

# **AS/400 Toolbox for Java and GUI Components**

## *Lab 4*

*Fall Common 1998*

**Clifton Nock and AS/400 Toolbox for Java Team**

**IBM Corporation**

## **Disclaimer**

This package is available for use AS IS. There is no support or service to the documentation and the code shipped with the package. IBM reserves all the rights to the lab material. This self-study material is provided for personal use. Reproduction of the material for commercial use is prohibited unless written agreement is provided by IBM.

# LAB: AS/400 Toolbox for Java - Exercises

Introduction .....	4
Overview .....	4
<b>The Java™ Language .....</b>	<b>4</b>
<b>AS/400 Toolbox for Java .....</b>	<b>5</b>
<b>The Lab .....</b>	<b>6</b>
<b>Lab Setup .....</b>	<b>7</b>
<b>Exercise 1: Command Call .....</b>	<b>8</b>
<b>Introduction .....</b>	<b>8</b>
<b>Goals of this exercise .....</b>	<b>8</b>
<b>Part 1: Create an AS400 object .....</b>	<b>9</b>
<b>Part 2: Create a CommandCall object .....</b>	<b>10</b>
<b>Part 3: Run the command .....</b>	<b>10</b>
<b>Part 4: Retrieve AS400Message objects .....</b>	<b>11</b>
<b>Run the program .....</b>	<b>12</b>
<b>Exercise 2: Data Queue .....</b>	<b>13</b>
<b>Introduction .....</b>	<b>13</b>
<b>Goals of this exercise .....</b>	<b>13</b>
<b>Part 1: Connect to the AS/400 Data Queue service .....</b>	<b>14</b>
<b>Part 2: Create a DataQueue object .....</b>	<b>15</b>
<b>Part 3: Create a Data Queue on the AS/400 .....</b>	<b>16</b>
<b>Part 4: Write a string to the Data Queue .....</b>	<b>16</b>
<b>Part 5: Peek an entry from the Data Queue .....</b>	<b>17</b>
<b>Run the program .....</b>	<b>18</b>
<b>Exercise 3: JDBC .....</b>	<b>19</b>
<b>Introduction .....</b>	<b>19</b>
<b>Goals of this exercise .....</b>	<b>19</b>
<b>Part 1: Register a JDBC Driver and Connect to an AS/400 Database .....</b>	<b>20</b>
<b>Part 2: Create a Collection .....</b>	<b>21</b>
<b>Part 3: Create a Table .....</b>	<b>22</b>
<b>Part 4: Insert Records into a Database .....</b>	<b>23</b>
<b>Part 5: Query Records from a Database .....</b>	<b>24</b>
<b>Part 6: Extract Information about a Result Set .....</b>	<b>25</b>
<b>Part 7: Extract Column Information from a Row .....</b>	<b>26</b>
<b>Run the program .....</b>	<b>27</b>
<b>Exercise 4: Record-level Access .....</b>	<b>28</b>
<b>Introduction .....</b>	<b>28</b>
<b>Goals of this exercise .....</b>	<b>28</b>
<b>Part 1: Create a Record Format .....</b>	<b>29</b>

<b>Part 2: Create a Sequential File object</b>	31
<b>Part 3: Set the Record Format and open a file</b>	32
<b>Part 4: Read a Record from a File</b>	32
<b>Part 5: Disconnect the Record-level access Service</b>	33
<b>Run the program</b>	34
<b>Exercise 5: Program Call</b>	35
<b>Introduction</b>	35
<b>Goals of this exercise</b>	35
<b>Part 1: Create a ProgramCall object</b>	36
<b>Part 2: Create a ProgramParameter list</b>	37
<b>Part 3: Run the AS/400 program</b>	38
<b>Part 4: Retrieve the Messages from the Message Queue object</b>	38
<b>Run the program</b>	40
<b>Exercise 6: Java on the AS/400 (Optional)</b>	41
<b>Introduction</b>	41
<b>Goals of this exercise</b>	41
<b>Part 1: Create Java Source Code on the AS/400</b>	41
<b>Part 2: Compile the Source Code on the AS/400</b>	43
<b>Part 3: Run the program on the AS/400</b>	44
<b>Exercise 7: SQL Result Set Table Pane</b>	45
<b>Introduction</b>	45
<b>Goals of this exercise</b>	45
<b>Part 1: Create an SQLConnection object</b>	45
<b>Part 2: Create an SQLResultSetTablePane object</b>	46
<b>Part 3: Run a query and load the results</b>	46
<b>Part 4: Setup an error handler</b>	47
<b>Run the program</b>	48
<b>Exercise 8: Navigate the Integrated File System</b>	50
<b>Introduction</b>	50
<b>Goals of this exercise</b>	50
<b>Part 1: Create a VIFSDirectory object</b>	51
<b>Part 2: Create and load an AS400ExplorerPane object</b>	52
<b>Run the program</b>	53
<b>Exercise 9: Command Call Button and Message List</b>	56
<b>Introduction</b>	56
<b>Goals of this exercise</b>	56
<b>Part 1: Create a CommandCallButton object</b>	57
<b>Part 2: Create a VMessageList object</b>	58
<b>Part 3: Create an AS400DetailsPane object</b>	58
<b>Part 4: Load a message list</b>	59
<b>Run the program</b>	60

## Exercise 10: Develop using VisualAge for Java

(Optional) .....	62
<b>Introduction</b> .....	62
<b>Goals of this exercise</b> .....	62
<b>Part 1: Start VisualAge for Java</b> .....	63
<b>Part 2: Create a project</b> .....	64
<b>Part 3: Create a JFrame object</b> .....	65
<b>Part 4: Create an AS400 object</b> .....	68
<b>Part 5: Create a RecordListFormPane object</b> .....	70
<b>Part 6: Load the contents of a database file</b> .....	72
<b>Part 7: Pack and show the JFrame object</b> .....	74
<b>Run the program</b> .....	76
<b>Conclusion</b> .....	77
<b>Appendix A: Solutions</b> .....	78
<b>Exercise 1: Command Call</b> .....	78
<b>Exercise 2: Data Queue</b> .....	79
<b>Exercise 3: JDBC</b> .....	80
<b>Exercise 4: Record-level access</b> .....	84
<b>Exercise 5: Program Call</b> .....	86
<b>Exercise 5: Program Call (RPG program)</b> .....	88
<b>Exercise 7: SQL Result Set Table Pane</b> .....	88
<b>Exercise 8: Navigate the Integrated File System</b> .....	90
<b>Exercise 9: Command Call Button and Message List</b> .....	91

## Introduction

This lab provides an overview of AS/400 Toolbox for Java. In this lab, you will build Java applications using the AS/400 Toolbox for Java. You will be presented with several Java examples and asked to create solutions to common data processing tasks with AS/400 Toolbox for Java.

## Overview

### The Java™ Language

Java began as part of a research project to develop advanced portable software for consumer electronics devices. The developers originally intended to use C++ as their development language, but they encountered many problems due to the nature of C++. Over time the developers decided to create a new programming language. From this decision, Java™ was born.

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 AS/400 ILE RPG service programs. They enable you to divide your program pieces into easily reused units. Packages are Java language constructs. They may contain multiple classes.
- **Classes** are similar to AS/400 ILE RPG modules. They enable you to divide your source code into functions (methods in Java, procedures and subroutines in RPG) and variables those functions need. Classes are typically self-contained groupings. They normally contain multiple fields (variables) and methods.
- **Methods** are similar to AS/400 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.

## AS/400 Toolbox for Java

Java programs can access AS/400 data and resources from any platform (including the AS/400) using the AS/400 Toolbox for Java. The AS/400 Toolbox for Java contains the infrastructure to access the following AS/400 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
- *many others!*

The AS/400 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 classes are 100% Pure Java, which means they will run on any JVM that supports JDK 1.1 or later. The classes are shipped in both a zip and jar file and can be accessed from either the AS/400 or on a client.

This lab requires Modification 1 of the AS/400 Toolbox for Java and Swing 1.0.2. See the AS/400 Toolbox for Java home page for more information: <http://www.as400.ibm.com/toolbox>

## The Lab

This lab consists of exercises that illustrate various components of the AS/400 Toolbox for Java. In the exercises, you will create and execute Java applications in a few different environments.

Hardware required to complete this lab:

Server - AS/400 - RISC model OS/400 V3R7+ or IMPI V3R2  
Client - PC - Intel based; 32 Mb Memory; 200 Mb hard drive  
OS/2, Windows 95 or Windows NT

Alternately, the client could also be running the Java Virtual Machine on AS/400, AIX, or any other Java platform.

This lab walks you through the steps needed to access AS/400 resources from a Java program. Each exercise uses different components of the AS/400 Toolbox for Java:

- **Command Call:** The program will call a non-interactive AS/400 command.
- **Data Queue:** The program will create and write records to a data queue on an AS/400.
- **JDBC:** The program will access an AS/400 database using the standard JDBC interface (defined by the java.sql package in JDK 1.1) and SQL (Structured Query Language).
- **Record Level Access:** The program will access the records in an AS/400 physical file.
- **Program Call:** The Java program will call an AS/400 program.
- **SQLResultSetTablePane:** The program will present the results of a database query in a table.
  
- **VIFSDirectory and AS400ExplorerPane:** The program will present an interface for navigating the Integrated File System of an AS/400.
- **CommandCallButton:** The program will display a button that calls an AS/400 command when clicked.
- **VMessageList and AS400DetailsPane:** The program will display the messages returned from an AS/400 command.
- **RecordListFormPane:** The program will display the contents of a database file, one record at a time.

In addition, you will move a program that you wrote on the PC to the AS/400. This illustrates that 100% Pure Java programs really do run anywhere. Finally, you will develop an application using a visual application builder.

Some of the exercises are labeled as *Optional*. This means you are free to skip over those exercises without affecting the following exercises.



## Lab Setup

During this lab you will need a userid and password for the AS/400. You will also need to know the name of the AS/400 system. The userid and password will be **JAVAx** where **xx** is the number 01 to 99 (two digits number). 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 AS/400 **userid** is \_\_\_\_\_

My AS/400 **password** is \_\_\_\_\_

Please note that if a lab uses an AS/400 library, the name of that library will be the same as your userid.

The name of my AS/400 **system** is \_\_\_\_\_.

You are now ready to begin the exercises.

## Exercise 1: Command Call

### Introduction

In this exercise, you will use the AS/400 Toolbox for Java to call an AS/400 command from your Java program.

The `CommandCall` object (part of the AS/400 Toolbox for Java) enables a Java program to call any non-interactive AS/400 command. The list of AS/400 messages that result from the command are available to the Java program when the AS/400 command completes.

In this exercise you will use **AS400**, **CommandCall**, and **AS400Message** classes to complete a Java program. Your program will run an AS/400 non-interactive command and return the results.

Much of the program has been provided for you. You will need to write Java code to connect to the AS/400, 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 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 between the beginning comments of Exercise #1 Part #1 and before the ending comments.

### Setup

1. Edit the CommandCallExample class found in the file CommandCallExample.java. You can use any editor you like. All Windows systems provide an editor called Notepad. You can start Notepad by typing “notepad” at the DOS prompt or using the Start menu, select “Programs”, “Accessories”, “Notepad”. To open a file using Notepad, select the “File”, “Open...” menu.
2. Locate the section for Lab Exercise #1 Part #1.

### Procedure

1. Create an AS400 object called *system* and specify your assigned AS/400 system name. Some prototypes that you may need are listed below. (The complete set of prototypes is provided in the documentation that is shipped in soft copy form with the AS/400 Toolbox for Java.) This AS400 object represents the connection to the AS/400 system.

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

### Prototypes

#### class AS400

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

## Part 2: Create a CommandCall object

### Setup

- Continue editing the CommandCallExample class found in the file CommandCallExample.java.
- Locate the section for Lab Exercise #1 Part #2.

### Procedure

1. Create a CommandCall object called *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)

## Part 3: Run the command

### Setup

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

### Procedure

1. The lab already has code that creates a String called *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 “The command was successful” or “The command failed.”

### Prototypes

#### class CommandCall

- public boolean run(String commandString)

## Part 4: Retrieve AS400Message objects

### Setup

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

### Procedure

1. Retrieve any messages that were generated by running the command. This is 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()

#### class AS400Message

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

## Run the program

Now it is time to run the CommandCallExample program.

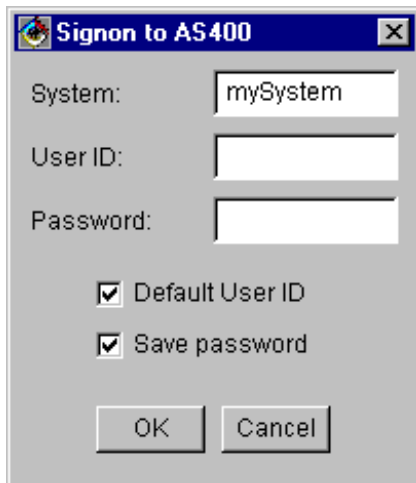
1. Make sure to save the modified CommandCallExample.java file.
2. Compile the program from a DOS prompt.

**javac CommandCallExample.java**

3. Run the program and specify the AS/400 command that you want to run.

**java CommandCallExample CRTLIB FRED**

4. The program will prompt you for a user ID and password. This happens automatically the first time you access the AS/400 using the AS/400 Toolbox for Java. Enter your assigned user ID and password.



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

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

## Exercise 2: Data Queue

### Introduction

In this exercise, you will use the AS/400 Toolbox for Java to enable your program to read and write data to a data queue.

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

- The data queue is a fast means of communication between jobs. Therefore, it is an excellent way to synchronize and pass data between jobs.
- Many jobs can access data queues simultaneously.
- Messages on a data queue do not require field definitions like database files. They are free format.
- The data queue can be used for either synchronous or asynchronous processing.
- Messages on a data queue can be ordered in one of three ways:
  1. Last in, first out (LIFO). The last (newest) message placed on the data queue is the first message taken off the queue.
  2. First in, first out (FIFO). The first (oldest) message placed on the data queue is the first message taken off the queue.
  3. Keyed. Each message on the data queue has a key associated with it. A message can only be taken off the queue by specifying the key that is associated with it.

The DataQueue class provides a complete set of interfaces to access AS/400 data queues from your Java program. It is an excellent way to communicate between Java programs and AS/400 programs. The AS/400 program can be written in any language.

In this exercise you will use **AS400**, **DataQueue**, and **DataQueueEntry** classes to complete a Java program. Your program will create a data queue on the AS/400, write a string to the data queue, and then read an entry from the data queue without removing it from the queue.

### Goals of this exercise

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

1. Connect to the AS/400 Data Queue service.
2. Create a DataQueue object.
3. Create a data queue on the AS400.
4. Write a string to the data queue.
5. Peek an entry from the data queue.

## Part 1: Connect to the AS/400 Data Queue service

Note that all the sections that need to be completed start with the comments:

```
// -----  
//           Lab Exercise #2 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 #2 Part #1** and before the ending comments.

### Setup

- Edit the `DataQueueExample` class found in the file `DataQueueExample.java`.
- Locate the section for Lab Exercise #2 Part #1.

### Procedure

1. Create an AS400 object called *system* and specify your assigned AS/400 system name.
2. Connect to the AS/400 Data Queue service.

**Note:** In the AS/400 Toolbox for Java, access to an AS/400 resource is called a service. A service corresponds to a server job on the AS/400 and is the interface to the data and resources on the AS/400. The AS400 object manages a set of socket connections to the server jobs on the AS/400 and it contains up to one connection per service type. Every connection for each service has its own job on the AS/400. Services provided are:

- JDBC
- Program call / command call
- Integrated file system
- Network print
- Data queue
- Record-level access

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. Java programs that want to control exactly when the connection is made can explicitly connect by calling the `connectService()` method on the AS400 object with the service type specified as a parameter.



## Prototypes

### class AS400

- public static final int DATAQUEUE
- public AS400()
- public AS400(String systemName)
- public void connectService(int service)
- public void setSystemName(String systemName)

## Part 2: Create a DataQueue object

### Setup

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

### Procedure

1. Create a DataQueue object called *dataQ* specifying the AS400 object that was created in Part 1 and the string “/QSYS.LIB/<userid>.LIB/<userid>.DTAQ” to represent the path on the AS/400 where the data queue will be stored. This DataQueue object represents a data queue, although at this point, the data queue has not been created on the AS/400.

## Prototypes

### class DataQueue

- public DataQueue()
- public DataQueue(AS400 system, String path)
- public void setSystem(AS400 system)
- public void setPath(String path)

## Part 3: Create a Data Queue on the AS/400

### Setup

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

### Procedure

1. In Part 2 the data queue object was instantiated to represent a data queue in the specified path on the AS/400 but was not actually created. In Part 3, you need to call the *create(int)* method to create the data queue on the AS/400. The *int* parameter for the *create()* method is the maximum number of bytes per data queue entry. In our example we will use *50* as the data queue entry length.

### Prototypes

#### class **DataQueue**

- public void create(int entryLength)

## Part 4: Write a string to the Data Queue

### Setup

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

### Procedure

1. Write the string “[The AS/400 Toolbox for Java is 100% Pure Java.](#)” to the data queue referenced by *dataQ* on the AS/400.

### Prototypes

#### class **DataQueue**

- public void write(String data)

## Part 5: Peek an entry from the Data Queue

### Setup

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

### Procedure

1. Peek the data queue referenced by *dataQ* on the AS/400. Peeking the data queue will read an entry from the data queue without removing it from the queue.
2. Print the data for the data queue entry as a string to System.out.

### Prototypes

#### class DataQueue

- public DataQueueEntry peek()

#### class DataQueueEntry

- public String getString()

## Run the program

Now it is time to try the DataQueueExample program.

1. Compile the program:

```
javac DataQueueExample.java
```

2. Run the program:

```
java DataQueueExample
```

3. You will see a userid and password prompt. Enter your userid and password at the signon prompt.



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

```
The AS/400 Toolbox for Java is 100% Pure Java!
```

## Exercise 3: JDBC

### Introduction

In this exercise, you will use the AS/400 Toolbox for Java to enable your program to populate and query an AS/400 database file using JDBC.

The AS/400 Toolbox for Java JDBC Driver enables Java programs to access AS/400 database files using standard JDBC interfaces, which enable the Java programmer to issue SQL statements and process results. JDBC is a standard part of Java.

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 generated by running a SQL query or DatabaseMetaData catalog method.
- The ResultSetMetaData interface determines the types and properties of the columns in a ResultSet.
- The DatabaseMetaData interface provides catalog methods which provide information about the database.

In this exercise you will use **AS400** and **AS400JDBCdriver** classes along with the following *java.sql* interfaces and classes to complete a Java program: **Connection**, **DriverManager**, **Statement**, **PreparedStatement**, **ResultSet**, and **ResultSetMetaData**. Your program will create, populate, and query an AS/400 database file. Your task is to fill in the missing JDBC code needed to interact with the AS/400 database and display the resulting customer information.

### Goals of this exercise

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

1. Register a JDBC driver and connect to an AS/400 database.
2. Create a collection.
3. Create a table.
4. Insert records into a database.
5. Query records from a database.
6. Extract information about a result set.
7. Extract column information from a row.

## Part 1: Register a JDBC Driver and Connect to an AS/400 Database

### Setup

- Edit the JDBCExample class found in the file JDBCExample.java.
- Locate the section for Lab Exercise #3 Part #1.

### Procedure

1. Create an AS400 object called *system* and specify your assigned AS/400 system name.
2. Register the JDBC driver using *java.sql.DriverManager*. To register the driver you should use the *DriverManager.registerDriver(Driver)* method specifying a new instance of the Toolbox's JDBC driver, *AS400JDBCdriver*, that is provided in the *AS/400 Toolbox for Java* access package.
3. Connect to the database with the *DriverManager.getConnection(String)* method specifying the URL for the database. The URL has the format "jdbc:as400://mySystem". The as400 in the URL indicates that **we would like** to use the driver that we just registered. You can use *system* to concatenate the system name to the end of the URL. Assign the method's resulting Connection object to a variable called *connection*.

### Prototypes

#### class AS400JDBCdriver

- public AS400JDBCdriver ()

#### class java.sql.DriverManager

- public static void registerDriver (Driver driver)
- public static Connection getConnection (String url)

## Part 2: Create a Collection

### Setup

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

### Procedure

1. Create a Statement object, *createCollection*, using the *Connection.createStatement( )* method from the previously created connection object, *connection*.
2. Execute the SQL statement: "CREATE COLLECTION" to create a collection on the AS/400 specifying the collection name, *collectionName\_*, that will be input later as the second parameter on the command line. Use the *Statement.executeUpdate(String)* method to create the collection. Catch and ignore any *java.sql.SQLException* so it does not fail if the collection already exists.

### Prototypes

#### class **java.sql.Connection**

- public Statement createStatement ( )

#### class **java.sql.Statement**

- public int executeUpdate (String sql)

## Part 3: Create a Table

### Setup

- Use your editor to locate the section for Lab Exercise #3 Part #3.

### Procedure

1. Create a table containing four fields: *custnum*, *name*, *limit*, and *balance*. This can be done using the SQL statement: “CREATE TABLE” specifying the collection name, *collectionName\_*, and table name, *tableName\_*, from the command line parameters.

The four field types are as follows:

- *custnum* is an integer
- *name* is a varchar(20)
- *limit* is a double
- *balance* is a double

Again, you can use the *Connection.createStatement( )* and *Statement.executeUpdate(String)* methods to complete this task.

### Prototypes

#### class **java.sql.Connection**

- public Statement createStatement ( )

#### class **java.sql.Statement**

- public int executeUpdate (String sql)



## Part 4: Insert Records into a Database

### Setup

- Use your editor to locate the section for Lab Exercise #3 Part #4.

### Procedure

1. Prepare a statement for inserting rows into the table. Since this statement will be used multiple times, we will use a *PreparedStatement* object and parameter markers. With a prepared statement, the statement is only compiled once making it more efficient when having to call it multiple times. Create the prepared statement object, *insertStatement*, using the *Connection.prepareStatement( )* method. To create the prepared statement you will use the SQL statement: “INSERT INTO”, specifying the collection name, *collectionName\_*, and table name, *tableName\_*, the four fields: *custnum*, *name*, *limit*, and *balance*, with values of “?” (undefined) for each.
2. Set the field values with example customer information. Use the methods *PreparedStatement.setString*, *PreparedStatement.setInt*, and *PreparedStatement.setDouble* to accomplish this task.
3. Execute the statement to insert the rows using *PreparedStatement.executeUpdate( )*.
4. Repeat steps 3 and 4 a few more times to insert multiple records into the table.

### Prototypes

#### class `java.sql.Connection`

- `public PreparedStatement prepareStatement (String sql)`

#### class `java.sql.PreparedStatement`

- `public int executeUpdate ( )`  
**Note:** *PreparedStatement.executeUpdate* will execute a SQL INSERT, UPDATE or DELETE statement.
- `public void setDouble (int parameterIndex, double x )`
- `public void setInt (int parameterIndex, int x )`
- `public void setString (int parameterIndex, String x )`

## Part 5: Query Records from a Database

### Setup

- Use your editor to locate the section for Lab Exercise #3 Part #5.

### Procedure

1. Execute a query to retrieve the rows from the table. This can be done by creating and executing a SQL “SELECT” statement specifying the collection and table name and placing the results in a *ResultSet* object. The *Statement.executeQuery(String)* method can be used to execute the query.

### Prototypes

#### class `java.sql.Connection`

- `public Statement createStatement ()`

#### class `java.sql.Statement`

- `public ResultSet executeQuery ()`

## Part 6: Extract Information about a Result Set

### Setup

- Use your editor to locate the section for Lab Exercise #3 Part #6.

### Procedure

1. Extract the number of columns in a row from the result set returned in Exercise #3 Part #5. The number of columns in a row, *columnCount*, can be obtained from the *ResultSet.getMetaData ( )* method which returns a *ResultSetMetaData* object. The *ResultSetMetaData.getColumnCount( )* method can then be used to get the number of columns in a row.

### Prototypes

#### class `java.sql.ResultSet`

- `public ResultSetMetaData getMetaData ( )`

#### class `java.sql.ResultSetMetaData`

- `public int getColumnCount ( )`

## Part 7: Extract Column Information from a Row

### Setup

- Use your editor to locate the section for Lab Exercise #3 Part #7.

### Procedure

1. While there is data in the result set, extract the next row in the table by calling the *ResultSet.next( )* method.

If the *ResultSet.next( )* method returns true, meaning a row was returned, all of the column's string values can be processed inside a for loop with the *ResultSet.getString(column#)* method.

Before we can print out this field information to *System.out* in a readable way it would be useful to know the display size of the column. This can also be obtained from the *ResultSetMetaData* object in Part #6 of this exercise.

Print out the field information to *System.out* by calling *printColumn*, a custom made method in *JDBCExample.java* which takes two parameters: *String* for the field data and an *int* for the size of the display column. This method takes the field string and pads the remaining display column with spaces to format the output in a readable way. Finally, outside of the field processing loop call *System.out.println ( )* to advance the pointer so the next row can be processed.

**Note:** The for loop should be indexed from one to the number of columns in a row, *columnCount* which was obtained in Part #6 of this exercise.

### Prototypes

#### class *java.sql.ResultSet*

- *public boolean next ( )*  
A *ResultSet* is initially positioned before its first row; the first call to *next* makes the first row the current row; the second call makes the second row the current row, etc.
- *public String getString (String columnName)*  
Get the value of a column in the current row as a Java *String*.

#### class *java.sql.ResultSetMetaData*

- *public int getColumnDisplaySize (int column )*

#### class *JDBCExample*

- *private static void printColumn (String data, int columnSize)*  
Prints out the column to *System.out* padding the remaining column display length with spaces.

## Run the program

1. Compile the program:

```
javac JDBCExample.java
```

2. Run the program.

```
java JDBCExample myCollection myTable
```

3. Wait for the signon dialog to appear.



Enter your userid and password at the signon prompt. Click the **OK** button.

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

CUSTNUM	NAME	LIMIT	BALANCE
500001	Mickey Mouse	150000.0	5897.95
500002	Donald Duck	80000.0	63000.25
500008	Goofy	11500.0	905.72

## Exercise 4: Record-level Access

### Introduction

In this exercise, you will use the AS/400 Toolbox for Java to enable your program to access an AS/400 physical file using Record Level Access.

In this exercise you will use **AS400**, **AS400Text**, **AS400ZonedDecimal**, **CharacterFieldDescription**, **QSYSObjectPathName**, **SequentialFile**, **Record**, **RecordFormat**, and **ZonedDecimalDescription** classes to complete a Java program. Your program will create a record format for the database file *QIWS/QCUSTCDT*, create, open, and read from a sequential file that represents the database file. Lastly, the program will disconnect the record-level access resources it used.

The database file *QIWS/QCUSTCDT* is a sample customer database that contains the following customer fields.

- Customer information: **number**, name (**last name** and **initials**), and address (**street**, **city**, **state**, **zip code**)
- Customer account information: **credit limit**, **charge code**, **balance** and **credit due**.

### Goals of this exercise

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

1. Create a record format
2. Create a sequential file.
3. Set the record format and open a file.
4. Read a record from a file.
5. Disconnect the record-level access service.

## Part 1: Create a Record Format

### Setup

- Edit the RLAExample class found in the file RLAExample.java.
- Locate the section for Lab Exercise #4 Part #1.

### Procedure

1. Create field description objects that will be used in describing a record in the database file. A *field description* allows 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. The fields we will use in this application are as follows:

Field Name	Field Type	Toolbox Class
CUSNUM	6-digit zoned decimal (0)	AS400ZonedDecimal
LSTNAM	8-character string	AS400Text
INIT	3-character string	AS400Text
STREET	13-character string	AS400Text
CITY	6-character string	AS400Text
STATE	2-character string	AS400Text
ZIPCOD	5-digit zoned decimal (0)	AS400ZonedDecimal
CDTLMT	4-digit zoned decimal (0)	AS400ZonedDecimal
CHGCOD	1-digit zoned decimal (0)	AS400ZonedDecimal
BALDUE	6-digit zoned decimal (2)	AS400ZonedDecimal
CDTDUE	6-digit zoned decimal (2)	AS400ZonedDecimal

Use the *ZonedDecimalFieldDescription* class for the zoned decimal fields and the *CharacterFieldDescription* class for the string fields. **Note:** For zoned decimal field types the number in parenthesis is the number of decimal positions in the zoned decimal number.

2. Create a record format object called *qcustcdt* to describe a record in the database file. Set the name of the record format object to “CUSREC”.  
A 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. A *record format* object contains a set of field descriptions. The field description can be accessed by index or by name.
3. Add the field descriptions to the record format object created in the previous step.

## Prototypes

### **class AS400Text**

- public AS400Text(int length)

### **class AS400ZonedDecimal**

- public AS400ZonedDecimal(int numDigits, int numDecimalPositions)

### **class CharacterFieldDescription**

- public CharacterFieldDescription(AS400Text dataType, String name)

### **class RecordFormat**

- public RecordFormat()
- public RecordFormat(String name)
- public void addFieldDescription(FieldDescription field)
- public void setName(String name)

### **class ZonedDecimalFieldDescription**

- public ZonedDecimalFieldDescription(AS400ZonedDecimal dataType, String name)



## Part 2: Create a Sequential File object

### Setup

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

### Procedure

1. Create an AS400 object called *system* and specify your assigned AS/400 system name.
2. Create a *QSYSObjectPathName* object to represent the sequential file in the integrated file system. The *QSYSObjectPathName* class is used to build the path to an object while letting you describe the object's location components such as the library name, object name, and object type.
3. Create the *SequentialFile* object specifying the AS/400 system and path name created in the previous steps.

### Prototypes

#### class AS400

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

#### class QSYSObjectPathName

- public QSYSObjectPathName()
- public QSYSObjectPathName(String library, String object, String type)
- public String getPath()
- public void setLibraryName(String library)
- public void setObjectName(String object)
- public void setObjectType(String type)

#### class SequentialFile

- public SequentialFile()
- public SequentialFile(AS400 system, String name)
- public void setPath(String name)
- public void setSystem(AS400 system)

## Part 3: Set the Record Format and open a file

### Setup

- Locate the section for Lab Exercise #4 Part #3.

### Procedure

1. Set the record format created in Part #1 for the sequential file created in Part #2.
2. Open the file as *read only*, with a blocking factor of *10*, and specify *no* commitment control record locking.

### Prototypes

#### class `SequentialFile`

- `public static final int COMMIT_LOCK_LEVEL_NONE`
- `public static final int READ_ONLY`
- `public void open(int openType, int blockingFactor, int commitLockLevel)`
- `public void setRecordFormat(RecordFormat recordFormat)`
- `public void setSystemName(String systemName)`

## Part 4: Read a Record from a File

### Setup

- Locate the section for Lab Exercise #4 Part #4.

### Procedure

1. Read the first record in the file.
2. While the record is not empty, get the *balance due* field information and if the balance due is greater than *zero*, display the customer's name (*last name* and *initials*) along with the *balance due*. Continue processing the remaining records in the file.

### Prototypes

#### class `SequentialFile`

- `public Record readNext()`

#### class `Record`

- `public Object getField(int index)`

#### class `java.math.BigDecimal`

- `public float floatValue()`

## Part 5: Disconnect the Record-level access Service

### Setup

- Locate the section for Lab Exercise #4 Part #5.

### Procedure

1. Disconnect from the record-level access service to close the socket connection and free up system resources.

### Prototypes

#### class AS400

- public static final int RECORDACCESS
- public void disconnectService(int service)

## Run the program

1. Compile the program:

```
javac RLAExample.java
```

2. Run the program.

```
java RLAExample
```

3. Wait for the signon dialog to appear.



Enter your userid and password at the signon prompt. Click the **OK** button.

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

G K	Henning	37.00
B D	Jones	100.00
S S	Vine	439.00
J A	Johnson	3987.50
K L	Stevens	58.75
J S	Alison	10.00
J W	Doe	250.00
E D	Williams	25.00
F L	Lee	489.50
M T	Abraham	500.00

## Exercise 5: Program Call

### Introduction

In this exercise, you will use the AS/400 Toolbox for Java to enable your program to run a program on the AS/400 system.

The ProgramCall object (part of the AS/400 Toolbox for Java) enables a Java program to call any AS/400 program passing it input, output, and input/output parameters. In this exercise you will use **AS400**, **AS400Message**, **AS400Text**, **MessageQueue**, **ProgramCall**, **ProgramParameter**, and **QueuedMessage** classes to complete a Java program. Your program will run an AS/400 RPG program and display the results from the user's message queue.

The AS/400 RPG program reads an entry from a sequential data queue that has a maximum data entry length of 50 and displays the results of the read in a message queue. The program has two input parameters: *library* and *name* allowing the caller to specify the library and name of an existing data queue. **Note:** The library name is used for the name of the message queue.

You do not have to create the RPG program, it is already loaded on the AS/400. The source code is listed in Appendix A for reference.

### Goals of this exercise

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

1. Create a ProgramCall object.
2. Create a ProgramParameter list.
3. Run the AS/400 program.
4. Retrieve the Messages from a MessageQueue object.

## Part 1: Create a ProgramCall object

### Setup

- Edit the ProgramCallExample class found in the file ProgramCallExample.java.
- Locate the section for Lab Exercise #5 Part #1.

### Procedure

1. Create an AS400 object called *system* and specify your assigned AS/400 system name.
2. Create a ProgramCall object called *pgm* and specify *system* as the AS400 object parameter.

### Prototypes

#### class AS400

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

#### class ProgramCall

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

## Part 2: Create a ProgramParameter list

### Setup

- Get the library and data queue name of the data queue that was created in Lab Exercise #2 Part #2.  
    **Name of the library is** \_\_\_\_\_  
  
    **Name of the data queue is** \_\_\_\_\_
- Locate the section for Lab Exercise #5 Part #2.

### Procedure

1. Create a ProgramParameter array, called *parmList*, that contains two ProgramParameter objects.
2. Create a AS400Text object called *libraryText* that is 10 bytes in length. The AS400Text object is used to convert character data between Java unicode and an EBCDIC code page and character set (CCSID).
3. Convert the library name of the existing data queue to a byte array named, *libraryName*. Use the library name of the existing data queue from Exercise #2.
4. Assign index *zero* of the parameter array to a new instantiation of ProgramParameter with *libraryName* as the input data.
5. Repeat steps 2-4 for the name of the existing data queue.
6. Set the program's parameter list to the ProgramParameter array created in step one.

### Prototypes

#### class ProgramParameter

- public ProgramParameter()
- public ProgramParameter(byte[] data)
- public void setInputData(byte[] data)

#### class AS400Text

- public AS400Text(int length)
- public byte[] toBytes(Object javaValue)

#### class ProgramCall

- public void setParameterList(ProgramParameter[])

## Part 3: Run the AS/400 program

### Setup

- Locate the section for Lab Exercise #5 Part #3.

### Procedure

1. Run the program. Notice that the run() method returns a boolean, which indicates whether or not the program was successful. If the program failed, retrieve any messages that were generated by running the program. This is stored as an array of AS400Message objects. Each AS400Message object in the array represents a message that was generated by the program.
2. Loop through the array of messages, and print each message's ID and text to System.out.

### Prototypes

#### class ProgramCall

- public AS400Message[] getMessageList()
- public boolean run()

#### class AS400Message

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

## Part 4: Retrieve the Messages from the Message Queue object

### Setup

- Locate the section for Lab Exercise #5 Part #4.

### Procedure

1. Create a MessageQueue object called *msgQueue* specifying the AS400 system object and current user's message queue.
2. Return the list of messages in the message queue placing the list in a Java.util.Enumeration object called *list*.
3. While *list* is not empty loop return the next element in the enumeration casting it to a QueuedMessage object and print each message's text to System.out.
4. Remove all the messages from the message queue on the AS/400. **Note:** The MessageQueue getMessages method does not remove the message from the message queue.

### Prototypes



**class MessageQueue**

- public static final String CURRENT
- public MessageQueue(AS400 system, String path)
- public Enumeration getMessages()
- public void remove()

**class QueueMessage**

- public String getText()

**class Enumeration**

- public boolean hasMoreElements()
- public Object nextElement()

## Run the program

Now it is time to try the ProgramCallExample program.

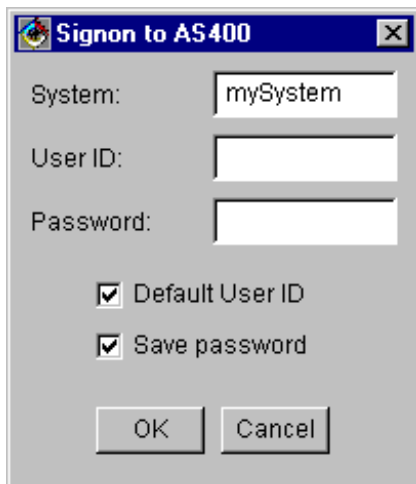
1. Compile the program:

```
javac ProgramCallExample.java
```

2. Run the program:

```
java ProgramCallExample
```

3. You will see a userid and password prompt. Enter your userid and password at the signon prompt.



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

```
The AS/400 Toolbox for Java is 100% Pure Java!
```

## Exercise 6: Java on the AS/400 (Optional)

### Introduction

In this exercise, you will use the AS/400 Toolbox for Java to run your program on the AS/400 system.

This exercise will use the AS/400 for both the development and runtime environment for the Java program. You will create a Java program using the *Source Entry Utility* (SEU) tool, compile it inside the *QSHHELL* environment, and then run the program from the command line.

In this exercise, the Java program you will be creating is almost identical to the program created in **Lab Exercise #2**. The **AS400**, **DataQueue**, and **DataQueueEntry** classes will be used to complete the Java program. Your program will create a data queue on the AS/400, write a string to the data queue, and then read an entry from the data queue and display the entry to System.out. For more information on data queues, please refer back to the *Introduction* section of **Lab Exercise #2**.

### Goals of this exercise

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

1. Create Java source code on the AS/400.
2. Compile the source code on the AS/400.
3. Run the program on the AS/400.

## Part 1: Create Java Source Code on the AS/400

### Setup

- Log onto the AS/400 using the userid and password you were given.
- Create a directory that is located in the integrated file system to place your java source code as well as your compiled Java programs. Create your directory with the same name as your userid.  
**MKDIR ('/<directory>')**
- Create an AS/400 library in which to place the source physical file. Again, use the same name as your userid.  
**CRTLIB <library>**
- Add the library to your library list.  
**ADDLIBLE <library>**

### Procedure

1. Create a Java source file using the AS/400 SEU tool. However, SEU stores the Java source code inside of a source member instead of the integrated file system. When you use the java compiler to compile your Java source, it needs to be in the integrated file system. You will

make use of the CPYTOSTMF command which copies the Java source from your source physical file into a file in the integrated file system.

- Issue **CRTSRCPF** and press **F4** to prompt on the command.
- Fill in the following information:

File: **jsource**  
Library: **<library>**  
Text 'description': **'java source'**

Press **enter**.

- Now issue **STRPDM**
- Take **option 3** to Work with members.
- Fill in the following information:

File: **jsource**  
Library: **<library>**

Press **enter**.

- Now we need to create the java source code with SEU. Press the **F6** key to Start Source Entry Utility.

- Fill in the following information:

Source member: **DQExample**  
Source type: **TXT**  
Text 'description': **'Toolbox Data Queue Example'**

Press **enter**.

- Type in the following source code. **Please note the punctuation as well as the capitalization.** Both are very important elements to Java's syntax. Once you have entered in the source into SEU, it's time to save the source code. Hit the **F3** key. Make sure **'N'** is set on *Return to editing*. Press **Enter**.

```
import com.ibm.as400.access.AS400;  
import com.ibm.as400.access.DataQueue;  
import com.ibm.as400.access.DataQueueEntry;  
  
public class DQExample extends Object  
{  
    public static void main(String[] args)  
    {  
        try  
        {  
            // Note: When running on the AS/400 the system, userid, and  
            // password does not need to be supplied. The userid and  
            // password of the job that started the program is used.  
            AS400 system = new AS400();  
  
            // Create a data queue object.  
            DataQueue dtaq = new DataQueue(system,  
                "/QSYS.LIB/myLibrary.LIB/myDQ.DTAQ");  
  
            // Create sequential data queue on the AS/400.  
            dtaq.create(100);  
  
            // Write an entry to the data queue.  
            dtaq.write("A Toolbox program can run on the AS/400 JVM.");  
        }  
        catch (Exception e)  
        {  
            e.printStackTrace();  
        }  
    }  
}
```

```
        // Read an entry from the data queue.
        DataQueueEntry dqEntry = dtaq.read();

        // Print the entry to System.out.
        System.out.println(dqEntry.getString());

    }
    catch (Exception e) {
        System.out.println ("Error: " + e);
    }
    System.exit (0);
}
}
```

## Prototypes

### class AS400

- public AS400()

### class DataQueue

- public DataQueue(AS400 system, String path)
- public void create(int entryLength)
- public DataQueueEntry read()
- public void write(String data)

### class DataQueueEntry

- public String getString()

## Part 2: Compile the Source Code on the AS/400

### Setup

- Now it is time to copy the source file from your library into the integrated file system. This is accomplished with the CPYTOSTMF command. Type in **CPYTOSTMF** and hit **F4** to prompt on it.
- Fill in the following information:

From database file member: **/qsys.lib/<library>.lib/jsource.file/DQExample.mbr**

To stream file: **/<directory>/DQExample.java**

Stream file code page: **819**

**Note:** Code page 819 is the code page used to convert the file from ASCII to EBCDIC.

Hit **enter twice** and the stream file *DQExample.java* will be created in the integrated file system. You are now ready to compile your Java program on the AS/400.

- Exit out from Work with Members Using PDM with **F3**.
- Exit out from the AS/400 Program Development Manager (PDM) with **F3**.

**Procedure**

1. Create an environment variable called CLASSPATH. This environment variable is the list of directories in the integrated file system that Java searches for class file used to compile/run.

**ADDENVVAR ENVVAR(CLASSPATH)  
VALUE('/<directory>:/QIBM/ProdData/HTTP/Public/jt400/lib/jt400.jar')**

**Note:** CLASSPATH is a case sensitive parameter, specify this in **UPPER** case, not lower. The AS/400 Toolbox for Java license program (**5763JC1**) is installed to the integrated file system in: **QIBM/ProdData/HTTP/Public/jt400/lib/jt400.jar**. The *jt400.jar* file contains all the Toolbox class files that are needed for a Java program to access the resources on the AS/400.

2. Compile the Java program on the AS/400.
  - Enter the QSH shell environment. **STRQSH**
  - Change into your directory. **cd <directory>**
  - Compile using the javac command. **javac DQExample.java**

If you are successful and your source compiles with no errors, a file called *DQExample.class* will be created inside of your integrated file system directory.

**Note:** If there any errors in your source, they will be printed to the screen. A dollar symbol, \$, signifies the end of the compile. You will need to go back and edit your source file with SEU to fix any errors. After you have saved your changes, be sure to recopy the source file to the integrated file system with the **CPYTOSTMF** command and specify **\*REPLACE** on the *Stream file option* parameter.

- Exit the QSH shell environment with **F3**.
- Verify that the class file has been created in the integrated file system.

**WRKLNK <directory>  
option 5**

Press **F3** to exit from WRKLNK.

**Part 3: Run the program on the AS/400****Procedure**

1. Run the program.

**java DQExample**
2. Verify that the program output looks similar to this:

**A Toolbox program can run on the AS/400 JVM.  
Java program completed**
3. Press **F3** to exit the Java Shell environment.

## Exercise 7: SQL Result Set Table Pane

### Introduction

In this exercise, you will use the AS/400 Toolbox for Java to present the results of a database query in a table.

The `SQLResultSetTablePane` object (part of the AS/400 Toolbox for Java) enables a Java program to present the results of a database query in a table. The table is a Java Swing component and can be imbedded inside any graphical user interface.

In this exercise you will use the `SQLConnection`, `SQLResultSetTablePane`, and `ErrorDialogAdapter` classes to complete a Java program. Your program will present the results of queries in a table. The queries are entered by the user.

The AWT/Swing part of the program has been provided for you. You will need to write Java code to connect to the AS/400 database, create the table pane object, and run the queries.

### Goals of this exercise

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

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

### Part 1: Create an `SQLConnection` object

#### Setup

- Edit the `SQLResultSetTablePaneExample` class found in the file `SQLResultSetTablePaneExample.java`.
- Locate the section for Lab Exercise #7 Part #1.

#### Procedure

1. Create an `SQLConnection` object, *connection*, that uses the JDBC URL “`jdbc:as400://systemName`”, where *systemName* is your assigned AS/400 system name. This `SQLConnection` object represents the JDBC connection to the AS/400 database.

#### Prototypes

##### class `SQLConnection`

- `public SQLConnection (String URL)`

## Part 2: Create an `SQLResultSetTablePane` object

### Setup

Locate the section for Lab Exercise #7 Part #2.

### Procedure

1. Create an `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)`

## Part 3: Run a query and load the results

### Setup

Locate the section for Lab Exercise #7 Part #3. **Note:** Part #3 is located in the `keyPressed(KeyEvent)` method.

### Procedure

1. The query text that the user types is stored in a `String` called *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()`, then the table will still appear, but it will be empty.

### Prototypes

#### class `SQLResultSetTablePane`

- `public void setQuery (String query)`
- `public void load ()`



## Part 4: Setup an error handler

### Setup

Locate the section for Lab Exercise #7 Part #4.

### Procedure

1. Any errors that occur when accessing the AS/400 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`, then this error handler will display a message box.

### Prototypes

#### class `ErrorDialogAdapter`

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

#### class `SQLResultSetTablePane`

- `public void addErrorListener (ErrorListener listener)`

## Run the program

Now it is time to run the `SQLResultSetTablePaneExample` program.

1. Compile the program from a DOS prompt.

```
javac SQLResultSetTablePaneExample.java
```

2. Run the program.

```
java SQLResultSetTablePaneExample
```

3. The program will display 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. A good one to try is:

**SELECT \* FROM QIWS.QCUSTCDT**

5. The program 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 AS/400 database 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 8: Navigate the Integrated File System

### Introduction

In this exercise, you will use the AS/400 Toolbox for Java to write an application which allows the user to navigate the integrated file system on the AS/400 using a familiar explorer-style interface.

A `VIFSDirectory` object (part of the AS/400 Toolbox for Java) presents a hierarchy of directories and files in the AS/400 integrated file system as part of a graphical user interface. A `AS400ExplorerPane` object will present the hierarchy in an explorer, which combines a tree with a detailed view of the files. `AS400ExplorerPane` objects are Java Swing components and can be imbedded inside any graphical user interface.

In this exercise you will use the `VIFSDirectory` and `AS400ExplorerPane` classes to complete a Java program.

The AWT/Swing part of the program has been provided for you. You will need to write Java code to create the AS/400 Toolbox for Java objects and load the message list after the AS/400 command has been run.

### Goals of this exercise

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

1. Create a `VIFSDirectory` object.
2. Create and load an `AS400ExplorerPane` object.

## Part 1: Create a VIFSDirectory object

### Setup

- Edit the VIFSDirectoryExample class found in the file VIFSDirectoryExample.java.
- Locate the section for Lab Exercise #8 Part #1.

### Procedure

1. Create an AS400 object, called *system*, which specifies the name of the AS/400 system which is assigned to you for this lab. This AS400 object represents the physical connection to the AS/400 system.
2. Create a VIFSDirectory object, called *root*, with the system specified as *system*, and the path specified as “/QIBM/ProdData”. This object represents the root directory that we want to present in the explorer. Note that is not a good idea to use “/” or “/QSYS.LIB” as the root, since both will result in a long download.

### Prototypes

#### class AS400

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

#### class VIFSDirectory

- public VIFSDirectory ()
- public VIFSDirectory (AS400 system, String path)
- public void setSystem (AS400 system)
- public void setPath (String path)

## Part 2: Create and load an AS400ExplorerPane object

### Setup

Locate the section for Lab Exercise #8 Part #2.

### Procedure

1. Create an AS400ExplorerPane object called *explorerPane*, setting its root to *root*. This is a graphical user interface component which we will use to display the intergrated file system hierarchy and various details about each directory and file. The “root” describes where we being the hierarchy.
2. Use load() to load the messages into the AS400ExplorerPane (*explorerPane*). By calling load(), the *explorerPane* object will actually load the contents of the root directory. If you forget to call load(), then the explorer will still appear, but it may be empty.

### Prototypes

#### class AS400ExplorerPane

- public AS400ExplorerPane ()
- public AS400ExplorerPane (VNode root)
- public void setRoot (VNode root)
- public void load ()

## Run the program

Now it is time to run the VIFSDirectoryExample program.

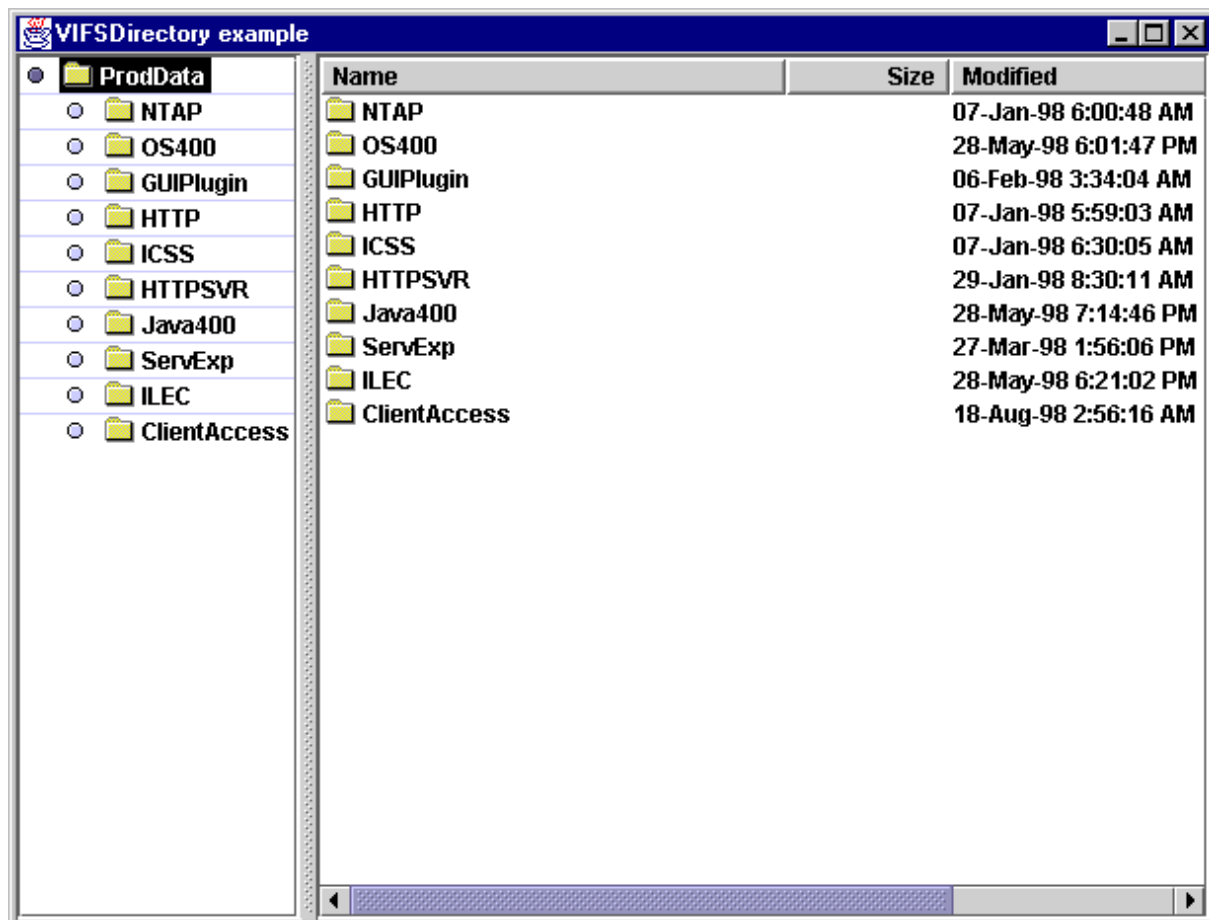
1. Compile the program from a DOS prompt.

```
javac VIFSDirectoryExample.java
```

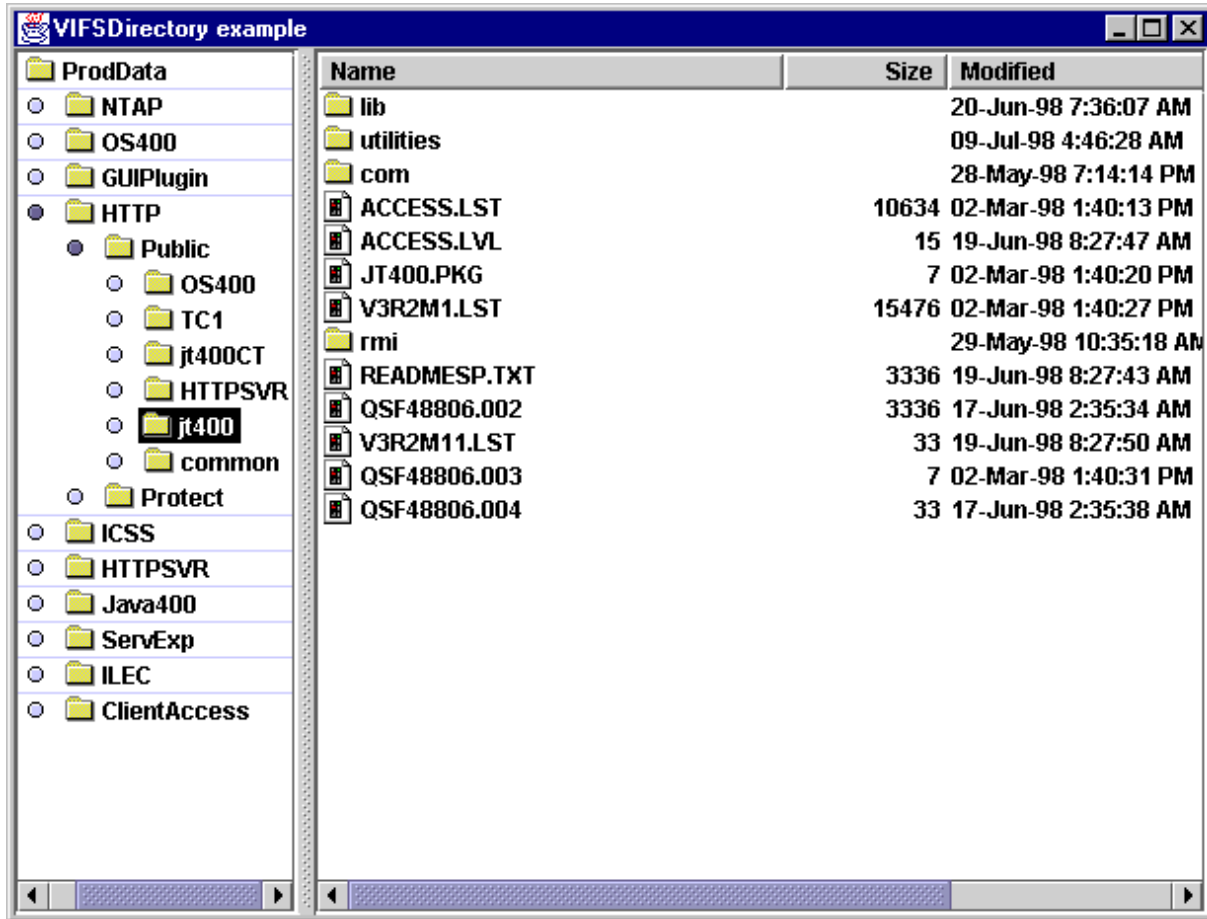
2. Run the program.

```
java VIFSDirectoryExample
```

3. The program will prompt you for a user ID and password. Enter your assigned user ID and password.
4. The program will display the graphical user interface below. It includes a tree view of the AS/400's integrated file system on the left and a details view of the selected directory on the right.

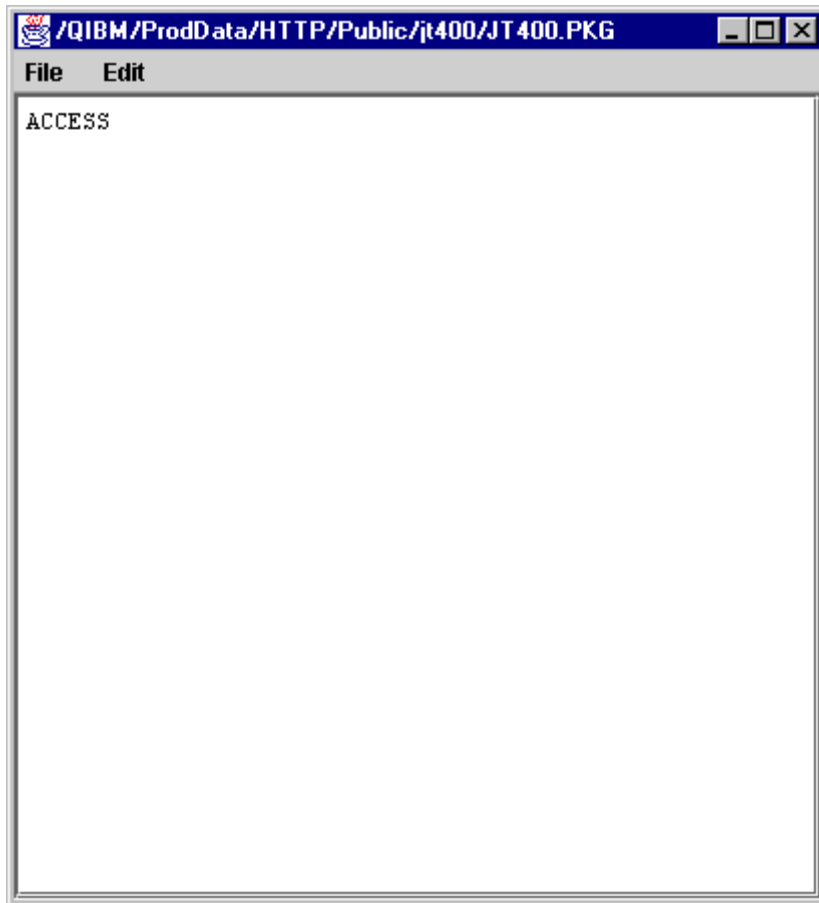


5. Navigate down the tree to the ProdData/HTTP/Public/jt400 directory. Verify that the graphical user interface looks similar to this:



- Right click on JT400.PKG. You will see a popup menu of actions that can be performed on this file. Choose "View". This will bring up a view window which displays the contents of the file JT400.PKG. *There are also other actions you can perform, like creating, renaming, deleting, and editing. If you want to try these during the lab, please make sure not to delete or change any existing files.*





7. Close the view window using the “X” in the upper right corner.
8. Close the explorer window using the “X” in the upper right corner.

## Exercise 9: Command Call Button and Message List

### Introduction

In this exercise, you will use the AS/400 Toolbox for Java to write an application with a button that calls an AS/400 command when it is clicked. In addition you will present the list of messages returned from the AS/400 command.

The `CommandCallButton` object (part of the AS/400 Toolbox for Java) enables a Java program to provide a button which calls an AS/400 when it is clicked. The button is a Java Swing button and can be used like any other button.

The `VMessageList` object (part of the AS/400 Toolbox for Java) presents a list of AS/400 messages, such as the messages returned by an AS/400 command. The `VMessageList` object can be presented as part of various other components, such as an `AS400DetailsPane` object. The `AS400DetailsPane` object will present the list of the messages in a table. The table is a Java Swing component and can be imbedded inside any graphical user interface.

In this exercise you will use the **`CommandCallButton`**, **`VMessageList`**, and **`AS400DetailsPane`** classes to complete a Java program.

The AWT/Swing part of the program has been provided for you. You will need to write Java code to create the AS/400 Toolbox for Java objects and load the message list after the AS/400 command has been run.

### Goals of this exercise

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

1. Create a `CommandCallButton` object.
2. Create a `VMessageList` object.
3. Create an `AS400DetailsPane` object.
4. Load a message list.

## Part 1: Create a CommandCallButton object

### Setup

- Edit the CommandCallButtonExample class found in the file CommandCallButtonExample.java.
- Locate the section for Lab Exercise #9 Part #1.

### Procedure

1. Create an AS400 object, called *system*, which specifies the name of the AS/400 system which is assigned to you for this lab. This AS400 object represents the physical connection to the AS/400 system.
2. Create a CommandCallButton object, called *button*, with the text “Verify TCP Connection”. This object is a button which will run an AS/400 command when clicked. The text appears on the button.
3. Use `setSystem()` to set the system for the button to *system*. This tells the button which AS400 object to use for calling commands.
4. Use `setCommand()` to set the command that the button will run. Set the command to “PING *systemName*”, where *systemName* is the AS/400 system assigned to you by the lab instructor. This command will check the TCP/IP connection and return several status messages.

### Prototypes

#### class AS400

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

#### class CommandCallButton

- `public CommandCallButton (String text)`
- `public void setSystem (AS400 system)`
- `public void setCommand (String command)`

## Part 2: Create a VMessageList object

### Setup

Locate the section for Lab Exercise #9 Part #2.

### Procedure

1. Create a VMessageList object called *messageList*. This represents a list of messages to be presented in a graphical user interface component.

### Prototypes

#### class VMessageList

- public VMessageList ()

## Part 3: Create an AS400DetailsPane object

### Setup

Locate the section for Lab Exercise #9 Part #3.

### Procedure

1. Create an AS400DetailsPane object called *detailsPane*, setting its root to *messageList*. This is a graphical user interface component which we will use to display a list of AS/400 messages and various details about each message. The “root” describes which list will be displayed.

### Prototypes

#### class AS400DetailsPane

- public AS400DetailsPane ()
- public AS400DetailsPane (VNode root)
- public void setRoot (VNode root)

## Part 4: Load a message list

### Setup

Locate the section for Lab Exercise #9 Part #4.

### Procedure

1. This code is called whenever the AS/400 command is called (as a result of the user clicking on the button). Use `setMessageList()` to set *messageList*'s message list to equal the list of messages returned by the command (accessible using *button*'s `getMessageList()`).
2. Use `load()` to load the messages into the `AS400DetailsPane` (*detailsPane*). By calling `load()`, the *detailsPane* object will actually change the GUI to reflect the new list of messages. If you forget to call `load()`, then the table will still appear, but it will be empty, or it will never change.

### Prototypes

#### class `VMessageList`

- `public void setMessageList (AS400Message[] messageList)`

#### class `CommandCallButton`

- `public AS400Message[] getMessageList ()`

#### class `AS400DetailsPane`

- `public void load ()`

## Run the program

Now it is time to run the CommandCallButtonExample program.

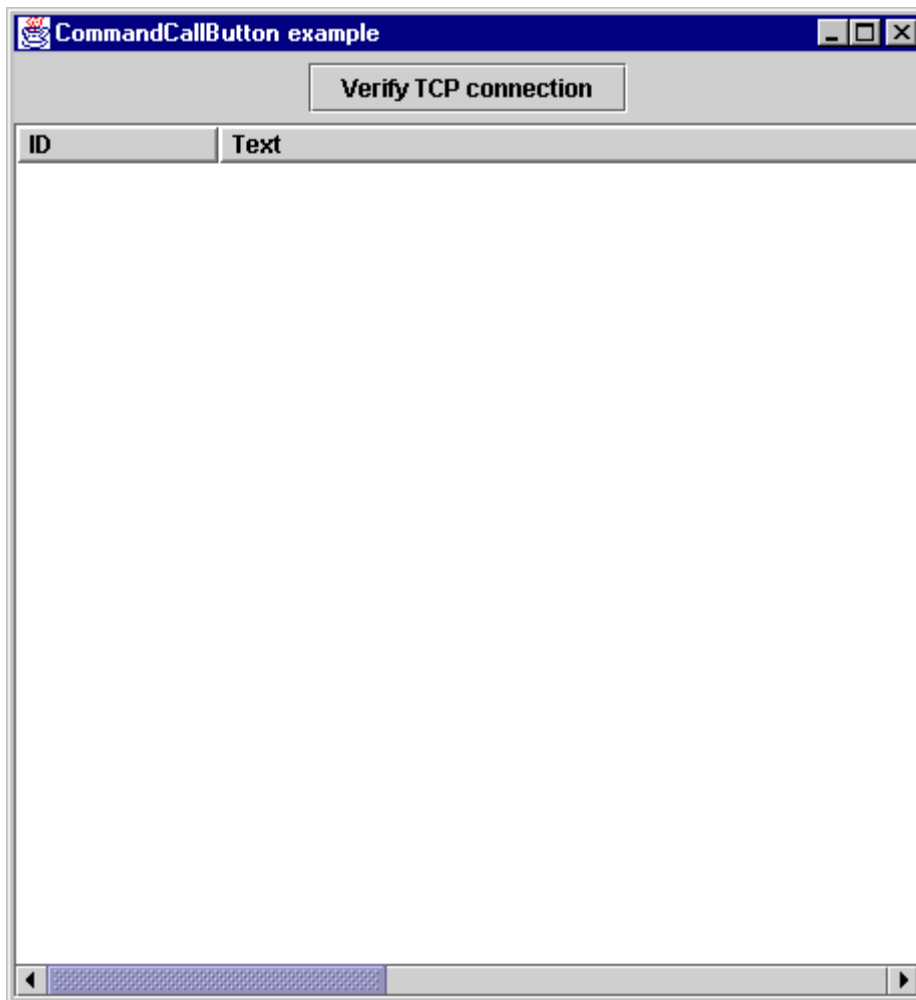
1. Compile the program from a DOS prompt.

```
javac CommandCallButtonExample.java
```

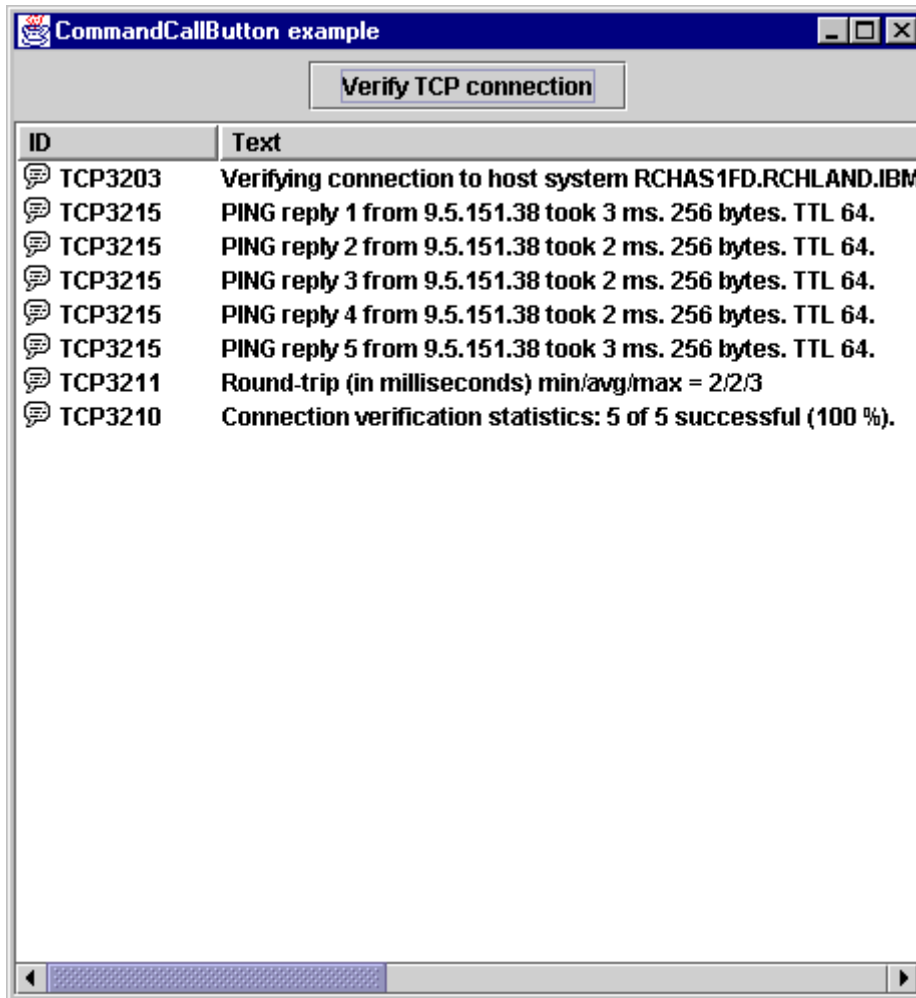
2. Run the program.

```
java CommandCallButtonExample
```

3. The program will display the graphical user interface below. It includes a button at the top, which calls the AS/400 command when clicked. It also displays an empty message list. The message list is empty since we have not yet called the AS/400 command.



4. Click on the button.
5. The program will prompt you for a user ID and password. This happens the first time you call a command because this is when the physical connection to the AS/400 database is made. Enter your assigned user ID and password.
6. This program will call the AS/400 command and load the messages into the details pane. Verify that the results in the table look similar to this:



7. Close the window using the “X” in the upper right corner.

## Exercise 10: Develop using VisualAge for Java (Optional)

### Introduction

Throughout this lab, you have written the Java code necessary for applications to communicate with an AS/400 and access AS/400 data and resources. In all cases, some of the Java code already existed and you added the AS/400 Toolbox for Java code. You have been using a simple editor and the compiler that comes with the Java Developers Kit. This setup works fine when you do a little bit of Java code or work on small applications. However, when your project gets larger and you are making frequent modifications, using a simple editor and the JDK tools can get cumbersome. There are many Java integrated development environment (IDEs) on the market to help you be more productive.

In this lab, you will use IBM VisualAge for Java with the AS/400 Toolbox for Java to develop a Java application from scratch. In particular, you will use VisualAge for Java's Visual Composition Editor which allows you to represent your application graphically, without writing any Java code. VisualAge for Java will take the application's graphical representation and generate the Java code for you. Congratulations, your job just got a little bit easier!

One caveat: there are many features of VisualAge for Java that we will not have time to cover. This lab will guide you through the steps necessary to develop the application -- and will hopefully give you a taste of what visual development is all about.

The AS/400 Toolbox for Java integrates well with VisualAge for Java. This is because many of the components in the AS/400 Toolbox for Java are Java Beans. Java Beans are components that follow a standard specification defined by Sun. These Beans can be used within many development and runtime tools. The VisualAge for Java Enterprise Edition comes with the AS/400 Toolbox for Java Beans preloaded.

In this lab, you will develop a Java application which displays the records from an AS/400 database file. The application will display the records one-at-a-time with field labels and buttons that allow the user to move forward and backward in the list. The RecordListFormPane object (part of the AS/400 Toolbox for Java) will be the component that does most of the work here. You will need to create the application around it.

### Goals of this exercise

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

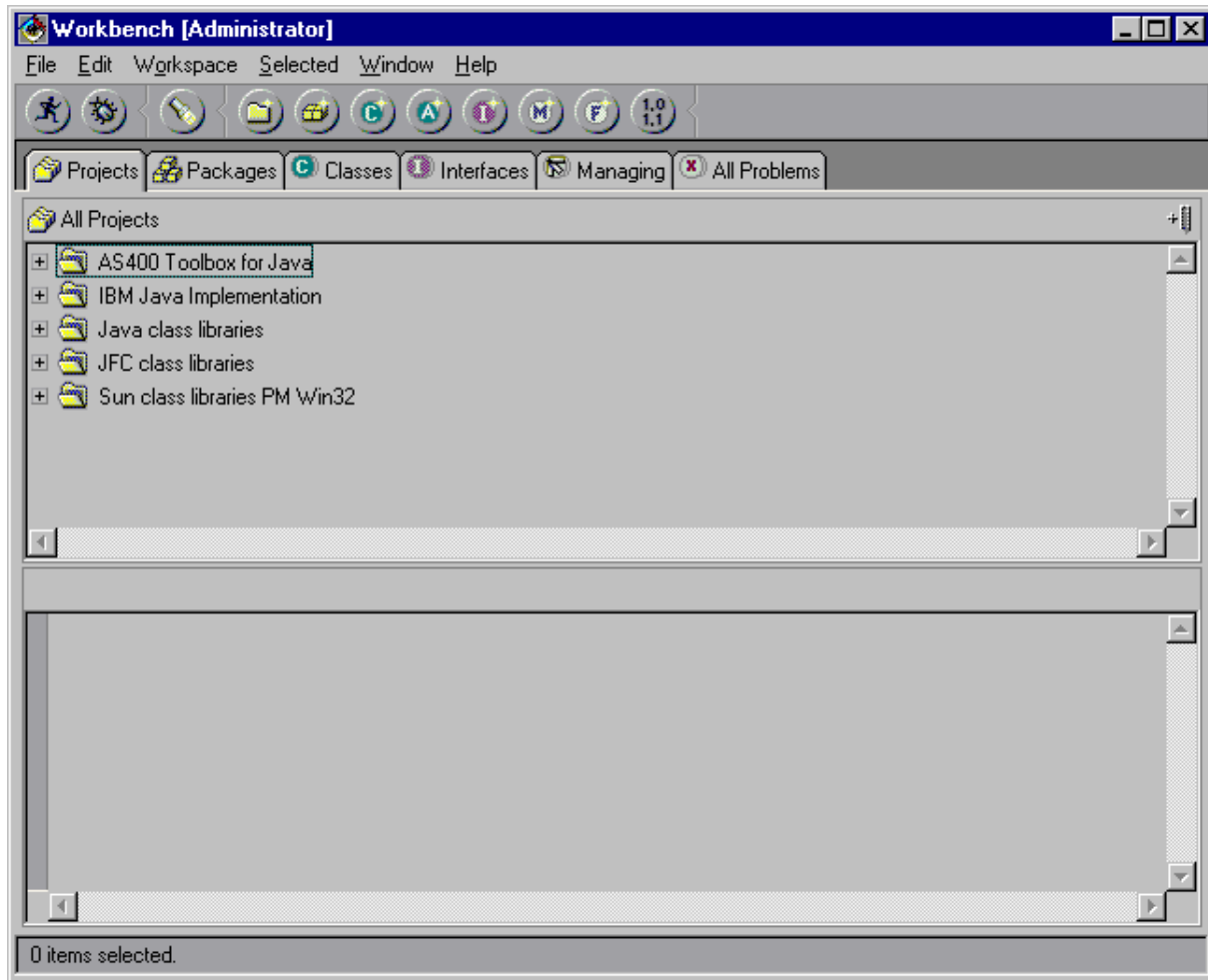
1. Start VisualAge for Java.
2. Create a project.
3. Create a JFrame object.
4. Create an AS400 object.
5. Create an RecordListFormPane object.
6. Load the contents of a database file.
7. Pack and show the JFrame object.



## Part 1: Start VisualAge for Java

### Procedure

1. Select the following menus: **Start - Programs - IBM VisualAge for Java for Windows - IBM VisualAge for Java.**
2. Verify that you see a similar window to this:

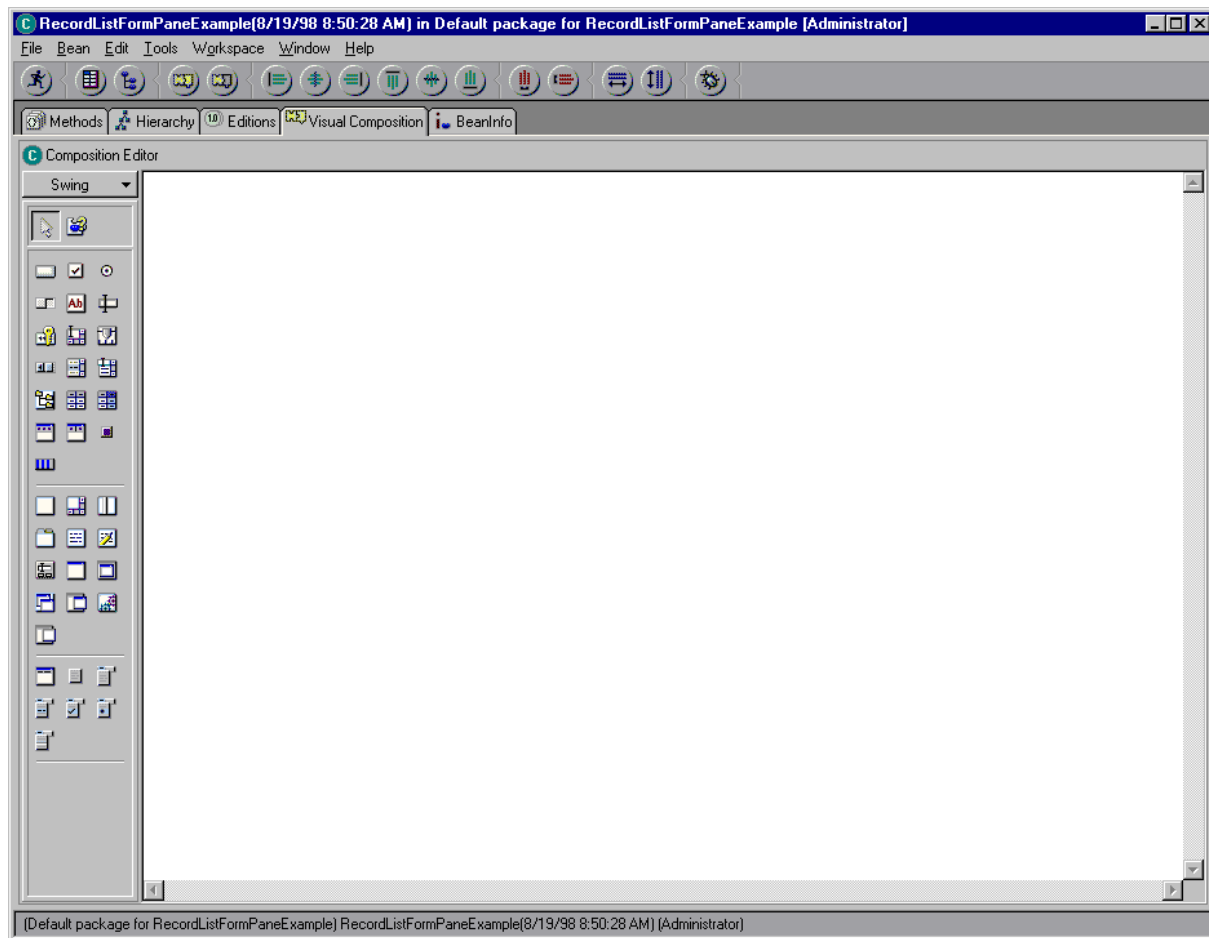


This is the main application window for VisualAge for Java. Note that there may be different projects listed when you run this.

## Part 2: Create a project

### Procedure


1. Select the following menus: **Selected - Add - Project...**
2. Under **Create a new project named:** type “RecordListFormPaneExample”. This is the name of our project.
3. Click **Finish**.
4. If VisualAge for Java tells you that there is already a project with this name in the repository, click **Ok** to replace the old project.
5. Verify that there is now a project named “RecordListFormPaneExample”.
6. Right click on this project and select: **Add - Class...**
7. Under **Class name:** type “RecordListFormPaneExample”.
8. Make sure that **Compose the class visually** is checked.
9. Click **Finish**.
10. Verify that the Visual Composition Editor started automatically:

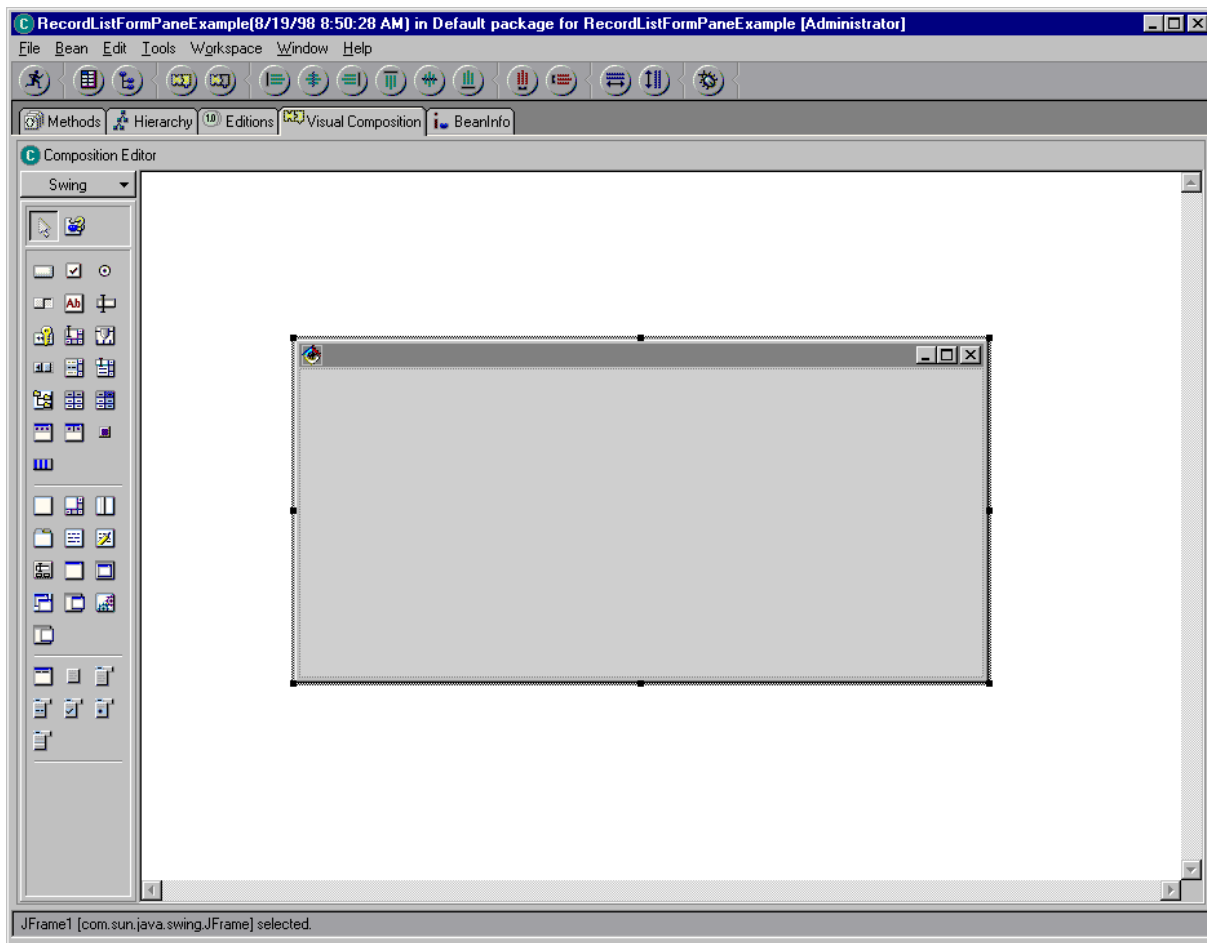


This is the window where we will develop our application.

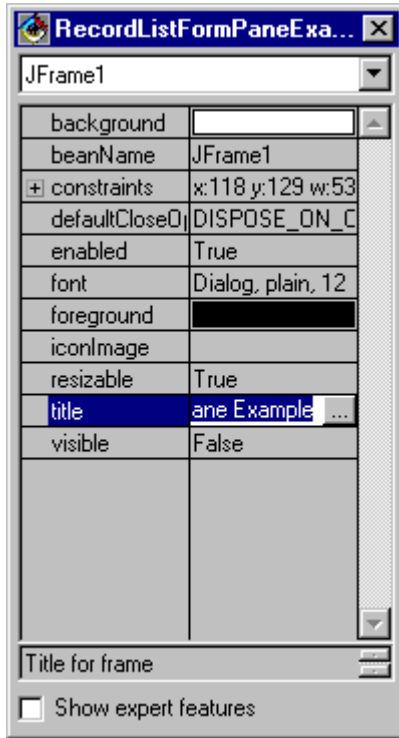
## Part 3: Create a JFrame object

### Procedure

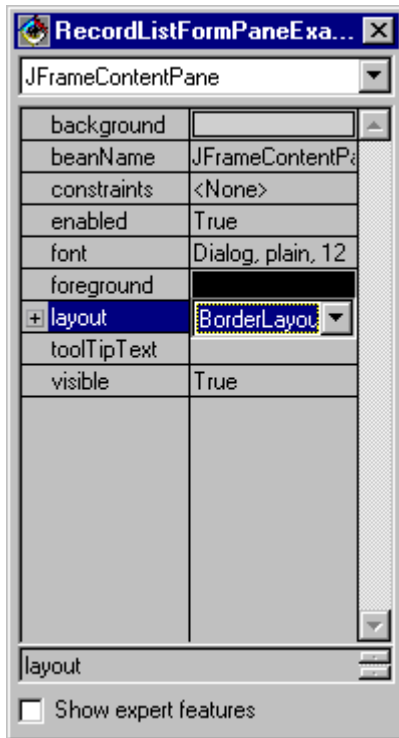
1. Notice the palette on the left side of the Visual Composition Editor. This shows icons for all of the Java Beans currently loaded into VisualAge for Java. Whenever you need a new object in your application, you can select it from the palette and then click on the Visual Composition Editor where you want the object to be located.
2. The first object you need for your application is a JFrame object. This is a Swing object which represents the application's main window. At the top of the palette, there is category listed. You can choose from several categories of Java Beans. Select **Swing**, since JFrame is a Swing object.
3. Select on the JFrame object  from the palette to the Visual Composition Editor. If you have trouble finding JFrame in the palette, remember that holding your mouse over any of the icons for a few seconds will cause its name to appear briefly. This is helpful to choose among several icons that look similar.
4. Click anywhere on the Visual Composition Editor to indicate where to drop the new JFrame object. Verify that the JFrame object appears in the Visual Composition Editor:



- There are a few properties of the JFrame object that you should set. The first is its title, which will show up in the top bar of the window. Right click on the *top bar* of the JFrame object and select **Properties**.
- This brings up a window which lists some of the properties of the JFrame object. Click on **title** and enter in a title for your application's main window: "Record List Form Pane Example".



- Click on the "X" in the upper right corner of the Properties window to make it go away. Your JFrame object should now reflect the title that you assigned.
- Another property that you need to set is the JFrame object's layout manager. The layout manager is responsible for positioning and sizing components correctly. Right click on the *middle* object JFrame object and select **Properties**.
- This brings up some more properties of the JFrame object (actually the JFrame object's "content pane"). Click on **layout** and select "BorderLayout" from the list of choices.

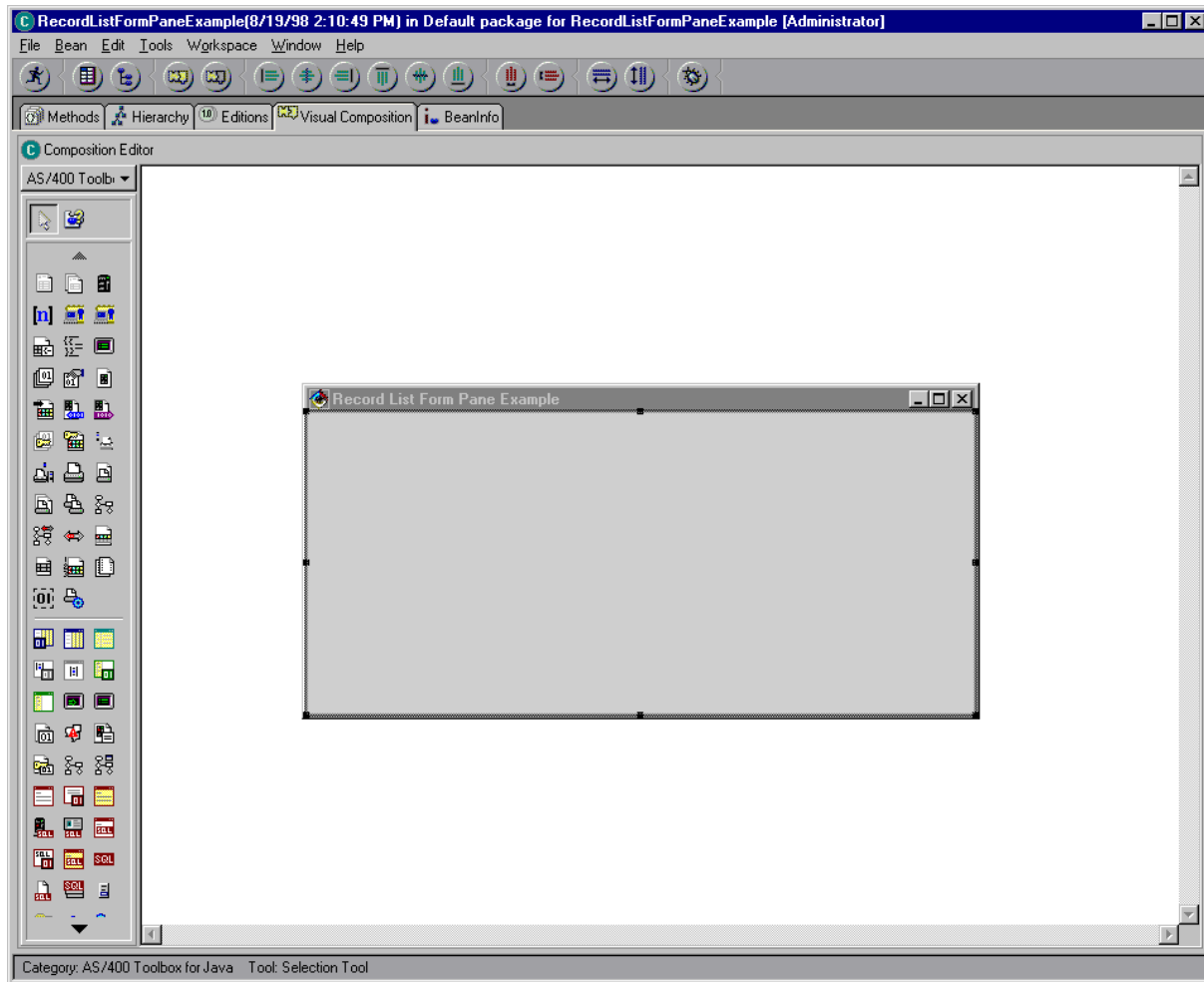



10. Click on the “X” in the upper right corner of the Properties window to make it go away. Your JFrame object should now be completely initialized.

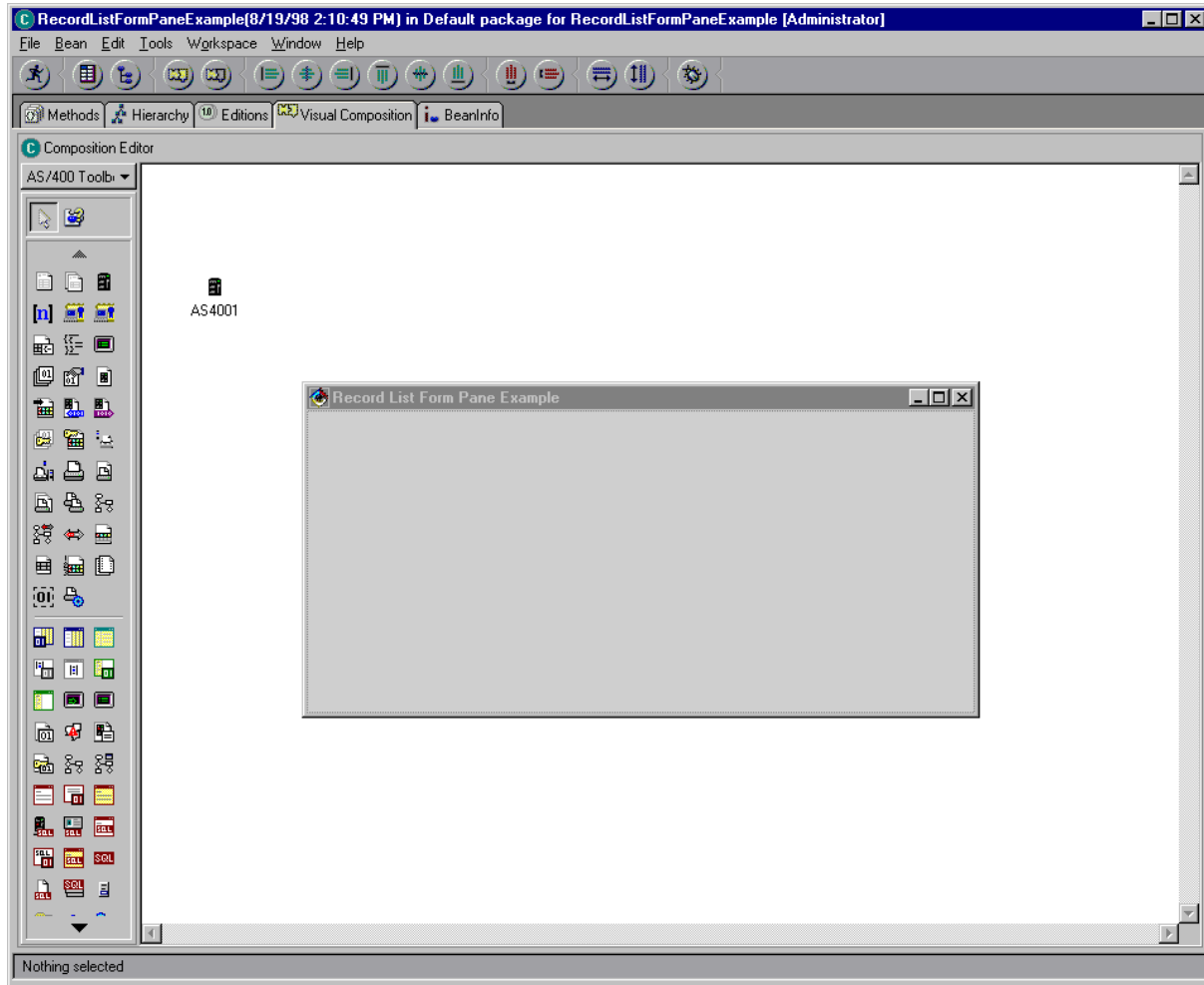
## Part 4: Create an AS400 object

### Procedure

1. At the top of the palette, select **AS/400 Toolbox for Java**. This will cause the palette to display all of the AS/400 Toolbox for Java's Beans. Remember that if you can not determine what Bean is using the icon, then hold the mouse over each icon and VisualAge for Java will briefly display the name of the Bean.




2. You will need an AS400 object to enable this application to communicate with an AS/400 system. You must create an AS400 object by selecting the AS400 object  from the palette and dropping anywhere on the Visual Composition Editor, except for on the JFrame object. The reason that you do not drop it on this object is because it is a "non-visual" Bean and is not part of the graphical user interface. Verify that the contents of the Visual Composition Editor now look similar to this:

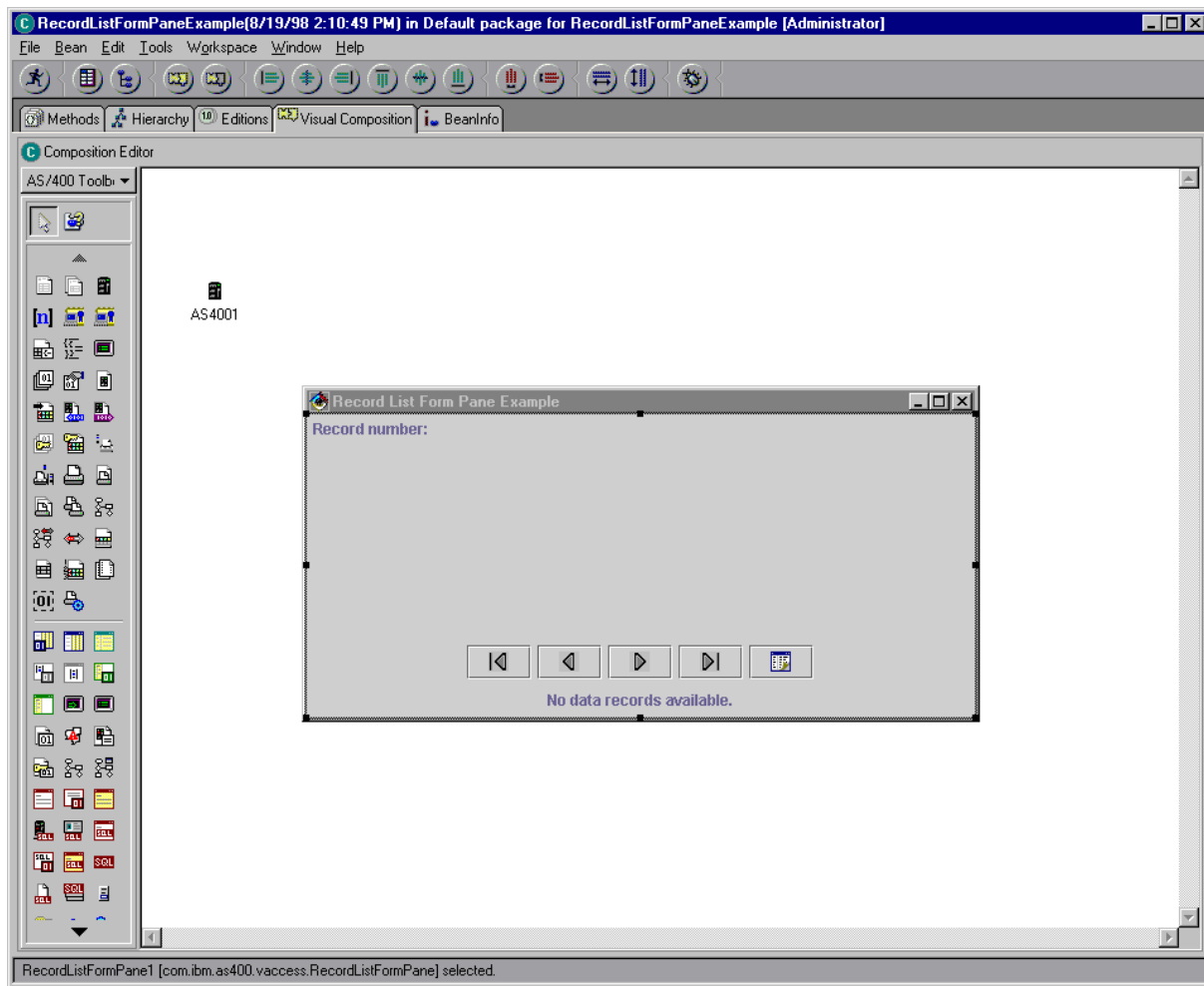


3. Right click on the AS400 object and select **Properties**. Set the **systemName** property to be the name of the AS/400 that you are using for this lab.

## Part 5: Create a RecordListFormPane object

### Procedure

1. Select on the RecordListFormPane object  from the palette to the Visual Composition Editor.
2. Click inside the center of the JFrame. This will place the RecordListFormPane in the JFrame in the Visual Composition Editor. RecordListFormPane is an AS/400 Toolbox for Java Bean that will display the contents of an AS/400 database file, one record at a time. It provides several buttons that allow the user to move to different records. Verify that the contents of the Visual Composition Editor now look similar to this:

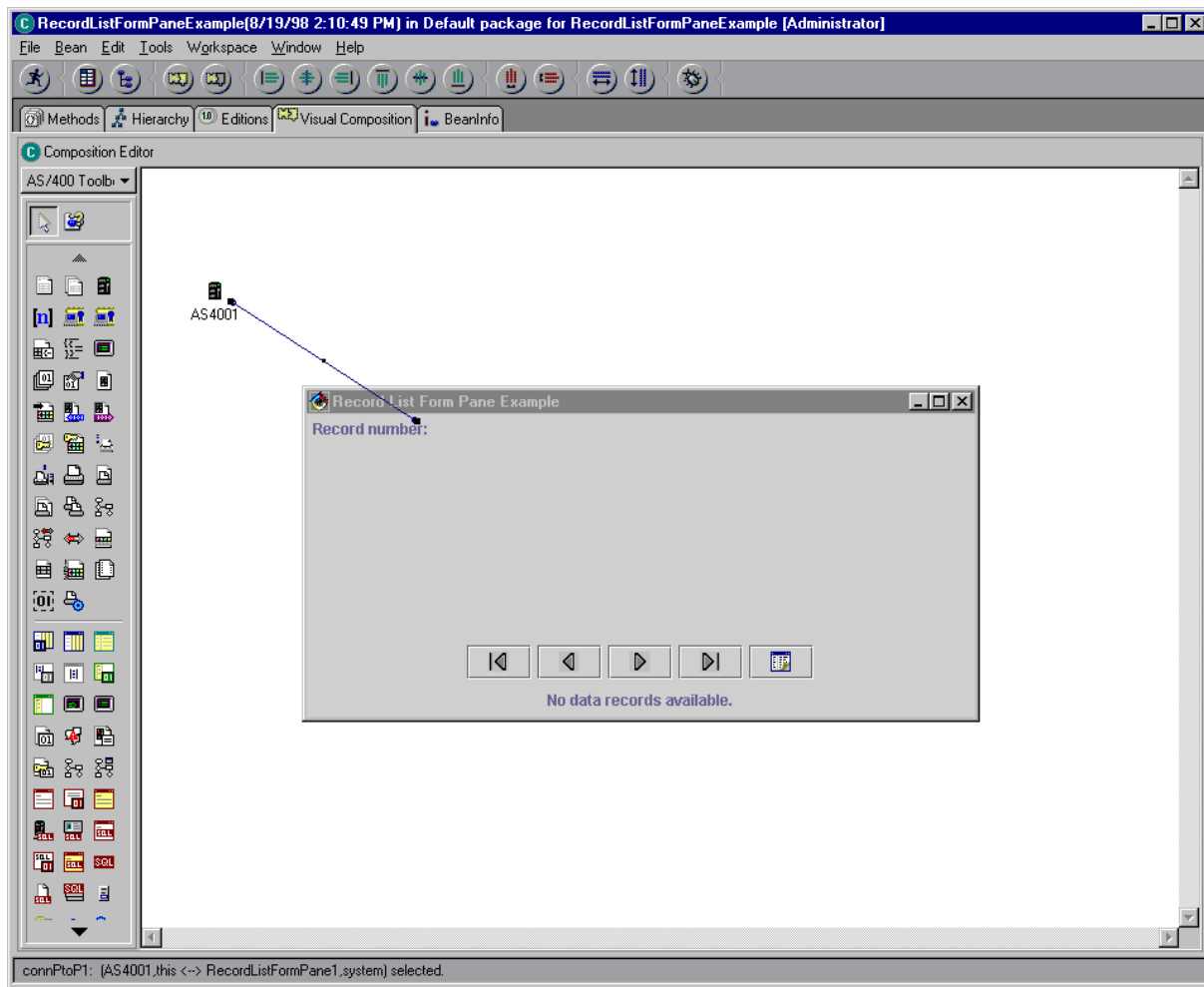


3. Note that at this point, the RecordListFormPane object does not refer to any particular AS/400 database file yet. For this, you need to set some properties. The first is the system where the file resides. The system is just an AS400 object that you created in the previous part. In the Visual Composition Editor, you define object relationships by drawing “connections” between two or more objects. Try not to confuse this use of the word “connection” with the AS/400 connection. You need to specify the **system** property of the



RecordListFormPane object to be equal to the AS400 object that you just defined. Right click on the RecordListFormPane object and select **Connect - Connectable Features...**

4. Choose **system** and click on **Ok**. This defines which property you want to set and the start of a connection. The connection is represented by a dashed line and there is a “spider” on the end, which means that you must click on the object which should be the other end of the connection.
5. Click on the AS400 object, since it is the value to which we want to set the property.
6. Select **this** on the popup menu that appears. This means that the value of the property is set to the entire AS400 object. You have just set the RecordListFormPane object’s **system** property to be equal the AS400 object, and you still have not written any code... Verify that the contents of the Visual Composition Editor now look similar to this:

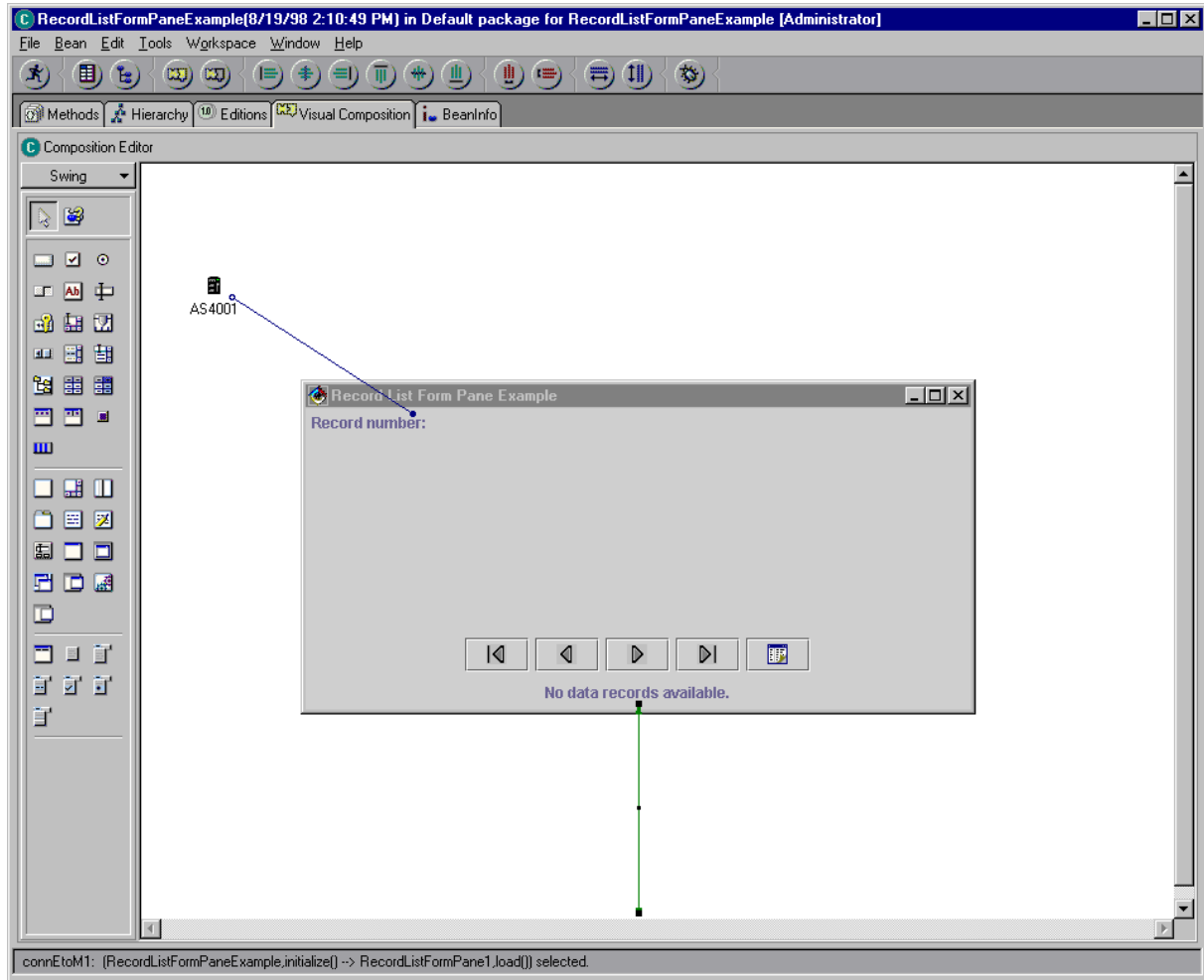


7. You still need to define which database file the RecordListFormPane object will display. Right click on the RecordListFormPane object and select **Properties**. Set the **fileName** property to “/QSYS.LIB/QIWS.LIB/QCUSTCDT.FILE”.

## Part 6: Load the contents of a database file

### Procedure

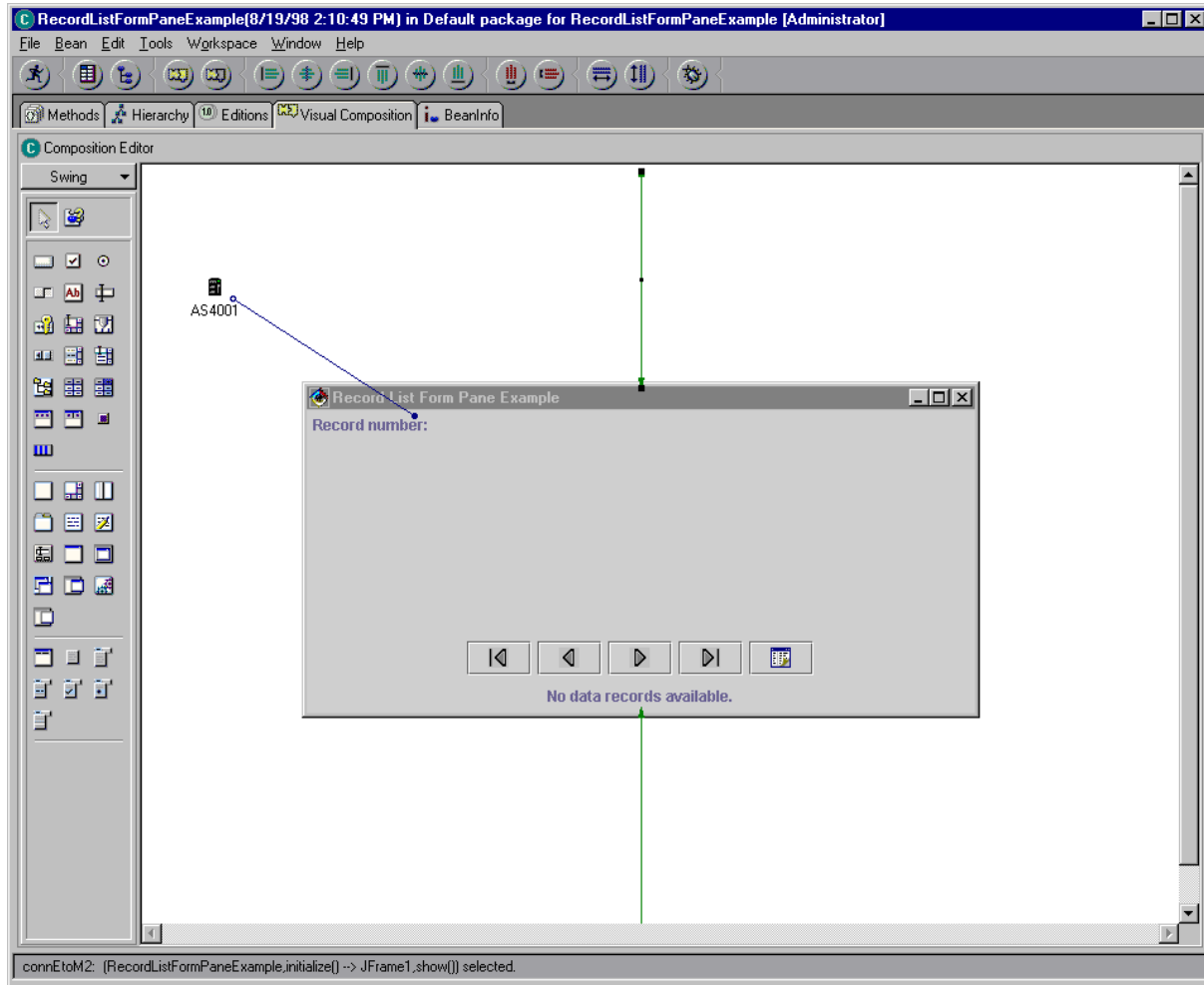
1. Another thing that you need to do is use the Visual Composition Editor to define when your application should actually load the contents of the database file. For this example, you will load the contents as soon as the application initializes. You can define this with another connection. This time you will connect the `initialize()` event for the application to the `load()` method of the `RecordListFormPane` object. Right click on any part of the white space, which represents the overall application. Select **Connect...**
2. You should be presented with a window title “Start Connection From (`RecordListFormPaneExample`)”. This window lists all of the properties and events for which you can start a connection relating to the overall application. Click on **Event** to list the events. Select **initialize()** and click on **Ok**. This defines the start of a connection. Your cursor will change to a “spider”, so it is time again to click on the target object of the connection.
3. Click inside the center of the `RecordListFormPane` object. You will see a popup menu that gives you some choices for the other end of the connection. Select **Connectable Features...** You should now see a window which gives you the choice of all properties and methods for which you can end this connection relating to the `RecordListFormPane` object. Click on **Method** to list the methods. Select **load()** and click on **Ok**. This defines the end of the connection. You have just made a connection that means that when the application initializes, it should call the `load()` method of the `RecordListFormPane` object. Verify that the contents of the Visual Composition Editor now look similar to this:



## Part 7: Pack and show the JFrame object

### Procedure

1. Your application is almost complete. The graphical user interface is initialized and ready to go. The last thing you need to do is use the Visual Composition Editor to define when your application should present (or “show”) the graphical user interface to the user. For this example, you will do this as soon as the application initializes. You can do this with yet another connection. This time you will connect the initialize() event for the application to the pack() and show() methods of the JFrame object. Right click on any part of the white space, which represents the overall application. Select **Connect...**
2. You should be presented with a window title “Start Connection From (RecordListFormPaneExample)”. This window lists all of the properties and events for which you can start a connection relating to the overall application. Click on **Event** to list the events. Select **initialize()** and click on **Ok**. This defines the start of a connection. Your cursor will change to a “spider”, so it is time again to click on the target object of the connection.
3. Click on the top bar of the JFrame object. You will see a popup menu that gives you some choices for the other end of the connection. Select **Connectable Features...** You should now see a window which gives you the choice of all properties and methods for which you can end this connection relating to the JFrame object. Click on **Method** to list the methods. Select **pack()** and click on **Ok**. This defines the end of the connection. You have just made a connection that means that when the application initializes, it should call the pack() method of the JFrame object. Verify that the contents of the Visual Composition Editor now look similar to this:




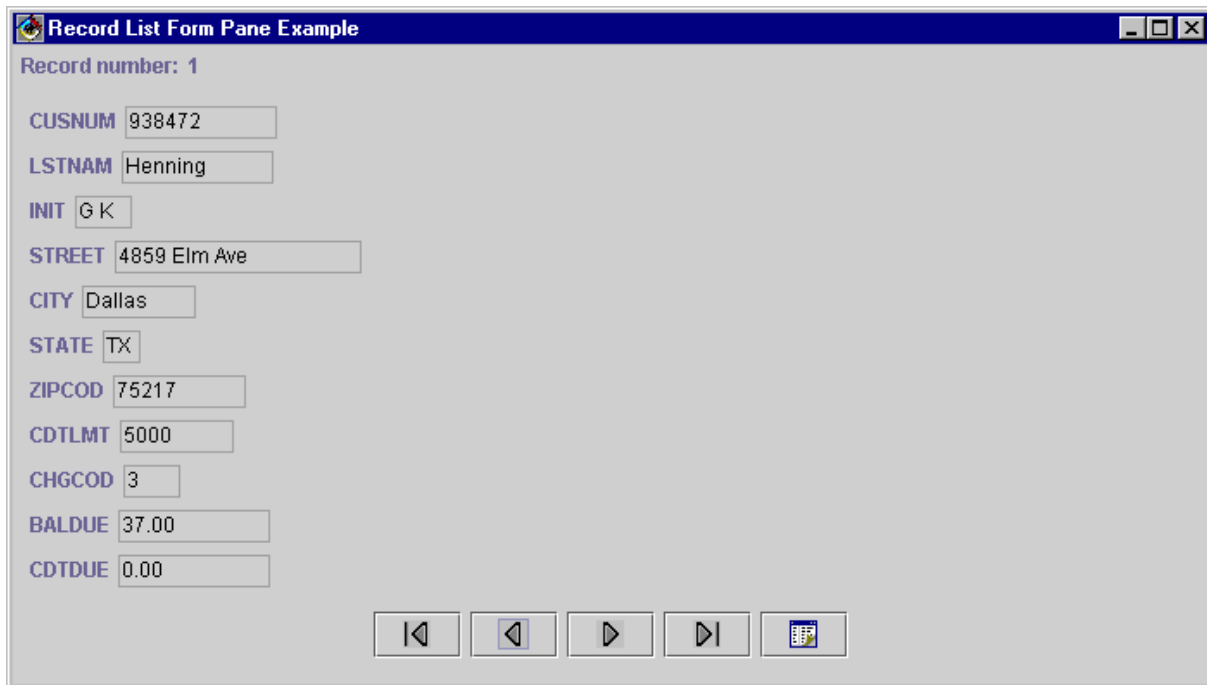
Note that some of the connection arrows may be positioned differently depending on where you dropped the JFrame object. This is fine as long as the correct objects are connected.

4. Right click on any part of the white space again, which represents the overall application. Select **Connect....**
5. You should be presented with a window title “Start Connection From (RecordListFormPaneExample)”. Click on **Event** to list the events. Select **initialize()** and click on **Ok**. This defines the start of another connection. Your cursor will again change to a “spider”, so it is time again to click on the target object of the connection.
6. Click on the top bar of the JFrame object. Select **Connectable Features....** You should see the window which gives you the choice of all properties and methods for which you can end this connection relating to the JFrame object. Click on **Method** to list the methods. Select **show()** and click on **Ok**. This defines the end of the connection. You have just made a connection that means that when the application initializes, it should call the show() method of the JFrame object.

## Run the program

Now it is time to run the RecordListFormPaneExample program.

1. In the Visual Composition Editor, click on the Run button . VisualAge for Java will save your application, generate Java code based on the objects and connections that you defined, and run the application. This may take a few minutes.
2. When the program runs, you will get the AS/400 Toolbox for Java signon prompt. Enter the user ID and password that you are assigned for this lab. This should all look familiar, since this is just another Java program using the AS/400 Toolbox for Java. The only difference is that VisualAge for Java generated the Java code instead of you!
3. After signing on you should see the JFrame object with the RecordListFormPane object inside:



4. You can step through the records using the buttons at the bottom of the graphical user interface.
5. Close the window using the “X” in the upper right corner.

## Conclusion

In this lab, you enabled various pieces of a client program to access an AS/400 various components from the AS/400 Toolbox for Java.

The AS/400 Toolbox for Java is implemented using 100% Pure Java. This means 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 and maintenance expenses usually associated with multiple platform application development.

You can get more information about the AS/400 Toolbox for Java and download a trial or beta version by going to the web address <http://www.as400.ibm.com/toolbox>.

## Appendix A: 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 extends Object
{
    public static void main(String[] args)
    {
        try
        {
            // -----
            //           Lab Exercise #1 Part #1 - Insert code here.
            // -----

            AS400 system = new AS400("mySystem");

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

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

            CommandCall command = new CommandCall(system);

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

            // Gather the command line arguments passed to this program.
            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))
                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.  
// -----  
  
AS400Message[] messageList = command.getMessageList();  
for (int i=0; i < messageList.length; i++)  
{  
    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: Data Queue

```
import com.ibm.as400.access.AS400;  
import com.ibm.as400.access.DataQueue;  
import com.ibm.as400.access.DataQueueEntry;  
  
public class DataQueueExample extends Object  
{  
    public static void main(String[] args)  
    {  
        try  
        {  
  
            // -----  
            //           Lab Exercise #2 Part #1 - Insert code here.  
            // -----  
  
            AS400 system = new AS400("mySystem");  
            system.connectService(AS400.DATAQUEUE);  
  
            // -----  
            //           End of code.  
            // -----  
  
            // -----  
            //           Lab Exercise #2 Part #2 - Insert code here.  
            // -----  
  
            DataQueue dataQ = new DataQueue(system,  
                                           "/QSYS.LIB/myLibrary.LIB/myDataQ.DTAQ");  
  
            // -----  
            //           End of code.  
            // -----  
  
        }  
        catch (Exception e) {  
            System.out.println ("Error: " + e);  
        }  
    }  
}
```

```
// -----  
  
// -----  
//           Lab Exercise #2 Part #3 - Insert code here.  
// -----  
  
dataQ.create(50);  
  
// -----  
//           End of code.  
// -----  
  
// -----  
//           Lab Exercise #2 Part #4 - Insert code here.  
// -----  
  
dataQ.write("The AS/400 Toolbox for Java is 100% Pure Java");  
  
// -----  
//           End of code.  
// -----  
  
// -----  
//           Lab Exercise #2 Part #5 - Insert code here.  
// -----  
  
DataQueueEntry dqEntry = dataQ.peek();  
System.out.println(dqEntry.getString());  
  
// -----  
//           End of code.  
// -----  
  
}  
catch (Exception e) {  
    System.out.println ("Error: " + e);  
}  
}  
System.exit (0);  
}  
}
```

### Exercise 3: JDBC

```
import com.ibm.as400.access.AS400;  
import com.ibm.as400.access.AS400JDBCdriver;  
import java.sql.Connection;  
import java.sql.DriverManager;  
import java.sql.PreparedStatement;  
import java.sql.ResultSet;  
import java.sql.ResultSetMetaData;  
import java.sql.SQLException;  
import java.sql.Statement;  
  
public class JDBCExample extends Object  
{  
    public static void main(String[] args)
```

```
{
  try
  {
    String collectionName_ = "";
    String tableName_ = "";

    // Evaluate the input parameters.
    if (args.length != 2)
    {
      System.out.println();
      System.out.println("Usage:");
      System.out.println();
      System.out.println("  JDBCExample collectionName tableName");
      System.out.println();
      System.out.println("For example:");
      System.out.println();
      System.out.println("  JDBCExample MyLibrary MyTable");
      System.out.println();
    }
    else
    {
      collectionName_ = args[0];
      tableName_ = args[1];
    }

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

    AS400 system = new AS400("mySystem");
    DriverManager.registerDriver(new AS400JDBCdriver());
    Connection connection = DriverManager.getConnection ("jdbc:as400://" +
                                                       system);

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

    // -----
    //           Lab Exercise #3 Part #2 - Insert code here.
    // -----

    try
    {
      Statement createCollection = connection.createStatement();
      createCollection.executeUpdate ("CREATE COLLECTION " +
                                     collectionName_);
    }
    catch (SQLException s)
    {
      // If collection already exists ignore and do nothing.
    }

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

    // Drop the table if it already exists.
  }
}
```

```
try
{
    Statement dropTable = connection.createStatement ();
    dropTable.executeUpdate ("DROP TABLE " + collectionName_ +
        "." + tableName_);
}
catch (SQLException sql)
{
    // Ignore, do nothing.
}

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

Statement createTable = connection.createStatement();
createTable.executeUpdate ("CREATE TABLE " + collectionName_ + "."
    + tableName_ + " (CUSTNUM INTEGER, NAME VARCHAR(20), "
    + "LIMIT DOUBLE, BALANCE DOUBLE)");

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

// -----
//           Lab Exercise #3 Part #4 - Insert code here.
// -----

PreparedStatement insertStatement = connection.prepareStatement(
    "INSERT INTO " + collectionName_ + "." + tableName_ +
    " (CUSTNUM, NAME, LIMIT, BALANCE)" + " VALUES (?, ?, ?, ?)");

// Insert customer records into the table
insertStatement.setInt (1, 500001);
insertStatement.setString (2, "Mickey Mouse");
insertStatement.setDouble (3, 150000.00);
insertStatement.setDouble (4, 5897.95);
insertStatement.executeUpdate ();

insertStatement.setInt (1, 500002);
insertStatement.setString (2, "Donald Duck");
insertStatement.setDouble (3, 80000.00);
insertStatement.setDouble (4, 63000.25);
insertStatement.executeUpdate ();

insertStatement.setInt (1, 500008);
insertStatement.setString (2, "Goofy");
insertStatement.setDouble (3, 11500.00);
insertStatement.setDouble (4, 905.72);
insertStatement.executeUpdate ();

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

// -----
//           Lab Exercise #3 Part #5 - Insert code here.
// -----
```

```

Statement select = connection.createStatement ();
ResultSet rs = select.executeQuery ("SELECT * FROM " + collectionName_
    + "." + tableName_);

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

// -----
//                               Lab Exercise #3 Part #6 - Insert code here.
// -----

ResultSetMetaData rsmd = rs.getMetaData ();
int columnCount = rsmd.getColumnCount ();

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

// Print the column headers to System.out
for (int x=1; x<= columnCount; ++x)
{
    String label = rsmd.getColumnLabel(x);
    int colSize = rsmd.getColumnDisplaySize(x);

    printColumn(label, colSize);
}
System.out.println();

// -----
//                               Lab Exercise #3 Part #7 - Insert code here.
// -----

while (rs.next ())
{
    for (int i=1; i<= columnCount; ++i)
    {
        String data = rs.getString(i);
        int columnSize = rsmd.getColumnDisplaySize(i);

        printColumn (data, columnSize);
    }
    System.out.println();
}

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

}
catch (Exception e)
{
    System.out.println("Error: " + e);
}
System.exit (0);
}

// -----
// Method: printColumn

```

```
// Description: Prints out the column information to System.out
//           padding remaining column display length with spaces.
// -----
private static void printColumn(String data, int columnSize)
{
    StringBuffer buffer = new StringBuffer(columnSize - data.length());

    for (int c=0; c< buffer.capacity(); ++c)
        buffer = buffer.insert(c, ' ');

    // Add a space for a column divider.
    String pad = buffer.append(' ').toString();

    System.out.print(data + pad);
}
}
```

### Exercise 4: Record-level access

```
import com.ibm.as400.access.AS400;
import com.ibm.as400.access.AS400Text;
import com.ibm.as400.access.AS400ZonedDecimal;
import com.ibm.as400.access.CharacterFieldDescription;
import com.ibm.as400.access.QSYSObjectPathName;
import com.ibm.as400.access.SequentialFile;
import com.ibm.as400.access.Record;
import com.ibm.as400.access.RecordFormat;
import com.ibm.as400.access.ZonedDecimalFieldDescription;
import java.math.BigDecimal;

public class RLAExample extends Object
{
    public static void main(String[] args)
    {
        try
        {
            // -----
            //           Lab Exercise #4 Part #1 - Insert code here.
            // -----

            // Create a record description for the file, QIWS\QCUSTCDT
            ZonedDecimalFieldDescription cusNum = new ZonedDecimalFieldDescription
                (new AS400ZonedDecimal(6,0), "CUSNUM");

            CharacterFieldDescription lastName = new CharacterFieldDescription
                (new AS400Text(8), "LSTNAM");
            CharacterFieldDescription initials = new CharacterFieldDescription
                (new AS400Text(3), "INIT");
            CharacterFieldDescription street = new CharacterFieldDescription
                (new AS400Text(13), "STREET");
            CharacterFieldDescription city = new CharacterFieldDescription
                (new AS400Text(6), "CITY");
            CharacterFieldDescription state = new CharacterFieldDescription
                (new AS400Text(2), "STATE");

            ZonedDecimalFieldDescription zipCode = new ZonedDecimalFieldDescription
                (new AS400ZonedDecimal(5,0), "ZIPCOD");
        }
    }
}
```

```

ZonedDecimalFieldDescription cdtLimit = new ZonedDecimalFieldDescription
    (new AS400ZonedDecimal(4,0), "CDTLMT");
ZonedDecimalFieldDescription chgCode = new ZonedDecimalFieldDescription
    (new AS400ZonedDecimal(1,0), "CHGCOD");
ZonedDecimalFieldDescription balDue = new ZonedDecimalFieldDescription
    (new AS400ZonedDecimal(6,2), "BALDUE");
ZonedDecimalFieldDescription cdtDue = new ZonedDecimalFieldDescription
    (new AS400ZonedDecimal(6,2), "CDTDUE");

RecordFormat qcustcdt = new RecordFormat("CUSREC");

// Add the field descriptions to the record format.
qcustcdt.addFieldDescription(cusNum);
qcustcdt.addFieldDescription(lastName);
qcustcdt.addFieldDescription(initials);
qcustcdt.addFieldDescription(street);
qcustcdt.addFieldDescription(city);
qcustcdt.addFieldDescription(state);
qcustcdt.addFieldDescription(zipCode);
qcustcdt.addFieldDescription(cdtLimit);
qcustcdt.addFieldDescription(chgCode);
qcustcdt.addFieldDescription(balDue);
qcustcdt.addFieldDescription(cdtDue);

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

// -----
//                               Lab Exercise #4 Part #2 - Insert code here.
// -----

AS400 system = new AS400();

// Create the object path name to the database file.
QSYSObjectPathName fileName = new QSYSObjectPathName("QIWS",
    "QCUSTCDT",
    "FILE");

// Create the sequential file object.
SequentialFile file = new SequentialFile(system, fileName.getPath());

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

// -----
//                               Lab Exercise #4 Part #3 - Insert code here.
// -----

// Set the file's record format.
file.setRecordFormat(qcustcdt);

// Open the file for read-only access.  Specify blocking factor of
// 10.  Don't use commitment control.
file.open(SequentialFile.READ_ONLY, 10,
    SequentialFile.COMMIT_LOCK_LEVEL_NONE);

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

```

```

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

// Read the file record of the file.
Record data = file.readNext();

// While there are records in the file, read the next record.
while (data != null)
{
    // Display customer and balance if balance due > 0.
    if ( ((BigDecimal)data.getField("BALDUE")).floatValue() > 0.0)
    {
        System.out.print( (String)data.getField("INIT") + " ");
        System.out.print( (String)data.getField("LSTNAM") + " ");
        System.out.println( (BigDecimal)data.getField("BALDUE"));
    }

    // Read the next record in the file.
    data = file.readNext();
}
// -----
//           End of code.
// -----

// -----
//           Lab Exercise #4 Part #5 - Insert code here.
// -----

system.disconnectService(AS400.RECORDACCESS);

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

}
catch (Exception e)
{
    System.out.println("Error: " + e);
}
System.exit (0);
}
}

```

## Exercise 5: Program Call

```

import com.ibm.as400.access.AS400;
import com.ibm.as400.access.AS400Message;
import com.ibm.as400.access.AS400Text;
import com.ibm.as400.access.MessageQueue;
import com.ibm.as400.access.ProgramCall;
import com.ibm.as400.access.ProgramParameter;
import com.ibm.as400.access.QSYSObjectPathName;
import com.ibm.as400.access.QueuedMessage;

public class ProgramCallExample extends Object

```



```

{
public static void main(String[] args)
{
    try
    {
        // -----
        //             Lab Exercise #5 Part #1 - Insert code here.
        // -----

        AS400 system = new AS400("mySystem");
        ProgramCall pgm = new ProgramCall(system);

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

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

        ProgramParameter[] parmList = new ProgramParameter[2];

        // Input parameter #1: Data Queue library name.
        AS400Text libraryText = new AS400Text(10);
        byte[] libraryName = libraryText.getBytes("myLibrary");
        parmList[0] = new ProgramParameter(libraryName);

        // Input parameter #2: Data Queue name.
        AS400Text dqText = new AS400Text(10);
        byte[] dqName = dqText.getBytes("myDataQ");
        parmList[1] = new ProgramParameter(dqName);

        pgm.setParameterList(parmList);

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

        pgm.setProgram(QSYSObjectPathName.toPath("JAVALIB",
                                                "DQRECEIVE",
                                                "PGM"));

        // -----
        //             Lab Exercise #5 Part #3 - Insert code here.
        // -----

        if (pgm.run() != true)
        {
            AS400Message[] messageList = pgm.getMessageList();
            for (int i=0; i < messageList.length; i++)
            {
                System.out.println (messageList[i].getID() + ":" +
                                    messageList[i].getText());
            }
        }

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

```

```

// -----
// -----
//          Lab Exercise #5 Part #4 - Insert code here.
// -----

MessageQueue msgQueue = new MessageQueue(system, MessageQueue.CURRENT);
Enumeration list = msgQueue.getMessages();

while (list.hasMoreElements())
{
    QueuedMessage message = (QueuedMessage)list.nextElement();
    System.out.println(message.getText());
}

msgQueue.remove();

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

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

System.exit (0);
}
}

```

## Exercise 5: Program Call (RPG program)

Source for the RPG program that is used in the 98 Fall Common Lab Exercise #5 (ProgramCallExample). JAVALIB.LIB/DQRECEIVE.PGM

This RPG program receives an entry from a sequential data queue with maximum data entry length of 50 (library and data queue name are input parameters on the program) and displays the results into a message queue (name from library parameter).

```

.CL0N01N02N03Factor1+++OpcdeFactor2+++ResultLenDHHiLoEq
***** Beginning of data *****
C          *ENTRY    PLIST
C          PARM      LIB      10
C          PARM      NAME     10
C          CALL 'QRCVDTAQ'
C          PARM NAME   JVDTAQ  10
C          PARM LIB    JVDLIB  10
C          PARM      DTQLEN  50
C          PARM      WK50   50
C          PARM 1      DTWAIT  50
C          WK50      DSPLYLIB
C          RETRN

```

## Exercise 7: 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.ResultSetTablePane;

import com.sun.java.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.beans.*;
import java.sql.*;

public class ResultSetTablePaneExample
extends KeyAdapter {

    private static ResultSetTablePane    tablePane_;
    private static JTextField            textField_;

    public static void main(String argv[])
    {
        try {
            // Register the AS/400 Toolbox for Java JDBC driver.
            DriverManager.registerDriver(new AS400JDBCdriver());

            // -----
            //                      Lab Exercise #7 Part #1 - Insert code here.
            // -----

            SQLConnection connection = new SQLConnection("jdbc:as400://mySystem");

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

            // -----
            //                      Lab Exercise #7 Part #2 - Insert code here.
            // -----

            ResultSetTablePane tablePane = new ResultSetTablePane();
            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 ResultSetTablePaneExample ());

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

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

// -----
//                Lab Exercise #7 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 #7 Part #3 - Insert code here.
            // -----

            tablePane.setQuery (queryText);
            tablePane.load ();

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

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

## Exercise 8: Navigate the Integrated File System

```
import com.ibm.as400.access.AS400;
import com.ibm.as400.vaccess.AS400ExplorerPane;
import com.ibm.as400.vaccess.ErrorDialogAdapter;
import com.ibm.as400.vaccess.VIFSDirectory;

import com.sun.java.swing.*;
import java.awt.*;
import java.awt.event.*;

public class VIFSDirectoryExample
{

    public static void main(String argv[])
    {
        try
        {
            // -----
            //             Lab Exercise #8 Part #1 - Insert code here.
            // -----

            AS400 system = new AS400 ("mySystem");
            VIFSDirectory root = new VIFSDirectory (system, "/QIBM/ProdData");

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

            // -----
            //             Lab Exercise #8 Part #2 - Insert code here.
            // -----

            AS400ExplorerPane explorerPane = new AS400ExplorerPane (root);
            explorerPane.load ();

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

            // Initialize the frame.
            JFrame frame = new JFrame ("VIFSDirectory example");
            frame.getContentPane ().setLayout (new BorderLayout ());
            frame.getContentPane ().add ("Center", explorerPane);

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

            ErrorDialogAdapter errorHandler = new ErrorDialogAdapter (explorerPane);
            explorerPane.addErrorListener (errorHandler);

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

            // Display the frame.

```

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

## Exercise 9: Command Call Button and Message List

```
import com.ibm.as400.access.ActionCompletedEvent;
import com.ibm.as400.access.ActionCompletedListener;
import com.ibm.as400.access.AS400;
import com.ibm.as400.vaccess.AS400DetailsPane;
import com.ibm.as400.vaccess.CommandCallButton;
import com.ibm.as400.vaccess.ErrorDialogAdapter;
import com.ibm.as400.vaccess.VMessageList;

import com.sun.java.swing.*;
import java.awt.*;
import java.awt.event.*;
```

```
public class CommandCallButtonExample
implements ActionCompletedListener {
```

```
    private static CommandCallButton    button_;
    private static AS400DetailsPane     detailsPane_;
    private static VMessageList         messageList_;
```

```
    public static void main(String argv[])
    {
        try {
```

```
            // -----
            //                Lab Exercise #9 Part #1 - Insert code here.
            // -----
```

```
            AS400 system = new AS400 ("mySystem");
            CommandCallButton button
                = new CommandCallButton ("Verify TCP connection");
            button.setSystem (system);
            button.setCommand ("PING " + system.getSystemName ());
```

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

```
            // -----
            //                Lab Exercise #9 Part #2 - Insert code here.
            // -----
```

```
            VMessageList messageList = new VMessageList ();
```

```
// -----  
//                               End of code.  
// -----  
  
// -----  
//                               Lab Exercise #9 Part #3 - Insert code here.  
// -----  
  
AS400DetailsPane detailsPane = new AS400DetailsPane (messageList);  
  
// -----  
//                               End of code.  
// -----  
  
// Set up the action completed listener.  
button_ = button;  
messageList_ = messageList;  
detailsPane_ = detailsPane;  
button.addActionListener (new CommandCallButtonExample ());  
  
// Initialize a panel to hold the button.  
JPanel panel = new JPanel ();  
panel.setLayout (new FlowLayout ());  
panel.add (button);  
  
// Initialize the frame.  
JFrame frame = new JFrame ("CommandCallButton example");  
frame.getContentPane ().setLayout (new BorderLayout ());  
frame.getContentPane ().add ("North", panel);  
frame.getContentPane ().add ("Center", detailsPane);  
  
// When the frame closes, exit the program.  
frame.addWindowListener (new WindowAdapter ()  
{  
    public void windowClosing (WindowEvent event) { System.exit (0);}  
});  
  
ErrorDialogAdapter errorHandler = new ErrorDialogAdapter (detailsPane);  
detailsPane.addErrorListener (errorHandler);  
  
// Display the frame.  
frame.pack();  
frame.show();  
} catch (Exception e) {  
    System.out.println ("Error: " + e);  
}  
}  
  
// This gets called whenever a command is run.  
public void actionPerformed (ActionCompletedEvent event)  
{  
    try {  
  
        CommandCallButton    button        = button_;  
        VMessageList          messageList   = messageList_;  
        AS400DetailsPane      detailsPane   = detailsPane_;
```

```
// -----  
//           Lab Exercise #9 Part #4 - Insert code here.  
// -----  
  
messageList.setMessageList (button.getMessageList ());  
detailsPane.load ();  
  
// -----  
//           End of code.  
// -----  
  
} catch (Exception e) {  
    System.out.println ("Error: " + e);  
}  
}  
}
```