a boolean, which indicates whether or not the command was successful. Add code to check this and print the appropriate message, either
   System.out.println("The command was successful");
   or
   System.out.println("The command failed.");

Prototypes
class CommandCall

- public boolean run(String commandString)

Example code: boolean successful = command.run("CRTLIB TESTLIB");

Part 4: Retrieve AS400Message objects

Setup

1. Continue editing the CommandCallExample.java source file.
2. Locate the section for Lab Exercise #1 Part #4.

Procedure

1. Retrieve any messages that were generated by running the command. The messages are stored as an array of AS400Message objects. Each AS400Message object in the array represents a message that was generated by the command.
2. Loop through the array of messages, and print each message's ID and text to System.out.

Prototypes

class CommandCall

- public AS400Message[] getMessageList()

    Example code: AS400Message[] message = command.getMessageList();

class AS400Message

- public String getID()
- public String getText()

    Example code: String iD = message.getID();

Run the application

Now it is time to run the CommandCallExample application.

1. Make sure to save the modified CommandCallExample.java file.
2. Compile the application from the DOS prompt. Hint: Java is case-sensitive even on Windows so the case of the file name must be correct. To compile, type:

   javac CommandCallExample.java

3. If you have errors, re-edit then re-compile the source file.
4. Run the application once you successfully compiled it. The application has one command line parameter, the command to run on the server. To run the program, type:

   java CommandCallExample CRTLIB FRED

5. The application will prompt you for a user ID and password. This happens automatically when your application accesses the server using the IBM Toolbox for Java. Enter your assigned user ID and password.

6. Verify that the application output (which appears in the DOS prompt) looks similar to this:

```
The command failed.
CPF2111 Library FRED already exists.
```

****NOTE**: If this is the first time Library FRED is created, the output will say the command completed successfully, otherwise the output will be as above.

Exercise 2: JDBC

Introduction

In this exercise, you will use the IBM Toolbox for Java JDBC driver to query a database on the server. JDBC stands for Java Database Connectivity. JDBC is the Java standard for SQL database access. The IBM Toolbox for Java provides a JDBC 3.0 implementation. With JDBC you can access databases using standard Java interfaces.

In this exercise you will use Toolbox JDBC objects to complete a Java application. Much of the application code has been provided for you. You will need to write Java code to register the Toolbox JDBC driver as the driver to use, connect to the server, query the database, and display the results.

Goals of this exercise

At the end of this exercise, you should be able to:

1. Register the Toolbox JDBC driver as the driver to use to get data.
2. Create an SQL Connection.
3. Create an SQL Statement.
4. Query the database to get database metadata.
5. Query the database to get a result set.

Part 1: Set System, Library and Database

Note all sections that need code written by you start with the comments:

```
// -------------------------------------------------------------
//             Lab Exercise #2 Part #x - Insert code here.
// -------------------------------------------------------------
```

and end with the comments:

```
// -------------------------------------------------------------
//                         End of code.
// -------------------------------------------------------------
```

Setup

1. Edit the JDBCQuery.java source file. You can use any editor you like. For this lab, we will use Windows Notepad. In the DOS prompt, type:

   notepad JDBCQuery.java

2. Locate the section for Lab Exercise #2 Part #1.

Procedure

Update the "xxxx"s to use the real values.

1. 'system' is the name of the server we are running to.
2. 'collectionName' is the name of the i5/OS library. Use **QIWS**. In SQL terminology, collections map to i5/OS libraries.
3. 'tableName' is the name of the database file. Use **QCUSTCDT**. In SQL terminology, tables map to i5/OS database files.

Remember that at any time during this lab, if you get stuck, you can either ask a lab attendant for help or consult Appendix B for the solutions.

Part 2: Register the Toolbox JDBC Driver

Setup

1. Continue editing the JDBCQuery.java source file.
2. Locate the section for Lab Exercise #2 Part #2.

Procedure

Register the Toolbox JDBC Driver. By registering the driver, the JDBC Driver Manager (part of every Java Virtual Machine) will route JDBC calls to the Toolbox driver. The Toolbox driver, in turn, will send them to the i5/OS host servers. Some prototypes that you may need are listed below.

Hints

1. The Toolbox driver is named *com.ibm.as400.access.AS400JDBCDriver*.
2. You must create a Toolbox driver object to use as a parameter on the registerDriver method.
3. The registerDriver method is static. Static means you don't need to "new up" an object to call the method, you just class qualify the method name. For example, suppose a driver is *com.greatStuff.JDBCDriver*. The line of code is:

```
DriverManager.registerDriver(new com.greatStuff.JDBCDriver());
```

Prototypes

class DriverManager

- public static void registerDriver(Driver driver)

Part 3: Establish a Connection

Setup

1. Continue editing the JDBCQuery.java source file.
2. Locate the section for Lab Exercise #2 Part #3.

Procedure

Connect to the database. Look up a few lines of code and you will see "Connection connection = null;". This code declares the variable 'connection' but nothing has happened yet. Now it is time to establish the connection. Write the line of code to connect the Toolbox JDBC driver to the DB2 on i5/OS.

Hints

1. You will again use the DriverManager class, this time to get a connection. Like the previous exercise, you will use a static method on DriverManager.
2. The Toolbox syntax is "jdbc:as400://<system>".
3. You already identified the system name in step 1. getConnection() requires a string so you will have to concatenate the Toolbox constant with the system name. Using the greatStuff driver in the hint section of the previous part, the line of code is:

```
connection = DriverManager.getConnection("jdbc:xxxx://" + system);
```

Prototypes

class DriverManager

- public static Connection getConnection(String identifier)

Part 4: Result Set

Setup

1. Continue editing JDBCQuery.java
2. Locate the section for Lab Exercise #2 Part #4.

Procedure

1. The next step is to create a **Statement** object, then use the statement to run a query and get back a **ResultSet**. The result set contains the result of the query. Creating the statement is already done for you. Look up one line in the code and you will see a line that creates a statement from the connection. To do actual work, you tell the statement to execute a query.
2. Add a line of code to get a result set called 'rs'containing all rows and columns of our database. You create a result set by executing a query with the connection object.

Hints

1. The SQL statement we will use is "SELECT * FROM xxxx". The "*" means all rows and columns. After the "from" is the database name, represented here by x's.
2. In the first part of this exercise you defined the collection and table names in String variables. You will need to concatenate Strings with constants to build the entire SQL statement.
3. Different database vendors use different separator characters to separate library from file name. You could hardcode the DB2 UDB for i5/OS separator character, but that would not be very portable. The better thing to do is ask the database what its separator character is.

Prototypes

class Statement

- public ResultSet executeQuery(String query)

  Example Code: ResultSet rs = statement.executeQuery("SELECT * FROM xxx");

class DatabaseMetaData

- public String getCatalogSeparator()

  Example Code: String separator = dmd.getCatalogSeparator();

Part 5: Printing Database Information

Setup

1. Continue editing the JDBCQuery.java source file.
2. Locate the section for Lab Exercise #2 Part #5 (there are two markers).

Procedure

No code is needed for this part, just a couple of things to look at.

1. Find the first marker. The code here uses **ResultSetMetaData** to find out the number of columns in the result set and the column names. The result set not only contains the data but also this metadata.
2. Find the second marker. The result set points at a row of data at a time. You use **ResultSet.next()** to get the next row of data. To get a field out of the row, you use **ResultSet.getString(int columnIndex).**

Run the application

Now it is time to run the JDBCQuery application.

1. Make sure to save the modified JDBCQuery.java file.
2. Compile the application from the DOS prompt:

   javac JDBCQuery.java

3. Run the application:

   java JDBCQuery

4. As in the previous exercise, the Toolbox will prompt you for a user ID and password. This happens automatically when your application accesses the server using the IBM Toolbox for Java.

5. The application will execute the query, and then print the contents of the database.

Exercise 3: SQL Result Set Table Pane

Introduction

In this exercise, you will use the IBM Toolbox for Java to present the results of an SQL database query in a graphical user interface.

The SQLResultSetTablePane object (part of the IBM Toolbox for Java) enables a Java program to present the results of a database query in a JTable. A JTable is a Java Swing component and can be embedded inside any graphical user interface.

In this exercise you will use **SQLConnection**, **SQLResultSetTablePane**, and **ErrorDialogAdapter** objects to complete a Java application. Your application will present the results of an SQL database query. The SQL query is entered by the user. The Swing and AWT parts of the application have been provided for you. You will need to write Java code to connect to the DB2 on i5/OS, create the SQLResultSetTablePane object, and run the queries.

Goals of this exercise

At the end of this exercise, you should be able to:

1. Create a SQLConnection object.
2. Create a SQLResultSetTablePane object.
3. Run a query and load the results.
4. Setup an error handler.

Part 1: Create an SQLConnection object

Setup

1. Edit the SQLResultSetTablePaneExample.java source file. In the DOS prompt, type:

   notepad SQLResultSetTablePaneExample.java

2. Locate the section for Lab Exercise #3 Part #1.

Procedure

1. Create a SQLConnection object, *connection*, that uses the JDBC URL "jdbc:as400://*systemName*", where *systemName* is your assigned i5/OS system name. This SQLConnection object represents the JDBC connection to the DB2 on i5/OS.

Prototypes

class SQLConnection

- public SQLConnection (String URL)

  Example Code: SQLConnection con = new SQLConnection("jdbc:as400://system");

Part 2: Create an SQLResultSetTablePane object

Setup
1. Continue editing the SQLResultSetTablePaneExample.java source file.
2. Locate the section for Lab Exercise #3 Part #2

Procedure
1. Create a SQLResultSetTablePane object called *tablePane*. This represents the graphical user interface component which presents the contents of the query to the user.
2. Use setConnection() to set *tablePane*'s connection to the SQLConnection object that you created in Part 1. This tells *tablePane* which JDBC connection to use for executing the query and gathering results.

Prototypes
class SQLResultSetTablePane
- public SQLResultSetTablePane ()
- public void setConnection (SQLConnection connection)

```
Example Code:  SQLResultSetTablePane tp = new SQLResultSetTablePane();
               tp.setConnection(connection);
```

Part 3: Run a query and load the results

Setup
1. Continue editing the SQLResultSetTablePaneExample.java source file.
2. Locate the section for Lab Exercise #3 Part #3. **Note**: This section is located in the *keyPressed*(*KeyEvent*) method.

Procedure
1. The query text that the user types is stored in a String named *queryText*. Use this value to set the query string to be run by *tablePane*.
2. Use load() to run the query and load the results into *tablePane*. By calling load(), the *tablePane* object will actually run the query (using JDBC), load its results, and present them in the table. If you forget to call load(), the table will still appear, but it will be empty.

Prototypes
class SQLResultSetTablePane
- public void setQuery (String query)
- public void load ()

```
Example Code:  tp.setQuery(query);
               tp.load();
```

Part 4: Setup an error handler

Setup

1. Continue editing the SQLResultSetTablePaneExample.java source file.
2. Locate the section for Lab Exercise #3 Part #4.

Procedure

1. Any errors that occur when accessing the server are not automatically displayed to the user. You need to set up an ErrorListener to handle errors. For this example, we will use an ErrorDialogAdapter, which is an ErrorListener that handles errors by displaying them in a message box for the user to see. You can also implement your own custom error handler if you have different requirements.
2. Create an ErrorDialogAdapter object called *errorHandler* and specify *tablePane* for the component. This initializes the error handler and tells it to use *tablePane* to determine the parent frame for any message box dialogs that it displays.
3. Use addErrorListener() to add *errorHandler* as an ErrorListener to *tablePane*. This sets up the error handler to "listen" to *tablePane*. Now, whenever an error occurs in *tablePane*, this error handler will display a message box.

Prototypes

class ErrorDialogAdapter

- public ErrorDialogAdapter()
- public ErrorDialogAdapter(Component component)
- public void setComponent(Component component)

```
Example Code:  ErrorDialogAdapter eda = new ErrorDialogAdapter();
               eda.setComponent(tp);
```

class SQLResultSetTablePane

- public void addErrorListener (ErrorListener listener)

```
Example Code:  tp.addErrorListener(eda);
```

Run the application

Now it is time to run the SQLResultSetTablePaneExample application.

1. Compile the application from a DOS prompt.

   javac SQLResultSetTablePaneExample.java

2. Run the application.

   java SQLResultSetTablePaneExample

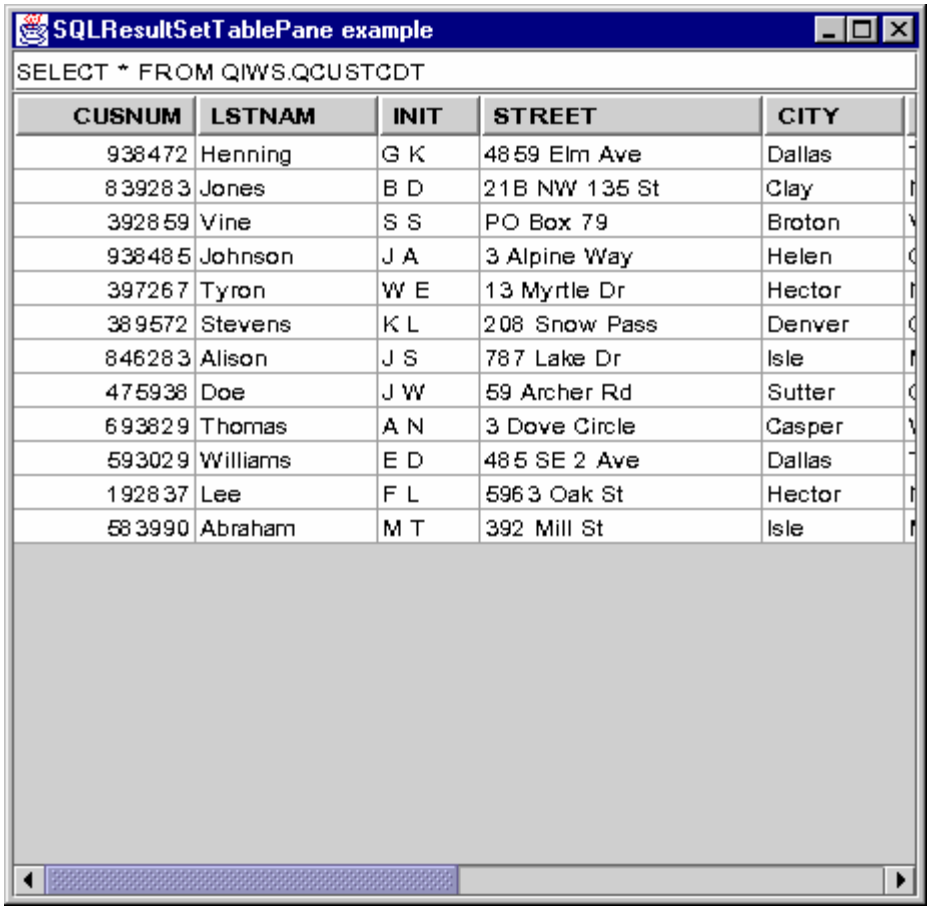
3. The application displays the graphical user interface below. It includes a text field at the top, where you can type in SQL queries. It also displays an empty table. The table is empty since we have not yet run any queries.

4. Enter an SQL query in the text field. Backspace over "Enter an SQL query here." to erase it so the SQL statement starts at the beginning of the text field. A good SQL statement to try is:

   SELECT * FROM QIWS.QCUSTCDT

5. The application will prompt you for a user ID and password. This happens the first time you run a query because this is when the physical connection to the DB2 on i5/OS is made. Enter your assigned user ID and password.

6. Verify that the results in the table look similar to this:



CUSNUM	LSTNAM	INIT	STREET	CITY
938472	Henning	G K	4859 Elm Ave	Dallas
839283	Jones	B D	21B NW 135 St	Clay
392859	Vine	S S	PO Box 79	Broton
938485	Johnson	J A	3 Alpine Way	Helen
397267	Tyron	W E	13 Myrtle Dr	Hector
389572	Stevens	K L	208 Snow Pass	Denver
846283	Alison	J S	787 Lake Dr	Isle
475938	Doe	J W	59 Archer Rd	Sutter
693829	Thomas	A N	3 Dove Circle	Casper
593029	Williams	E D	485 SE 2 Ave	Dallas
192837	Lee	F L	5963 Oak St	Hector
583990	Abraham	M T	392 Mill St	Isle

7. Close the window using the "X" in the upper right corner.

Exercise 4: Program Call

Introduction

In this exercise, you will use the IBM Toolbox for Java ProgramCall and ProgramParameter classes to call an i5/OS API**.**

Goals of this exercise

At the end of this exercise, you should be able to:

1. Use Toolbox converter classes to convert numeric and string data between Java and i5/OS types.
2. Build an array of program parameters.
3. Call an i5/OS program.
4. Parse the results of the program.

Part 1: Character Conversion

Note that all sections that need code written by you start with the comments:

```
// ------------------------------------------------------------
//             Lab Exercise #4 Part #x - Insert code here.
// ------------------------------------------------------------
```

and end with the comments:

```
// ------------------------------------------------------------
//                         End of code.
// ------------------------------------------------------------
```

Type the code for each exercise between the beginning comments of Exercise #4 Part #x and before the ending comments.

Setup

1. Edit the **qsyrusri2.java** source file. You can use any editor you like. For this lab, we will use Windows Notepad. In the DOS prompt, type:

   notepad qsyrusri2.java

2. Locate the section for Lab Exercise #4 Part #1.

Procedure

Create a text conversion object called 'char10' that converts a 10-character string between Java Unicode and i5/OS EBCDIC. Note the constructor of the converter object includes an AS400 object. The AS400 object is specified just in case a connection to the server is needed (under certain environments). For most applications, though, the Toolbox's built-in conversion classes handle String conversions between Unicode and EBCDIC without ever making a connection to the server. (Hint -- just above the line of code you enter are converters for shorter strings).

Prototypes

class AS400Text

- public AS400Text(int length, int ccsid, AS400 system*)*

  Example Code: AS400Text char6 = new AS400Text(6, system.getCcsid(), system);

class AS400

- public int getCcsid()

  Example Code: int ccsid = system.getCcsid();

Remember that at any time during this lab, if you get stuck, you can either ask a lab attendant for help or consult Appendix B for the solutions.

Part 2: Set Up the ProgramCall Object

Setup
1. Continue editing the qsyrusri2.java source file.
2. Locate the section for Lab Exercise #4 Part #2.

Procedure
Enter two lines of code. The first will create a ProgramCall object called 'pc'. On the constructor pass the AS400 object *system*. The program to call is on that server. The second line of code will set the name of the program to call. Program names are fully qualified in terms of the integrated file system. The program we are calling is QSYRUSRI.PGM which is located in QSYS.LIB.

Prototypes
class ProgramCall
- public ProgramCall(AS400 system)
- public void setProgram(String program);

Example Code: ProgramCall pc = new ProgramCall(system);
pc.setProgram("/QSYS.LIB/QSYRUSRI.PGM");

Part 3: Program Parameters

Setup
1. Continue editing the qsyrusri2.java source file.
2. Locate the section for Lab Exercise #4 Part #3.

Procedure
Add the line of code to create the third program parameter. This parameter is the data format. We will be using format USRI0100.

Hints
1. Java numbers array elements starting at 0 so the array index will be 2.
2. You must convert the string from a Java String to EBCDIC as you store the data in the program parameter.
3. Your conversion must result in the correct number of bytes (8 in this case). You should use the *char8* AS400Text object.
4. The next line of code shows setting the fourth parameter which is much like this one.

Prototypes
class ProgramParameter
- public ProgramParameter(byte[] data)

Example Code: parm[2] = new ProgramParameter(char8.toBytes("USRI0100"));

Part 4: Parsing Returned Data

Setup

1. Continue editing qsyrusri2.java
2. Locate the section for Lab Exercise #4 Part #4.

Procedure

At this point you have successfully called the program and have data back. You designated parameter 0 as the parameter getting output data and have retrieved the data as a byte array called *data*. Now you have to parse the byte array to retrieve individual values. Create a String object called 'strValue' that retrieves the profile name. It is 10 characters long, starting at byte 8 in the byte array. Since the profile name is 10 characters long, you should use the AS400Text object you created in part 1.

Prototypes

class AS400Text

- public Object toObject(byte[] serverValue, int offset)

  Example Code: Object obj = char10.toObject(data, 8);

Run the application

Now it is time to run the application.

1. Make sure to save the modified qsyrusri2.java file.
2. Compile the application from the DOS prompt:

   javac qsyrusri2.java

3. Run the application:

   java qsyrusri2

4. As in previous exercises, the Toolbox will prompt you for system name, user ID and password. This happens automatically when your application accesses the server using the IBM Toolbox for Java.

5. The application will call the i5/OS program and print results.

Conclusion

In this lab, you enabled various pieces of a client to access a server running i5/OS using various components from the IBM Toolbox for Java.

The IBM Toolbox for Java is implemented so that there are no platform dependencies in the code. Since Java is portable across many environments, pure Java programs will run on any Java enabled platform. The important implication to you as an application developer is that one version of your program will run on many platforms. This can reduce the duplicate development, maintenance, and code porting expenses usually associated with multiple platform application development.

You can get more information about the IBM Toolbox for Java and download a trial or open source version by going to the web address **http://www.ibm.com/servers/eserver/iseries/toolbox.**

Appendix A: Bonus Exercises

Bonus Exercise 1: Program Call via PCML

Introduction

In the previous exercise you called an i5/OS program using Toolbox ProgramCall. In this exercise you will call the same program but this time you will call it using the Program Call Markup Language (PCML) support in the Toolbox.

Many Toolbox objects (for example, User, Job, and DataArea) are implemented using Toolbox ProgramCall. For these objects, Toolbox presents an easy to use Java interface. Under the covers Toolbox implements these interfaces by calling i5/OS APIs. However, the ProgramCall classes can be too hard to use to call some of the more complex APIs. PCML, an implementation of XML, was created to solve the complexity problem. PCML offers the following advantages over ProgramCall:

- Parameter formats are described via HTML-like syntax. You no longer have to count offsets. You describe the format of the data and the PCML does the offset calculation for you. This is especially useful for variable length and nested structures. PCML does the messy math for you.
- PCML does data conversion for you. You describe the type and size of your data and PCML does the conversion for you.
- Parameter formats can be changed without re-compiling the application. The PCML source file is parsed at run time. If you need to change it, you simple change the .pcml file and rerun the application.

Goals of this exercise

At the end of this exercise, you should be able to use PCML to call an i5/OS program.

Part 1: Create the ProgramCallDocument Object

Note that all sections that need code written by you start with the comments:

```
// ------------------------------------------------------------
//             Lab Bonus Exercise #1 Part #x - Insert code here.
// ------------------------------------------------------------
```

and end with the comments:

```
// ------------------------------------------------------------
//                         End of code.
// ------------------------------------------------------------
```

Type the code for each exercise between the beginning comments of Lab Bonus Exercise #1 Part #x and before the ending comments.

Setup

1. Edit the qsyrusri.java source file. You can use any editor you like. For this lab, we will use Windows Notepad. In the DOS prompt, type:

   notepad qsyrusri.java

2. Locate the section for Lab Bonus Exercise #1 Part #1.

Procedure

Initialize the variable called *pcml* to the proper ProgramCallDocument object. ProgramCallDocument takes two parameters: the AS400 object representing the server and the name of the file containing the PCML source. Our PCML source file is called **qsyrusri** and the AS400 object representing the server is called **system**.

Prototypes

class ProgramCallDocument

- public ProgramCallDocument(AS400 system, String PCMLFileName)

  Example Code: ProgramCallDocument pcml = new ProgramCallDocument(system, “qsysrusri”);

Remember that at any time during this lab, if you get stuck, you can either ask a lab attendant for help or consult Appendix B for the solutions.

Part 2: Retrieve Output

Setup

1. Continue editing the qsyrusri.java source file.
2. Locate the section for Lab Bonus Exercise #1 Part #2.

Procedure

At this point you successfully called the i5/OS program and have data back. You will now pull data from the output parameter. As in the last exercise, you will retrieve the user profile name. You will let PCML do most of the work. All you have to do is name the field. PCML will convert it and return the data to you.

To determine the identifier, you need to look at the PCML source file. Edit **qsyrusri.pcml**.

- The top section of the file has a description of the USRI0100 format. Each field has a name, type and length (among other things). Type and length are used by PCML to properly process the data. Names are used to identify what field you want.
- The bottom section of the file describes the parameters of the program. Looking at the .java program you see no program name, format name, profile name, etc. That is because these parameters are really constants. 'Init' is used to set their values so these parameters are totally handled in the .pcml file.

To determine the name of the user profile field, you have to 'fully qualify' the name of the field using periods to separate the values. The first part is the program name label (qsyrusri). A PCML file can contain descriptions for many program calls so you have to identify which one to use. The second part is the name of the parameter (receiver). Note its type is 'struct' which means the structure at the top of the .pcml file describes the format of the data. The last part is the name of the field in the structure (userProfile). The full label becomes *qsyrusri.receiver.userProfile*. Add code to store the user profile in the String **value**.

Prototypes

class ProgramCallDocument

- public String getValue(String label)

  Example Code: String val = pcml.getValue("qsyrusri.receiver.userProfile");

Run the application

Now it is time to run the application.

1. Make sure to save the modified qsyrusri.java file.
2. Compile the application from the DOS prompt:

   javac qsyrusri.java

3. Run the application:

   java qsyrusri

4. As in the previous exercise, the Toolbox will prompt you for a system name, user ID and password. This happens automatically when your application accesses the server using the IBM Toolbox for Java.

5. The application will call the program and print the results.

Compare ProgramCall and PCML

Once you have both programs working, take a minute to compare the two programs (qsyrusri.java and qsyrusri2.java). The obvious difference is parameter handling. The program that uses standard program call devotes a lot of code to creating parameters and converting between i5/OS formats and Java formats. The program that uses PCML does not need that code. It describes the parameters then leaves building the parameter list and converting data to the PCML code provided by the Toolbox.

Bonus Exercise 2: GUI Builder

Introduction

Designing graphical user interfaces (GUIs) using Java can be time consuming. The IBM Toolbox for Java provides an XML[1] application called Panel Definition Markup Language (PDML) which can save you a lot of time coding complex GUIs. You can describe the components and layout of your GUI using PDML. PDML looks a lot like HTML, but the tags are different. The PDML code that you write describes the types of components (e.g., lists, text fields, buttons) that appear on your GUI and where on the GUI they are to be located. In most cases, using PDML will take much less effort than writing the corresponding Java code.

The IBM Toolbox for Java also provides a tool called GUIBuilder, which allows you to design your GUI interactively. GUIBuilder is a Java application which automatically generates PDML code for you.

In this lab, you will use the GUIBuilder to design a simple GUI. The GUI will display text boxes where the user can read and write entries to an i5/OS data queue. You will also write the Java code necessary to show the GUI on the screen.

Goals of this exercise

At the end of this exercise, you should be able to:

1. Start GUIBuilder.
2. Set the panel title.
3. Add text boxes to the panel.
4. Add buttons to the panel.
5. Create a PanelManager object.
6. Show the PanelManager object.

[1]XML stands for eXtensible Markup Language. XML is a language for describing other specific languages and is popular for describing data in a format that is easily readable by humans and easily parsable by computers.

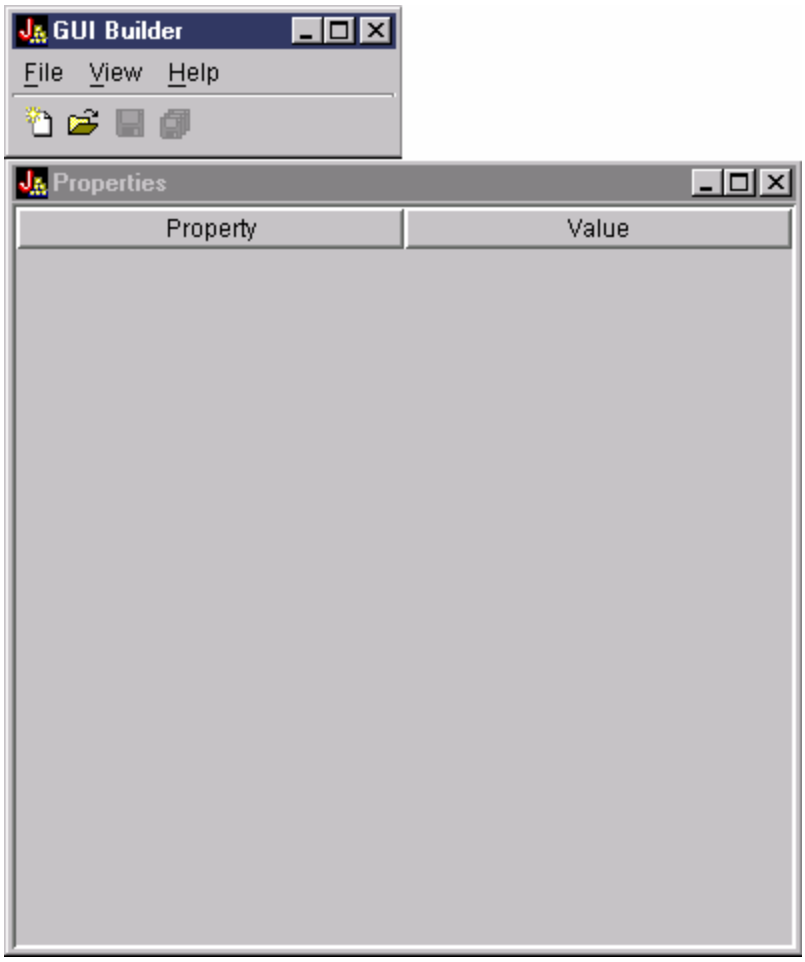
Part 1: Start GUIBuilder

Procedure

1. In the DOS prompt, type:

   java com.ibm.as400.ui.tools.GUIBuilder

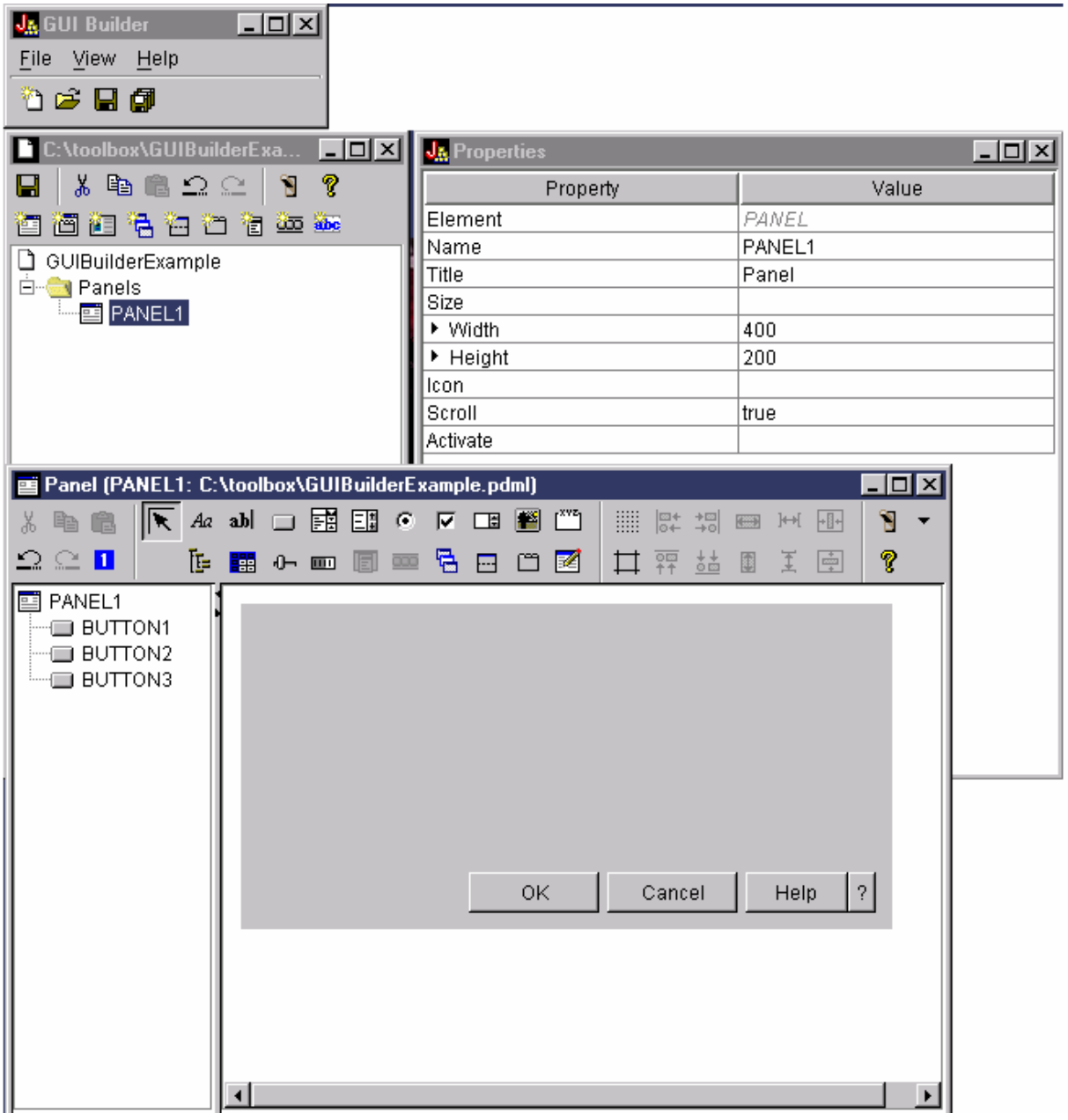
2. Verify that you see windows similar to these:



This is the main application window for GUIBuilder. There are two windows:

- The **GUIBuilder** window shows the menus for GUIBuilder.
- The **Properties** window shows the properties defined for the selected component. Since there are no components currently selected, there is nothing in this window.

3. In the **GUIBuilder** window, select the **File - New File** menu. This creates a new file for your GUI.
4. In the GUIBuilder window, select the **File-Save As** menu. This prompts for a file name. Enter **GUIBuilderExample.pdml**. Now your GUI definition will be saved in this file.
5. Click on the "Insert Panel" button which is directly above the "GUIBuilderExample" icon.
6. Verify that your windows look like this:

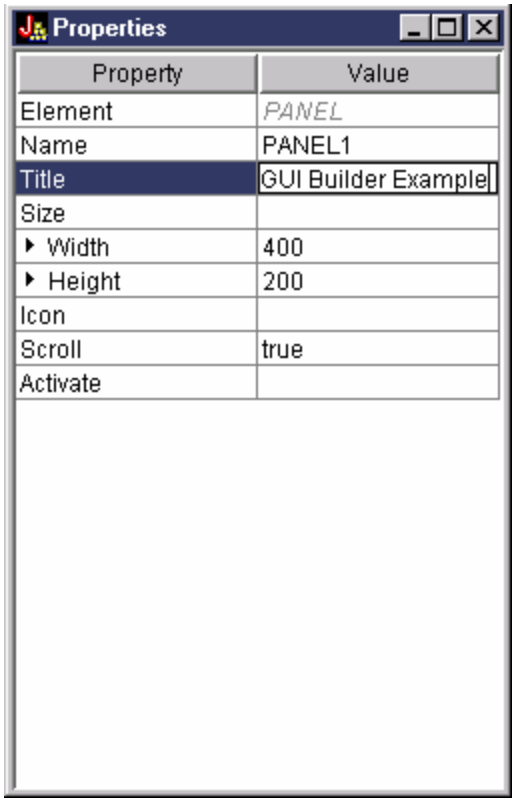


Now you are ready to design the GUI.

Part 2: Set the panel title

Procedure

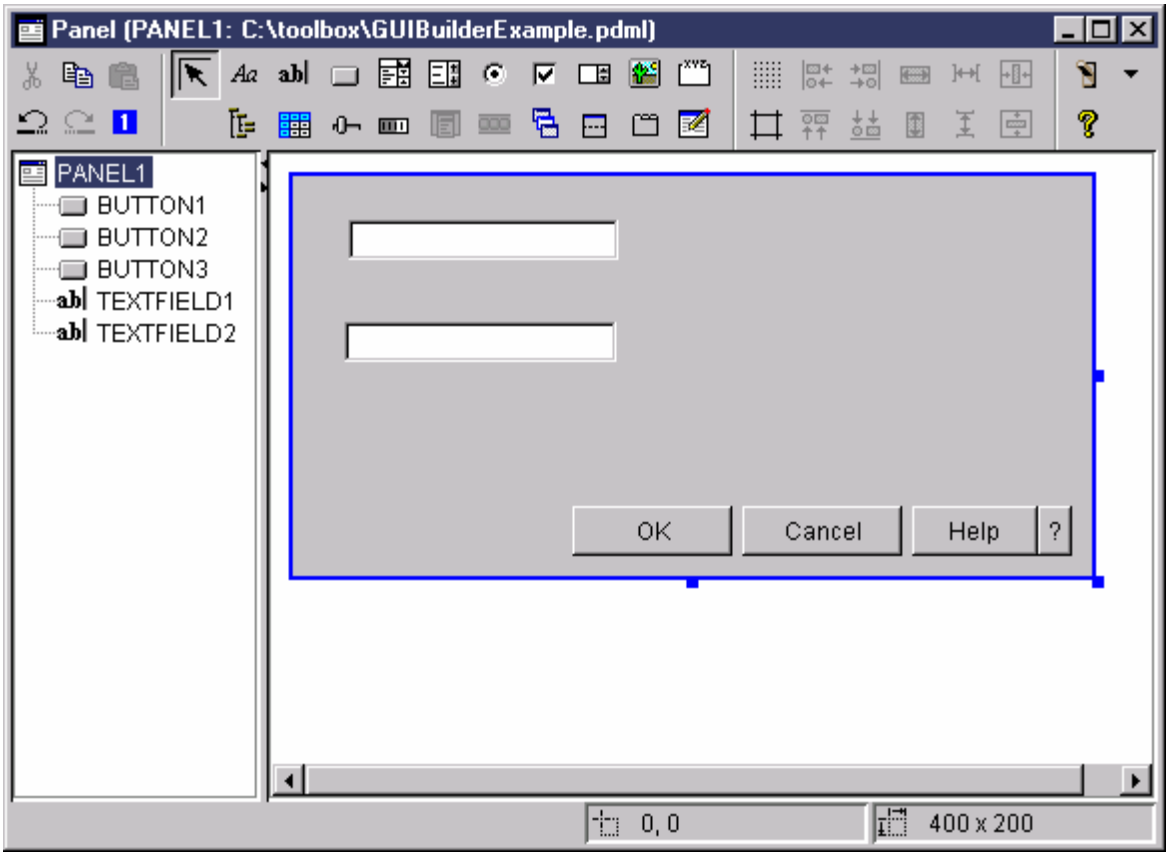
1. Make sure the **PANEL1** icon is selected. The **Properties** windows will show the properties of the panel.
2. In the **Properties** window, locate the **Title** property. Change its value to "GUI Builder Example".



Part 3: Add text boxes to the panel

Procedure

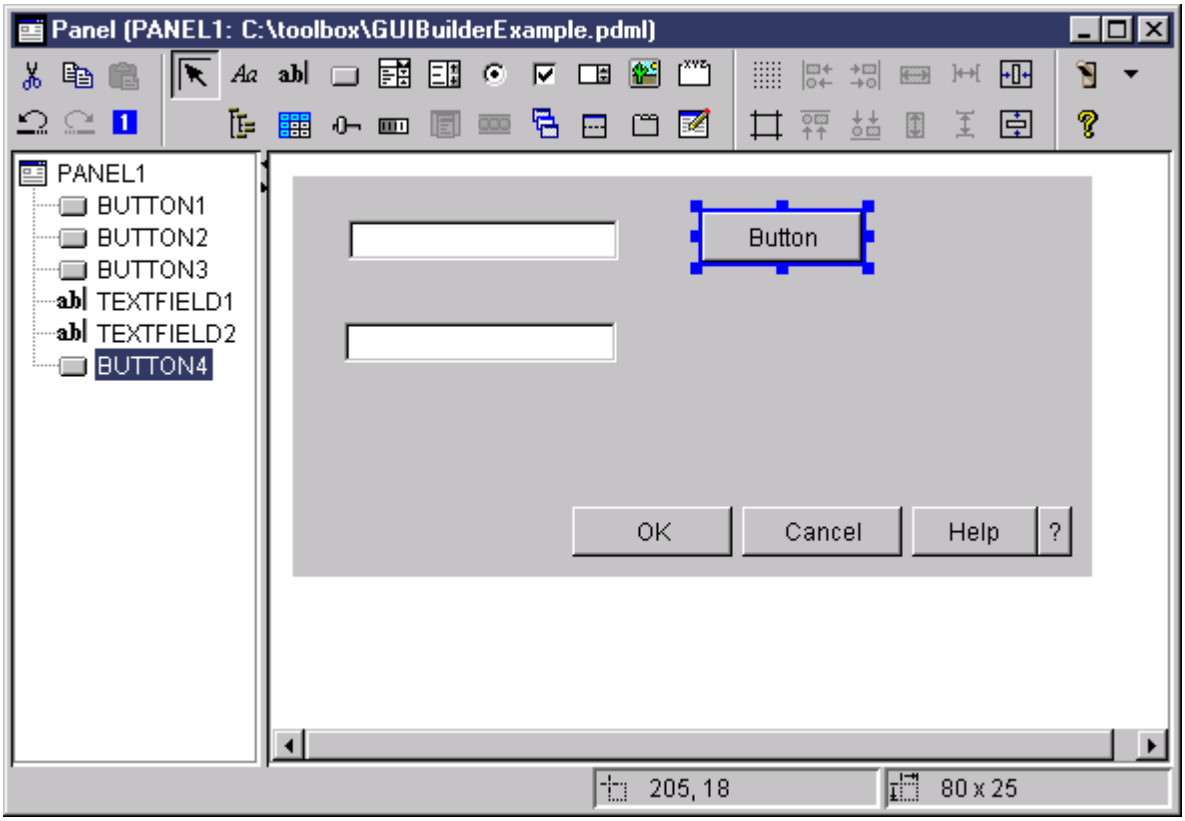
1. Locate the  icon in the **Panel** window. This represents a text box. Left click on this icon.
2. Left click anywhere in the **Panel** window where you want to place the text box. For this example, we will place it near the upper-left corner of the panel.
3. You can make the text box wider than its default width. Left click on the small dot on the right side of the text box. Drag the right side to the desired width.
4. Repeat steps 1-3 to add another text box to the panel.



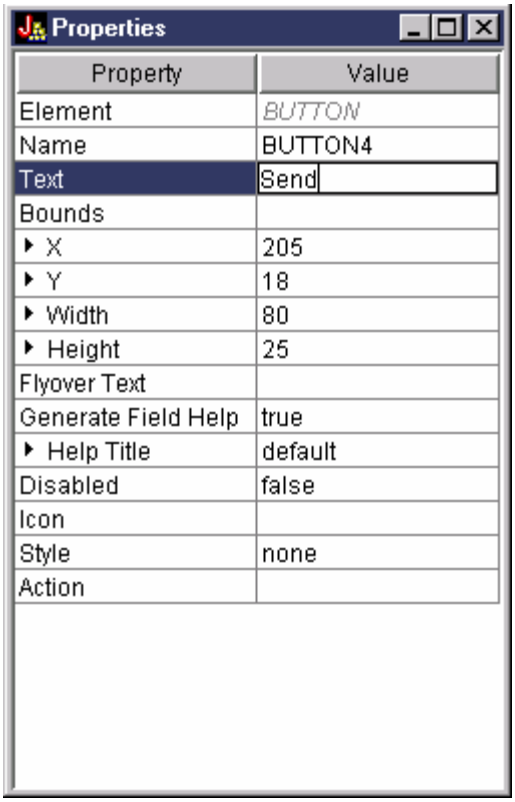
Part 4: Add buttons to the panel

Procedure

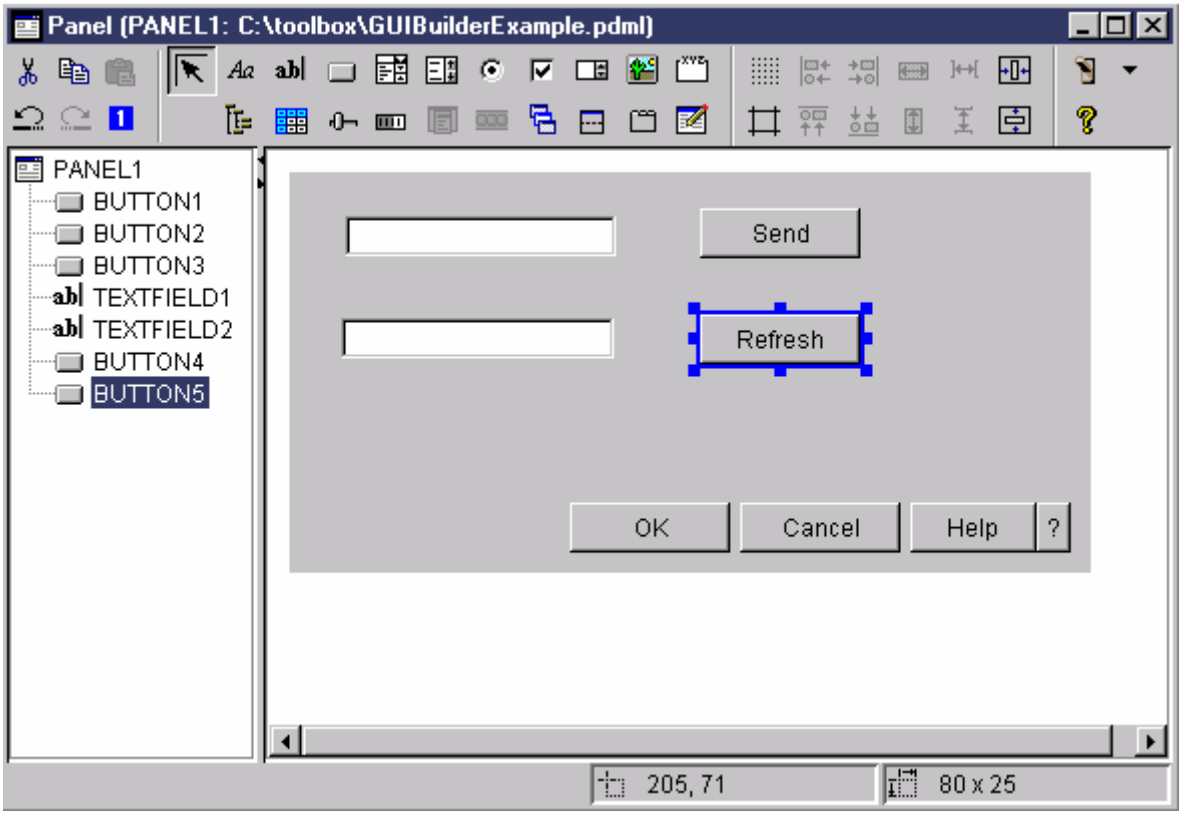
1. Locate the  icon in the **Panel** window. This represents a button. Left click on this icon.
2. Left click anywhere in the **Panel** window where you want to place the button. For this example, we will place it to the right of the first text field.



3. Notice that the **Properties** window shows the properties of the button. Locate the **Text** property. Change its value to "Send" and press enter. Notice that the button's label changes in the **Panel** window.



4. Repeat steps 1-3 to add another button with the label “Refresh”.



5. In the **GUIBuilder** window, select the **File - Save** menu. This will save the GUI definition in a file called GUIBuilderExample.pdml.
6. In the **GUIBuilder** window, select the **File - Exit** menu. This will exit GUIBuilder.

Part 5: Create a PanelManager object

The GUI definition that you have just completed does not work by itself. All you have done is described what the GUI looks like. You need to write Java code that displays the GUI and implements the behavior of the program. For this part of the lab, most of the Java application code is provided for you. You will need to write Java code to load the GUI definition and display it on the screen.

Setup

1. Edit the GUIBuilderMain.java source file. In the DOS prompt, type:

   notepad GUIBuilderMain.java

2. Locate the section for Lab Bonus Exercise #2 Part #5.

Procedure

1. Create a PanelManager object named *pm*. In the constructor, specify "GUIBuilderExample" as the base name of the GUI definition, "PANEL1" as the name of the panel, and *null* for the data beans parameter.

Prototypes
class PanelManager

- public PanelManager(String baseName, String panelName, DataBean[] dataBeans)

  Example Code: PanelManager pm = new PanelManager("Example", "Panel", null);

Part 6: Show the PanelManager object

Setup

1. Continue editing the GUIBuilderMain.java source file.
2. Locate the section for Lab Bonus Exercise #2 Part #6.

Procedure

1. Show the GUI by calling the setVisible(true) method on *pm*, the PanelManager object.

Prototypes
class PanelManager

- public void setVisible(boolean show)

  Example Code: pm.setVisible(true);

Run the application

Now it is time to run the GUIBuilderMain application.

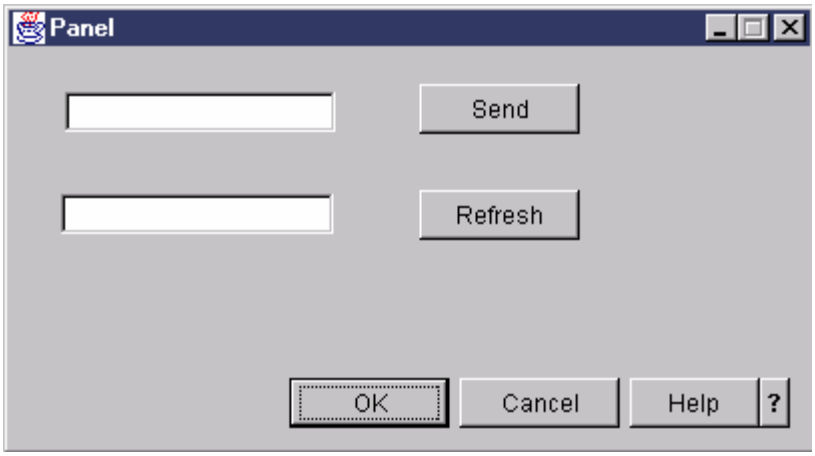
1. Compile the application from a DOS prompt.

   javac GUIBuilderMain.java

2. Run the application.

   java GUIBuilderMain

   You may see a **java.util.MissingResourceException** message. This is just a warning message that no help text was found. (GUIBuilder allows you to define help text for the panel, but we did not define any for this exercise.) You can ignore this message.
3. The application displays the user ID and password prompt. Once signed on, it will display the GUI that you designed.
4. Enter some text into the first text box and click the **Send** button. This will write the text as an entry to an i5/OS data queue.



5. Click the **Refresh** button. This will retrieve the most recent data queue entry from the server and display it in the second text box. Remember that this data queue entry may have been added by someone else in the lab.
6. Close the window using the “X” in the upper right corner.

Appendix B: Solutions

Exercise 1: Command Call

```
import com.ibm.as400.access.AS400;
import com.ibm.as400.access.AS400Message;
import com.ibm.as400.access.CommandCall;

public class CommandCallExample
{
  public static void main(String[] args)
  {
    try
    {

      // ----------------------------------------------------------
      //             Lab Exercise #1 Part #1 - Insert code here.
      // ----------------------------------------------------------

      AS400 system = new AS400("mySystem");     // used to get a connection

      // ----------------------------------------------------------
      //                           End of code.
      // ----------------------------------------------------------

      // ----------------------------------------------------------
      //             Lab Exercise #1 Part #2 - Insert code here.
      // ----------------------------------------------------------

      CommandCall command = new CommandCall(system); // used to run a command

      // ----------------------------------------------------------
      //                           End of code.
      // ----------------------------------------------------------

      // Gather the command line arguments passed to this .
      StringBuffer buffer = new StringBuffer();
      for (int i = 0; i < args.length; ++i)
      {
        buffer.append (args[i]);
        buffer.append (" ");
      }
      String commandString = buffer.toString ();

      // ----------------------------------------------------------
      //             Lab Exercise #1 Part #3 - Insert code here.
      // ----------------------------------------------------------

      if (command.run(commandString)) // runs the command, returns a boolean
        System.out.println("The command was successful.");
      else
        System.out.println("The command failed.");
```

```
      // -------------------------------------------------------------
      //                        End of code.
      // -------------------------------------------------------------


      // -------------------------------------------------------------
      //             Lab Exercise #1 Part #4 - Insert code here.
      // -------------------------------------------------------------

      // gets the list of messages returned after the command runs
      AS400Message[] messageList = command.getMessageList();
      for (int i=0; i < messageList.length; i++)     // outputs each message
      {
        System.out.println (messageList[i].getID() + ":" +
                            messageList[i].getText());
      }

      // -------------------------------------------------------------
      //                        End of code.
      // -------------------------------------------------------------

    }
    catch (Exception e) {
      System.out.println ("Error: " + e);
    }

    System.exit (0);
  }
}
```

Exercise 2: JDBC Query

```
import java.sql.*;      // Part of the JDK
import com.ibm.as400.access.*;      // import the Toolbox's JDBC classes

public class JDBCQuery
{
    // Format a string so that it has the specified width.
    private static String format (String s, int width)
    {
        String formattedString;

        // The string is shorter than specified width,
        // so we need to pad with blanks.
        if (s.length() < width)
        {
            StringBuffer buffer = new StringBuffer (s);

            for (int i = s.length(); i < width; ++i)
                buffer.append (" ");

            formattedString = buffer.toString();
        }

        // Otherwise, we need to truncate the string.
        else
            formattedString = s.substring (0, width);

        return formattedString;
    }


    public static void main (String[] parameters)
    {
      // -------------------------------------------------------------
      //             Intro Lab Exercise #2 Part #1 - Finish code here.
      // -------------------------------------------------------------
        String system        = "mySystem"; // name of the system to connect to
        String collectionName= "QIWS";    // collection/library name for table
        String tableName     = "QCUSTCDT"; // table/file name
      // -------------------------------------------------------------
      //                            End of code.
      // -------------------------------------------------------------

        Connection connection   = null;   // Creates a connection object

        try {

      // -------------------------------------------------------------
      //             Intro Lab Exercise #2 Part #2 - Insert code here.
      // -------------------------------------------------------------
        // Load the IBM Toolbox for Java JDBC driver.
        DriverManager.registerDriver(new
                                com.ibm.as400.access.AS400JDBCDriver());
      // -------------------------------------------------------------
      //                            End of code.
      // -------------------------------------------------------------
```

```
// -------------------------------------------------------------
//             Intro Lab Exercise #2 Part #3 - Insert code here.
// -------------------------------------------------------------
  // Get a connection to the database.  Since we do not
  // provide a user id or password, a prompt will appear
  connection = DriverManager.getConnection ("jdbc:as400://" + system);
// -------------------------------------------------------------
//                           End of code.
// -------------------------------------------------------------

      // retrieve information about the database
      DatabaseMetaData dmd = connection.getMetaData ();

      // Execute the query.
      Statement select = connection.createStatement ();

// -------------------------------------------------------------
//             Intro Lab Exercise #2 Part #4 - Insert code here.
// -------------------------------------------------------------
      // Executes the query and retrieves the results
      ResultSet rs = select.executeQuery ("SELECT * FROM "
          + collectionName + dmd.getCatalogSeparator() + tableName);
// -------------------------------------------------------------
//                           End of code.
// -------------------------------------------------------------

      // Get information about the result set.  Set the column
      // width to whichever is longer: the length of the label
      // or the length of the data.
// -------------------------------------------------------------
//             Intro Lab Exercise #2 Part #5 -
// -------------------------------------------------------------
      // Retrieve information about the results
      ResultSetMetaData rsmd = rs.getMetaData ();
      int columnCount = rsmd.getColumnCount () - 1;  // skip last column

      // code below formats how the data in the ResultSet is displayed
      String[] columnLabels = new String[columnCount];
      int[] columnWidths = new int[columnCount];

      for (int i = 1; i <= columnCount; ++i)
      {
          columnLabels[i-1] = rsmd.getColumnLabel (i);
          columnWidths[i-1] = Math.max (columnLabels[i-1].length(),
              rsmd.getColumnDisplaySize (i));
      }

      // Output the column headings.
      for (int i = 1; i <= columnCount; ++i)
      {
          System.out.print (format (rsmd.getColumnLabel(i),
                                    columnWidths[i-1]));
          System.out.print (" ");
      }
      System.out.println ();

      // Output a dashed line.
      StringBuffer dashedLine;
      for (int i = 1; i <= columnCount; ++i)
      {
          for (int j = 1; j <= columnWidths[i-1]; ++j)
          {
```

```
                    System.out.print ("-");
                }
                System.out.print (" ");
            }
            System.out.println ();

            // Iterate throught the rows in the result set and output
            // the columns for each row.
        // ------------------------------------------------------------
        //             Intro Lab Exercise #2 Part #5 -
        // ------------------------------------------------------------
            // goes through the ResultSet one row at a time
            while (rs.next ())
            {
                for (int i = 1; i <= columnCount; ++i)
                {
                    String value = rs.getString (i); // retrieves column value
                    if (rs.wasNull ())
                    {
                        value = "<null>";
                    }
                    System.out.print (format (value, columnWidths[i-1]));
                    System.out.print (" ");
                }
                System.out.println ();
            }

        }
        catch (Exception e)
        {
           System.out.println ();
           System.out.println ("ERROR: " + e.getMessage());
        }

        // Clean up.
        try
        {
           if (connection != null)
              connection.close (); // closes the connection and any statements
        }
        catch (SQLException e) { }         // Ignore errors.

        System.exit (0);
    }
}
```

Exercise 3: SQL Result Set Table Pane

```
import com.ibm.as400.access.AS400JDBCDriver;
import com.ibm.as400.vaccess.ErrorDialogAdapter;
import com.ibm.as400.vaccess.SQLConnection;
import com.ibm.as400.vaccess.SQLResultSetTablePane;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.beans.*;
import java.sql.*;

public class SQLResultSetTablePaneExample
extends KeyAdapter
{
  private static SQLResultSetTablePane    tablePane_;
  private static JTextField               textField_;

  public static void main(String argv[])
  {
    try {
      // Register the IBM Toolbox for Java JDBC driver.
      DriverManager.registerDriver(new AS400JDBCDriver());

      // -------------------------------------------------------------
      //             Lab Exercise #3 Part #1 - Insert code here.
      // -------------------------------------------------------------

      // Get a connection to the database
      SQLConnection connection = new SQLConnection("jdbc:as400://mySystem");

      // -------------------------------------------------------------
      //                           End of code.
      // -------------------------------------------------------------

      // -------------------------------------------------------------
      //             Lab Exercise #3 Part #2 - Insert code here.
      // -------------------------------------------------------------
      SQLResultSetTablePane tablePane = new SQLResultSetTablePane();
      tablePane.setConnection (connection);
      // -------------------------------------------------------------
      //                           End of code.
      // -------------------------------------------------------------

      // Store the table pane in a static variable.
      tablePane_ = tablePane;

      // Initialize the text area.
      textField_ = new JTextField ("Enter an SQL query here.");
      textField_.addKeyListener (new SQLResultSetTablePaneExample ());

      // Initialize the frame.
```

```
      JFrame frame = new JFrame ("SQLResultSetTablePane example");
      frame.getContentPane ().setLayout (new BorderLayout ());
      frame.getContentPane ().add ("North", textField_);
      frame.getContentPane ().add ("Center", tablePane_);

      // When the frame closes, exit the .
      frame.addWindowListener (new WindowAdapter ()
      {
        public void windowClosing (WindowEvent event) { System.exit (0);}
      });

      // ---------------------------------------------------------------
      //             Lab Exercise #3 Part #4 - Insert code here.
      // ---------------------------------------------------------------
      ErrorDialogAdapter errorHandler = new ErrorDialogAdapter (tablePane);
      tablePane.addErrorListener (errorHandler);
      // ---------------------------------------------------------------
      //                           End of code.
      // ---------------------------------------------------------------

      // Display the frame.
      frame.pack();
      frame.show();
    } catch (Exception e) {
      System.out.println ("Error: " + e);
    }
  }

  // This gets called whenever a key is pressed in the text area.
  public void keyPressed (KeyEvent event)
  {
    try {
      // Check for the Enter key.
      if (event.getKeyCode () == KeyEvent.VK_ENTER) {
        String queryText = textField_.getText ();

        SQLResultSetTablePane tablePane = tablePane_;

        // ---------------------------------------------------------------
        //             Lab Exercise #3 Part #3 - Insert code here.
        // ---------------------------------------------------------------
        tablePane.setQuery (queryText);
        tablePane.load ();
        // ---------------------------------------------------------------
        //                           End of code.
        // ---------------------------------------------------------------
      }
    } catch (Exception e) {
      System.out.println ("Error: " + e);
    }
  }
}
```

Exercise 4: Program Call

```
import com.ibm.as400.access.*;

 //
 // This program calls i5/OS API QSYRUSRI (Retrieve User Information)
 // to get information on the current user.
 //

 public class qsyrusri2
 {
     public static void main(String[] argv)
     {
         try
         {
             String msgId, msgText;

             AS400 system = new AS400();  // used to get a connection

             // Create a binary converter to go between an i5/OS
             // bin 4 and a Java int.
             AS400Bin4 bin4   = new AS400Bin4();

             // Create text converters for the various sizes
             // of strings will use.
             AS400Text char6  = new AS400Text(6, system.getCcsid(),
                                              system);   // text, size 6
             AS400Text char7  = new AS400Text(7, system.getCcsid(),
                                              system);   // text, size 7
             AS400Text char8  = new AS400Text(8, system.getCcsid(),
                                              system);   // text, size 8
      // -------------------------------------------------------------
      //             Intro Lab Exercise #4 Part #1 - Insert code here.
      // -------------------------------------------------------------
             AS400Text char10 = new AS400Text(10, system.getCcsid(),
                                               system);  // text, size 10
      // -------------------------------------------------------------
      //                           End of code.
      // -------------------------------------------------------------

             // Reroute System.error to System.out
             System.setErr(System.out);

             // Create a program call object.  Then
             // set the program name to "/QSYS.LIB/QSYRUSRI.PGM"
      // -------------------------------------------------------------
      //             Intro Lab Exercise #4 Part #2 - Insert code here.
      // -------------------------------------------------------------
             ProgramCall pc = new ProgramCall(system);
             pc.setProgram("/QSYS.LIB/QSYRUSRI.PGM");
      // -------------------------------------------------------------
      //                           End of code.
      // -------------------------------------------------------------

             // The program has five parameters.
             ProgramParameter[] parms = new ProgramParameter[5];

             // First parm is the output area that contains the result.
             // The parameter value 100 means expect 100 bytes back.
             parms[0] = new ProgramParameter(100);
```

```
        // Second parm is the size of the output area
        // (100 bytes in our case).
        parms[1] = new ProgramParameter(bin4.toBytes(100));

        // Third parm is the output format to use.  The
        // server expects the format to be in EBCDIC so we
        // use our character converter to convert from a Java
        // Unicode string to a byte array containing EBCDIC bytes.
// ------------------------------------------------------------
//             Intro Lab Exercise #4 Part #3 - Insert code here.
// ------------------------------------------------------------
        parms[2] = new ProgramParameter(char8.toBytes("USRI0100"));
// ------------------------------------------------------------
//                           End of code.
// ------------------------------------------------------------

        // Fourth parm is the user profile.  It is also converted
        // from Unicode to EBCDIC.
        parms[3] = new ProgramParameter(char10.toBytes("*CURRENT"));

        // Fifth parm is the 32 byte error area.
        byte[] errorArea = new byte[32];
        parms[4] = new ProgramParameter(errorArea, 32);

        System.out.println("Beginning ProgramCall Example..");
        System.out.println("    Setting input parameters...");

        // Give the list of parameters to the program call object.
        pc.setParameterList(parms);

        // Call the program.  If the return code is false,
        // we received messages from the server saying why the
        // program failed.  For example, program not found,
        // not authorized to program, etc.  The failure to connect
        // to the server will show up as an exception, not a message.
        if (pc.run() == false)
        {
            // Retrieve list of i5/OS messages
            AS400Message[] msgs = pc.getMessageList();

            // Iterate through messages and write them to standard output
            for (int m = 0; m < msgs.length; m++)
            {
                msgId = msgs[m].getID();
                msgText = msgs[m].getText();
                System.out.println("    " + msgId + " - " + msgText);
            }
            System.out.println("** Call to QSYRUSRI failed. **");
        }
        // else we were able to call the program and received data.
        else
        {
            // Pull the data out of the output parm (the first parameter)
            byte[] data = parms[0].getOutputData();

            // Print various values out of the return data.  Note
            // the programmer has to know where in the buffer the
            // data starts.  Also note the data at this point is
            // binary / EBCDIC.  To be used by Java it must be
            // converted to Java types (int, long, ...) and Java
            // Strings.
```

```
                int value   = ((Integer) bin4.toObject(data)).intValue();
                System.out.println("        Bytes returned:      " + value);

                value   = ((Integer) bin4.toObject(data, 4)).intValue();
                System.out.println("        Bytes available:     " + value);

        // -------------------------------------------------------------
        //             Intro Lab Exercise #4 Part #4 - Insert code here.
        // -------------------------------------------------------------
                String strValue = (String) char10.toObject(data, 8);
        // -------------------------------------------------------------
        //                           End of code.
        // -------------------------------------------------------------
                System.out.println("        Profile name:        " +
                                                                strValue);

                strValue = (String) char7.toObject(data, 18);
                System.out.println("        Previous signon date:" +
                                                                strValue);

                strValue = (String) char6.toObject(data, 25);
                System.out.println("        Previous signon time:" +
                                                                strValue);
            }
        }
        catch (Exception e)
        {
            System.out.println("Unexpected Exception ");
            e.printStackTrace();
            System.out.println("*** Call to QSYRUSRI failed. ***");
        }

        System.exit(0);
    }
}
```

Bonus Exercise 1: PCML (Java Source)

```
import com.ibm.as400.data.ProgramCallDocument;
import com.ibm.as400.data.PcmlException;
import com.ibm.as400.access.AS400;
import com.ibm.as400.access.AS400Message;

//
// This program calls i5/OS API QSYRUSRI (Retrieve User Information)
// to get information on the current user.
//

public class qsyrusri {

    public static void main(String[] argv)
    {
        ProgramCallDocument pcml;
        boolean rc = false;
        String msgId, msgText;
        Object value;

        System.setErr(System.out);     // Redirect System.error to System.out

        // Construct AS400 without parameters, user will be prompted
        // for system name, userid and password.
        AS400 system = new AS400(); // used to get a connection

        try
        {
            System.out.println("Beginning PCML Example..");
            System.out.println("   Constructing ProgramCallDocument for QSYRUSRI API...");

            // Construct ProgramCallDocument.
            // First parameter is system to connect to.
            // Second parameter is pcml resource name. In this example,
            // serialized PCML file "qsyrusri.pcml.ser" or
            // PCML source file "qsyrusri.pcml" must be found in the classpath.
     // -------------------------------------------------------------
     //             Intro Lab Bonus Exercise #1 Part #1 - Insert code here.
     // -------------------------------------------------------------
            pcml = new ProgramCallDocument(system, "qsyrusri");
     // -------------------------------------------------------------
     //                           End of code.
     // -------------------------------------------------------------

            // Set input parameters. Several parameters have default values
            // specified in the PCML source. No need to set them using Java code.
            System.out.println("    Setting input parameters...");
            pcml.setValue("qsyrusri.receiverLength",
                          new Integer((pcml.getOutputsize("qsyrusri.receiver"))));

            // Request to call the API
            // User will be prompted to sign on to the system
            System.out.println("    Calling QSYRUSRI API.");
            rc = pcml.callProgram("qsyrusri");

            // If return code is false, we received messages from the i5/OS
            if(rc == false)
            {
                // Retrieve list of i5/OS messages
                AS400Message[] msgs = pcml.getMessageList("qsyrusri");

                // Iterate through messages and write them to standard output
                for (int m = 0; m < msgs.length; m++)
                {
                    msgId = msgs[m].getID();
                    msgText = msgs[m].getText();
                    System.out.println("    " + msgId + " - " + msgText);
```

```
                }

                System.out.println("** Call to QSYRUSRI failed.  **");
                System.exit(0);
            }
            // Return code was true, call to QSYRUSRI succeeded
            // Write some of the results to standard output
            else
            {
                value = pcml.getValue("qsyrusri.receiver.bytesReturned");
                System.out.println("        Bytes returned:      " + value);
                value = pcml.getValue("qsyrusri.receiver.bytesAvailable");
                System.out.println("        Bytes available:     " + value);
      // -------------------------------------------------------------
      //             Intro Lab Bonus Exercise #1 Part #2 - Insert code here.
      // -------------------------------------------------------------
                value = pcml.getValue("qsyrusri.receiver.userProfile");
      // -------------------------------------------------------------
      //                           End of code.
      // -------------------------------------------------------------
                System.out.println("        Profile name:        " + value);
                value = pcml.getValue("qsyrusri.receiver.previousSignonDate");
                System.out.println("        Previous signon date:" + value);
                value = pcml.getValue("qsyrusri.receiver.previousSignonTime");
                System.out.println("        Previous signon time:" + value);
            }
        }
        catch (PcmlException e)
        {
            System.out.println(e.getLocalizedMessage());
            e.printStackTrace();
            System.out.println("*** Call to QSYRUSRI failed. ***");
            System.exit(0);
        }

        System.exit(0);
    }
}
```

Bonus Exercise #1: PCML (PCML Source)

```
<pcml version="1.0">

 <!-- PCML source for calling "Retreive user Information" (QSYRUSRI) API -->

   <!-- Format USRI0150 - Other formats are available -->
   <struct name="usri0100">
     <data name="bytesReturned"              type="int"    length="4"  usage="output"/>
     <data name="bytesAvailable"             type="int"    length="4"  usage="output"/>
     <data name="userProfile"                type="char"   length="10" usage="output"/>
     <data name="previousSignonDate"         type="char"   length="7"  usage="output"/>
     <data name="previousSignonTime"         type="char"   length="6"  usage="output"/>
     <data                                   type="byte"   length="1"  usage="output"/>
     <data name="badSignonAttempts"          type="int"    length="4"  usage="output"/>
     <data name="status"                     type="char"   length="10" usage="output"/>
     <data name="passwordChangeDate"         type="byte"   length="8"  usage="output"/>
     <data name="noPassword"                 type="char"   length="1"  usage="output"/>
     <data                                   type="byte"   length="1"  usage="output"/>
     <data name="passwordExpirationInterval" type="int"    length="4"  usage="output"/>
     <data name="datePasswordExpires"        type="byte"   length="8"  usage="output"/>
     <data name="daysUntilPasswordExpires"   type="int"    length="4"  usage="output"/>
     <data name="setPasswordToExpire"        type="char"   length="1"  usage="output"/>
     <data name="displaySignonInfo"          type="char"   length="10" usage="output"/>
   </struct>

   <!-- Program QSYRUSRI and its parameter list for retrieving USRI0100 format -->
   <program name="qsyrusri" path="/QSYS.lib/QSYRUSRI.pgm">
     <data name="receiver"                   type="struct"             usage="output"
           struct="usri0100"/>
     <data name="receiverLength"             type="int"    length="4"  usage="input" />
     <data name="format"                     type="char"   length="8"  usage="input"
           init="USRI0100"/>
     <data name="profileName"                type="char"   length="10" usage="input"
           init="*CURRENT"/>
     <data name="errorCode"                  type="int"    length="4"  usage="input"
           init="0"/>
   </program>

 </pcml>
```

Bonus Exercise #2: GUI Builder

```
import com.ibm.as400.access.*;
import com.ibm.as400.ui.framework.java.*;
import javax.swing.*;

import java.awt.*;
import java.awt.event.*;

public class GUIBuilderMain
implements ActionListener
{

    private static DataQueue    dq;
    private static JTextField   sendField;
    private static JButton      sendButton;
    private static JTextField   refreshField;
    private static JButton      refreshButton;



    public static void main(String[] args)
    {

        try {

            // -------------------------------------------------------------
            //        Intro Lab Bonus Exercise #2 Part #5 - Insert code here.
            // -------------------------------------------------------------

            PanelManager pm = new PanelManager("GUIBuilderExample",
                                               "PANEL1",
                                               null);

            // -------------------------------------------------------------
            //                           End of code.
            // -------------------------------------------------------------

            // Initialize the components as defined in GUIBuilderExample.pdml.
            sendField       = (JTextField)pm.getComponent("TEXTFIELD1");
            sendButton      = (JButton)pm.getComponent("BUTTON4");
            refreshField    = (JTextField)pm.getComponent("TEXTFIELD2");
            refreshButton   = (JButton)pm.getComponent("BUTTON5");

            // Initialize the AS400 object.
            AS400 system = new AS400();

            // Initialize the data queue object.  Create the data queue if
            // it is not already created.
            dq = new DataQueue(system, "/QSYS.LIB/COMMON.LIB/COMMON.DTAQ");
            try {
                dq.create(500, "*ALL", true, false, false, "");
```

```
            }
            catch(Exception e) {
                // Ignore.  This means that the data queue is already
                // created.
            }
            // Add a listener which makes the Send button write
            // an entry to the data queue whenever it is pressed.
            sendButton.addActionListener(new GUIBuilderMain());

            // Add a listener which makes the Refresh button
            // peek the data queue whenever it is pressed.
            refreshButton.addActionListener(new GUIBuilderMain());

            // ------------------------------------------------------------
            //      Intro Lab Bonus Exercise #2 Part #6 - Insert code here.
            // ------------------------------------------------------------

            pm.setVisible(true);

            // ------------------------------------------------------------
            //                            End of code.
            // ------------------------------------------------------------
        }
        catch(Exception e) {
            e.printStackTrace();
        }
    }


    public void actionPerformed(ActionEvent event)
    {
        try {

            // Write an entry to the data queue when the send
            // button is pressed.
            if (event.getSource() == sendButton) {
                dq.write(sendField.getText());
            }

            // Peek the data queue when the refresh button
            // is pressed.
            else {
                String text = dq.peek().getString();
                refreshField.setText(text);
            }
        }
        catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```