Table of Contents

IBM Toolbox for Java

IBM Toolbox for Java

IBM Toolbox for Java is a set of Java(TM) classes that allow you to use Java programs to access data on your iSeries and AS/400e servers. You can use these classes to write client/server applications, applets, and servlets that work with data on your iSeries. You can also run Java applications that use the IBM Toolbox for Java classes on the iSeries Java virtual machine (JVM).

IBM Toolbox for Java uses the iSeries Host Servers as access points to the system. Because Toolbox for Java uses communication functions built into Java, you do not need to use IBM iSeries Client Access Express to use Toolbox for Java. Each server runs in a separate job on the server, and each server job sends and receives data streams on a socket connection.

The following information will help you understand and use IBM Toolbox for Java:

Installing and configuring IBM Toolbox for Java
See how to set up and configure IBM Toolbox for Java on your system. Before installing, you need to ensure that your workstations and servers meet the requirements for IBM Toolbox for Java. This information also directs you on how to configure an HTTP server and configure your system properties.

IBM Toolbox for Java classes
See how the IBM Toolbox for Java classes enable you to work with iSeries and AS/400e server data:

- Access classes enable you to access and manage resources on your iSeries
- »Resource classes use a common framework to access and manage iSeries resources«
- Graphical User Interface (GUI) classes allow you to visually present and manipulate data
- Utility classes enable you to do administrative tasks, such as using the AS400ToolboxInstaller class and AS400JarMaker class
- Security classes make secured connections with the server and verify the identity of a user working on the iSeries system
- HTML classes allow you to quickly create HTML forms and tables
- Servlet classes assist in retrieving and formatting data for use in Java servlets

Javadocs for IBM Toolbox for Java
View the javadoc reference information for the IBM Toolbox for Java classes.

Using the Graphical Toolbox to create your own GUI panels
Use the Graphical Toolbox to create custom user interface panels in Java, which you can incorporate into your Java applications, applets, or Operations Navigator plug-ins.

Setting up proxy servers and clients
Learn how to use IBM Toolbox for Java proxy support. Proxy support is the processing that IBM Toolbox for Java needs to carry out a task on one JVM when the application is running on a different JVM. Proxy support includes using the Secure Sockets Layer (SSL) protocol to encrypt data.

Using PCML to call iSeries programs
Understand how to use Program Call Markup Language (PCML), a tag language that helps you call iSeries programs while using less Java code. PCML, which is based on XML, is a tag syntax that fully describes input and output parameters for iSeries programs called by your Java application.

Additional information
This information includes:

- A summary of V5R1 new and changed IBM Toolbox for Java functions
- Programming tips that will help you get the most from the IBM Toolbox for Java
- A tutorial with detailed descriptions of what is happening in the example code
- Convenient links to many code examples provided throughout this topic
- A reference section that links to more information about HTML, XHTML, Java, and servlets
- A PDF of the IBM Toolbox for Java information that you can print or download

What's new for V5R1

IBM Toolbox for Java is available in three forms:

- The licensed program for IBM Toolbox for Java, 5722-JC1, Version 5 Release 1 (V5R1) installs on V4R4 and later of OS/400. From a client, IBM Toolbox for Java connects back to V4R3 and later of OS/400.
- OS/400 also includes the non-graphical classes of IBM Toolbox for Java optimized for use when running IBM Toolbox for Java classes on the iSeries Java virtual machine (JVM). So if, for example, you do not have a need for the graphical functionality of the licensed program, you can still easily use IBM Toolbox for Java. For more information, see Jar files.
- IBM Toolbox for Java is also now open source. You can download the code and get information from the JTOpen web site.

Packages

IBM Toolbox for Java V5R1 includes the following new package:

- com.ibm.as400.resource provides a generic framework and consistent programming interface for working with various iSeries objects and lists.

New classes

IBM Toolbox for Java V5R1 also features many new classes in existing packages: access, html, servlet, and vaccess. The new classes enable you to:

- Convert text in bidirectional languages, such as Arabic and Hebrew, between iSeries and Java formats
- Manage a pool of AS400 objects to share and manage connections to your server, including JDBC(TM) connections
- Increase the variety of HTML tags you can include in your Java programs, including headings, meta tags, and lists
- Display IFS directory contents in an HTML list or a hierarchical tree
- Graphically present information about JDBC data sources and user-defined resources
- Display any resource in a servlet
- Use provided support for JDBC 2.0 Optional Package extensions
- Work with iSeries environment variables
- Request a license for a product installed on the iSeries
- Query and modify the state and configuration of the AS/400 NetServer
- Ping the Host Servers from a command line or within a Java program

For more information about new classes, see the associated package documentation.

Enhanced classes

IBM Toolbox for Java V5R1 also includes enhancements to existing classes. These enhancements offer:

- Improvements in IFS list processing
- Added methods to CommandCall, ProgramCall and ServiceProgramCall to:
    - Get the Job ID of the job actually running the command, program or service program
    - Provide a way to directly call the command, program or service program on the same thread as the application, instead of calling by way of a server (when running on the iSeries JVM)
    - Indicate if the iSeries command, program, or service program is thread-safe
- The ability to use SSL to encrypt connections between proxy clients and proxy servers
- User-defined grouping of system values
- An easier way to load and access additional message help text

New or enhanced functions

IBM Toolbox for Java V5R1 also features new and enhanced functions:

- Faster string conversion provided by IBM Toolbox for Java incorporating its own conversion tables
- Better NLS support for system and error messages
- HTTP tunneling available for connecting between proxy client and proxy server

Additional functions and features in Graphical Toolbox

The Graphical Toolbox incorporates new features:

- The PDML framework now uses Java Help(TM)
- GUI Builder now includes context sensitive help
- Changes in MessageViewer allow you to retrieve second level text for iSeries messages and include your own HTML detail help

Compatibility

You can not use this release of IBM Toolbox for Java to deserialize some objects that you serialized using earlier releases.

If you are using Secure Sockets Layer (SSL) to encrypt data flowing between the client and the server, you must use the SSL object delivered in the V5R1 version of IBM iSeries Client Encryption licensed program 5722-CE2 or 5722-CE3. This release of IBM Toolbox for Java will not work with previous versions SSL.

IBM Toolbox for Java continues to provide support for

- Swing 1.1, which is required to use GUI classes or the Graphical Toolbox
- Java 2 Platform, Standard Edition (J2SE), with continued support of Java Platform 1.1.x
- Linux workstations

You should also review compatibility issues involving OS/400 requirements for running IBM Toolbox for Java.

How to see what's new or changed

To help you see where technical changes have been made, this information (but not the javadocs) uses the following symbols:

- » marks the beginning of new or changed information
- « marks the end of new or changed information

Print this topic

You can view, print, or download a PDF version of this information. To access the PDF document, you must have Adobe(R) Acrobat(R) Reader installed. You can download a copy from Adobe.

To view or download the PDF version, select IBM Toolbox for Java PDF (about 936 KB or 274 pages).

To save a PDF on your workstation for viewing or printing:

1. Open the PDF in your browser (click the link above).
2. In the menu of your browser, click **File**.
3. Click **Save As**.
4. Navigate to the directory in which you would like to save the PDF.
5. Click **Save**.

Note: The **IBM Toolbox for Java** contains some information that is not included in the PDF files.

Download IBM Toolbox for Java information in a zipped package

A a zipped package of the **IBM Toolbox for Java** HTML files, including the javadocs, is available at the IBM Toolbox for Java Web site.

Note: The **IBM Toolbox for Java** information may have links to documents that are not included in the zipped package. These links will not work in the files that you download to your workstation.

Setting up IBM Toolbox for Java

The IBM Toolbox for Java allows you to access iSeries resources, data, and programs through Java applets, servlets, or applications.

To set up IBM Toolbox for Java do these tasks:

1. Ensure that your workstation meets the requirements for IBM Toolbox for Java.
2. Ensure that your server meets the requirements for IBM Toolbox for Java.
3. Install the program files.

You also need to consider the following:

- Configuring an HTTP server for use with IBM Toolbox for Java if you want to use applets or servlets from an iSeries server that uses IBM Toolbox for Java classes served from the same iSeries. If you are interested in working with Secure Sockets Layer (SSL), you must configure an HTTP server.
- Copying the IBM Toolbox for Java class files on your workstation for information on copying files to your workstation. The class files are all contained in jar or zip files. For more information about using jar files, see Jar files.
- »Specify system properties to configure various aspects of the IBM Toolbox for Java, including the definition of a proxy server or a level of tracing.«

Workstation requirements for IBM Toolbox for Java

To run IBM Toolbox for Java, your workstation must meet the requirements listed below.

Requirements for running Java applications

- A Java virtual machine that fully supports »JDK 1.1.8« or any later JDK, including Java 2. If your program uses the Graphical Toolbox or classes in the vaccess package, Swing 1.1 is also required. The following environments have been tested:
  - Windows 98
  - Windows 95
  - Windows NT Workstation 4.0
  - »AIX Version 4.3.3.1«
  - »Sun Solaris Version 5.7«
  - »OS/400 Version 4 Release 4 or later«
  - OS/2 Version 4 Release 5
  - Linux 5.2
- TCP/IP installed and configured

Requirements for running Java applets

- A browser that has a compatible Java virtual machine. The following environments have been tested:
  - »Netscape Communicator 4.7«
  - »Microsoft Internet Explorer 5«
- TCP/IP installed and configured
- The workstation must connect to a server that is running » OS/400 V4R3 or later«.

IBM Toolbox for Java switched to support Swing 1.1 in V4R5, and this release continues that support. Switching to Swing required programming changes in the IBM Toolbox for Java classes. So, if your programs use the Graphical Toolbox or the vaccess classes from releases before V4R5, you will need to change your programs as well.

In addition to a programming change, the Swing classes must be in the CLASSPATH when the program is run. The Swing classes are part of the Java 2 platform. If you don't have the Java 2 platform, you can download the Swing 1.1 classes from Sun Microsystems, Inc.

OS/400 requirements for running IBM Toolbox for Java

To efficiently run IBM Toolbox for Java on your iSeries system, you need to understand the following issues:

- Compatibility with different levels of OS/400
- Native optimizations when running on OS/400 JVM
- Dependencies on OS/400 options and other licensed programs

Compatibility with different levels of OS/400

Because you can use IBM Toolbox for Java both on your server and your client, the compatibility issues affect both running on a server and connecting from a client back to a server.

Running IBM Toolbox for Java, Version 5 Release 1, on your servers

To install IBM Toolbox for Java (licensed program 5722-JC1 V5R1M0) on an iSeries system, the server must be running one of the following:

- OS/400 Version 5 Release 1
- OS/400 Version 4 Release 5
- OS/400 Version 4 Release 4

You can install only one version of the IBM Toolbox for Java licensed program on the system. To install a different version, first remove the existing IBM Toolbox for Java licensed program.

Using IBM Toolbox for Java to connect from a client back to the server

You can use different versions of IBM Toolbox for Java on a client and on the server to which you are connecting. To use IBM Toolbox for Java, Version 5 Release 1, to access data and resources on an iSeries system, the **server to which you are connecting** must be running one of the following:

- OS/400 Version 5 Release 1
- OS/400 Version 4 Release 5
- OS/400 Version 4 Release 4
- OS/400 Version 4 Release 3

»The table below shows how different versions of IBM Toolbox for Java are compatible with different releases of OS/400.

Table: Compatibility requirements for installing on and connecting back to different versions of OS/400

Toolbox modification	Ships with OS/400	LPP	Installs on OS/400	Connects back to OS/400
Mod 0	V4R2	5763-JC1 V3R2M0	V3R2 and above	V3R2 and above
Mod 1	V4R3	5763-JC1 V3R2M1	V3R2 and above	V3R2 and above
Mod 2	V4R4	5769-JC1 V4R2M0	V4R2 and above	V4R2 and above
Mod 3	V4R5	5769-JC1 V4R5M0	V4R3 and above	V4R2 and above
Mod 4	V5R1	5722-JC1 V5R1M0	V4R4 and above	V4R3 and above

Native optimizations when running on the iSeries JVM

Native optimizations are a set of functions that affect operation of IBM Toolbox for Java when running on the iSeries JVM. The optimizations are:

- Signon: When no userid or password is specified in the AS400 object, the userid and password of the current job are used
- Directly calling OS/400 APIs instead of making socket calls to host servers:
    - Record-level database access, data queues and user space when security requirements are met.
    - Program call and command call when security requirements and thread safety requirements are met.
    - Native JDBC driver when your Java program is running on the iSeries JVM and the database is on the same machine. In this case, the Toolbox for Java driver passes requests to the native driver.

The optimizations exist to make the IBM Toolbox for Java classes work the way a user would expect them to work when running on OS/400.

Note: It is very important to understand that your Java programs use native optimizations only when you use the version of IBM Toolbox for Java that matches the version of OS/400 on your server.

No change to the Java application is needed to get the optimizations. IBM Toolbox for Java automatically enables the optimizations when appropriate.

When the versions of IBM Toolbox for Java and OS/400 do not match, native optimizations are not available. In this case, IBM Toolbox for Java works as if it is running on a client.

The table below shows which versions of IBM Toolbox for Java and OS/400 you must run to get use native optimizations. This table documents only compatibility issues that affect native optimizations. The previous table shows general compatibility issues between IBM Toolbox for Java and OS/400.

Table: Compatibility requirements for using native optimizations

Level of OS/400	**Level of IBM Toolbox for Java**				
	V3R2M0	V3R2M1	V4R2M0	V4R5M0	V5R1M0
V4R2	No IBM Toolbox for Java performance enhancements are available.				
V4R3		X	X		
V4R4		X	X		
V4R5				X	
V5R1					X

An 'X' indicates that performance enhancements are enabled for a given configuration of IBM Toolbox for Java and OS/400.

In order to gain the performance improvements, you need to make sure to use the jar file that includes OS/400 native optimizations. For more information, see Note 1 in Jar files.«

Dependencies on OS/400 and other licensed programs

Using IBM Toolbox for Java may require you to address some dependencies that it has on OS/400 and on other licensed programs.

Dependencies on OS/400

Running IBM Toolbox for Java in a client/server environment requires that you enable the QUSER user profile, start the host servers, and have TCP/IP running. Perform these actions from an iSeries command line by running the following commands:

1. Type **DSPUSRPRF USRPRF(QUSER)** and press ENTER to enable the QUSER user profile. The resulting display shows the status for QUSER.
2. Type **STRHOSTSVR** and press ENTER to start the OS/400 host servers.

3. Type **STRTCPSVR SERVER(*DDM)** to start the TCP/IP server (with *DDM specified for the Server parameter).

Note: To run TCP/IP, you must have the TCP/IP Connectivity Utilities for AS/400 (licensed program » 5722-TC1«) installed on the server. For more information on host server options and TCP/IP, see the TCP/IP topic in the iSeries Information Center.

Dependencies on other licensed programs

When you want to use the spooled file viewer functions (SpooledFileViewer class) of IBM Toolbox for Java, ensure that you have installed host option 8 (AFP Compatibility Fonts) on your server.

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

When you want to use Secure Sockets Layer (SSL), ensure that you have installed the following:

- IBM HTTP Server for iSeries licensed program, » 5722-DG1«
- OS/400 Option 34 (Digital Certificate Manager)
- One of the IBM Cryptographic Access Provider for iSeries licensed programs:
  - IBM Cryptographic Access Provider 56-bit for iSeries, » 5722-AC2«
  - IBM Cryptographic Access Provider 128-bit for iSeries, » 5722-AC3«
- One of the IBM iSeries Client Encryption licensed programs:
  - IBM iSeries Client Encryption (56-bit), » 5722-CE2«
  - IBM iSeries Client Encryption (128-bit), » 5722-CE3«

»The V5R1 version of IBM Toolbox for Java requires that you use the V5R1 version of Client Encryption. The 128-bit version of Client Encryption is compatible with both the 56- and 128-bit versions of Cryptographic Access Provider listed above. In other words, when 5722-CE3 is installed, you can use 5722-AC3 or 5722-AC2.«

Note: You can use SSL support only when connecting to V4R4 and later iSeries systems. For more information on SSL, see Secure Sockets Layer.

Installing IBM Toolbox for Java on iSeries

To install IBM Toolbox for Java licensed program, you must be on »V4R4« or later. Then follow these steps:

1. On an iSeries command line, enter **GO LICPGM**.
2. Select **11. Install licensed program**.
3. »Select **5722-JC1 IBM Toolbox for Java**.«

For more information on installing licensed programs, see the Information Center topic, Managing software and licensed programs.

Configuring an HTTP server for use with IBM Toolbox for Java

If you want to use applets, servlets, SSL, or the AS400ToolboxInstaller class on the iSeries system, you must set up an HTTP server and install the class files on the iSeries system. For more information on the IBM HTTP Server, see the IBM HTTP Server for AS/400 Webmaster's Guide, GC41-5434, at the following URL: http://www.iseries.ibm.com/products/http/docs/doc.htm. The Webmaster's Guide is available in both HTML and PDF formats.

For information on the Digital Certificate Manager and how to create and work with digital certificates using the IBM HTTP Server, see Digital Certificate Management.

Copying the IBM Toolbox for Java class files on your workstation

Copying the class files to your workstation allows you to serve the files from your workstation. You can use the AS400ToolboxInstaller class or rely on existing mechanisms for obtaining server updates on your workstation.

»The class files are packaged in various jar files. You need to copy one or more of these jar files to your workstation. For more information, see Jar files. The example below assumes you want to copy jt400.jar, which contains the core IBM Toolbox for Java classes.«

Before you copy the files from the server to your workstation, you must decide if you want to use the AS400ToolboxInstaller class or manually copy the jar file.

- For more information about using the AS400ToolboxInstaller class to copy the jar files, see Client installation and update classes .

To manually copy the jar file, complete the following steps:

1. Find the jt400.jar file in the following directory:
   /QIBM/ProdData/HTTP/Public/jt400/lib
2. Copy jt400.jar from the server to your workstation. You can do this in a variety of ways:
   - Use Client Access/400 to map a network drive on your workstation to the server, then copy the file.
   - Use file transfer protocol (FTP) to send the file (making sure to use binary mode) to your workstation.
3. Update the CLASSPATH environment variable of your workstation.
   - For example, if you are using Windows NT and you copied jt400.jar to C:\jt400\lib, add the following string to the end of your CLASSPATH:

```
;C:\jt400\lib\jt400.jar
```

IBM Toolbox for Java access classes

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

The following classes provide access to iSeries and AS/400e resources:

- AS400 - manages sign-on information, creates and maintains socket connections, and sends and receives data
- SecureAS400 class - enables you to use an AS400 object when sending or receiving encrypted data
- »AS400JPing - allows your Java program to query the host servers to see which services are running and which ports are in service«
- »BidiTransform - enables you to do your own conversions of bidirectional text«
- Command call - runs iSeries batch commands
- »Connection pool - manages a pool of AS400 objects, which is used to share connections and manage the number of connections a user can have to an iSeries server«
- Data area - creates, accesses, and deletes data areas
- Data conversion and description - converts and handles data, and describes the record format of a buffer of data
- Data queues - creates, accesses, changes, and deletes data queues
- Digital certificates - manages digital certificates on iSeries servers
- »Environment variable - manages iSeries environment variables«
- »Event log - provides a way to log exceptions and messages independent of the device used to display them«
- Exceptions - throws errors when, for example, device errors or programming errors occur
- FTP - provides you with a programmable interface to FTP functions
- Integrated file system - accesses files, opens files, opens input and output streams, and lists the contents of directories
- Java application call - calls a Java program on an iSeries server that runs on the iSeries Java virtual machine
- JDBC - accesses DB2 UDB for iSeries data
- Jobs - accesses iSeries jobs and job logs
- Messages - accesses messages and message queues on the iSeries system
- »NetServer configuration - accesses and modifies the state and configuration of the AS/400 NetServer«
- Permission - displays and changes authorities in AS400 objects
- Print - manipulates iSeries print resources
- »Product license - manage licenses for iSeries products«
- Program call - calls an iSeries program
- QSYS object path name - represents objects in the iSeries integrated file system
- Record-level access - creates, reads, updates, and deletes iSeries files and members
- Service program call - calls an iSeries service program
- System status - displays system status information and allows access to system pool information
- System values - retrieves and changes system values and network attributes
- Trace (serviceability) - logs trace points and diagnostic messages
- Users and groups - accesses iSeries users and groups
- User space - accesses an iSeries user space

Note: When possible, use a resource class instead of an access class. Resource classes provide a generic framework and a consistent programming interface for working with various iSeries objects and lists.

AS400 class

The AS400 class manages the following:

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

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

The AS400 object handles connections, user IDs, and passwords differently when it is running in the iSeries Java virtual machine. For more information, see iSeries Java virtual machine.

See managing connections for information on managing connections to the iSeries through the AS400 object. »See AS400ConnectionPool for information on reducing initial connect time by requesting connections from a connection pool.«

AS400 class provides the following sign-on functions:

- Authenticate the user profile
- Get profile token credential
- Manage default user IDs
- Cache passwords
- Prompt for user ID
- Change a password
- Get the version and release of the iSeries

For information about using an AS400 object when sending or receiving encrypted data, see the SecureAS400 class.

Secure AS/400 Class

When an AS400 object communicates with the server, user data (except the user password) is sent unencrypted to the server. So, IBM Toolbox for Java objects associated with an AS400 object exchange data with the server over a normal connection.

When you want to use IBM Toolbox for Java to exchange sensitive data with the server, you can encrypt data by using Secure Sockets Layer (SSL). Use the SecureAS400 object to designate which data you want to encrypt. IBM Toolbox for Java objects associated with a SecureAS400 object exchange data with the server over a secure connection.

The SecureAS400 class is a subclass of the AS400 class.

You can set up a secure AS/400 connection by creating an instance of a SecureAS400 object as indicated below:

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

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

```
 // Create a secure AS400 object.  This is the only statement that changes
 // from the non-SSL case.
SecureAS400 sys = new SecureAS400("mySystem.myCompany.com");

// Create a command call object
CommandCall cmd = new CommandCall(sys, "myCommand");

// Run the commands.  A secure connection is made when the
// command is run.  All the information that passes between the
// client and server is encrypted.
cmd.run();
```

For more information, see secure sockets layer.

»AS400JPing

The AS400JPing allows your java program to query the host servers to see which services are running and which ports are in service. To query the servers from a command line, use the JPing class.

The AS400JPing class provides several methods:

- Ping the server
- Ping a specific service on the server
- Set a PrintWriter object to which you want to log ping information
- Set the time out for the ping operation

Example: Using AS400JPing within a Java program to ping the iSeries Remote Command Service:

```
AS400JPing pingObj = new AS400JPing("myAS400", AS400.COMMAND, false);
  if (pingObj.ping())
    System.out.println("SUCCESS");
  else
    System.out.println("FAILED");
```

«

»BidiTransform class

The AS400BidiTransform class provides layout transformations that enable you to convert bidirectional text in iSeries format (after first converting it to Unicode) to bidirectional text in Java format, or from Java format to iSeries format.

The AS400BidiTransform class allows you to:

- Get and set the system CCSID
- Get and set the string type of iSeries data
- Get and set the string type of Java data
- Convert data from a Java layout to iSeries
- Convert data from an iSeries layout to Java

Example: Using the AS400BidiTransform class to transform bidirectional text

The following example shows how you can use the AS400BidiTransform class to transform bidirectional text:

```
// Java data to AS/400 layout:
AS400BidiTransform abt;
abt = new AS400BidiTransform(424);
String dst = abt.toAS400Layout("some bidirectional string");
```

«

Command call

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

Input to CommandCall is as follows:

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

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

Using the CommandCall class causes the AS400 object to connect to the iSeries.

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

```
                      // Create an AS400 object.
    AS400 sys = new AS400("mySystem.myCompany.com");

                      // Create a command call object. This
                      // program sets the command to run
                      // later. It could set it here on the
                      // constructor.
    CommandCall cmd = new CommandCall(sys);

                      // Run the CRTLIB command
    cmd.run("CRTLIB MYLIB");

                      // Get the message list which
                      // contains the result of the
                      // command.
    AS400Message[] messageList = cmd.getMessageList();

                      // ... process the message list.

                      // Disconnect since I am done sending
                      // commands to the AS/400
    sys.disconnectService(AS400.COMMAND);
```

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

»The default behavior is for the IBM Toolbox for Java to look up the thread safety for the command on the system. If threadsafe, the command is run on-thread. You can suppress the run-time lookup by explicitly specifying thread-safety for the command by using using the setThreadSafe() method.«

Example

Run a command that is specified by the user.

»Connection pooling

Use a connection pool to share and manage connections to an iSeries server.

In terms of performance, connecting to the server is an expensive operation. Using a connection pool can increase performance by enabling you to use, reuse, and share existing connections instead of repeatedly connecting and disconnecting.

The AS400ConnectionPool class manages a pool of AS400 objects. The AS400JDBCConnectionPool class represents a pool of AS/400JDBCConnections that are available for use by a Java program.

A connection pool of either type keeps track of the number of connections it creates. Using methods inherited from ConnectionPool, you can set several connection pool properties, including:

- the maximum number of connections that can be given out by a pool
- the maximum lifetime of a connection
- the maximum inactivity time of a connection

Retrieve a connection using an AS400ConnectionPool by specifying the system name, user id, the password (optional), and the service (optional).

Return connections to an AS400ConnectionPool by using the returnConnectionToPool() method.

Note: When connections are not returned to the pool, the connection pool continues to grow in size and connections are not reused.

Ensure that the time used to create connections occurs when the pool is created by filling the pool with active (preconnected) connections. A user can then retrieve a connection from the pool, use the connection in an application, and return the connection to the pool for reuse. Note that it is the responsibility of each application to return connections to the pool for reuse.

See managing connections for more information about managing when a connection to the iSeries is opened when using the AS400ConnectionPool classes.

Example: Using an AS400ConnectionPool to reuse AS400 objects«

Data area

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

Using the DataArea class, you can do the following:

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

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

CharacterDataArea

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

Using the CharacterDataArea class, you can do the following:

- Clear the data area so that it contains all blanks.
- Create a character data area on the AS/400 system using default property values
- Create a character data area with specific attributes
- Delete the data area from the AS/400 system where the data area exists
- Return the integrated file system path name of the object represented by the data area.
- Read all of the data that is contained in the data area
- Read a specified amount of data from the data area starting at offset 0 or the offset that you specified
- Set the fully qualified integrated file system path name of the data area
- Write data to the beginning of the data area
- Write a specified amount of data to the data area starting at offset 0 or the offset that you specified

DecimalDataArea

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

Using the DecimalDataArea class, you can do the following:

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

- Write data to the beginning of the data area

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

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

LocalDataArea

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

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

Using the LocalDataArea class, you can do the following:

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

LogicalDataArea

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

Using the LogicalDataArea class, you can do the following:

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

DataAreaEvent

The DataAreaEvent class represents a data area event.

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

- Get the identifier for the event

DataAreaListener

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

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

- Clear
- Create
- Delete
- Read
- Write

Data conversion and description

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

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

Data types

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

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

Conversion specifying a record format

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

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

Conversion through record format uses three classes:

- FieldDescription classes identify a field or parameter with a data type and a name.
- A RecordFormat class describes a group of fields.
- A Record class joins the description of a record (in the RecordFormat class) with the actual data.
- »A LineDataRecordWriter class writes a record to an OutputStream in line data format«

Examples

Two examples illustrate using the record format conversion classes with data queues:

- Example: Using the Record and RecordFormat classes to put data on a queue
- »Example: Using the FieldDescription, RecordFormat, and Record classes«

Data queues

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

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

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

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

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

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

Sequential and keyed data queues

The data queue classes support the following data queues:

- Sequential data queues
- Keyed data queues

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

When data is read from a data queue, the data is placed in a DataQueueEntry object. This object holds the data for both keyed and sequential data queues. Additional data available when reading from a keyed data queue is placed in a KeyedDataQueueEntry object that extends the DataQueueEntry class.

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

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

```
                       // Create an AS400 object
     AS400 sys = new AS400("mySystem.myCompany.com");

                       // Create the DataQueue object
     DataQueue dq = new DataQueue(sys, "/QSYS.LIB/MYLIB.LIB/MYQUEUE.DTAQ");

                       // read data from the queue
```

```
DataQueueEntry dqData = dq.read();

                  // get the data out of the DataQueueEntry object.
byte[] data = dqData.getData();

                  // ... process the data

                  // Disconnect since I am done using data queues
sys.disconnectService(AS400.DATAQUEUE);
```

Sequential data queues

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

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

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

Examples

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

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

Keyed data queues

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

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

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

Examples

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

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

Digital certificates

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

Digital certificates comprise the following:

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

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

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

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

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

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

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

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

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

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

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

Listing certificates

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

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

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

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


                  // Create the certificate object.
AS400CertificateVldlUtil certificateList =
         new AS400CertificateVldlUtil(sys, "/QSYS.LIB/MYLIB.LIB/CERTLIST.VLDL");


                  // Create the certificate attribute
                  // list. We only want certificates
                  // for a single person so the list
                  // consists of only one element.
AS400CertificateAttribute[] attributeList = new AS400CertificateAttribute[1];
attributeList[0] = new AS400CertificateAttribute(AS400CertificateAttribute.SUBJECT_COMMON_NAME, "Jane Doe");


                  // Retrieve the list that matches
                  // the criteria. User space "myspace"
                  // in library "mylib" will be used
                  // for storage of the certificates.
                  // The user space must exist before
                  // calling this API.
int count = certificateList.listCertificates(attributeList,
                                             "/QSYS.LIB/MYLIB.LIB/MYSPACE.USRSPC");


                  // Retrieve the certificates from
                  // the user space.
AS400Certificates[] certificates = certificateList.getCertificates("/QSYS.LIB/MYLIB.LIB/MYSPACE.USRSPC", 0, 8);


                  // ... process the certificates
```

» EnvironmentVariable class

The EnvironmentVariable class and the EnvironmentVariableList class enable you to access and set iSeries **system-level** environment variables.

Each variable has unique identifiers: the system name and the environment variable name. Each environment variable is associated with a CCSID, which is by default the CCSID of the current job, which describes where the contents of the variable are stored.

Note: Environment variables are different than system values, although they are often used for the same purpose. See SystemValues for more information on how to access system values.

Use an EnvironmentVariable object to perform the following actions on an environment variable:

- Get and set the name
- Get and set the system
- Get and set the value (which allows you to change the CCSID)
- Refresh the value

Example: Creating, setting, and getting environment variables

The following example creates two EnvironmentVariables and sets and gets their values.

```
// Create the iSeries system object.
AS400 system = new AS400("mySystem");
// Create the foreground color environment variable and set it to red.
EnvironmentVariable fg = new EnvironmentVariable(system, "FOREGROUND");
fg.setValue("RED");
// Create the background color environment variable and get its value.
EnvironmentVariable bg = new EnvironmentVariable(system, "BACKGROUND");
String background = bg.getValue();
```

«

Exceptions

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

Most exceptions contain three pieces of information:

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

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

```
                  // ... all the setup work to delete
                  // a file on the AS/400 through the
                  // IFSFile class is done. Now try
                  // deleting the file.
try
{
   aFile.delete();

}

                  // The delete failed.
catch (ExtendedIOException e)
{
                  // Display the translated string
                  // containing the reason that the
                  // delete failed.
   System.out.println(e);

                  // Get the return code out of the
                  // exception and display additional
                  // information based on the return
                  // code.
   int rc = e.getReturnCode()

   switch (rc)
   {
      case ExtendedIOException.FILE_IN_USE:
         System.out.println("Delete failed, file is in use "):
         break;

      case ExtendedIOException.PATH_NOT_FOUND:
         System.out.println("Delete failed, path not found ");
         break;

                  // ... for every specific error you
                  // want to track

      default:
         System.out.println("Delete failed, rc = ");
         System.out.println(rc);
   }
}
```

See exceptions inheritance structure for more information about exceptions.

Exceptions thrown by the IBM Toolbox for Java access classes

The following table describes when various exceptions are thrown.

Exception	Description
AS400Exception	Thrown when the iSeries system returns an error message.
AS400SecurityException	Thrown when a security or authority error occurs.
ConnectionDroppedException	Thrown when the connection is dropped unexpectedly.
» ConnectionPoolException	Thrown when a problem occurs with the connection pool.«
ErrorCompletingRequestException	Thrown when an error occurs before the request is completed.
ExtendedIOException	Thrown when an error occurs while communicating with the iSeries.
ExtendedIllegalArgumentException	Thrown when an argument is not valid.
ExtendedIllegalStateException	Thrown when the AS400 object is not in the proper state to perform the operation.
IllegalObjectTypeException	Thrown when the AS400 object is not of the required type.
IllegalPathNameException	Thrown when an integrated file system path name is not valid.
InternalErrorException	Thrown when an internal problem occurs. When this type of exception is thrown, contact your service representative to report the problem.
»LicenseException	Thrown when an error condition occurs while trying to retrieve a license.«
ObjectAlreadyExistsException	Thrown when the AS400 object already exists.
ObjectDoesNotExistException	Thrown when the AS400 object does not exist.
»ProxyException	Thrown when an error occurs while communicating with the proxy server.«
RequestNotSupportedException	Thrown when the requested function is not supported because the iSeries system is not at the correct level.
ReturnCodeException	An **interface** for exceptions that contain a return code. The return code is used to further identify the cause of an error.
ServerStartupException	Thrown when the iSeries server cannot be started.

See Inheritance structure for exceptions for more information about exceptions thrown by the IBM Toolbox for Java.

Inheritance structure for exceptions

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

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

FTP class

The FTP class provides a programmable interface to FTP functions. You no longer have to use java.runtime.exec() or tell your users to run FTP commands in a separate application. That is, you can program FTP functions directly into your application. So, from within your program, you can do the following:

- Connect to an FTP server
- Send commands to the server
- List the files in a directory
- Get files from the server **and**
- Put files to the server

For example, with the FTP class, you can copy a set of files from a directory on a server:

```
FTP client = new FTP("myServer", "myUID", "myPWD");
client.cd("/myDir");
client.setDataTransferType(FTP.BINARY);
String [] entries = client.ls();

for (int i = 0; i < entries.length; i++)
{
  System.out.println("Copying " + entries[i]);
  try
  {
    client.get(entries[i], "c:\\ftptest\\" + entries[i]);
  }
  catch (Exception e)
  {
    System.out.println("  copy failed, likely this is a directory");
  }
}

client.disconnect();
```

FTP is a generic interface that works with many different FTP servers. Therefore, it is up to the programmer to match the semantics of the server.

FTP subclass

While the FTP class is a generic FTP interface, the AS400FTP subclass is written specifically for the FTP server on the AS/400. That is, it understands the semantics of the FTP server on the AS/400, so the programmer doesn't have to. For example, this class understands the various steps needed to transfer an AS/400 save file to the AS/400 and performs these steps automatically. AS400FTP also ties into the security facilities of the IBM Toolbox for Java. As with other IBM Toolbox for Java classes, AS400FTP depends on the AS400 object for system name, user ID, and password.

The following example puts a save file to the AS/400. Note the application does not set data transfer type to binary or use Toolbox CommandCall to create the save file. Since the extension is .savf, AS400FTP class detects the file to put is a save file so it does these steps automatically.

```
AS400 system = new AS400();
AS400FTP ftp = new AS400FTP(system);
ftp.put("myData.savf", "/QSYS.LIB/MYLIB.LIB/MYDATA.SAVF");
```

Integrated file system

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

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

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

- Specify a file sharing mode to deny access to the file while it is in use
- Specify a file creation mode to open, create, or replace the file
- Lock a section of the file and deny access to that part of the file while it is in use
- List the contents of a directory more efficiently
- »Cache the contents of a directory to improve performance by limiting calls to AS/400«
- Determine the number of bytes available on the iSeries file system
- Allow a Java applet to access files in the iSeries file system
- Read and write data as text instead of as binary data
- »Determine the type of the file object (logical, physical, save, and so on) when the object is in the QSYS.LIB file system «

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

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

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

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

The integrated file system classes are as follows.

Class	Description
IFSFile	Represents a file in the integrated file system
IFSJavaFile	Represents a file in the integrated file system (extends java.io.File)
IFSFileInputStream	Represents an input stream for reading data from an iSeries file
IFSTextFileInputStream	Represents a stream of character data read from a file
IFSFileOutputStream	Represents an output stream for writing data to an iSeries file
IFSTextFileOutputStream	Represents a stream of character data being written to a file
IFSRandomAccessFile	Represents a file on the iSeries for reading and writing data
IFSFileDialog	Allows the user to move within the file system and to select a file within the file system

Examples

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

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

IFSFile class

The IFSFile class represents an object in the iSeries integrated file system. The methods on IFSFile represent operations that are done on the object as a whole. You can use IFSFileInputStream, IFSFileOutputStream, and IFSRandomAccessFile to read and write to the file. The IFSFile class allows the Java program to do the following:

- Determine if the object exists and is a directory or a file
- Determine if the Java program can read from or write to a file
- Determine the length of a file
- Determine the permissions of an object and set the permissions of an object
- Create a directory
- Delete a file or directory
- Rename a file or directory
- Get or set the last modification date of a file
- List the contents of a directory
- »List the contents of a directory and save the attribute information to a local cache«
- Determine the amount of space available on the system
- Determine the type of the file object when it is in the QSYS.LIB file system

»You can get the list of files in a directory by using either the list() method or the listFiles() method:

- The listFiles() method caches information for each file on the initial call. After calling listFiles(), using other methods to query file details results in better performance because the information is retrieved from the cache. For example, calling isDirectory() on an IFSFile object returned from listFiles() does not require a call to the server.
- The list() method retrieves information on each file in a separate request to the server, making it slower and more demanding of server resources.

Note: Using listFiles() means that the information in the cache may eventually become stale, so you may need to refresh the data by calling listFiles() again.«

Examples

The following examples show how to use the IFSFile class:

- **Example:** Creating a directory
- **Example:** Using exceptions to track errors
- **Example:** Listing files with a .txt extension
- »**Example:** Using listFiles() to list the contents of a directory«

IFSJavaFile class

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

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

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

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

»You can get the list of files in a directory by using either the list() method or the listFiles() method:

- The listFiles() method performs better because it retrieves and caches information for each file on the initial call. After that, information on each file is retrieved from the cache.
- The list() method retrieves information on each file in a separate request, making it slower and more demanding of server resources.

Note: Using listFiles() means that the information in the cache eventually become stale, so you may need to refresh the data.«

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

```
// Work with /Dir/File.txt on the system flash.
AS400 as400 = new AS400("flash");
IFSJavaFile file = new IFSJavaFile(as400, "/Dir/File.txt");


// Determine the parent directory of the file.
String directory = file.getParent();


// Determine the name of the file.
String name = file.getName();


// Determine the file size.
long length = file.length();


// Determine when the file was last modified.
Date date = new Date(file.lastModified());


// Delete the file.
if (file.delete() == false)
{
  // Display the error code.
  System.err.println("Unable to delete file.");
}


try
{
  IFSFileOutputStream os = new IFSFileOutputStream(file.getSystem(),
                              file,
                              IFSFileOutputStream.SHARE_ALL,
                              false);
  byte[] data = new byte[256];
  int i = 0;
```

```
          for (; i < data.length; i++)
          {
            data[i] = (byte) i;
            os.write(data[i]);
          }
          os.close();
        }
        catch (Exception e)
        {
          System.err.println ("Exception: " + e.getMessage());
        }
```

IFSFileInputStream

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

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

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

The following example shows how to use the IFSFileInputStream class.

```
                  // Create an AS400 object.
AS400 sys = new AS400("mySystem.myCompany.com");

                  // Open a file object that
                  // represents the file.
IFSFileInputStream aFile =
              new IFSFileInputStream(sys,"/mydir1/mydir2/myfile");

                  // Determine the number of bytes in
                  // the file.
int available = aFile.available();

                  // Allocate a buffer to hold the data
byte[] data = new byte[10240];

                  // Read the entire file 10K at a time
for (int i = 0; i < available; i += 10240)
{
    aFile.read(data);
}

                  // Close the file.
aFile.close();
```

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

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

IFSTextFileInputStream

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

The following example shows how to use the IFSTextFileInputStream:

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

                  // Read the first four characters of
                  // the file.
String s = file.read(4);

                  // Display the characters read. Read
                  // the first four characters of the
                  // file. If necessary, the data is
                  // converted to unicode by the
                  // IFSTextFileInputStream object.
System.out.println(s);

                  // Close the file.
file.close();
```

IFSFileOutputStream

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

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

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

The following example shows how to use the IFSFileOutputStream class.

```
                  // Create an AS400 object
AS400 sys = new AS400("mySystem.myCompany.com");

                  // Open a file object that
                  // represents the file.
IFSFileOutputStream aFile =
              new IFSFileOutputStream(sys,"/mydir1/mydir2/myfile");

                  // Write to the file
byte i = 123;
aFile.write(i);

                  // Close the file.
aFile.close();
```

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

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

IFSTextFileOutputStream

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

The following example shows how to use IFSTextFileOutputStream:

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

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

                       // Close the file.
     file.close();
```

IFSRandomAccessFile

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

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

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

```
                  // Create an AS400 object.
AS400 sys = new AS400("mySystem.myCompany.com");

                  // Open a file object that represents
                  // the file.
IFSRandomAccessFile aFile =
              new IFSRandomAccessFile(sys,"/mydir1/myfile", "rw");

                  // Establish the data to write.
byte i = 123;

                  // Write to the file 10 times at 1K
                  // intervals.
for (int j=0; j<10; j++)
{
                  // Move the current offset.
   aFile.seek(j * 1024);

                  // Write to the file. The current
                  // offset advances by the size of
                  // the write.
   aFile.write(i);
}

                  // Close the file.
aFile.close();
```

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

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

IFSFileDialog

The IFSFileDialog class allows you to traverse the file system and select a file. This class uses the IFSFile class to traverse the list of directories and files in the integrated file system on the AS/400. Methods on the class allow a Java program to set the text on the push buttons of the dialog and to set filters. Note that an IFSFileDialog class based on Swing 1.1 is also available.

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

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

```
                  // Create an AS400 object.
AS400 sys = new AS400("mySystem.myCompany.com");

                  // Create a dialog object setting
                  // the text of the dialog's title
                  // bar and the AS/400 to traverse.
IFSFileDialog dialog = new IFSFileDialog(this, "Title Bar Text", sys);

                  // Create a list of filters then set
                  // the filters in the dialog. The
                  // first filter will be used when
                  // the dialog is first displayed.
FileFilter[] filterList = {new FileFilter("All files (*.*)", "*.*"),
                           new FileFilter("HTML files (*.HTML", "*.HTM")};

dialog.setFileFilter(filterList, 0);

                  // Set the text on the buttons of
                  // the dialog.
dialog.setOkButtonText("Open");
dialog.setCancelButtonText("Cancel");

                  // Show the dialog. If the user
                  // selected a file by pressing the
                  // Open button, get the file the
                  // user selected and display it.
if (dialog.showDialog() == IFSFileDialog.OK)
   System.out.println(dialog.getAbsolutePath());
```

JavaApplicationCall

The JavaApplicationCall class provides you with the ability to easily run a Java program residing on the AS/400 from a client with the Java virtual machine for AS/400.

After establishing a connection to the AS/400 from the client, the JavaApplicationCall class lets you configure the following:

1. Set the CLASSPATH environment variable on the AS/400 with the setClassPath() method
2. Define your program's parameters with the setParameters() method
3. Run the program with run()
4. Send input from the client to the Java program. The Java program reads the input via standard input which is set with the sendStandardInString() method. You can redirect standard output and standard error from the Java program to the client via the getStandardOutString() and getStandardErrorString()

JavaApplicationCall is a class you call from your Java program. However, the IBM Toolbox for Java also provides utilities to call AS/400 Java programs. These utilities are complete Java programs you can run from your workstation. See RunJavaApplication class for more information.

Example

This example shows you how to run a program on the AS/400 from the client that outputs "Hello World!".

JDBC

JDBC$^{(TM)}$ is an application programming interface (API) included in the Java platform that enables Java programs to connect to a wide range of databases.

The IBM Toolbox for Java JDBC driver allows you to use JDBC API interfaces to issue structured query language (SQL) statements to and process results from AS/400 databases. The server also has a JDBC driver that is optimized for use on the server.

»When you are running an application on the server, your application often uses the IBM Developer Kit for Java JDBC Driver by default. You can override this by using the driver property.«

Different versions of JDBC

»Different versions of the JDBC API exist, and the IBM Toolbox for Java JDBC driver supports the following versions:

- JDBC 1.2 API (the java.sql package) is included in the Java Platform 1.1 core API and JDK 1.1.
- JDBC 2.1 core API (the java.sql package) is included in both the Java 2 Platform, Standard Edition (J2SE) and the Java 2 Platform Enterprise Edition (J2EE).
- JDBC 2.0 Optional Package API (the javax.sql package) is included in J2EE and is available as a separate download from Sun. These extensions were formerly named the JDBC 2.0 Standard Extension API.«

Supported interfaces

The table below lists the supported JDBC interfaces and the API required to use them:

Supported interface	API required
Blob provides access to binary large objects (BLOBs)	JDBC 2.1 core
CallableStatement runs SQL stored procedures	JDK 1.1
Clob provides access to character large objects (CLOBs)	JDBC 2.1 core
Connection represents a connection to a specific database	JDK 1.1
» ConnectionPoolDataSource represents a factory for pooled AS400JDBCPooledConnection objects	JDBC 2.0 Optional Package «
DatabaseMetaData provides information about the database as a whole.	JDK 1.1
»DataSource represents a factory for database connections.	JDBC 2.0 Optional Package «
Driver creates the connection and returns information about the driver version.	JDK 1.1
PreparedStatement runs compiled SQL statements	JDK 1.1
ResultSet provides access to a table of data that is generated by running a SQL query or DatabaseMetaData catalog method	JDK 1.1
ResultSetMetaData provides information about a specific ResultSet	JDK 1.1
»RowSet is a connected row set that encapsulates a ResultSet	JDBC 2.0 Optional Package «
Statement runs SQL statements and obtains the results	JDK 1.1
»XAConnection is a database connection which participates in global XA transactions	JDBC 2.0 Optional Package «
»XAResource is resource manager for use in XA transactions	JDBC 2.0 Optional Package «

We have included a table that lists JDBC properties for easy reference.

Examples

The following examples illustrate ways to use the IBM Toolbox for Java JDBC driver.

- Using the JDBC driver to create and populate a table
- Using the JDBC driver to query a table and output its contents
- »Using the JDBC 2.0 Optional Package extensions to create and update a table«

Jobs

The IBM Toolbox for Java Jobs classes (in the access package) allow a Java program to retrieve and change job information.

»**Note:** When possible, use a resource class instead a class from the access package. Resource classes provide a generic framework and a consistent programming interface for working with various AS400 objects and lists. The resource classes for working with jobs are RJob, RJobList, and RJobLog.«

Use the Jobs classes to work with the following type of job information:

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

The job classes in the access package are as follows:

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

Examples

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

Display the messages in a job log.

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

```
try {
    // Creates AS400 object.
    AS400 as400 = new AS400("systemName");
    // Constructs a Job object
    Job job = new Job(as400,"QDEV002");
    // Gets job information
    System.out.println("User of this job :" + job.getUser());
    System.out.println("CPU used :" + job.getCPUUsed();
    System.out.println("Job enter system date : " + job.getJobEnterSystemDate());
    // Sets cache mode
    job.setCacheChanges(true);
    // Changes will be store in the cache.
    job.setRunPriority(66);
    job.setDateFormat("*YMD");
    // Commit changes. This will change the value on the iSeries.
    job.commitChanges();
    // Set job information to system directly(without cache).
    job.setCacheChanges(false);
    job.setRunPriority(60);
} catch (Exception e)
{
    System.out.println(quot;error :" + e)
}
```

Job

The job class (in the access package) allows a java program to retrieve and change iSeries jobs information.

Note: When possible, use a resource class instead of a class from the access package. Resource classes provide a generic framework and a consistent programming interface for working with various AS400 objects and lists. The resource classes for working with jobs are RJob, RJobList, and RJobLog.

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

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

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

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

JobList

You can use JobList class (in the access package) to list iSeries jobs.

»**Note:** When possible, use a resource class instead of a class from the access package. Resource classes provide a generic framework and a consistent programming interface for working with various AS400 objects and lists. The resource classes for working with jobs are RJob, RJobList, and RJobLog.«

With the JobList class, you can retrieve the following:

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

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

The following example lists all active jobs on the system:

```
                  // Create an AS400 object. List the
                  // jobs on this iSeries.
AS400 sys = new AS400("mySystem.myCompany.com");

                  // Create the job list object.
JobList jobList = new JobList(sys);

                  // Get the list of active jobs.
Enumeration list = jobList.getJobs();

                  // For each active job on the system
                  // print job information.
while (list.hasMoreElements())
{
    Job j = (Job) list.nextElement();

    System.out.println(j.getName() + "." +
                       j.getUser() + "." +
                       j.getNumber());
}
```

JobLog

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

»**Note:** When possible, use a resource class instead of a class from the access package. Resource classes provide a generic framework and a consistent programming interface for working with various AS400 objects and lists. The resource classes for working with jobs are RJob, RJobList, and RJobLog.«

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

```
                  // ... Setup work to create an AS400
                  // object and a jobList object has
                  // already been done

                  // Get the list of active jobs on
                  // the iSeries
Enumeration list = jobList.getJobs();

                  // Look through the list to find a
                  // job for the specified user.
while (list.hasMoreElements())
{
   Job j = (Job) list.nextElement();

   if (j.getUser().trim().equalsIgnoreCase(userID))
   {
                  // A job matching the current user
                  // was found. Create a job log
                  // object for this job.
      JobLog jlog = new JobLog(system,
                               j.getName(),
                               j.getUser(),
                               j.getNumber());

                  // Enumerate the messages in the job
                  // log then print them.
      Enumeration messageList = jlog.getMessages();

      while (messageList.hasMoreElements())
      {
          AS400Message message = (AS400Message) messageList.nextElement();
          System.out.println(message.getText());
      }

   }
}
```

Messages

AS400Message

AS400Message object allows the Java program to retrieve an iSeries message that is generated from a previous operation (for example, from a command call). From a message object, the Java program can retrieve the following:

- The iSeries library and message file that contain the message
- The message ID
- The message type
- The message severity
- The message text
- The message help text

The following example shows how to use the AS400Message object:

```
                       // Create a command call object.
     CommandCall cmd = new CommandCall(sys, "myCommand");

                       // Run the command
     cmd.run();

                       // Get the list of messages that are
                       // the result of the command that I
                       // just ran
     AS400Message[] messageList = cmd.getMessageList();

                      // Iterate through the list
                      // displaying the messages
     for (int i = 0; i < messageList.length; i++)
     {
        System.out.println(messageList[i].getText());
     }
```

Examples

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

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

QueuedMessage

The QueuedMessage class extends the AS400Message class.

»**Note:** When possible, use a resource class instead of a class from the access package. Resource classes provide a generic framework and a consistent programming interface for working with various AS400 objects and lists. The resource class for working with queued messages is RQueuedMessage.«

The QueuedMessage class accesses information about a message on an iSeries message queue. With this class, a Java program can retrieve:

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

- The message reply status

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

```
                // The message queue is on this iSeries.
AS400 sys = new AS400(mySystem.myCompany.com);

                // Create the message queue object.
                // This object will represent the
                // queue for the current user.
MessageQueue queue = new MessageQueue(sys, MessageQueue.CURRENT);

                // Get the list of messages currently
                // in this user's queue.
Enumeration e = queue.getMessages();

                // Print each message in the queue.
while (e.hasMoreElements())
{
   QueuedMessage msg = e.getNextElement();
   System.out.println(msg.getText());
}
```

MessageFile

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

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

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

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

MessageQueue

The MessageQueue class allows a Java program to interact with an iSeries message queue.

»**Note:** When possible, use a resource class instead of a class from the access package. Resource classes provide a generic framework and a consistent programming interface for working with various iSeries objects and lists. The resource class for working with message queues is RMessageQueue.«

The MessageQueue class acts as a container for the QueuedMessage class. The getMessages() method, in particular, returns a list of QueuedMessage objects. The MessageQueue class can do the following:

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

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

```
                // The message queue is on this iSeries.
AS400 sys = new AS400(mySystem.myCompany.com);

                // Create the message queue object.
                // This object will represent the
                // queue for the current user.
MessageQueue queue = new MessageQueue(sys, MessageQueue.CURRENT);

                // Get the list of messages currently
                // in this user's queue.
Enumeration e = queue.getMessages();

                // Print each message in the queue.
while (e.hasMoreElements())
{
   QueuedMessage msg = e.getNextElement();
   System.out.println(msg.getText());
}
```

»NetServer

The NetServer class represents the NetServer service on an iSeries server. NetServer objects allow you to query and modify the state and configuration of the NetServer service.

For example, you can use the NetServer class to:

- Start or stop the NetServer
- Get a list of all current file shares and print shares
- Get a list of all current sessions
- Query and change attribute values (using methods inherited from ChangeableResource)

Note: In order to use the NetServer class, you need a server user profile that has *IOSYSCFG authority.

The NetServer class is an extension of ChangeableResource and Resource, so it provides a collection of "attributes" to represent the various NetServer values and settings. You query or change the attributes in order to access or change the configuration of your NetServer. Some of the NetServer attributes are:

- NAME
- NAME_PENDING
- DOMAIN
- ALLOW_SYSTEM_NAME
- AUTOSTART
- CCSID
- WINS_PRIMARY_ADDRESS

Pending attributes

Many of the NetServer attributes are pending (for example, NAME_PENDING). Pending attributes represent NetServer values that take effect the next time you start (or restart) the NetServer on the server.

When you have a pair of related attributes and one attribute is pending while the other is nonpending:

- The pending attribute is read/write, so you can change it
- The nonpending attribute is read-only, so you can query it but you can not change it

Other NetServer classes

Related NetServer classes allow you to get and set detailed information about specific connections, sessions, file shares, and print shares:

- NetServerConnection - represents a NetServer connection
- NetServerFileShare - represents a NetServer file server share
- NetServerPrintShare - represents a NetServer print server share
- NetServerSession - represents a NetServer session
- NetServerShare - represents a NetServer share

Example: Using a NetServer object to change the name of the NetServer

```
// Create a system object to represent the iSeries server.
AS400 system = new AS400("MYSYSTEM", "MYUSERID", "MYPASSWD");
// Create an object with which to query and modify the NetServer.
```

```
NetServer nServer = new NetServer(system);
// Set the "pending name" to NEWNAME.
nServer.setAttributeValue(NetServer.NAME_PENDING, "NEWNAME");
// Commit the changes.  This sends the changes to the server.
nServer.commitAttributeChanges();
// The NetServer name will get set to NEWNAME the next time the NetServer
// is ended and started.«
```

Permission classes

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

Permission class

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

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

Example

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

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

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

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

UserPermission class

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

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

The UserPermission class allows you to do the following:

- Determine if the user profile is a group profile
- Return the user profile name
- Indicate whether the user has authority
- Set the authority of authorization list management

Example

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

```
// Create a system object.
AS400 sys = new AS400("MYAS400", "USERID", "PASSWORD");

// Represent the permissions to an object on the system, such as a library.
Permission objectInQSYS = new Permission(sys, "/QSYS.LIB/FRED.LIB");

// Retrieve the various users/groups that have permissions set on that object.
Enumeration enum = objectInQSYS.getUserPermissions();
while (enum.hasMoreElements())
{
  // Print out the user/group profile names one at a time.
  UserPermission userPerm = (UserPermission)enum.nextElement();
  System.out.println(userPerm.getUserID());
}
```

DLOPermission

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

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

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

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

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

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

Example

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

```
// Create a system object.

AS400 sys = new AS400("MYAS400", "USERID", "PASSWORD");
// Represent the permissions to a DLO object.
Permission objectInQDLS = new Permission(sys, "/QDLS/MyFolder");

// Print the object pathname and retrieve its permissions.
System.out.println("Permissions on "+objectInQDLS.getObjectPath()+" are as follows:");
Enumeration enum = objectInQDLS.getUserPermissions();
while (enum.hasMoreElements())
{
  // For each of the permissions, print out the user profile name
  // and that user's authorities to the object.
  DLOPermission dloPerm = (DLOPermission)enum.nextElement();
  System.out.println(dloPerm.getUserID()+": "+dloPerm.getDataAuthority());
}
```

QSYSPermission

QSYSPermission is a subclass of the UserPermission class. QSYSPermission allows you to display and set the permission a user has for an object in the traditional AS/400 library structure stored in QSYS.LIB. An object stored in QSYS.LIB can set its authorities by setting a single object authority value or by setting individual object and data authorities.

Use the getObjectAuthority() method to display the current object authority or the setObjectAuthority() method to set the current object authority using a single value. The following table lists the valid values:

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

Use the appropriate set method to set the detailed object authority values on or off:

- setAlter()
- setExistence()
- setManagement()
- setOperational()
- setReference()

Use the appropriate set method to set the detailed data authority values on or off:

- setAdd()
- setDelete()
- setExecute()
- setRead()
- setUpdate()

The single authority actually represents a combination of the detailed object authorities and the data authorities. Selecting a single authority automatically turns on the appropriate detailed authorities. Likewise, selecting various detailed authorities changes the appropriate single authority values. The following table illustrates the relationships:

	Detailed Object Authority					**Detailed Data Authority**				
Basic Authority	**Opr**	**Mgt**	**Exist**	**Alter**	**Ref**	**Read**	**Add**	**Upd**	**Dlt**	**Exe**
All	**Y**	**Y**	**Y**	**Y**	**Y**	**Y**	**Y**	**Y**	**Y**	**Y**
Change	**Y**	n	n	n	n	**Y**	**Y**	**Y**	**Y**	**Y**
Exclude	n	n	n	n	n	n	n	n	n	n
Use	**Y**	n	n	n	n	**Y**	n	n	n	**Y**
Autl	Only valid with a specified authorization list and user (*PUBLIC). Detailed Object and Data authorities are determined by the list.									

"**Y**" refers to those authorities that can be assigned.
"n" refers to those authorities that cannot be assigned.

If a combination of detailed object authority and data authority does not map to a single authority value, then the single value becomes "User Defined." For more information on object authorities, refer to the AS/400 CL commands Grant Object Authority (GRTOBJAUT) and Edit Object Authority (EDTOBJAUT).

Example

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

```
// Create a system object.
AS400 sys = new AS400("MYAS400", "USERID", "PASSWORD");

// Represent the permissions to a QSYS object.
Permission objectInQSYS = new Permission(sys, "/QSYS.LIB/FRED.LIB");

// Print the object pathname and retrieve its permissions.
System.out.println("Permissions on "+objectInQSYS.getObjectPath()+" are as follows:");
Enumeration enum = objectInQSYS.getUserPermissions();
while (enum.hasMoreElements())
{
  // For each of the permissions, print out the user profile name
  // and that user's authorities to the object.
  QSYSPermission qsysPerm = (QSYSPermission)enum.nextElement();
  System.out.println(qsysPerm.getUserID()+": "+qsysPerm.getObjectAuthority());
}
```

RootPermission

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

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

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

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

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

Example

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

```
// Create a system object.
AS400 sys = new AS400("MYAS400", "USERID", "PASSWORD");

// Represent the permissions to an object in the root file system.
Permission objectInRoot = new Permission(sys, "/fred");

// Print the object pathname and retrieve its permissions.
System.out.println("Permissions on "+objectInRoot.getObjectPath()+" are as follows:");
Enumeration enum = objectInRoot.getUserPermissions();
while (enum.hasMoreElements())
{
  // For each of the permissions, print out the user profile name
  // and that user's authorities to the object.
  RootPermission rootPerm = (RootPermission)enum.nextElement();
  System.out.println(rootPerm.getUserID()+": "+rootPerm.getDataAuthority());
}
```

Print classes

Print objects include spooled files, output queues, printers, printer files, writer jobs, and Advanced Function Printing (AFP) resources, which include fonts, form definitions, overlays, page definitions, and page segments. AFP resources are accessible only on Version 3 Release 7 (V3R7) and later AS/400 systems. (Trying to open an AFPResourceList to a system that is running an earlier version than V3R7 generates a RequestNotSupportedException exception.)

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

Use the print classes for the following:

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

Examples

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

Product license

The ProductLicense class enables you to request licenses for products installed on the iSeries. To be compatible with other iSeries license users, the class works through iSeries product license support when requesting or releasing a license.

The class does not enforce the license policy but returns enough information such that the application can enforce the policy. When a license is requested the ProductLicense class will return the status of the request -- license granted or denied. If the request is denied the application must disable the behavior that required the license because the IBM Toolbox for Java does not know which function to disable.

Use the ProductLicense class with iSeries license support to enforce the license of your application:

- The server side of your application registers your product and license terms with iSeries license support.
- The client side of your application uses the ProductLicense object to request and release licenses.

Example: ProductLicense scenario

For example, suppose your customer bought 15 concurrent use licenses for your product. Concurrent use means 15 users can use the product at the same time, but it doesn't have to be 15 specific users. It can be any 15 users in the organization. This information is registered with iSeries license support. As users connect your application uses the ProductLicense class to request a license.

- When the number of concurrent users is fewer than 15, the request is successful and your application runs.
- When the 16th user connects, the ProductLicense request fails. Your application then displays an error message and terminates.

When a user stops running the application, your application releases the license by way of the ProductLicense class. The license is now available for someone else to use.

For more information and a code example, refer to the ProductLicense javadoc.

Program call

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

Required parameters are as follows:

- The program and parameters to run
- The AS400 object that represents the iSeries system that has the program.

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

The ProgramCall object class causes the AS400 object to connect to the iSeries.

The following example shows how to use the ProgramCall class:

```
                  // Create an AS400 object.
AS400 sys = new AS400("mySystem.myCompany.com");

                  // Create a program object. I choose
                  // to set the program to run later.
ProgramCall pgm = new ProgramCall(sys);

                  // Set the name of the program.
                  // Because the program does not take
                  // any parameters, pass null for the
                  // ProgramParameter[] argument.
pgm.setProgram(QSYSObjectPathName.toPath("MYLIB",
                                         "MYPROG",
                                         "PGM"));

                  // Run the program. My program has
                  // no parms. If it fails to run, the failure
                  // is returned as a set of messages
                  // in the message list.
if (pgm.run() != true)
{
                  // If you get here, the program
                  // failed to run. Get the list of
                  // messages to determine why the
                  // program didn't run.
   AS400Message[] messageList = pgm.getMessageList();

                  // ... Process the message list.
}

                  // Disconnect since I am done
                  // running programs
sys.disconnectService(AS400.COMMAND);
```

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

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

»The default behavior is for iSeries programs to run in a separate server job, even when the Java program and the iSeries program are on the same server. You can override the default behavior and have the iSeries program run in the Java job using the setThreadSafe() method.«

Using ProgramParameter objects

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

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

```
                  // Create an AS400 object
AS400 sys = new AS400("mySystem.myCompany.com");

                  // My program has two parameters.
                  // Create a list to hold these
                  // parameters.
ProgramParameter[] parmList = new ProgramParameter[2];

                  // First parameter is an input
                  // parameter
byte[] name = {1, 2, 3};
parmList[0] = new ProgramParameter(key);

                  // Second parameter is an output
                  // parameter. A four-byte number
                  // is returned.
parmList[1] = new ProgramParameter(4);

                  // Create a program object
                  // specifying the name of the
                  // program and the parameter list.
ProgramCall pgm = new ProgramCall(sys,
                                  "/QSYS.LIB/MYLIB.LIB/MYPROG.PGM",
                                  parmList);

                  // Run the program.
if (pgm.run() != true)
{

                  // If the iSeries cannot run the
                  // program, look at the message list
                  // to find out why it didn't run.
   AS400Message[] messageList = pgm.getMessageList();

}
else
{
                  // Else the program ran. Process the
                  // second parameter, which contains
                  // the returned data.

                  // Create a converter for this
                  // iSeries data type
   AS400Bin4 bin4Converter = new AS400Bin4();

                  // Convert from iSeries type to Java
                  // object. The number starts at the
                  // beginning of the buffer.
   byte[] data = parmList[1].getOutputData();
   int i = bin4Converter.toInt(data);
}

                  // Disconnect since I am done
                  // running programs
```

```
            sys.disconnectService(AS400.COMMAND);
```

QSYSObjectPathName class

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

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

The following examples show how to use the QSYSObjectPathName class:

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

```
                  // Create an AS400 object.
AS400 sys = new AS400("mySystem.myCompany.com");

                  // Create a program call object.
ProgramCall pgm = new ProgramCall(sys);

                  // Create a path name object that
                  // represents program PRINT_IT in
                  // library REPORTS.
QSYSObjectPathName pgmName = new QSYSObjectPathName("REPORTS",
                                                    "PRINT_IT",
                                                    "PGM");

                  // Use the path name object to set
                  // the name on the program call
                  // object.
pgm.setProgram(pgmName.getPath());

                  // ... run the program, process the
                  // results
```

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

```
                  // Create an AS400 object.
AS400 sys = new AS400("mySystem.myCompany.com");

                  // Create a program call object.
ProgramCall pgm = new ProgramCall(sys);

                  // Use the toPath method to create
                  // the name that represents program
                  // PRINT_IT in library REPORTS.
pgm.setProgram(QSYSObjectPathName.toPath("REPORTS",
                                         "PRINT_IT",
                                         "PGM"));

                  // ... run the program, process the
                  // results
```

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

```
                  // Create a path name object from
                  // the fully qualified integrated
                  // file system name.
QSYSObjectPathName ifsName = new QSYSObjectPathName(pathName);
```

```
                                  // Use the path name object to get
                                  // the library, name and type of
                                  // AS/400 object.
            String library = ifsName.getLibraryName();
            String name    = ifsName.getObjectName();
            String type    = ifsName.getObjectType();
```

Record-level access

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

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

  Note: The record format of the file is not retrieved in its entirety. The record formats retrieved are meant to be used when setting the record format for an AS400File object. Only enough information is retrieved to describe the contents of a record of the file. Record format information, such as column headings and aliases, is not retrieved.
- Access the records in an iSeries file sequentially, by record number, or by key.
- Write records to an iSeries file.
- Update records in an iSeries file sequentially, by record number, or by key.
- Delete records in an iSeries file sequentially, by record number, or by key.
- Lock an iSeries file for different types of access.
- Use commitment control to allow a Java program to do the following:
    - Start commitment control for the connection.
    - Specify different commitment control lock levels for different files.
    - Commit and rollback transactions.
- Delete iSeries files.
- Delete a member from an iSeries file.

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

The following classes perform these functions:

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

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

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

The record-level access classes use the following:

- The RecordFormat class to describe a record of the database file
- The Record class to provide access to the records of the database file
- »The LineDataRecordWriter class to write a record in line data format«

These classes are described in the data conversion section.

Examples

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

AS400File

The AS400File class provides the methods for the following:

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

KeyedFile

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

To position the cursor, use the following methods:

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

To delete a record, use the following method :

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

The read methods are:

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

To update a record, use the following method:

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

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

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

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

Specifying the key

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

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

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

```
theKey[2] = "2227 John Doe Lane, ANYTOWN, NY 11199";
```

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

For example:

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

                  // Example of an INVALID partial key
Object[] INVALIDPartialKey = new Object[2];
INVALIDPartialKey[0] = new Integer(445123);
INVALIDPartialKey[1] = "2227 John Doe Lane, ANYTOWN, NY 11199";
```

Null keys and null key fields are not supported.

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

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

```
                  // Create an AS400 object, the file exists on this
                  // AS/400.
AS400 sys = new AS400("mySystem.myCompany.com");

                  // Create a file object that represents the file
KeyedFile myFile = new KeyedFile(sys, "/QSYS.LIB/MYLIB.LIB/MYFILE.FILE/%FILE%.MBR");

                  // Assume that the AS400FileRecordDescription class
                  // was used to generate the code for a subclass of
                  // RecordFormat that represents the record format
                  // of file MYFILE in library MYLIB.  The code was
                  // compiled and is available for use by the Java program.
RecordFormat recordFormat = new MYKEYEDFILEFormat();

                  // Set the record format for myFile.  This must
                  // be done prior to invoking open()
myFile.setRecordFormat(recordFormat);

                  // Open the file.
myFile.open(AS400File.READ_WRITE, 0, AS400File.COMMIT_LOCK_LEVEL_NONE);

                  // The record format for the file contains
                  // four key fields, CUSTNUM, CUSTNAME, PARTNUM
                  // and ORDNUM in that order.
                  // The partialKey will contain 2 key field
                  // values.  Because the key field values must be
                  // in order, the partialKey will consist of values for
                  // CUSTNUM and CUSTNAME.
Object[] partialKey = new Object[2];
partialKey[0] = new Integer(1);
partialKey[1] = "John Doe";

                  // Read the first record matching partialKey
Record keyedRecord = myFile.read(partialKey);

                  // If the record was not found, null is returned.
if (keyedRecord != null)
{ // Found the record for John Doe, print out the info.
```

```
  System.out.println("Information for customer " + (String)partialKey[1] + ":");
  System.out.println(keyedRecord);
}

                ....

                  // Close the file since I am done using it
myFile.close();

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

SequentialFile

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

To position the cursor, use the following methods:

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

To delete a record, use the following method:

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

To read a record, use the following methods:

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

To update a record, use the following method:

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

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

The following example shows how to use the SequentialFile class:

```
                  // Create an AS400 object, the file exists on this
                  // AS/400.
AS400 sys = new AS400("mySystem.myCompany.com");

                  // Create a file object that represents the file
SequentialFile myFile = new SequentialFile(sys, "/QSYS.LIB/MYLIB.LIB/MYFILE.FILE/%FILE%.MBR");

                  // Assume that the AS400FileRecordDescription class
                  // was used to generate the code for a subclass of
                  // RecordFormat that represents the record format
                  // of file MYFILE in library MYLIB.  The code was
                  // compiled and is available for use by the Java program.
RecordFormat recordFormat = new MYFILEFormat();

                  // Set the record format for myFile.  This must
                  // be done prior to invoking open()
myFile.setRecordFormat(recordFormat);

                  // Open the file.
myFile.open(AS400File.READ_WRITE, 0, AS400File.COMMIT_LOCK_LEVEL_NONE);

                  // Delete record number 2.
myFile.delete(2);

                  // Read record number 5 and update it
Record updateRec = myFile.read(5);
updateRec.setField("CUSTNAME", newName);

                  // Use the base class' update() method since I am
                  // already positioned on the record.
myFile.update(updateRec);

                  // Update record number 7
updateRec.setField("CUSTNAME", nextNewName);
updateRec.setField("CUSTNUM", new Integer(7));
myFile.update(7, updateRec);

                ....

                  // Close the file since I am done using it
myFile.close();
```

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

AS400FileRecordDescription

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

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

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

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

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

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

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

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

```
                       // Create an AS400 object, the file exists on this
                       // AS/400.
     AS400 sys = new AS400("mySystem.myCompany.com");

                       // Create an AS400FileRecordDescription object that represents the file
     AS400FileRecordDescription myFile = new AS400FileRecordDescription(sys, "/QSYS.LIB/MYLIB.LIB/MYFILE.FILE");

                       // Create the Java source file in the current working directory.
                       // Specify "package com.myCompany.myProduct;" for the
                       // package statement in the source since I will ship the class
                       // as part of my product.
     myFile.createRecordFormatSource(null, "com.myCompany.myProduct");

                       // Assuming that the format name for file MYFILE is FILE1, the
                       // file FILE1Format.java will be created in the current working directory.
                       // It will overwrite any file by the same name.  The name of the class
                       // will be FILE1Format.  The class will extend from RecordFormat.
```

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

```
                       // Create an AS400 object, the file exists on this
                       // AS/400.
     AS400 sys = new AS400("mySystem.myCompany.com");

                       // Create an AS400File object that represents the file
     SequentialFile myFile = new SequentialFile(sys, "/QSYS.LIB/MYLIB.LIB/MYFILE.FILE");

                       // Set the record format
                       // This assumes that import.com.myCompany.myProduct.FILE1Format;
                       // has been done.

     myFile.setRecordFormat(new FILE1Format());

                       // Open the file and read from it
                       ....

                       // Close the file since I am done using it
     myFile.close();

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

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

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

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

```
                  // Create an AS400 object, the file exists on this
                  // AS/400.
AS400 sys = new AS400("mySystem.myCompany.com");

                  // Create an AS400FileRecordDescription object that represents the file
AS400FileRecordDescription myFile = new AS400FileRecordDescription(sys, "/QSYS.LIB/MYLIB.LIB/MYFILE.FILE");

                  // Retrieve the record format for the file
RecordFormat[] format = myFile.retrieveRecordFormat();

                  // Create an AS400File object that represents the file
SequentialFile myFile = new SequentialFile(sys, "/QSYS.LIB/MYLIB.LIB/MYFILE.FILE");

                  // Set the record format
myFile.setRecordFormat(format[0]);

                  // Open the file and read from it
                  ....

                  // Close the file since I am done using it
myFile.close();

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

System Status

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

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

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

Example

This example shows you how to use caching with the SystemStatus class:

```
AS400 system = new AS400("MyAS400");
SystemStatus status = new SystemStatus(system);

// Turn on caching. It is off by default.
status.setCaching(true);

// This will retrieve the value from the system.
// Every subsequent call will use the cached value
// instead of retrieving it from the system.
int jobs = status.getJobsInSystem();


// ... Perform other operations here ...


// This determines if caching is still enabled.
if (status.isCaching())
{
  // This will retrieve the value from the cache.
  jobs = status.getJobsInSystem();
}

// Go to the system next time, regardless if caching is enabled.
status.refreshCache();

// This will retrieve the value from the system.
jobs = status.getJobsInSystem();

// Turn off caching. Every subsequent call will go to the system.
status.setCaching(false);

// This will retrieve the value from the system.
jobs = status.getJobsInSystem();
```

System values

The system value classes allow a Java program to retrieve and change system values and network attributes. »You can also define your own group to contain the system values you want.«

A SystemValue object primarily contains the following information:

- Name
- Description
- Release
- » Value«

Using the SystemValue class, retrieve a single system value by using the getValue() method and change a system value by using the setValue() method.

»You can also retrieve group information about a particular system value:

- To retrieve the system-defined group to which a system value belongs, use the getGroup() method.
- To retrieve the user-defined group to which a SystemValue object belongs (if any), use the getGroupName() and getGroupDescription() methods.«

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

System value list

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

»System value group

SystemValueGroup represents a user-defined collection of system values and network attributes. Rather than a container, it is instead a factory for generating and maintaining unique collections of system values.

You can create a SystemValueGroup by specifying one of the system-defined groups (one of the constants in the SystemValueList class) or by specifying an array of system value names.

You can individually add the names of system values to include in the group by using the add() method. You can also remove them by using the remove() method.

Once the SystemValueGroup is populated with the desired system value names, obtain the real SystemValue objects from the group by calling the getSystemValues() method. In this way, a SystemValueGroup object takes a set of system value names and generates a Vector of SystemValue objects, all having the system, group name, and group description of the SystemValueGroup.

To refresh a Vector of SystemValue objects all at once, use the refresh() method. «

Examples of using the SystemValue and SystemValueList classes

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

```
//Create an AS400 object
AS400 sys = new AS400("mySystem.myCompany.com");

//Create a system value representing the current second on the system.
SystemValue sysval = new SystemValue(sys, "QSECOND");

//Retrieve the value.
String second = (String)sysval.getValue();

//At this point QSECOND is cached. Clear the cache to retrieve the most
//up-to-date value from the system.
sysval.clear();
second = (String)sysval.getValue();
```

```
//Create a system value list.
SystemValueList list = new SystemValueList(sys);

//Retrieve all the of the date/time system values.
Vector vec = list.getGroup(SystemValueList.GROUP_DATTIM);

//Disconnect from the system.
sys.disconnectAllServices();
```

»Examples of using the SystemValueGroup class

The following example shows how to build a group of system value names and then manipulate them:

```
//Create an AS400 object
AS400 sys = new AS400("mySystem.myCompany.com");

//Create a system value group initially representing all of the network attributes on the system.
String name = "My Group";
String description = "This is one of my system values.";
SystemValueGroup svGroup = new SystemValueGroup(sys, name, description, SystemValueList.GROUP_NET);

//Add some more system value names to the group and remove some we do not want.
svGroup.add("QDATE");
svGroup.add("QTIME");
svGroup.remove("NETSERVER");
svGroup.remove("SYSNAME");

//Obtain the actual SystemValue objects. They are returned inside a Vector.
Vector sysvals = svGroup.getSystemValues();

//You will notice that this is one of my system values.
SystemValue mySystemValue = (SystemValue)sysvals.elementAt(0);
System.out.println(mySystemValue.getName()+" - "+mySystemValue.getGroupDescription());

//We can add another SystemValue object from another system into the group.
AS400 sys2 = new AS400("otherSystem.myCompany.com");
SystemValue sv = new SystemValue(sys2, "QDATE");
sysvals.addElement(sv);

//Now refresh the entire group of system values all at once.
//It does not matter if some system values are from different iSeries systems.
//It does not matter if some system values were generated using SystemValueGroup and some were not.
SystemValueGroup.refresh(sysvals);

//Disconnect from the systems.
sys.disconnectAllServices();
sys2.disconnectAllServices();
```

«

Trace

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

The Trace class logs the following categories of information:

Conversion	Logs character set conversions between Unicode and code pages. This category should be used only by the IBM Toolbox for Java classes.
Information	Traces the flow through a program.
Warning	Logs information about errors the program was able to recover from.
Error	Logs additional errors that cause an exception.
Diagnostic	Logs state information.
Data stream	Logs the data that flows between the iSeries and the Java program. This category should be used only by the IBM Toolbox for Java classes.
Proxy	This category is used by IBM Toolbox for Java classes to log data flow between the client and the proxy server.
All	This category is used to enable or disable tracing for all of the above categories at once. Trace information can not be directly logged to this category.

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

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

»**You can send trace data for different components to separate logs.** Trace data, by default, is written to the default log. Use component tracing to write application-specific trace data to a separate log or standard output. By using component tracing, you can easily separate trace data for a specific application from other data.«

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

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

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

The following examples show how to use the Trace class.

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

```
// Enable diagnostic, information, and warning logging.
Trace.setTraceDiagnosticOn(true);
Trace.setTraceInformationOn(true);
Trace.setTraceWarningOn(true);

// Turn tracing on.
Trace.setTraceOn(true);

// ... At this point in the Java program, write to the log.
Trace.log(Trace.INFORMATION, "Just entered class xxx, method xxx");
```

```
// Turning tracing off.
Trace.setTraceOn(false);
```

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

```
// Method 1 - build a trace record
// then call the log method and let the trace class determine if the
// data should be logged. This will work but will be slower than the
// following code.
String traceData = new String("Just entered class xxx, data = ");
traceData = traceData + data + "state = " + state;
Trace.log(Trace.INFORMATION, traceData);

// Method 2 - check the log status before building the information to
// log. This is faster when tracing is not active.
if (Trace.isTraceOn() && Trace.isTraceInformationOn())
{
   String traceData = new String("just entered class xxx, data = ");
   traceData = traceData + data + "state = " + state;
   Trace.log(Trace.INFORMATION, traceData);
}
```

»**Example 3:** The following shows how you can use component tracing.

```
// Create a component string. It is more efficient to create an
// object than many String literals.
String myComponent1 = "com.myCompany.xyzComponent";
String myComponent2 = "com.myCompany.abcComponent";

// Send Toolbox and the component trace data each to separate files.
// The Toolbox trace will contain all trace information, while each
// component log file will only contain trace information specific to
// that component.  If a Trace file is not specified, all trace data
// will go to standard out with the component specified in front of
// each trace message.

// Trace.setFileName("c:\\bit.bucket");
// Trace.setFileName(myComponent1, "c:\\Component1.log");
// Trace.setFileName(myComponent2, "c:\\Component2.log");

Trace.setTraceOn(true);             // Turn trace on.
Trace.setTraceInformationOn(true);  // Enable information messages.

// Log component specific trace data or general toolbox
// trace data.

Trace.setFileName("c:\\bit.bucket");
Trace.setFileName(myComponent1, "c:\\Component1.log");
```

The results of the example, if you do not specify any trace files, look like this:

```
Toolbox for Java - Version 5 Release 1 Modification level 0
[com.myCompany.xyzComponent] Tue Oct 24 16:02:44 CDT 2000 I am here
[com.myCompany.abcComponent] Tue Oct 24 16:02:44 CDT 2000 I am there
Tue Oct 24 16:02:44 CDT 2000 I am everywhere«
```

Users and groups

The user and group classes allow you to get a list of users and user groups on the iSeries system as well as information about each user through a Java program.

≫**Note:** When possible, use a resource class instead of a class from the access package. Resource classes provide a generic framework and a consistent programming interface for working with various AS400 objects and lists. The resource classes for working with users are RUser and RUserList.≪

Some of the user information you can retrieve includes previous sign-on date, status, date the password was last changed, date the password expires, and user class. When you access the User object, you should use the setSystem() method to set the system name and the setName() method to set the user name. After those steps, you use the loadUserInformation() method to get the information from the iSeries.

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

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

```
    // Create an AS400 object.
    AS400 system = new AS400 ("mySystem.myCompany.com");

    // Create the UserList object.
    UserList userList = new UserList (system);

    // Get the list of all users and groups.
    Enumeration enum = userList.getUsers ();

    // Iterate through the list.
    while (enum.hasMoreElements ())
    {
        User u = (User) enum.nextElement ();
        System.out.println  (u);
    }
```

Retrieving information about users and groups

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

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

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

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

Example

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

User space

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

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

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

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

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

```
                  // Create an AS400 object.
AS400 sys = new AS400("mySystem.myCompany.com");

                  // Create a user space object.
UserSpace US = new UserSpace(sys,
          "/QSYS.LIB/MYLIB.LIB/MYSPACE.USRSPC");

                  // Use the create method to create the user space on
                  // the AS/400.
US.create(10240,                         // The initial size is 10K
          true,                          // Replace if the user space already exists
          " ",                           // No extended attribute
          (byte) 0x00,                   // The initial value is a null
          "Created by a Java program",   // The description of the user space
          "*USE");                       // Public has use authority to the user space


                  // Use the write method to write bytes to the user space.
US.write("Write this string to the user space.", 0);
```

Graphical Toolbox

Overview

The Graphical Toolbox, a set of UI tools, makes it easy to create custom user interface panels in Java. You can incorporate the panels into your Java applications, applets, or Operations Navigator plug-ins. The panels may contain data obtained from the iSeries, or data obtained from another source such as a file in the local file system or a program on the network.

The **GUI Builder** is a WYSIWYG visual editor for creating Java dialogs, property sheets and wizards. With the GUI Builder you can add, arrange, or edit user interface controls on a panel, and then preview the panel to verify the layout behaves the way you expected. The panel definitions you create can be used in dialogs, inserted within property sheets and wizards, or arranged into splitter, deck, and tabbed panes. The GUI Builder also allows you to build menu bars, toolbars, and context menu definitions. »Incorporate JavaHelp in your panels, including context sensitive help.«

The **Resource Script Converter** converts Windows resource scripts into an XML representation that is usable by Java programs. With the Resource Script Converter you can process Windows resource scripts (RC files) from your existing Windows dialogs and menus. These converted files can then be edited with the GUI Builder. Property sheets and wizards can be made from RC files using the resource script converter along with the GUI Builder.

Underlying these two tools is a new technology called the **Panel Definition Markup Language**, or **PDML**. PDML is based on the Extensible Markup Language (XML) and defines a platform-independent language for describing the layout of user interface elements. Once your panels are defined in PDML, you can use the runtime API provided by the Graphical Toolbox to display them. The API displays your panels by interpreting the PDML and rendering your user interface using the Java Foundation Classes.

Benefits of the Graphical Toolbox

Write Less Code and Save Time

With the Graphical Toolbox you have the ability to create Java-based user interfaces quickly and easily. The GUI Builder lets you have precise control over the layout of UI elements on your panels. Because the layout is described in PDML, you are not required to develop any Java code to define the user interface, and you do not need to recompile code in order to make changes. As a result, significantly less time is required to create and maintain your Java applications. The Resource Script Converter lets you migrate large numbers of Windows panels to Java quickly and easily.

Custom Help

Defining user interfaces in PDML creates some additional benefits. Because all of a panel's information is consolidated in a formal markup language, the tools can be enhanced to perform additional services on behalf of the developer. For example, both the GUI Builder and the Resource Script Converter are capable of generating HTML skeletons for the panel's online help. You decide which help topics are required and the help topics are automatically built based on your requirements. Anchor tags for the help topics are built right into the help skeleton, which frees the help writer to focus on developing appropriate content. The Graphical Toolbox runtime environment automatically displays the correct help topic in response to a user's request.

Automatic Panel to Code Integration

In addition, PDML provides tags that associate each control on a panel with an attribute on a JavaBean. Once you have identified the bean classes that will supply data to the panel and have associated a attribute with each of the appropriate controls, you can request that the tools generate Java source code skeletons for the bean objects. At runtime, the Graphical Toolbox automatically transfers data between the beans and the controls on the panel that you identified.

Platform Independent

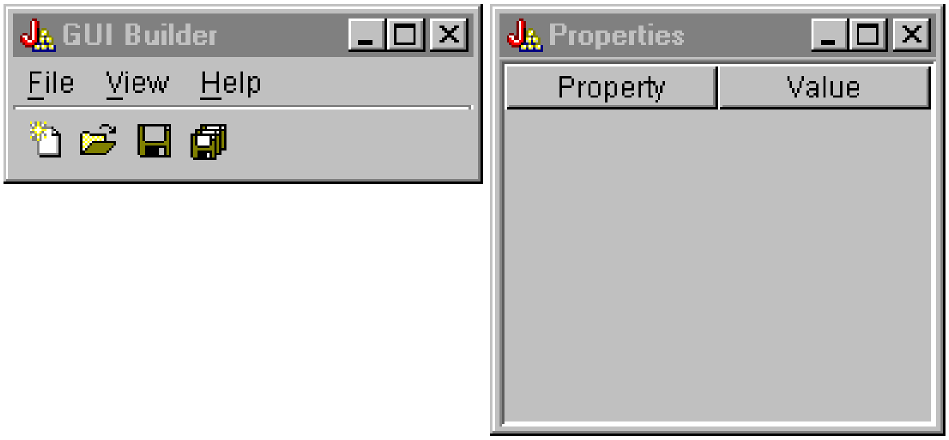
The Graphical Toolbox runtime environment provides support for event handling, user data validation, and common types of interaction among the elements of a panel. The correct platform look and feel for your user interface is automatically set based on the underlying operating system, and the GUI Builder lets you toggle the look and feel so that you can evaluate how your panels will look on different platforms.

Inside the Graphical Toolbox

The Graphical Toolbox provides you with two tools and, therefore, two ways of automating the creation of your user interfaces. You can use the GUI Builder to quickly and easily create new panels from scratch, or you can use the Resource Script Converter to convert existing Windows-based panels to Java. The converted files can then be edited with GUI Builder. Both tools support internationalization.

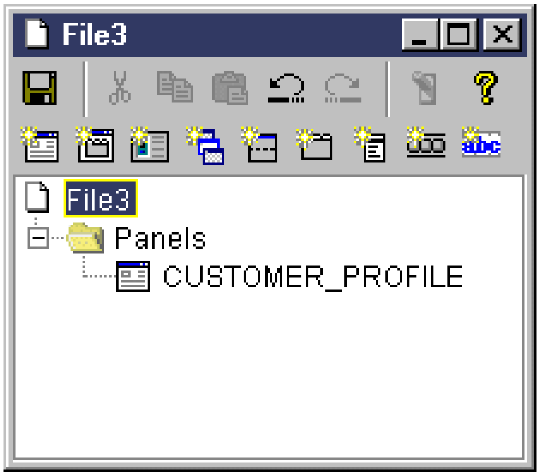
GUI Builder
Two windows are displayed when you invoke the GUI Builder for the first time, as shown below:

GUI Builder windows



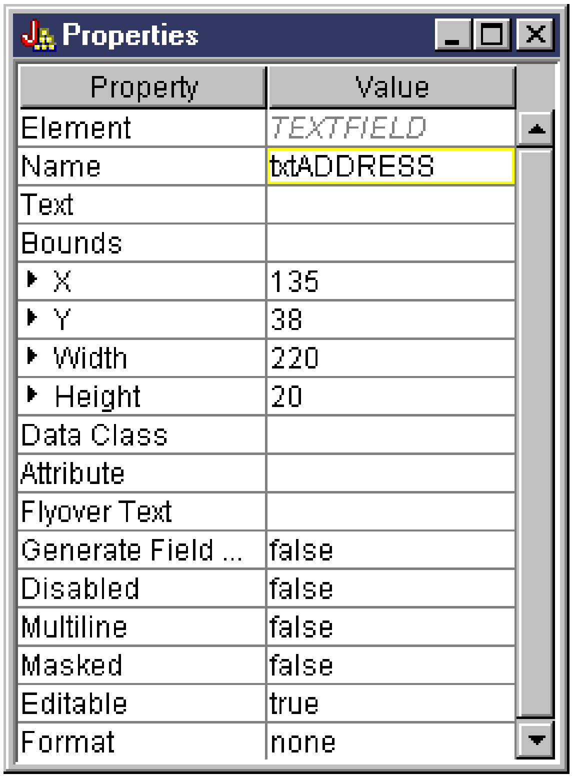
Use the File Builder window to create and edit your PDML files.

File Builder window



Use the Properties window to view or change the properties of the currently selected control.

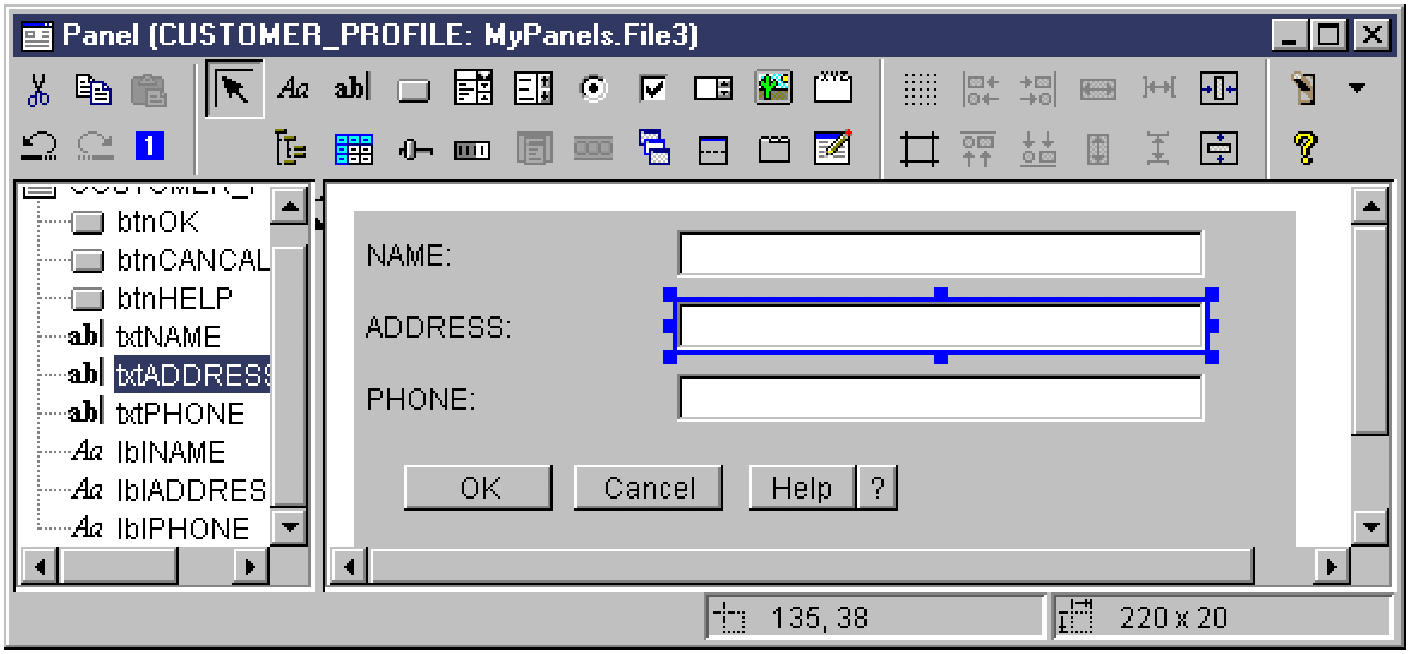
Properties window



Property	Value
Element	TEXTFIELD
Name	txtADDRESS
Text	
Bounds	
▸ X	135
▸ Y	38
▸ Width	220
▸ Height	20
Data Class	
Attribute	
Flyover Text	
Generate Field ...	false
Disabled	false
Multiline	false
Masked	false
Editable	true
Format	none

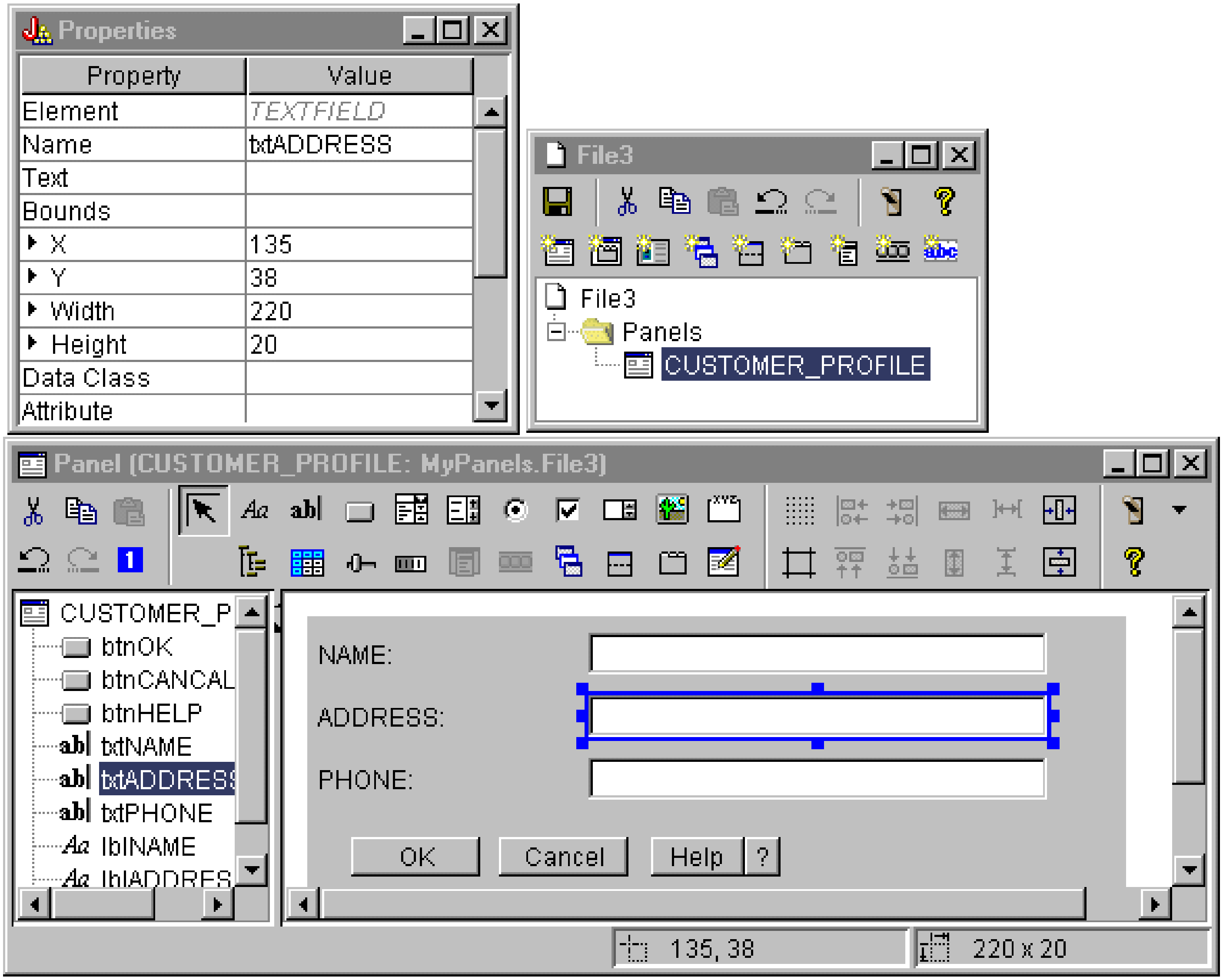
Use the Panel Builder window to create and edit your graphical user interface components. Select the desired component from the toolbar and click on the panel to place it where ever you want. The toolbar also facilities for aligning groups of controls, for previewing the panel, and for requesting online help for a GUI Builder function. See Explanation of the Toolbox Widgets for a description of what each icon does.

Panel Builder window



The panel being edited is displayed in the Panel Builder window. The figure below demonstrates how the windows work together:

Example of how GUI Builder windows work together



Properties

Property	Value
Element	TEXTFIELD
Name	txtADDRESS
Text	
Bounds	
X	135
Y	38
Width	220
Height	20
Data Class	
Attribute	

File3

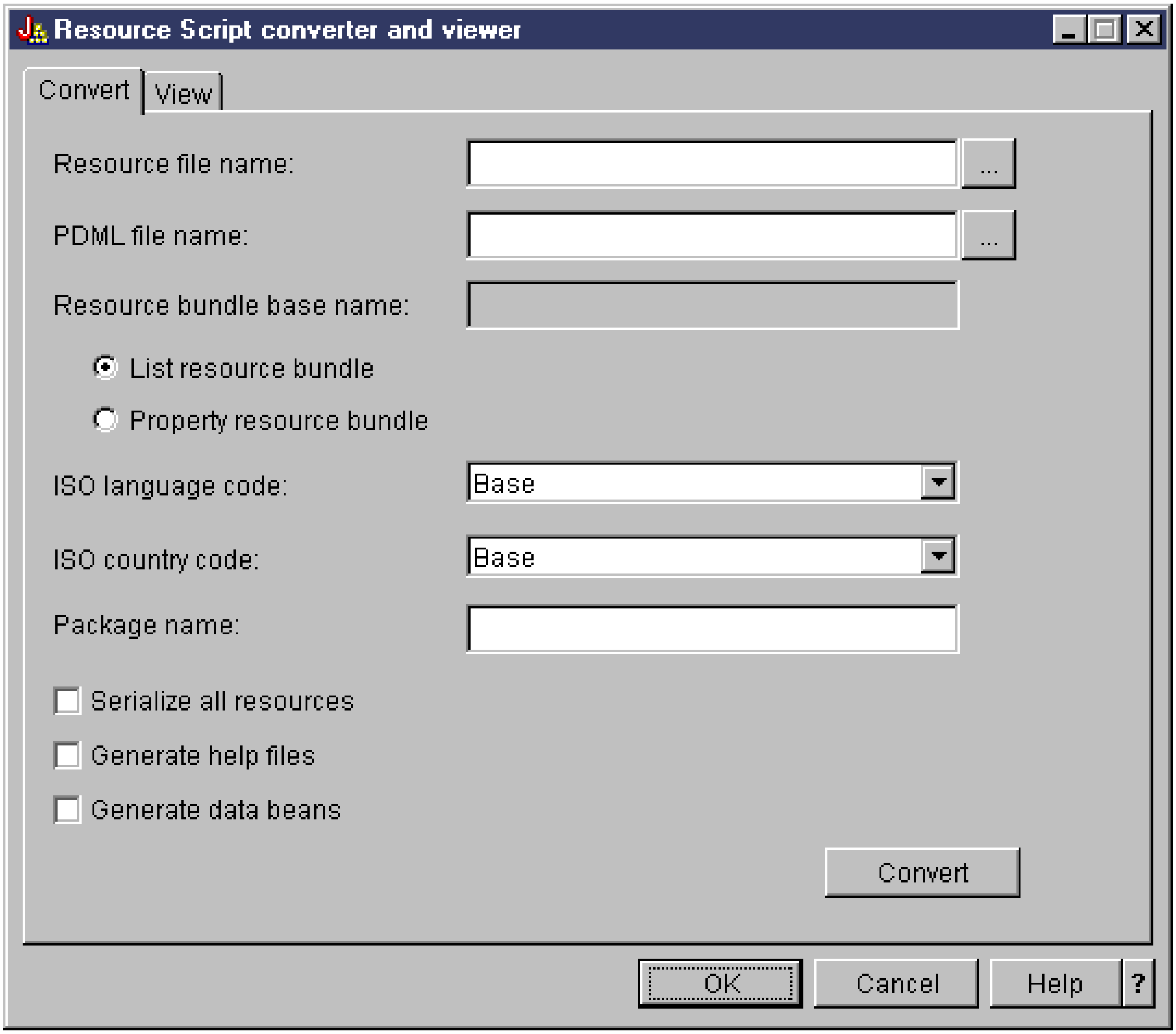
- File3
  - Panels
    - CUSTOMER_PROFILE

Panel (CUSTOMER_PROFILE: MyPanels.File3)

- CUSTOMER_P
  - btnOK
  - btnCANCAL
  - btnHELP
  - txtNAME
  - txtADDRESS
  - txtPHONE
  - lblNAME
  - lblADDRES

Resource Script Converter
The Resource Script Converter consists of a two-paned tabbed dialog. On the **Convert** pane you specify the name of the Microsoft or VisualAge for Windows RC file that is to be converted to PDML. You can specify the name of the target PDML file and associated Java resource bundle that will contain the translated strings for the panels. In addition, you can request that online help skeletons be generated for the panels, generate Java source code skeletons for the objects that supply data to the panels, and serialize the panel definitions for improved performance at runtime. The Converter's online help provides a detailed description of each input field on the Convert pane.

Resource Script Converter: Convert pane



Resource Script converter and viewer

Convert | View

Resource file name:

PDML file name:

Resource bundle base name:

● List resource bundle

○ Property resource bundle

ISO language code: Base

ISO country code: Base

Package name:

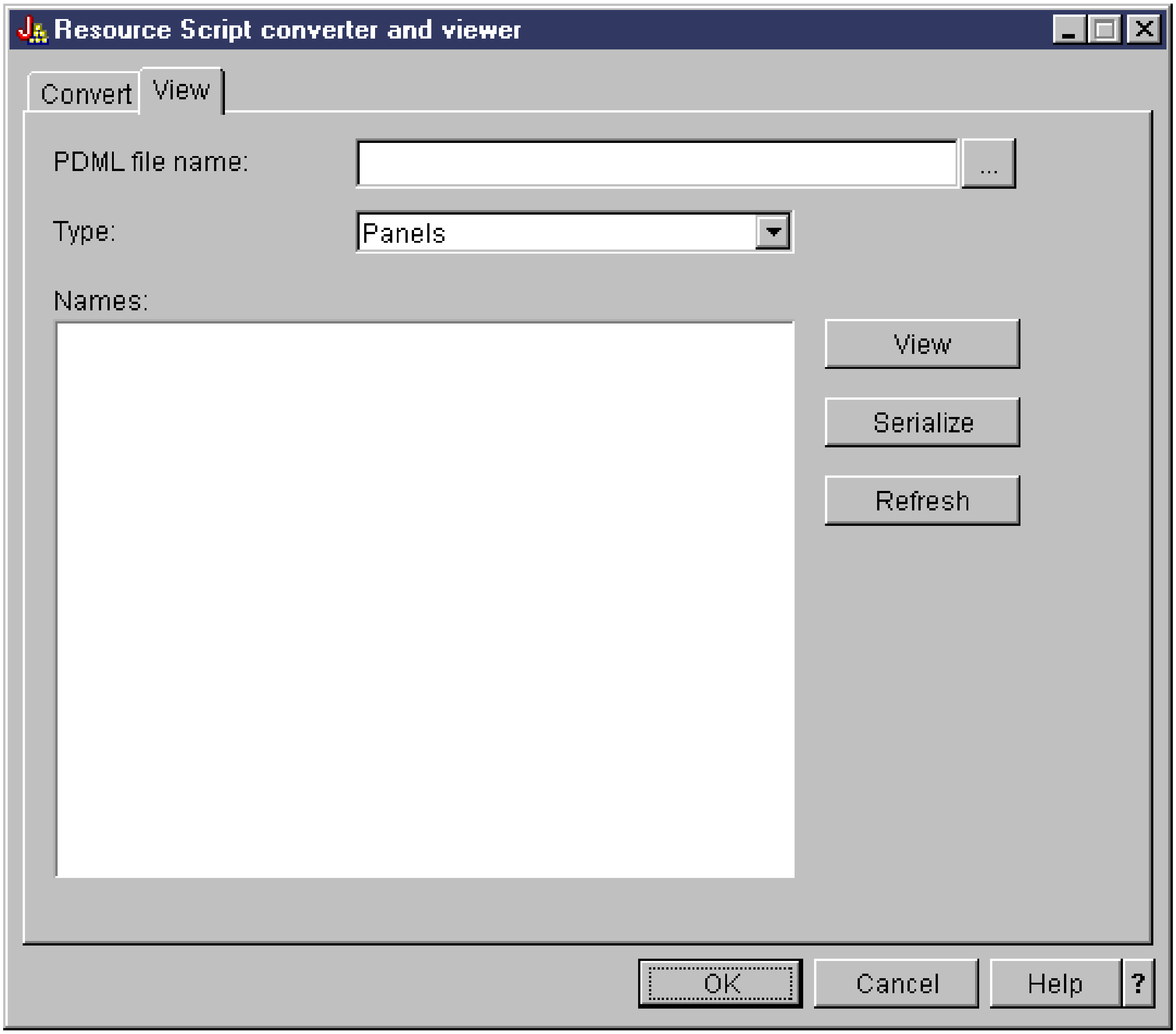
□ Serialize all resources

□ Generate help files

□ Generate data beans

After the conversion has run successfully, you can use the **View** pane to view the contents of your newly-created PDML file, and preview your new Java panels. You can use the GUI Builder to make minor adjustments to a panel if needed. The Converter always checks for an existing PDML file before performing a conversion, and attempts to preserve any changes in case you need to run the conversion again later.

Resource Script Converter: View pane



Getting started with the Graphical Toolbox

Use the following topics to to learn more about the Graphical Toolbox:

- Setting up the Graphical Toolbox
- Creating your user interface
- Displaying your panels at runtime
- Generating online help files
- Graphical Toolbox example
- Using the Graphical Toolbox in a browser

Setting up the Graphical Toolbox

The Graphical Toolbox is delivered as a set of JAR files. To set up the Graphical Toolbox you must install the JAR files on your workstation and set your CLASSPATH environment variable.

»You must also ensure that your workstation meets the requirements to run IBM Toolbox for Java.«

Installing the Graphical Toolbox on your workstation

To develop Java programs using the Graphical Toolbox, first install the Graphical Toolbox JAR files on your workstation. There are two ways to do this:

Transfer the JAR Files

Note: The following list represents three different ways to transfer the JAR files. The IBM Toolbox for Java licensed program must be installed on your iSeries. » Additionally, you need to download the JAR file for JavaHelp, jhall.jar, from the Sun JavaHelp Web site. «

- Use FTP (ensure you transfer the files in binary mode) and copy the JAR files from the directory **/QIBM/ProdData/HTTP/Public/jt400/lib** to a local directory on your workstation
- Use IBM iSeries Client Access Express to map a network drive.
- The AS400ToolboxInstaller class that comes with the IBM Toolbox for Java can also be used to install the Graphical Toolbox JAR files - specify OPNAV for the package name. For more information, see Client installation and update classes.

Install JAR files with Client Access Express

You can also install the Graphical Toolbox when you install Client Access Express. The IBM Toolbox for Java is now shipped as part of Client Access Express. If you are installing Client Access Express for the first time, choose Custom Install and select the **IBM Toolbox for Java** component on the install menu. If you have already installed Client Access Express, you can use the Selective Setup program to install this component if it is not already present.

Setting your classpath

To use the Graphical Toolbox, you must add these JAR files to your CLASSPATH environment variable (or specify them on the `classpath` option on the command line).

For example, if you have copied the files to the directory **C:\gtbox\lib** on your workstation, you must add the following path names to your classpath:

```
 C:\gtbox\lib\uitools.jar;
 C:\gtbox\lib\jui400.jar;
 C:\gtbox\lib\data400.jar;
 C:\gtbox\lib\util400.jar;
 C:\gtbox\lib\x4j400.jar;
»C:\gtbox\lib\jhall.jar;«
```

If you have installed the Graphical Toolbox using Client Access Express, the JAR files (except jhall.jar) will all reside in the directory **\Program Files\Ibm\Client Access\jt400\lib** on the drive where you have installed Client Access Express. »Client Access Express installs jhall.jar in the **\Program Files\Ibm\Client Access\jre\lib** directory.« The path names in your classpath should reflect this.

JAR File Descriptions

- **uitools.jar** Contains the GUI Builder and Resource Script Converter tools.
- **jui400.jar** Contains the runtime API for the Graphical Toolbox. Java programs use this API to display the panels constructed using the tools. These classes may be redistributed with applications.
- **data400.jar** Contains the runtime API for the Program Call Markup Language (PCML). Java programs use this API to call iSeries programs whose parameters and return values are identified using PCML. These classes may be redistributed

with applications.

- **util400.jar** Contains utility classes for formatting iSeries data and handling iSeries messages. These classes may be redistributed with applications.
- **x4j400.jar** Contains the XML parser used by the API classes to interpret PDML and PCML documents.
- »**jhall.jar** Contains the JavaHelp classes that displays the online help and context sensitive help for the panels you build with the GUI Builder. «

Note: Internationalized versions of the GUI Builder and Resource Script Converter tools are available. To run a non-U.S. English version you must add the correct version of **uitools.jar** for your language and country to your Graphical Toolbox installation. These JAR files are available on the iSeries in **/QIBM/ProdData/HTTP/Public/jt400/Mri29xx**, where 29*xx* is the 4-digit OS/400 NLV code corresponding to your language and country. (The names of the JAR files in the various Mri29xx directories include the correct 2-character Java language and country code suffixes.) This additional JAR file should be added to your classpath ahead of **uitools.jar** in the search order.

Using the Graphical Toolbox

Once you have installed the Graphical Toolbox, follow these links to learn how to use the tools:

- Using the GUI Builder
- Using the Resource Script Converter

Creating your user interface

Running the GUI Builder

To start the GUI Builder, invoke the Java interpreter as follows:

```
java com.ibm.as400.ui.tools.GUIBuilder [-plaf look and feel]
```

If you did not set your CLASSPATH environment variable to contain the Graphical Toolbox JAR files, then you will need to specify them on the command line using the `classpath` option. See Setting Up the Graphical Toolbox.

Options

`-plaf` *`look and feel`*

The desired platform look and feel. This option lets you override the default look and feel that is set based on the platform you are developing on, so you can preview your panels to see how they will look on different operating system platforms. The following look and feel values are accepted:

- Windows
- Metal
- Motif

Currently, additional look and feel attributes that Swing 1.1 may support are not supported by the GUI Builder

Types of user interface resources

When you start the GUI Builder for the first time you will create a new PDML file by clicking **New File** on the **File** pulldown. Once you have created your new PDML file, you can define any of the following types of UI resources to be contained within it.

Panel

The fundamental resource type. It describes a rectangular area within which UI elements are arranged. The UI elements may consist of simple controls, such as radio buttons or text fields, images, animations, custom controls, or more sophisticated subpanels (see Split Pane, Deck Pane and Tabbed Pane below). A panel may define the layout for a stand-alone window or dialog, or it may define one of the subpanels that is contained in another UI resource.

Menu

A popup window containing one or more selectable actions, each represented by a text string ("Cut", "Copy" and "Paste" are examples). You can define mnemonics and accelerator keys for each action, insert separators and cascading submenus, or define special checked or radio button menu items. A menu resource may be used as a stand-alone context menu, as a drop-down menu in a menu bar, or it may itself define the menu bar associated with a panel resource.

Toolbar

A window consisting of a series of push buttons, each representing a possible user action. Each button may contain text, an icon or both. You can define the toolbar as floatable, which lets the user drag the toolbar out of a panel and into a stand-alone window.

Property Sheet

A stand-alone window or dialog consisting of a tabbed panels and OK, Cancel, and Help buttons. Panel resources define the layout of each tabbed window.

Wizard

A stand-alone window or dialog consisting of a series of panels that are displayed to the user in a predefined sequence, with Back, Next, Cancel, Finish, and Help buttons. The wizard window may also display a list of tasks to the left of the panels which track the user's progress through the wizard.

Split Pane

A subpane consisting of two panels separated by a splitter bar. The panels may be arranged horizontally or vertically.

Tabbed Pane

A subpane that forms a tabbed control. This tabbed control can be placed inside of another panel, split pane, or deck pane.

Deck Pane

A subpane consisting of a collection of panels. Of these, only one panel can be displayed at a time. For example, at runtime the deck pane could change the panel which is displayed depending on a given user action.

String Table

A collection of string resources and their associated resource identifiers.

Generated files

The translatable strings for a panel are not stored in the PDML file itself, but in a separate Java resource bundle. The tools let you specify how the resource bundle is defined, either as a Java PROPERTIES file or as a ListResourceBundle subclass. A ListResourceBundle subclass is a compiled version of the translatable resources, which enhances the performance of your Java application. However, it will slow down the GUI Builder's saving process, because the ListResourceBundle will be compiled in each save operation. Therefore it's best to start with a PROPERTIES file (the default setting) until you're satisfied with the design of your user interface.

You can use the tools to generate HTML skeletons for each panel in the PDML file. At runtime, the correct help topic is displayed when the user clicks on the panel's Help button or presses F1 while the focus is on one of the panel's controls. You should insert your help content at the appropriate points in the HTML, within the scope of the <!-- `HELPDOC:SEGMENTBEGIN` --> and <!-- `HELPDOC:SEGMENTEND` --> tags. For more specific help information see Editing Help Documents generated by GUI builder.

You can generate source code skeletons for the JavaBeans that will supply the data for a panel. Use the Properties window of the GUI Builder to fill in the DATACLASS and ATTRIBUTE properties for the controls which will contain data. The DATACLASS property identifies the class name of the bean, and the ATTRIBUTE property specifies the name of the gettor/settor methods that the bean class implements. Once you've added this information to the PDML file, you can use the GUI Builder to generate Java source code skeletons and compile them. At runtime, the appropriate gettor/settor methods will be called to fill in the data for the panel.

Note: The number and type of gettor/settor methods is dependent on the type of UI control with which the methods are associated. The method protocols for each control are documented in the class description for the DataBean class.

Finally, you can serialize the contents of your PDML file. Serialization produces a compact binary representation of all of the UI resources in the file. This greatly improves the performance of your user interface, because the PDML file does not have to be interpreted in order to display your panels.

To summarize: If you have created a PDML file named **MyPanels.pdml**, the following files will also be produced based on the options you have selected on the tools:

- **MyPanels.properties** if you have defined the resource bundle as a PROPERTIES file
- **MyPanels.java** and **MyPanels.class** if you have defined the resource bundle as a ListResourceBundle subclass
- **<panel name>.html** for each panel in the PDML file, if you have elected to generate online help skeletons
- **<dataclass name>.java** and **<dataclass name>.class** for each unique bean class that you have specified on your DATACLASS properties, if you have elected to generate source code skeletons for your JavaBeans
- **<resource name>.pdml.ser** for each UI resource defined in the PDML file, if you've elected to serialize its contents.

Note: The conditional behavior functions (SELECTED/DESELECTED) will not work if the panel name is the same as the one in which the conditional behavior function is being attached. For instance, if PANEL1 in FILE1 has a conditional behavior reference attached to a field that references a field in PANEL1 in FILE2, the conditional behavior event will not work. To fix this, simply rename PANEL1 in FILE2 and then update the conditional behavior event in FILE1 to reflect this change.

Running the Resource Script Converter

To start the Resource Script Converter, invoke the Java interpreter as follows:

```
java com.ibm.as400.ui.tools.PDMLViewer
```

If you did not set your CLASSPATH environment variable to contain the Graphical Toolbox JAR files, then you will need to specify them on the command line using the `classpath` option. See Setting Up the Graphical Toolbox.

You can also run the Resource Script Converter in batch mode using the following command:

```
java com.ibm.as400.ui.tools.RC2XML file [options]
```

Where *file* is the name of the resource script (RC file) to be processed.

Options

`-x` *name*

The name of the generated PDML file. Defaults to the name of the RC file to be processed.

`-p` *name*

The name of the generated PROPERTIES file. Defaults to the name of the PDML file.

`-r` *name*

The name of the generated ListResourceBundle subclass. Defaults to the name of the PDML file.

`-package` *name*

The name of the package to which the generated resources will be assigned. If not specified, no package statements will be generated.

`-l` *locale*

The locale in which to produce the generated resources. If a locale is specified, the appropriate 2-character ISO language and and country codes will be suffixed to the name of the generated resource bundle.

`-h`

Generate HTML skeletons for online help.

`-d`

Generate source code skeletons for JavaBeans.

`-s`

Serialize all resources.

Mapping Windows Resources to PDML

All dialogs, menus, and string tables found in the RC file will be converted to the corresponding Graphical Toolbox resources in the generated PDML file. You can also define DATACLASS and ATTRIBUTE properties for Windows controls that will be propagated to the new PDML file by following a simple naming convention when you create the identifiers for your Windows resources. These properties will be used to generate source code skeletons for your JavaBeans when you run the conversion.

The naming convention for Windows resource identifiers is:

```
IDCB_<class name>_<attribute>
```

where `<class name>` is the fully-qualified name of the bean class that you wish to designate as the DATACLASS property of the control, and `<attribute>` is the name of the bean property that you wish to designate as the ATTRIBUTE property of the control.

For example, a Windows text field with the resource ID `IDCB_com_MyCompany_MyPackage_MyBean_SampleAttribute` would produce a DATACLASS property of **com.MyCompany.MyPackage.MyBean** and an ATTRIBUTE property of **SampleAttribute**. If you elect to generate JavaBeans when you run the conversion, the Java source file **MyBean.java** would be produced, containing the package statement **package com.MyCompany.MyPackage**, and gettor and settor methods for the **SampleAttribute** property.

Displaying your panels at runtime

The Graphical Toolbox provides a redistributable API that your Java programs can use to display user interface panels defined using PDML. The API displays your panels by interpreting the PDML and rendering your user interface using the Java Foundation Classes.

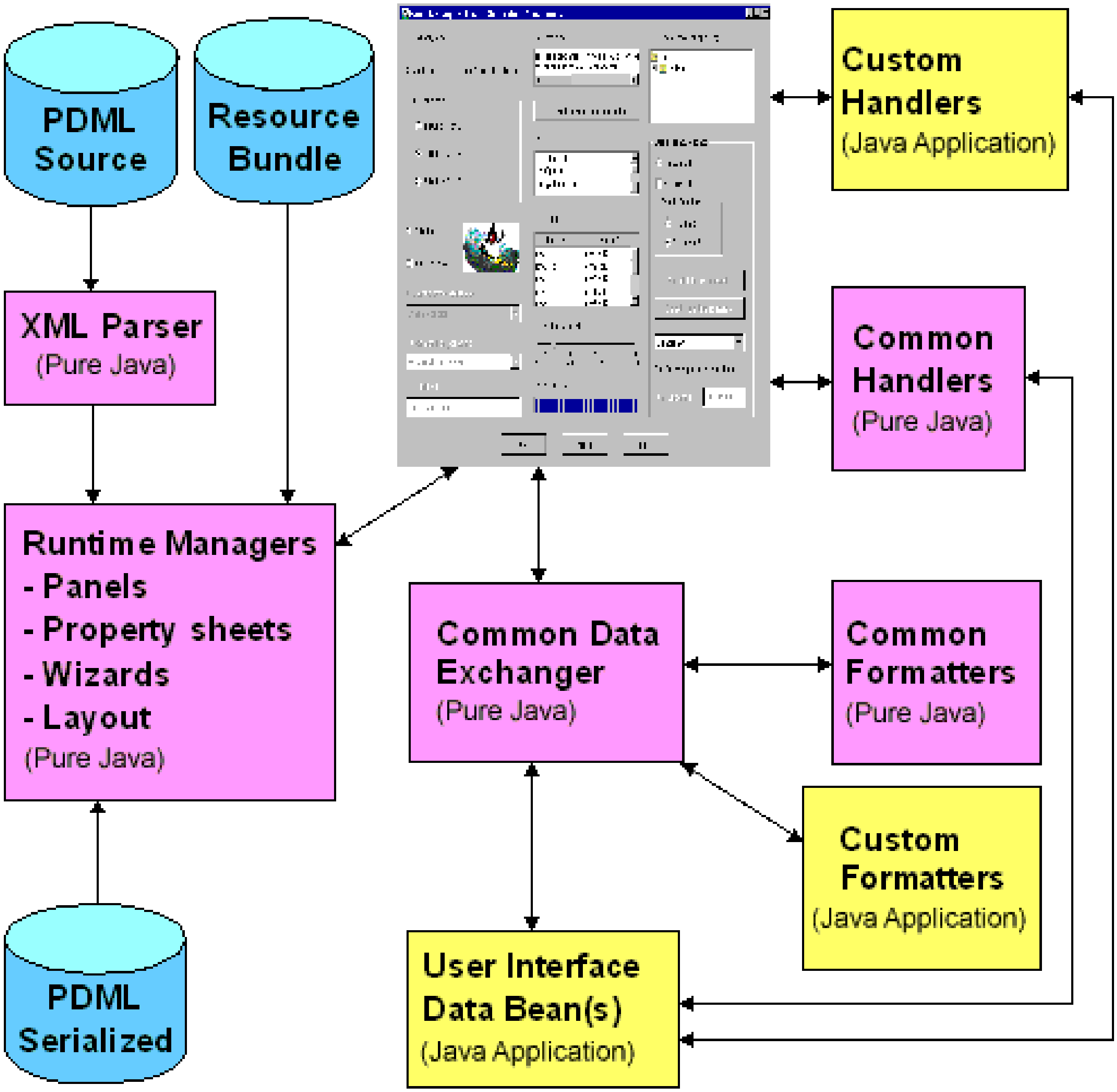
The Graphical Toolbox runtime environment provides the following services:

- Handles all data exchanges between user interface controls and the JavaBeans that you identified in the PDML.
- Performs validation of user data for common integer and character data types, and defines an interface that allows you to implement custom validation. If data is found to be invalid, an error message is displayed to the user.
- Defines standardized processing for Commit, Cancel and Help events, and provides a framework for handling custom events.
- Manages interactions between user interface controls based on state information defined in the PDML. (For example, you may want to disable a group of controls whenever the user selects a particular radio button.)

The package com.ibm.as400.ui.framework.java contains the Graphical Toolbox runtime API.

The elements of the Graphical Toolbox runtime environment are shown in the figure below. Your Java program is a client of one or more of the objects in the **Runtime Managers** box.

Graphical Toolbox Runtime Environment



Examples

Assume that the panel **MyPanel** is defined in the file **TestPanels.pdml**, and that a properties file **TestPanels.properties** is associated with the panel definition. Both files reside in the directory **com/ourCompany/ourPackage**, which is accessible either from a directory defined in the classpath or from a ZIP or JAR file defined in the classpath.

Example: Creating and displaying a panel

The following code creates and displays the panel:

```
import com.ibm.as400.ui.framework.java.*;

// Create the panel manager. Parameters:
// 1. Resource name of the panel definition
// 2. Name of panel
// 3. List of DataBeans omitted
```

```
PanelManager pm = null;
try {
  pm = new PanelManager("com.ourCompany.ourPackage.TestPanels",
                        "MyPanel", null);
}

catch (DisplayManagerException e) {
  e.displayUserMessage(null);
  System.exit(-1);
}

// Display the panel
pm.setVisible(true);
```

Example: Creating a dialog

Once the DataBeans that supply data to the panel have been implemented and the attributes have been identified in the PDML, the following code may be used to construct a fully-functioning dialog:

```
import com.ibm.as400.ui.framework.java.*;
import java.awt.Frame;

// Instantiate the objects which supply data to the panel
TestDataBean1 db1 = new TestDataBean1();
TestDataBean2 db2 = new TestDataBean2();

// Initialize the objects
db1.load();
db2.load();

// Set up to pass the objects to the UI framework
DataBean[] dataBeans = { db1, db2 };

// Create the panel manager. Parameters:
// 1. Resource name of the panel definition
// 2. Name of panel
// 3. List of DataBeans
// 4. Owner frame window

Frame owner;
...
PanelManager pm = null;
try {
  pm = new PanelManager("com.ourCompany.ourPackage.TestPanels",
                        "MyPanel", dataBeans, owner);
}

catch (DisplayManagerException e) {
  e.displayUserMessage(null);
  System.exit(-1);
}

// Display the panel
pm.setVisible(true);
```

Example: Using the dynamic panel manager

A new service has been added to the existing panel manager. The dynamic panel manager dynamically sizes the panel at runtime. Let's look at the **MyPanel** example again, using the dynamic panel manager:

```
import com.ibm.as400.ui.framework.java.*;
```

```
// Create the dynamic panel manager. Parameters:
// 1. Resource name of the panel definition
// 2. Name of panel
// 3. List of DataBeans omitted

DynamicPanelManager dpm = null;
try {
  pm = new DynamicPanelManager("com.ourCompany.ourPackage.TestPanels",
                               "MyPanel", null);
}

catch (DisplayManagerException e) {
  e.displayUserMessage(null);
  System.exit(-1);
}

// Display the panel
pm.setVisible(true);
```

When you instantiate this panel application you can see the dynamic sizing feature of the panels. Move your cursor to the edge of the GUI's display and, when you see the sizing arrows, you can change the size of the panel.

Graphical Toolbox examples

We have provided examples to show you how to implement the tools within Graphical Toolbox for your own UI programs.

- Construct and display a panel: Shows you how to construct a simple panel. The example then shows you how to build a small Java application that displays the panel. When the user enters data in the text field and clicks on the Close button, the application will echo the data to the Java console. This example illustrates the basic features and operation of the Graphical Toolbox environment as a whole.
- Create and display a panel: Shows you how to create and display a panel when the panel and properties file are in the same directory.
- Construct a fully-functional dialog: Once the DataBeans that supply data to the panel have been implemented and the attributes have been identified in the PDML this example shows you how to construct a fully-functioning dialog
- Size a panel using the dynamic panel manager: The dynamic panel manager dynamically sizes the panel at runtime.
- Editable combobox: Shows you a data bean coding example for an editable combobox.

The following examples show you how the GUIBuilder can help you to create:

- Panels: Shows you how to create a sample panel and the data bean code that runs the panel
- Deckpanes: Shows you how to create a deckpane and what a final deckpane may look like
- Property sheets: Shows you how to create a property sheet and what a final property sheet may look like
- Split panes: Shows you how to create a split pane and what a final split pane may look like
- Tabbed panes: Shows you how to create a tabbed pane and what a final tabbed pane may look like
- Wizards: Shows you how to create a wizard and what the final product may look like
- Toolbars: Shows you how to create a tool bar and what a final tool bar may look like
- Menu bars: Shows you how to create a menu bar and what a final menu bar may look like
- Help: Shows how a Help Document is generated and ways to split the Help Document into topic pages. Also, see Editing Help Documents generated by GUI builder
- Sample: Shows what a whole PDML program may look like, including panels, a property sheet, a wizard, select/deselect, and menu options.

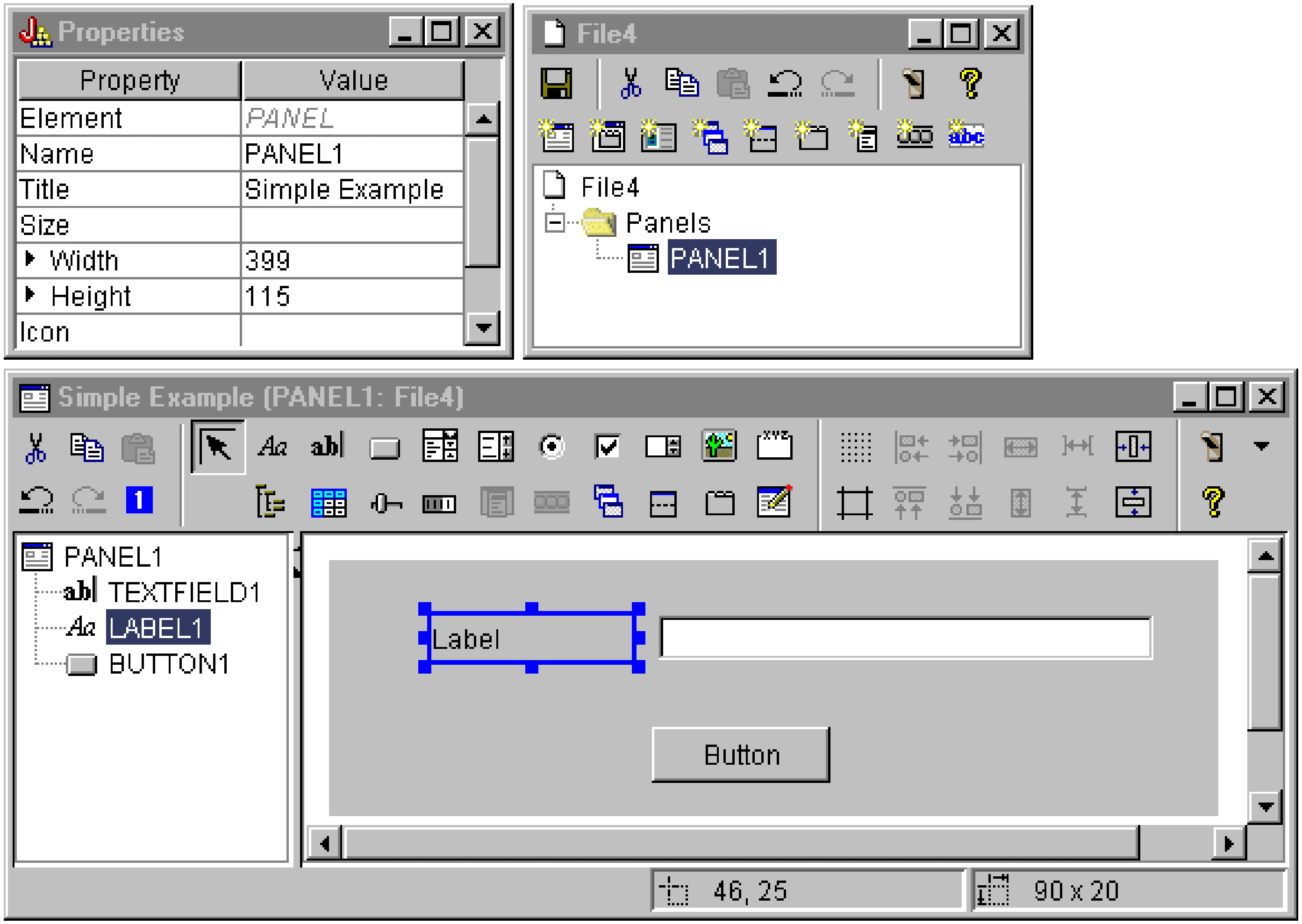
Graphical Toolbox Example

This example demonstrates how to use the Graphical Toolbox by constructing a simple panel. It is an overview that illustrates the basic features and operation of the Graphical Toolbox environment. After showing you how to construct a panel, the example goes on to show you how to build a small Java application that displays the panel. In this example, the user enters data in a text field and clicks on the **Close** button. The application then echos the data to the Java console.

Constructing the panel

When you start the GUI Builder, the Properties and GUI Builder windows appear. Create a new file named "MyGUI.pdml". For this example, insert a new panel. Click the "Insert Panel" icon in File Builder window. Its name is "PANEL1". Change the title by modifying information in the Properties window; type "Simple Example" in the "Title" field. Remove the three default buttons by selecting them with your mouse and pressing "Delete". Using the buttons in the Panel Builder window, add the three elements shown in the figure below: a label, a text field, and a pushbutton.

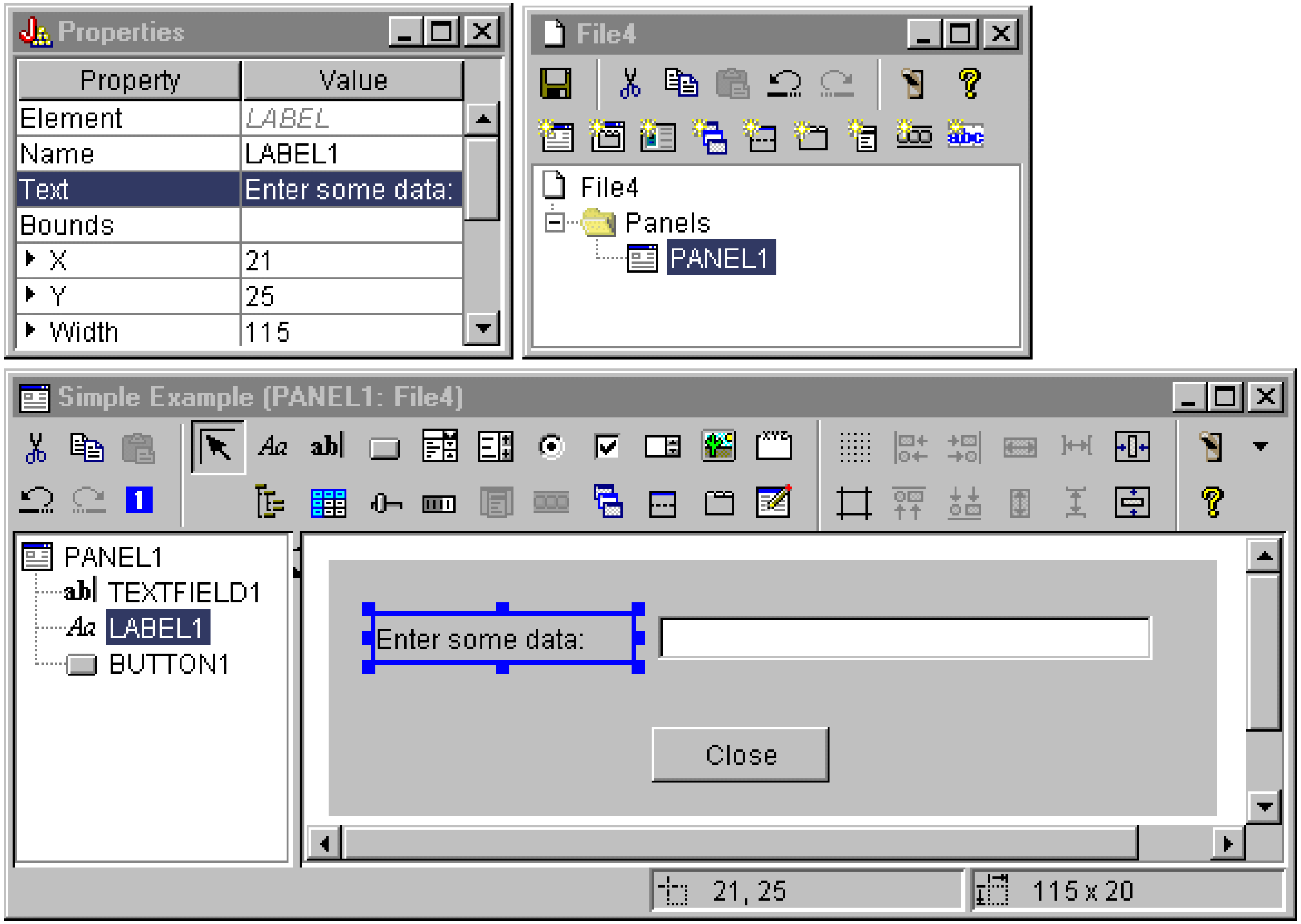
GUI Builder windows: Beginning to construct a panel



Property	Value
Element	PANEL
Name	PANEL1
Title	Simple Example
Size	
▸ Width	399
▸ Height	115
Icon	

By selecting the label, you can change its text in the Properties window. In this example, the same has been done for the pushbutton, changing its text to "Close".

GUI Builder windows: Changing text in the Properties window

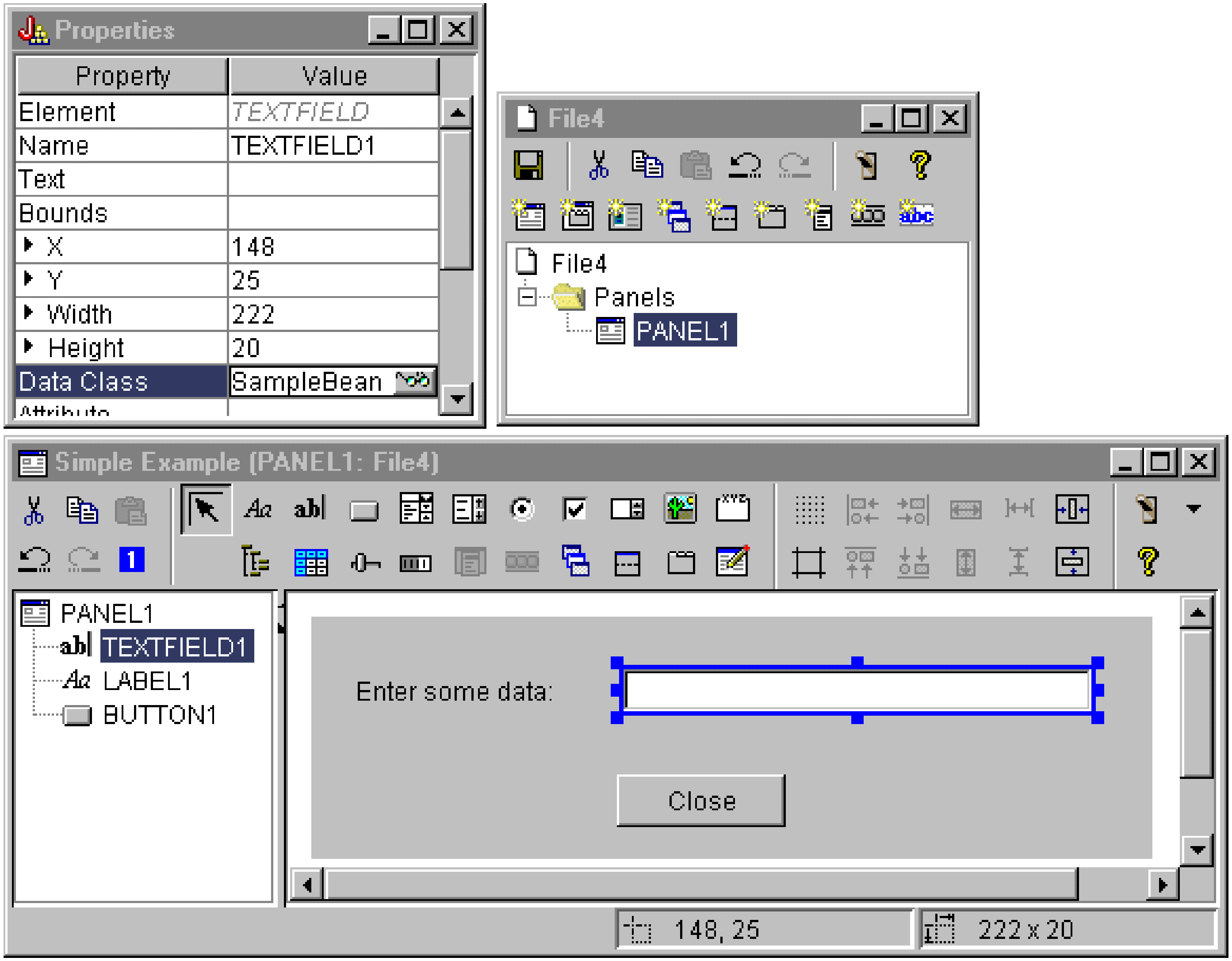


Property	Value
Element	LABEL
Name	LABEL1
Text	Enter some data:
Bounds	
X	21
Y	25
Width	115

Text field

The text field will contain data and, therefore, you can set several properties that will allow the GUI Builder to perform some additional work. For this example, you set the Data Class property to the name of a bean class named **SampleBean**. This databean will supply the data for this text field.

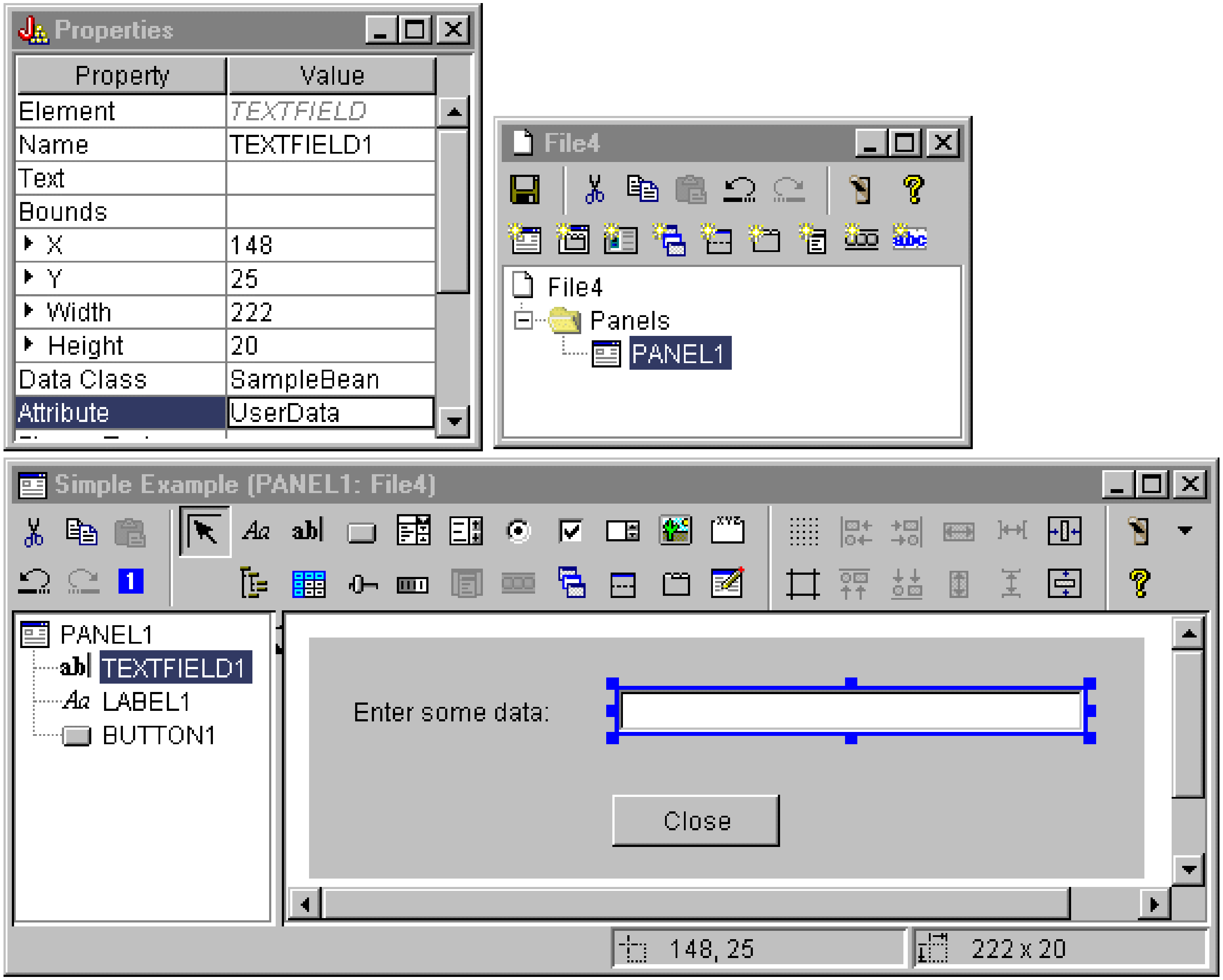
GUI Builder windows: Setting the Data Class property



Property	Value
Element	TEXTFIELD
Name	TEXTFIELD1
Text	
Bounds	
X	148
Y	25
Width	222
Height	20
Data Class	SampleBean
Attribute	

Set the Attribute property to the name of the bean property that will contain the data. In this case, the name is **UserData**.

GUI Builder windows: Setting the Attribute property

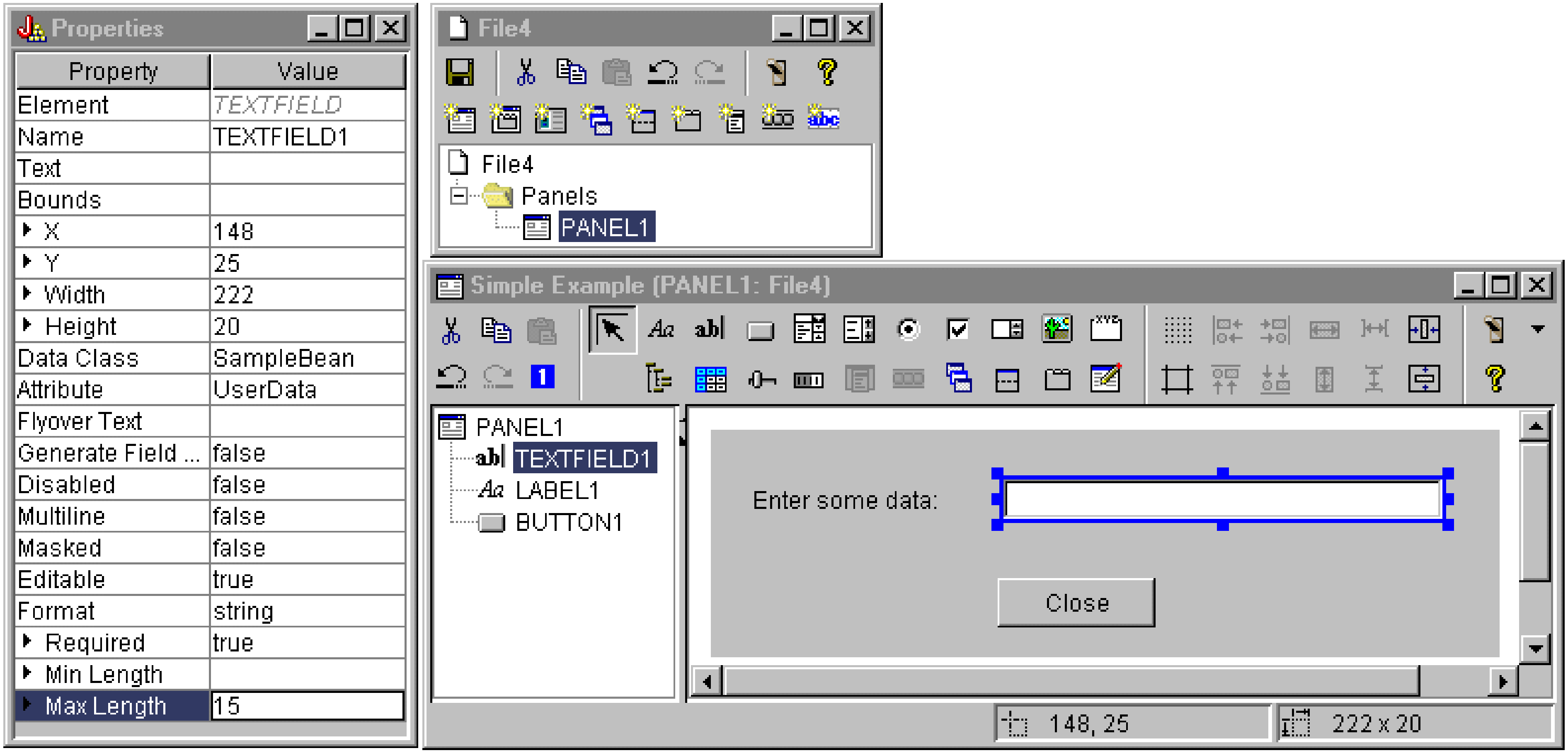


Property	Value
Element	TEXTFIELD
Name	TEXTFIELD1
Text	
Bounds	
X	148
Y	25
Width	222
Height	20
Data Class	SampleBean
Attribute	UserData

Following the above steps binds the **UserData** property to this text field. At run-time, the Graphical Toolbox obtains the initial value for this field by calling **`SampleBean.getUserData`**. The modified value is then sent back to the application when the panel closes by calling **`SampleBean.setUserData`**.

Specify that the user is required to supply some data, and that the data must be a string with a maximum length of 15 characters.

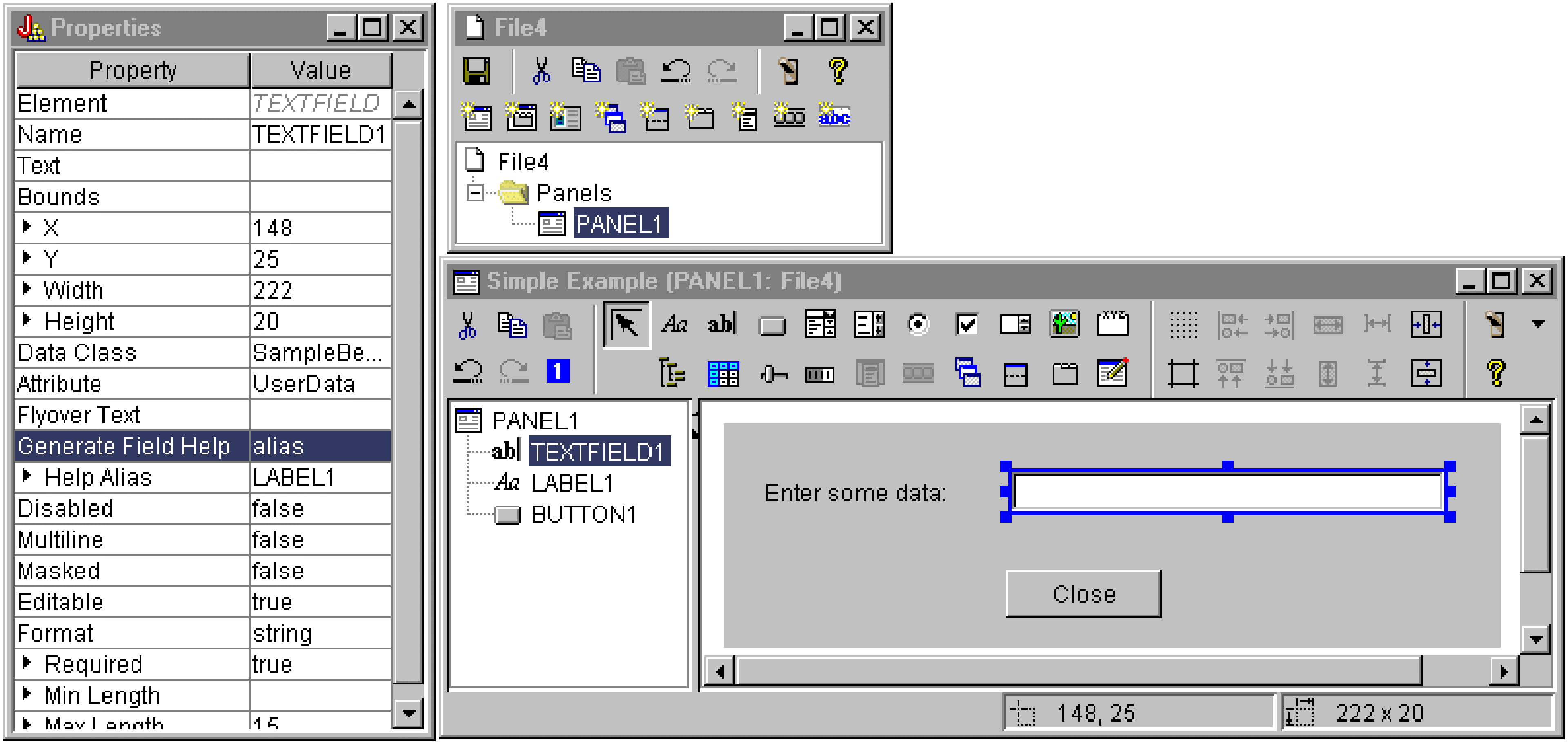
GUI Builder windows: Setting the maximum length of the text field



Property	Value
Element	TEXTFIELD
Name	TEXTFIELD1
Text	
Bounds	
X	148
Y	25
Width	222
Height	20
Data Class	SampleBean
Attribute	UserData
Flyover Text	
Generate Field ...	false
Disabled	false
Multiline	false
Masked	false
Editable	true
Format	string
Required	true
Min Length	
Max Length	15

Indicate that the context-sensitive help for the text field will be the help topic associated with the label "Enter some data".

GUI Builder windows: Setting context-sensitive help for the text field

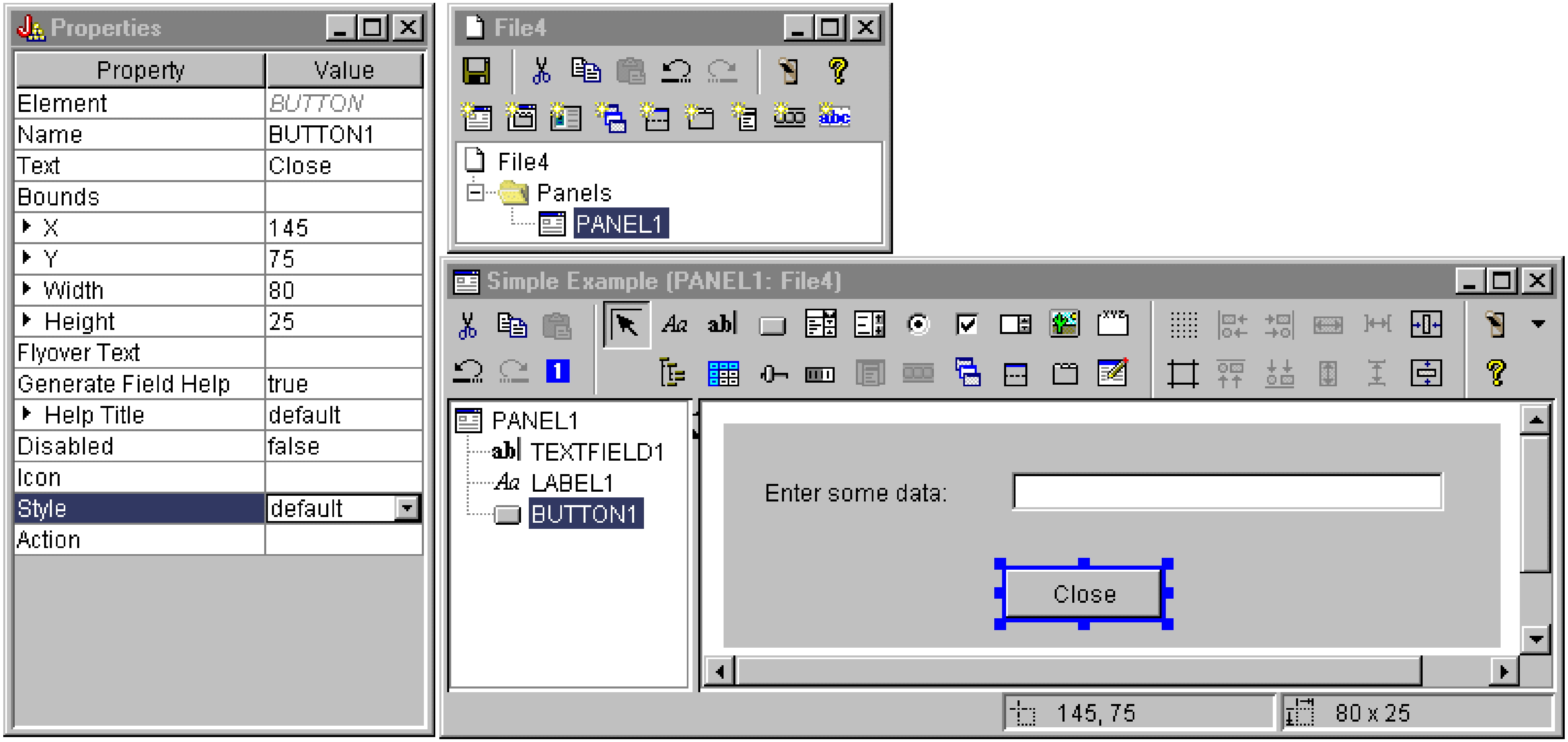


Property	Value
Element	TEXTFIELD
Name	TEXTFIELD1
Text	
Bounds	
X	148
Y	25
Width	222
Height	20
Data Class	SampleBe...
Attribute	UserData
Flyover Text	
Generate Field Help	alias
Help Alias	LABEL1
Disabled	false
Multiline	false
Masked	false
Editable	true
Format	string
Required	true
Min Length	
Max Length	15

Button

Modify the style property to give the button default emphasis.

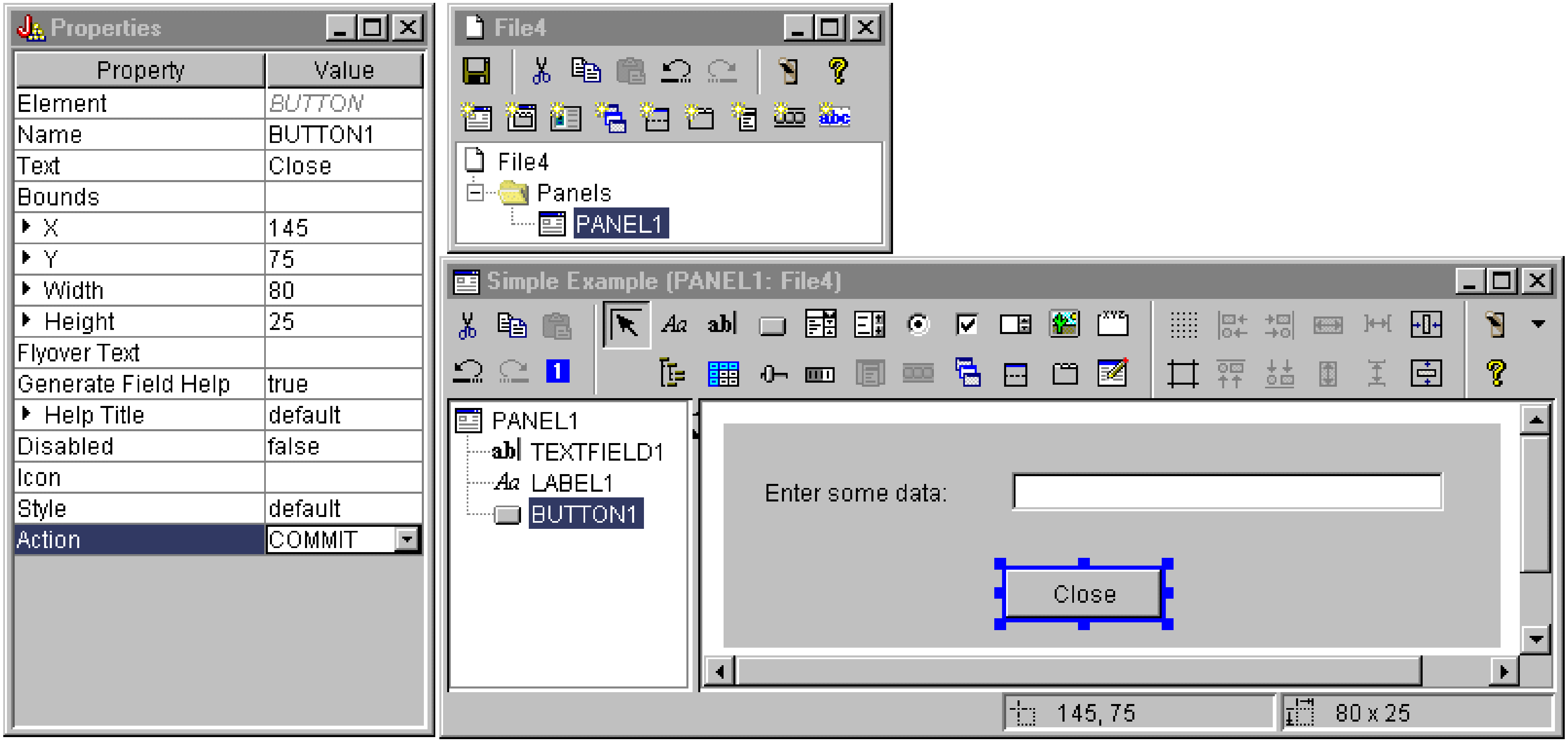
GUI Builder windows: Setting the Style property to give the button default emphasis



Property	Value
Element	*BUTTON*
Name	BUTTON1
Text	Close
Bounds	
▸ X	145
▸ Y	75
▸ Width	80
▸ Height	25
Flyover Text	
Generate Field Help	true
▸ Help Title	default
Disabled	false
Icon	
Style	default
Action	

Set the ACTION property to COMMIT, which causes the `setUserData` method on the bean to be called when the button is selected.

GUI Builder windows: Setting the Action property to COMMIT

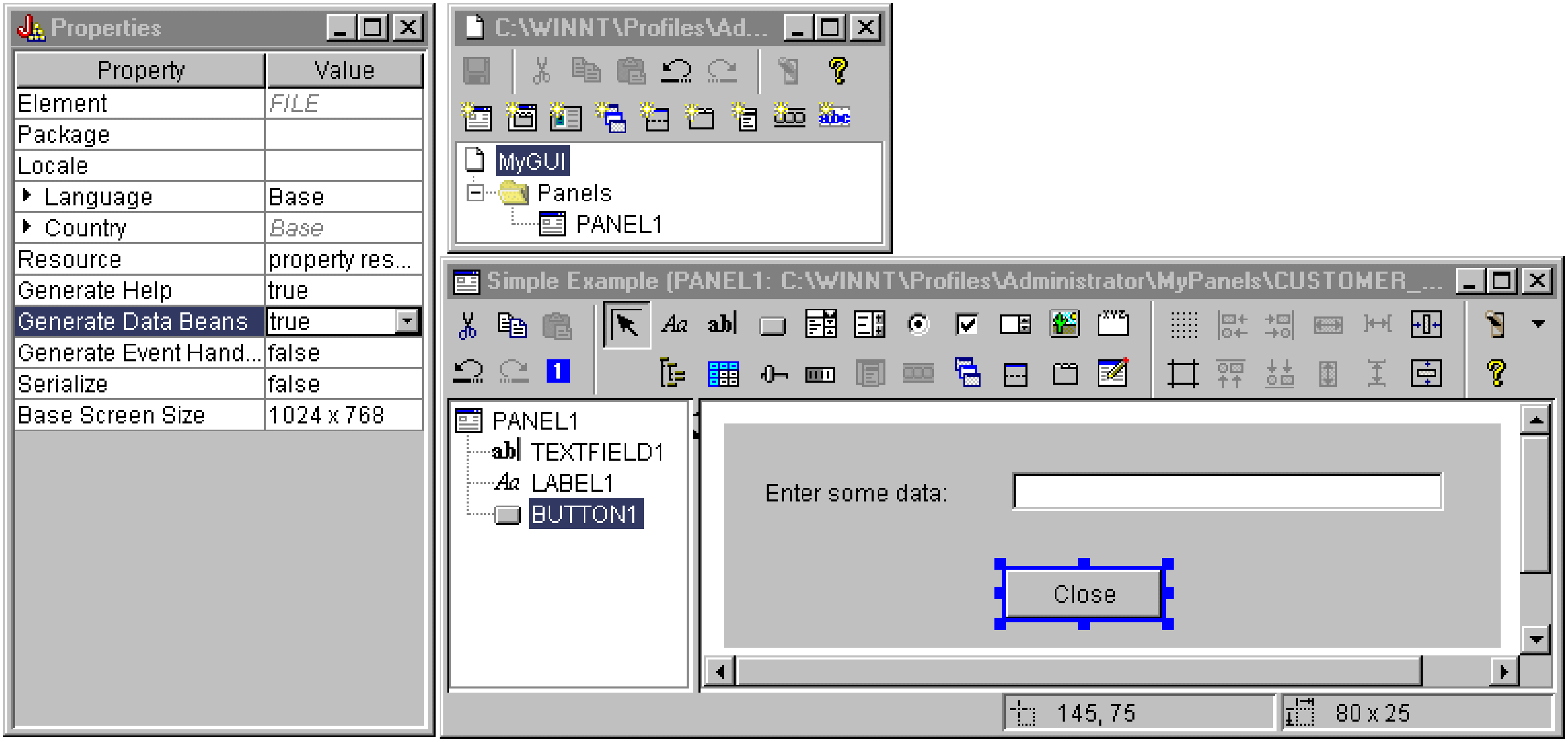


Properties

Property	Value
Element	BUTTON
Name	BUTTON1
Text	Close
Bounds	
X	145
Y	75
Width	80
Height	25
Flyover Text	
Generate Field Help	true
Help Title	default
Disabled	false
Icon	
Style	default
Action	COMMIT

Before you save the panel, set properties at the level of the PDML file to generate both the online help skeleton and the Java bean. Then you save the file by clicking on the 💾 icon in the main GUI Builder window. When prompted, specify a file name of **MyGUI.pdml**.

GUI Builder windows: Setting properties to generate the online help skeleton and the Java bean



Property	Value
Element	FILE
Package	
Locale	
Language	Base
Country	Base
Resource	property res...
Generate Help	true
Generate Data Beans	true
Generate Event Hand...	false
Serialize	false
Base Screen Size	1024 x 768

Generated files

After you save the panel definition, you can look at the files produced by the GUI Builder.

PDML file

Here is the content of **MyGUI.pdml** to give you an idea of how the Panel Definition Markup Language works. Because you use PDML only through the tools provided by the Graphical Toolbox, it is not necessary to understand the format of this file in detail:

```
<!-- Generated by GUIBUILDER -->
<PDML version="2.0" source="JAVA" basescreensize="1280x1024">

 <PANEL name="PANEL1">
  <TITLE>PANEL1</TITLE>
  <SIZE>351,162</SIZE>
  <LABEL name="LABEL1"">
   <TITLE>PANEL1.LABEL1</TITLE>
   <LOCATION>18,36</LOCATION>
   <SIZE>94,18</SIZE>
   <HELPLINK>PANEL1.LABEL1</HELPLINK>
  </LABEL>
  <TEXTFIELD name="TEXTFIELD1">
   <TITLE>PANEL1.TEXTFIELD1</TITLE>
   <LOCATION>125,31</LOCATION>
   <SIZE>191,26</SIZE>
   <DATACLASS>SampleBean</DATACLASS>
   <ATTRIBUTE>UserData</ATTRIBUTE>
   <STRING minlength="0" maxlength="15"/>
   <HELPALIAS>LABEL1</HELPALIAS>
  </TEXTFIELD>
  <BUTTON name="BUTTON1">
   <TITLE>PANEL1.BUTTON1</TITLE>
   <LOCATION>125,100</LOCATION>
   <SIZE>100,26</SIZE>
   <STYLE>DEFAULT</STYLE>
   <ACTION>COMMIT</ACTION>
   <HELPLINK>PANEL1.BUTTON1</HELPLINK>
  </BUTTON>
 </PANEL>

</PDML>
```

Resource bundle

Associated with every PDML file is a resource bundle. In this example, the translatable resources were saved in a PROPERTIES file, which is called **MyGUI.properties**. Notice that the PROPERTIES file also contains customization data for the GUI Builder.

```
##Generated by GUIBUILDER
BUTTON_1=Close
TEXT_1=
@GenerateHelp=1
@Serialize=0
@GenerateBeans=1
LABEL_1=Enter some data:
PANEL_1.Margins=18,18,18,18,18,18
PANEL_1=Simple Example
```

JavaBean

The example also generated a Java source code skeleton for the JavaBean object. Here is the content of **SampleBean.java**:

```
import com.ibm.as400.ui.framework.java.*;

public class SampleBean extends Object
    implements DataBean
{
    private String m_sUserData;
    public String getUserData()
    {
        return m_sUserData;
    }

    public void setUserData(String s)
    {
        m_sUserData = s;
    }

    public Capabilities getCapabilities()
    {
        return null;
    }

    public void verifyChanges()
    {
    }

    public void save()
    {
    }

    public void load()
    {
        m_sUserData ="";
    }
}
```

Note that the skeleton already contains an implementation of the gettor and settor methods for the `UserData` property. The other methods are defined by the `DataBean` interface and, therefore, are required.

The GUI Builder has already invoked the Java compiler for the skeleton and produced the corresponding class file. For the purposes of this simple example, you do not need to modify the bean implementation. In a real Java application you would typically modify the `load` and `save` methods to transfer data from an external data source. The default implementation of the other two methods is often sufficient. For more information, see the documentation on the **`DataBean`** interface in the javadocs for the PDML runtime framework.

Help file

The GUI Builder also creates an HTML framework called a Help Document. Help writers can easily manage help information by editing this file. For more information, see the following topics:

- Creating the Help Document
- Editing Help Documents generated by GUI builder

Constructing the application

Once the panel definition and the generated files have been saved, you are ready to construct the application. All you need is a new Java source file that will contain the main entry point for the application. For this example, the file is called **SampleApplication.java**. It contains the following code:

```
import com.ibm.as400.ui.framework.java.*;
import java.awt.Frame;
```

```
public class SampleApplication
{
    public static void main(String[] args)
    {
        // Instantiate the bean object that supplies data to the panel
        SampleBean bean = new SampleBean();

        // Initialize the object
        bean.load();

        // Set up to pass the bean to the panel manager
        DataBean[] beans = { bean };

        // Create the panel manager. Parameters:
        // 1. PDML file as a resource name
        // 2. Name of panel to display
        // 3. List of data objects that supply panel data
        // 4. An AWT Frame to make the panel modal

        PanelManager pm = null;
        try { pm = new PanelManager("MyGUI", "PANEL_1", beans, new Frame()); }
        catch (DisplayManagerException e)
        {
            // Something didn't work, so display a message and exit
            e.displayUserMessage(null);
            System.exit(1);
        }

        // Display the panel - we give up control here
        pm.setVisible(true);

        // Echo the saved user data
        System.out.println("SAVED USER DATA: '" + bean.getUserData() + "'");

        // Exit the application
        System.exit(0);
    }
}
```

It is the responsibility of the calling program to initialize the bean object or objects by calling **load**. If the data for a panel is supplied by multiple bean objects, then each of the objects must be initialized before passing them to the Graphical Toolbox environment.

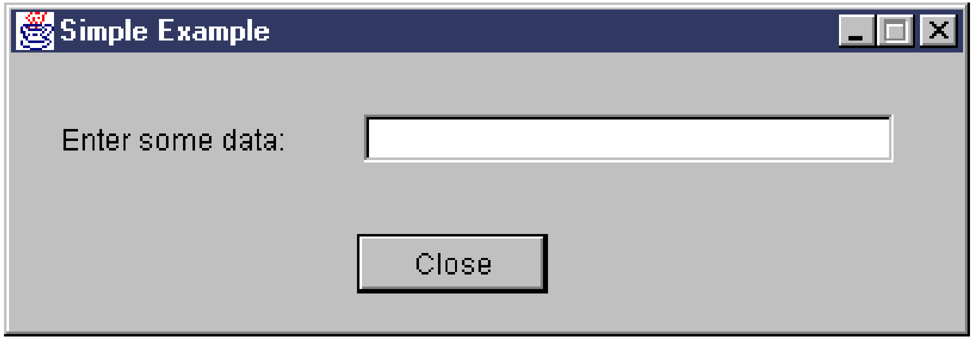
The class **com.ibm.as400.ui.framework.java.PanelManager** supplies the API for displaying standalone windows and dialogs. The name of the PDML file as supplied on the constructor is treated as a resource name by the Graphical Toolbox - the directory, ZIP file, or JAR file containing the PDML must be identified in the classpath.

Because a **Frame** object is supplied on the constructor, the window will behave as a modal dialog. In a real Java application, this object might be obtained from a suitable parent window for the dialog. Because the window is modal, control does not return to the application until the user closes the window. At that point, the application simply echoes the modified user data and exits.

Running the application

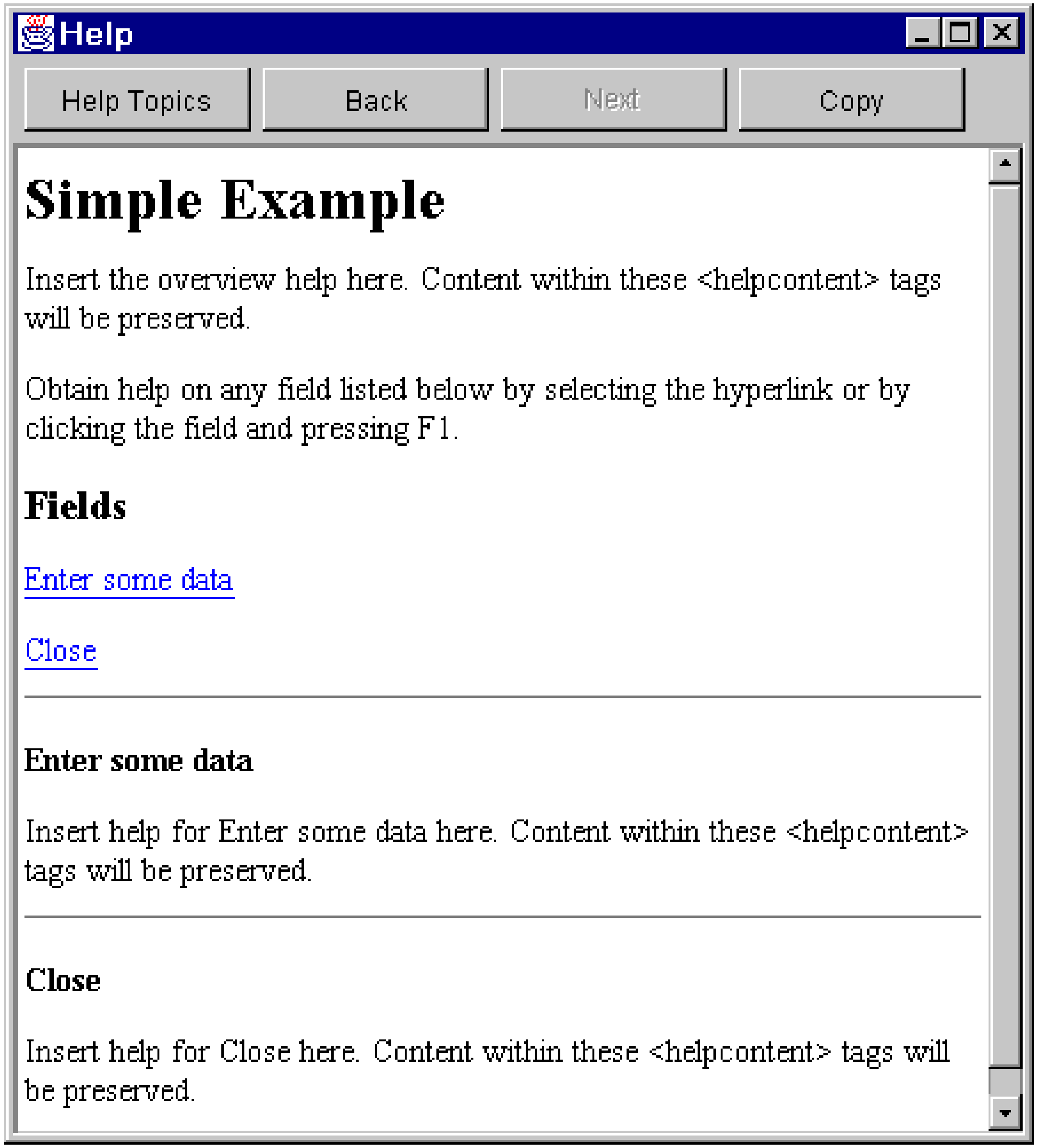
Here is what the window looks like when the application is compiled and run:

The Simple Example application window



If the user presses F1 while focus is on the text field, the Graphical Toolbox will display a help browser containing the online help skeleton that the GUI Builder generated.

The Simple Example online help skeleton



Simple Example

Insert the overview help here. Content within these <helpcontent> tags will be preserved.

Obtain help on any field listed below by selecting the hyperlink or by clicking the field and pressing F1.

Fields

Enter some data

Close

Enter some data

Insert help for Enter some data here. Content within these <helpcontent> tags will be preserved.

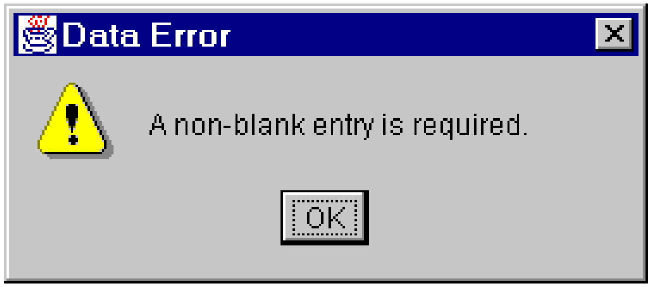
Close

Insert help for Close here. Content within these <helpcontent> tags will be preserved.

You can edit the HTML and add actual help content for the help topics shown.

If the data in the text field is not valid (for example, if the user clicked on the **Close** button without supplying a value), the Graphical Toolbox will display an error message and return focus to the field so that data can be entered.

Data Error message



A non-blank entry is required.

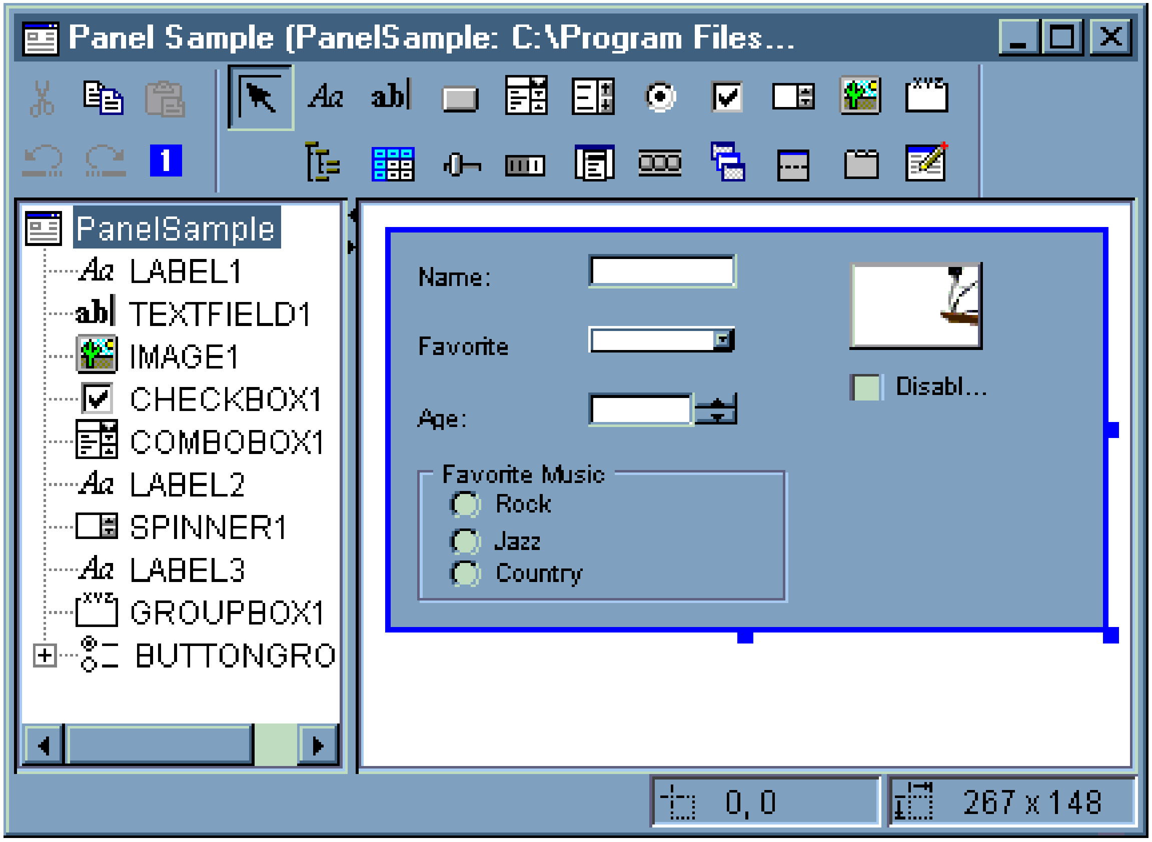
For information on how to run this sample as an applet, see Using the Graphical Toolbox in a Browser.

Creating a panel with GUIBuilder

Creating a panel with GUIBuilder is simple. From the GUIBuilder menu bar, select File, then select New File. Then select the "Insert New Panel" icon: The icons in the toolbar represent various components that you can add to the panel. Select the component you want and then click on the place you want to position it.

The following picture shows a panel that has been created with several of the options available to you:

Sample panel created with GUI Builder



This sample panel uses the following DataBean code to bring together the various components:

```
import com.ibm.as400.ui.framework.java.*;

public class PanelSampleDataBean extends Object
    implements DataBean
{
    private String m_sName;
    private Object m_oFavoriteFood;
    private ChoiceDescriptor[] m_cdFavoriteFood;
    private Object m_oAge;
    private String m_sFavoriteMusic;

    public String getName()
    {
        return m_sName;
    }

    public void setName(String s)
    {
        m_sName = s;
    }

    public Object getFavoriteFood()
    {
        return m_oFavoriteFood;
    }

    public void setFavoriteFood(Object o)
    {
        m_oFavoriteFood = o;
    }

    public ChoiceDescriptor[] getFavoriteFoodChoices()
    {
        return m_cdFavoriteFood;
    }

    public Object getAge()
    {
        return m_oAge;
    }

    public void setAge(Object o)
    {
        m_oAge = o;
    }

    public String getFavoriteMusic()
    {
        return m_sFavoriteMusic;
    }

    public void setFavoriteMusic(String s)
    {
        m_sFavoriteMusic = s;
    }

    public Capabilities getCapabilities()
    {
        return null;
    }

    public void verifyChanges()
    {
    }

    public void save()
    {
        System.out.println("Name = " + m_sName);
        System.out.println("Favorite Food = " + m_oFavoriteFood);
        System.out.println("Age = " + m_oAge);
        String sMusic = "";
        if (m_sFavoriteMusic != null)
        {
            if (m_sFavoriteMusic.equals("RADIOBUTTON1"))
                sMusic = "Rock";
            else if (m_sFavoriteMusic.equals("RADIOBUTTON2"))
                sMusic = "Jazz";
            else if (m_sFavoriteMusic.equals("RADIOBUTTON3"))
                sMusic = "Country";
        }
        System.out.println("Favorite Music = " + sMusic);
    }
    public void load()
    {
        m_sName = "Sample Name";
        m_oFavoriteFood = null;
        m_cdFavoriteFood = new ChoiceDescriptor[0];
        m_oAge = new Integer(50);
        m_sFavoriteMusic = "RADIOBUTTON1";
    }
}
```

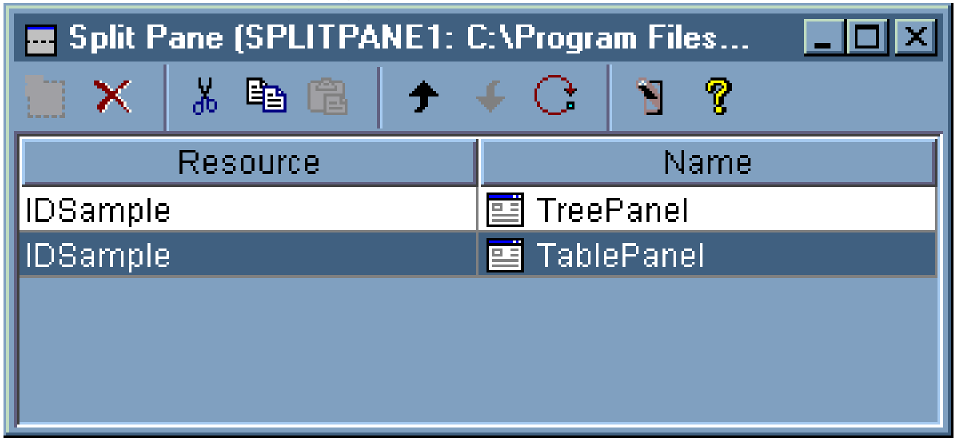
The panel is the most simple component available within the GUIBuilder, but from a simple panel, you can build great UI applications.

Creating a split pane with GUIBuilder

GUIBuilder makes creating a split pane simple. From the GUIBuilder menu bar, select File, then select New File.

Select the "Insert Split Pane" icon: The GUIBuilder creates a panel builder where you can insert the two components you want in your split pane:

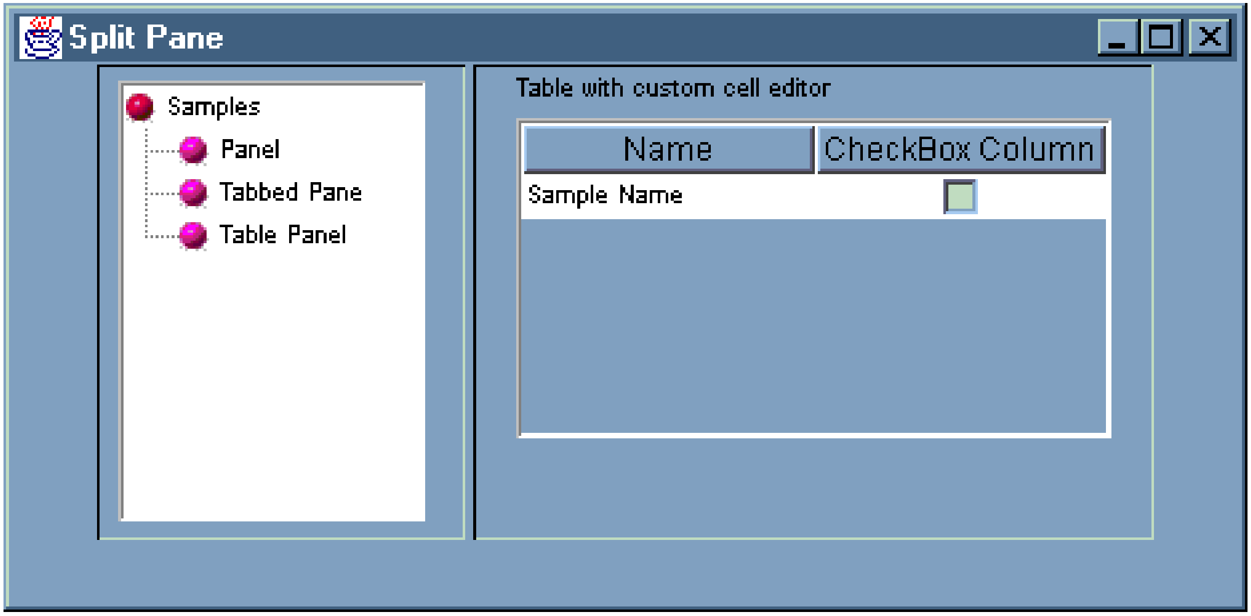
GUI Builder window: Building a split pane



Resource	Name
IDSample	TreePanel
IDSample	TablePanel

When you have created the split pane, use the icon to preview it. You will see both components you have selected like in the the following example:

Example split pane



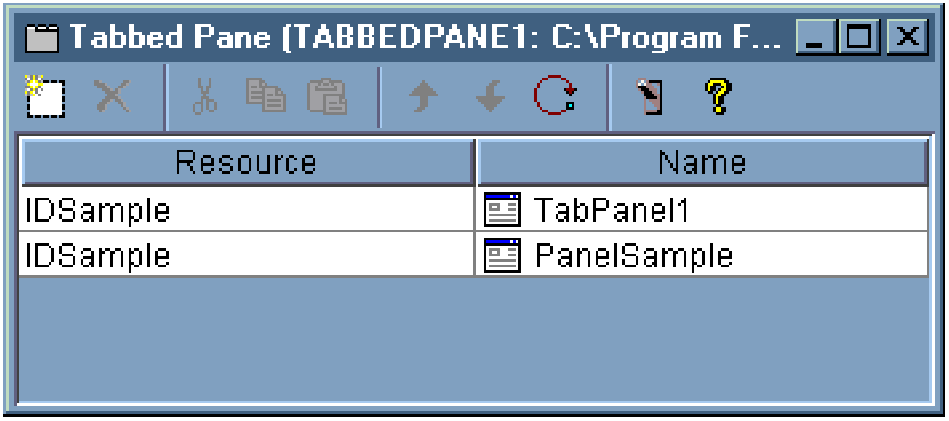
Name	CheckBox Column
Sample Name	□

Creating a tabbed pane with GUIBuilder

GUIBuilder makes creating a tabbed pane simple. From the GUIBuilder menu bar, select File, then select New File.

Select the "Insert Tabbed Pane" icon: The GUIBuilder creates a panel builder where you can insert the components for your tabbed pane:

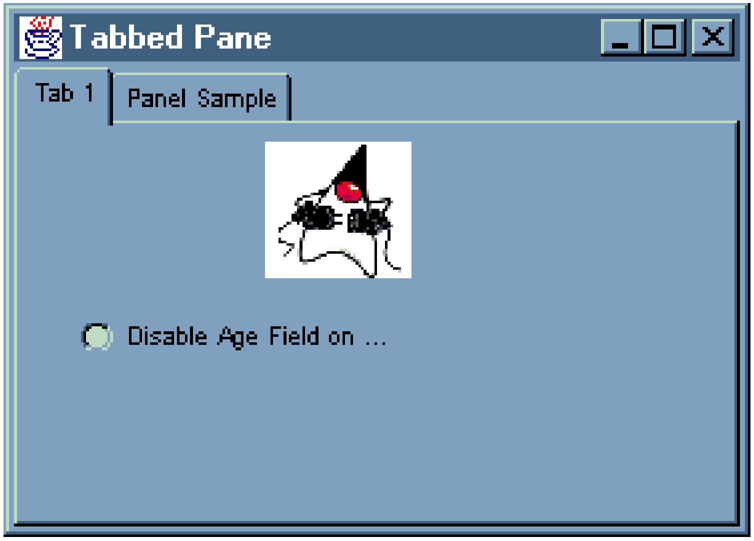
GUI Builder window: Building a tabbed pane



Resource	Name
IDSample	TabPanel1
IDSample	PanelSample

For this example, the tabbed pane looks like this:

Example tabbed pane

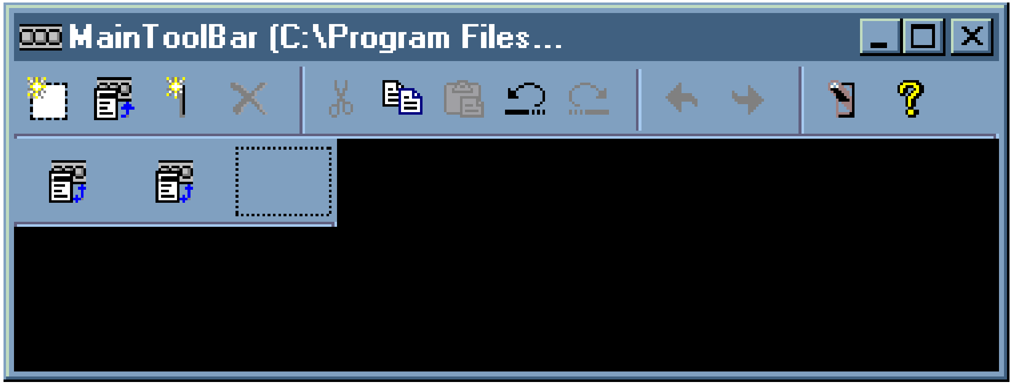


Creating a toolbar with GUIBuilder

GUIBuilder makes creating a toolbar simple. From the GUIBuilder menu bar, select File, then select New File.

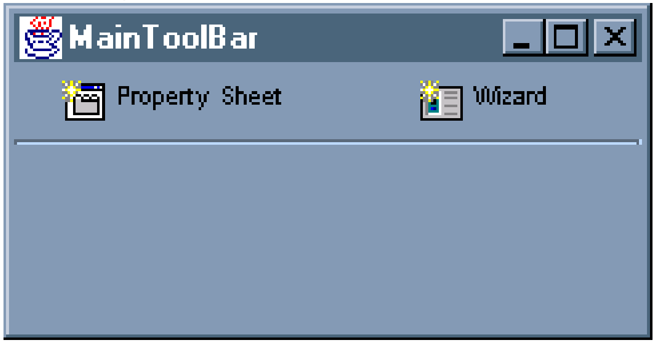
Select the "Insert Tool Bar" icon. The GUIBuilder creates a panel builder where you can insert the components for your toolbar:

GUI Builder window: Inserting a tool bar



When you have created the toolbar, use the icon to preview it. For this example, you can choose to either display a property sheet or wizard from the toolbar:

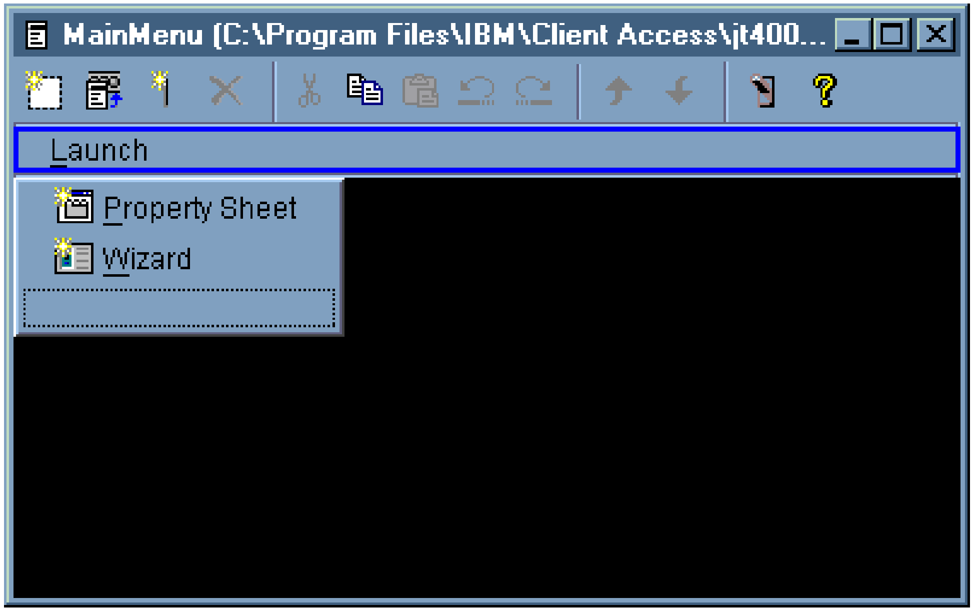
Previewing the tool bar



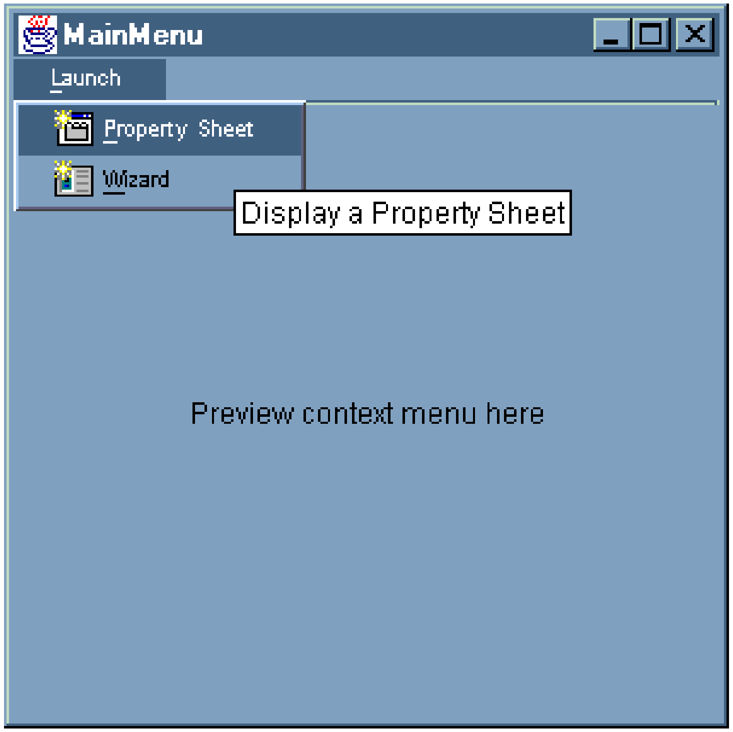
Creating a menubar with GUIBuilder

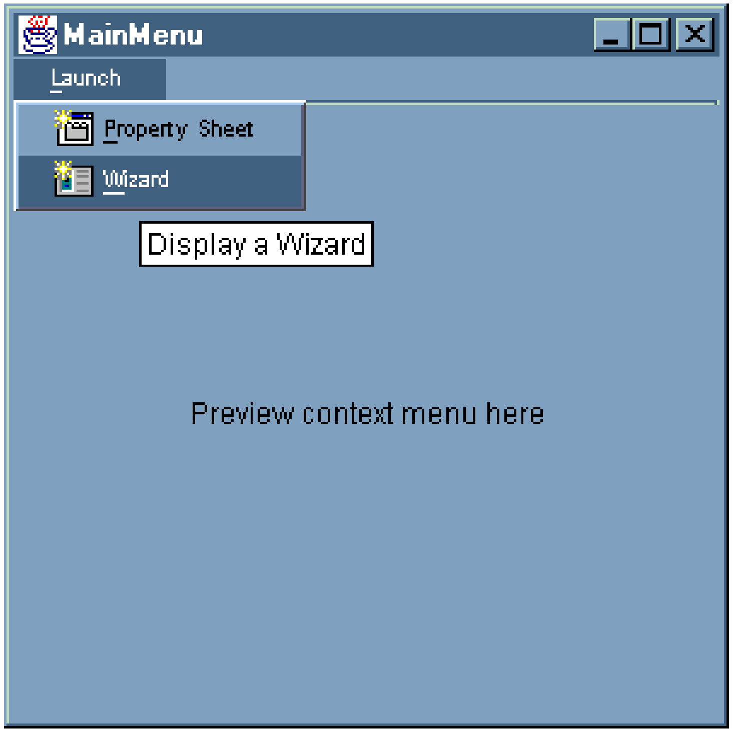
GUIBuilder makes creating a menubar simple. From the GUIBuilder menu bar, select File, then select New File.

Select the "Insert Menu" icon. The GUIBuilder creates a panel builder where you can insert the components for your menu:



When you have created the menu, use the icon to preview it. From the top bar, select "Launch". For this example, you can choose to display either a property sheet or a wizard:


MainMenu
Launch
Property Sheet
Wizard
Display a Property Sheet
Preview context menu here


MainMenu
Launch
Property Sheet
Wizard
Display a Wizard
Preview context menu here

Spinner

The spinner class is a component of the Graphical Toolbox. It has two small direction buttons that let the user scroll a list of predetermined values and select one. In some instances, the user may enter a new legal value.

There are several specific spinner classes that you can use:

- Calendar spinner
- Date spinner
- Time spinner
- Numeric spinner

Properties

Property	Description	Data Type	Flags*	Default Value	Valid Value
background	The background color of this component.	java.awt.Color	R, W, B	java.awt.Color.white	any instance of Color
foreground	The foreground color of this component.	java.awt.Color	R, W, B	java.awt.Color.black	any instance of Color
font	The font of this component.	java.awt.Font	R, W, B	new java.awt.Font("dialog", java.awt.Font.PLAIN, 12)	any instance of Font
opaque	Marks whether this component is opaque.	boolean	R, W, B	True	True - opaque False - not opaque
columns	The columns of the input field, which is used to validate the layout. **Note**: columns is a property derived from Swing JTextField. The function of columns in Spinner is similar to its function in Swing JTextField.	int	R, W, B	20	columns>0
enabled	Marks whether this component is enabled.	boolean	R, W, B	True	True - enabled False - disabled
editable	Marks whether this component is editable.	boolean	R, W, B	True	True - editable False - not editable
incrButtonArrowColor	The color of the increment arrow button.	java.awt.Color	R, W, B	java.awt.Color.black	Any instance of Color
decrButtonArrowColor	The color of the decrement arrow button.	java.awt.Color	R, W, B	java.awt.Color.black	Any instance of Color

orientation	The orientation of the arrow button.	int	R, W, B	0	0 - SPIN_VERTICAL (display name: VERTICAL) 1 - SPIN_HORIZONTAL (display name: HORIZONTAL)
wrap	Marks whether the wrap action is allowed. If wrap is true, the wrap action is allowed. In this case, assume Spinner's value is set to the maximum. If Spinner is scrolled up, the value will change to its minimum. Assuming Spinner's value is set to the minimum, if Spinner is scrolled down, the value will change to its maximum. If wrap is false, the wrap action is forbidden. If Spinner's value is set to the maximum, it cannot be scrolled up. If Spinner's Value is set to the minimum, it cannot be scrolled down.	boolean	R, W, B	True	True - the wrap action is allowed False - the wrap action is forbidden
* R = read, W = write, B = bound, E = expert					

Events

The spinner bean suite fires the following events:

- ChangeEvent
  - The change event notifies its registered listeners when the value of Spinner changes.

    Listener method: stateChanged

    increased(javax.swing.event.ChangeEvent)

    decreased(javax.swing.event.ChangeEvent)
- SpinnerErrorEvent
  - The SpinnerErrorEvent is used to notify you that some internal error has occurred. From the error code and error message, you can identify the error.

    Listener method:

internalError(com.ibm.spinner.SpinnerErrorEvent)

- PropertyChangeEvent
  - The PropertyChangeEvent is fired whenever the new spinner is different from the old one.

    Listener method:

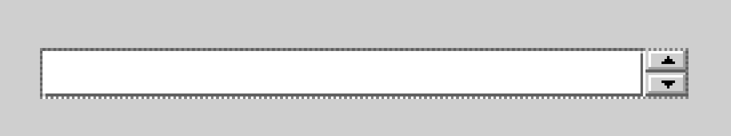
    propertyChange(java.beans.PropertyChangeEvent)

Methods

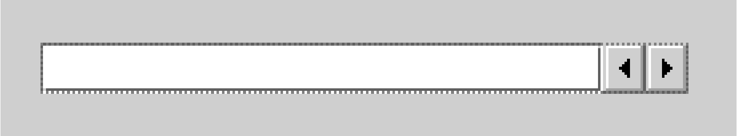
The spinner bean suite implements the following methods inherited from the class com.ibm.spinner.SpinnerGUI:

- public void scrollUp(): Increments the spinner's value
- public void scrollDown(): Decrements the spinner's value

Vertical Button Screen
The vertical button orientation spinner bean is shown below at its default orientation.



Horizontal Button Screen
The vertical button orientation spinner bean shown below is an example of after you set the "orientation" property to "HORIZONTAL".



The value that is currently selected is displayed in an input field. You can set the current value either by clicking on the arrow buttons or by typing a string into the input field.

Calendar spinner

The calendar spinner displays and spins the date and time. The date and time values can be changed by clicking on different sub fields and spinning on them. Alternatively, you can type in a date string to set the current value.

Properties

Property	Description	Data Type	Flags*	Default Value	Valid Value
wrapAssociated	Marks whether the changes of different field values of the date are associated. For example if wrapAssociated is true and the current value is "1998,12", the **year** will change from 1998 to 1999 when the value of the **month** is incremented.	boolean	R, W, B	True	True - different field values are associated False - different field values are not associated
timeZone	Represents a time zone offset. It is also makes changes for daylight savings time.	String	R, W, B	The system's local time zone.	valid time zone string
year	The year	int	R, W, B, E	The system's current year.	valid year
month	The month **Note**: the property value of month is from 0 to 11, but it displays 1 to 12 as its value in the UI. This is to keep it consistent with the JDK »and SDK.«	int	R, W, B, E	The system's current month.	integer from 1 to 12
day	The day.	int	R, W, B, E	The system's current day.	valid day
hour	The hour	int	R, W, B, E	The system's current hour.	valid hour
minute	The minute	int	R, W, B, E	The system's current minute.	valid minute
second	The second	int	R, W, B, E	The system's current second.	valid second
formatString	The user-defined pattern string for formatting and parsing date and time.	String	R, W, B	"dd-MMM-yy h:mm:ss a"	FULL - "EEEE,MMMM d,yyyy h:mm:ss 'o'clock' a z" LONG - "MMMM d,yyyy h:mm:ss a z" MEDIUM - "dd-MMM-yy h:mm:ss a" SHORT - "M/d/yy h:mm a"

formattingStyle	The ID of the format string. **Note**:This property is the same as the formatString property. However, to keep compatible with »a legacy API,« it cannot be eliminated. Therefore, it is a hidden property and can only be manipulated by the set/get methods.	int	R, W, B, H	2	0 - FULL 1 - LONG 2 - MEDIUM 3 - SHORT
caretPos	The caret position representing the current field to be changed. It can be one of YEAR, MONTH, DATE, HOUR, MINUTE, and SECOND. **Note**:The "caretPos" property of CalendarSpinner is not similar to the caret position defined in the TextField. Therefore, when you manipulate CalendarSpinner, the caret position displayed in the input field may not be consistent with the value of the caret position displayed in the property sheet.	int	R, W, B	0	0 - YEAR 1 - MONTH 2 - DATE 3 - HOUR 4 - MINUTE 5 - SECOND
datePartValue	The date value in long.	long	R, W, B, E	The current system date.	minimum<datePartValue <maximum
timePartValue	The time value in long.	long	R, W, B, E	The current system time.	minimum<timePartValue <maximum
calendar	The calendar value	java.util.Calendar	R, W, B	The current system calendar.	any instance of Calendar
value	The calendar value in long.	long	R, W, B, E	The current system calendar in long.	minimum<value<maximum
maximum	The maximum value.	java.util.Calendar	R, W, B	12/31/2050 11:59:59 PM	maximum>minimum
minimum	The minimum value.	java.util.Calendar	R, W, B	01/01/1950 12:00:00 AM	any instance of Calendar
dateString	The date and time shown in the entry field.	String	R, W, B, E	The current system date.	the instance of date string
date	The current date and time.	java.util.Date	R, W, B, H	The current system date and time.	any instance of Date
* R = read, W = write, B = bound,E =expert, H = hidden					

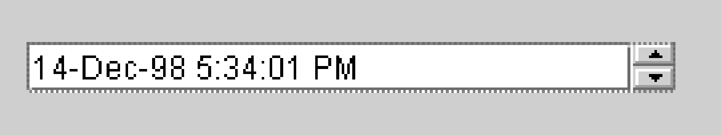
Events

The CalendarSpinner fires the DateChangedEvent. The listener method for this event is dateChanged(com.ibm.spinner.DateChangedEvent)

User interface

This section shows what the CalendarSpinner bean looks like and how to use it at runtime.

The currently selected value is displayed in an input field. The following picture shows what this may look like:

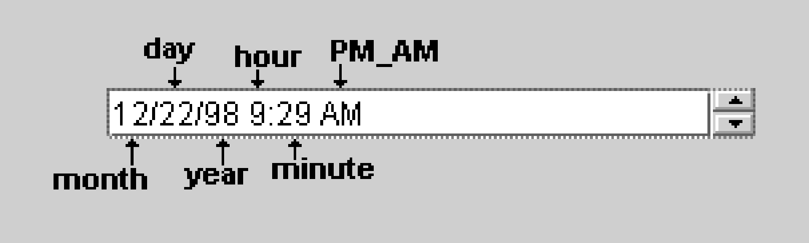


14-Dec-98 5:34:01 PM

You can change the date or time value by clicking on different sub fields within the input field and using the arrow buttons to spin them. Or you can set the values by typing a date or time string into the input field. If the input is invalid, the CalendarSpinner restores the previous value after you press the "Enter" or "Tab" key or when you change the focus to another component.

SHORT Style Screen

The SHORT Style CalendarSpinner Bean is shown below. This bean appears when you set the "formatString" property to "SHORT".



12/22/98 9:29 AM

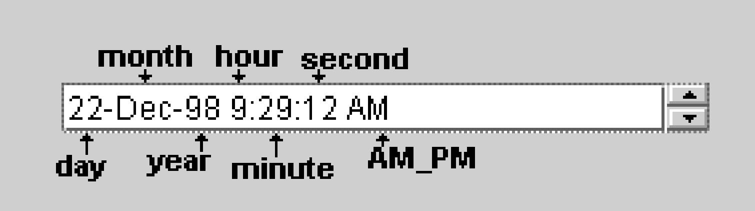
The short style includes six sub fields:

- Month
- Day
- Year
- Hour
- Minute
- AM/PM

The first five sub fields show digital values and can be changed by either by scrolling or by inputting a digital value. The AM/PM subfield can only be changed by scrolling.

MEDIUM Style Screen

The MEDIUM Style CalendarSpinner Bean is shown below. This bean appears when you set the "formatString" property to "MEDIUM".



22-Dec-98 9:29:12 AM

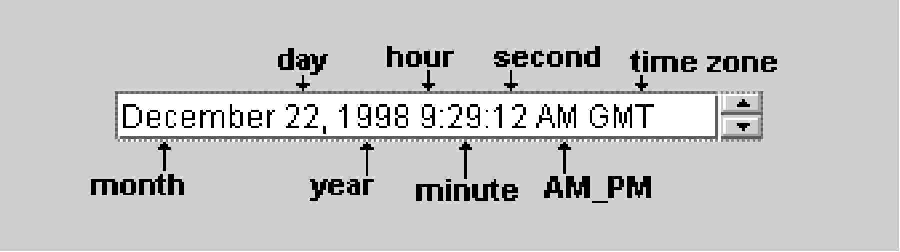
The medium style includes seven sub fields:

- Day
- Month
- Year
- Hour
- Minute
- Second
- AM/PM

The "day", "year", "hour", "minute", and "second" values can be changed either by scrolling or by inputting a digital value. The "month" and "PM_AM" values can only be changed by scrolling.

LONG Style Screen

The LONG Style CalendarSpinner Bean is shown below. This bean appears when you set the "formatString" property to "LONG".


December 22, 1998 9:29:12 AM GMT

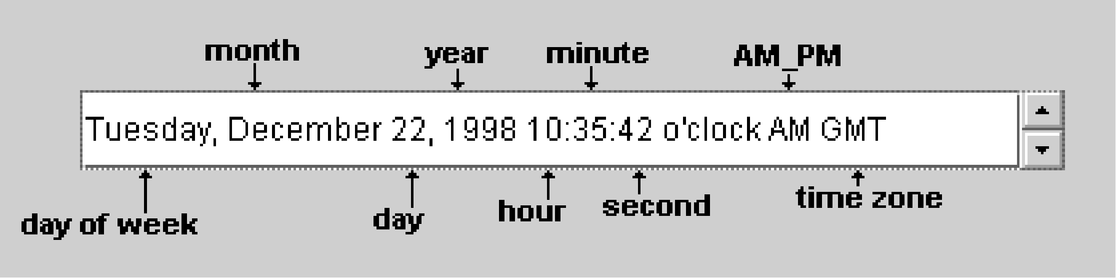
The long style includes eight sub fields.

- Month
- Day
- Year
- Hour
- Minute
- Second
- AM/PM
- Time zone

The "day", "year", "hour", "minute", and "second" values can be changed either by scrolling or by inputting a digital value. The "month", "PM_AM", and "time zone" values can be changed only by scrolling.

FULL Style Screen

The FULL Style CalendarSpinner Bean is shown below. This bean appears when you set the "formatString" property to "FULL".


Tuesday, December 22, 1998 10:35:42 o'clock AM GMT

The full style includes nine sub fields.

- Day of the week
- Month

- Day
- Year
- Hour
- Minute
- Second
- AM/PM
- Time zone

The "day", "year", "hour", "minute", and "second" values can be changed either by scrolling or by inputting a digital value. The "day of week", "month", "PM_AM" and "time zone" values can only be changed by scrolling.

Date spinner

The date spinner displays and spins the date. You can change the date value by clicking on different sub fields and spinning on them. You can also type in a date string to set the current value.

Properties

Property	Description	Data Type	Flags*	Default Value	Valid Value
wrapAssociated	Marks whether the changes of different field values are associated. For example if wrapAssociated is true and the current value is "1998,12", the **year** will change from 1998 to 1999 when the value of the **month** is incremented.	boolean	R, W, B	True	True - field values are associated False - field values are not associated
year	The year.	int	R, W, B, E	The current system year.	valid year
month	The month. **Note**: the property value of month is from 0 to 11, but it displays 1 to 12 as its value in the UI. This is to keep it consistent with WebRunner and JDK	int	R, W, B, E	The current system month.	integer from 1 to 12
day	The day	int	R, W, B, E	The current system day.	valid days
formatString	The user-defined pattern string for formatting and parsing the date.	String	R, W, B	"dd - MMM - yy"	FULL - "EEEE,MMMM d,yyyy" LONG - "MMMM d,yyyy" MEDIUM - "dd-MMM-yy" SHORT - "M/d/yy"
formattingStyle	The ID of format string. **Note**:This property is the same as the formatString property. However, to keep compatible with Webrunner API, it cannot be eliminated. Therefore, it is a hidden property and can only be manipulated by the set/get methods.	int	R, W, B, H	2	0 - FULL 1 - LONG 2 - MEDIUM 3 - SHORT

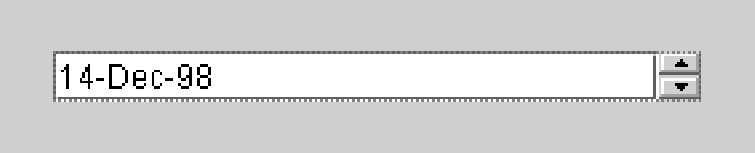
caretPos	The caret position which represents the current field to be changed. It can be one of YEAR, MONTH, or DATE. **Note**:The "caretPos" property of DateSpinner not similar to the caret position defined in the TextField. Therefore, when you manipulate DateSpinner, the caret position displayed in the input field may not be consistent with the value of the caret position displayed in the property sheet.	int	R, W, B	0	0 - YEAR 1 - MONTH 2 - DATE
value	The calendar value in long.	long	R, W, B, E	The current system calendar in long.	minimum<value <maximum
calendar (display name:Date)	The calendar value in calendar.	java.util.Calendar	R, W, B	The current system calendar.	any instance of Calendar
maximum	The maximum value.	java.util.Calendar	R, W, B	12/31/2050	maximum>minimum
minimum	The minimum value.	java.util.Calendar	R, W, B	01/01/1950	any instance of Calendar
dateString	The date shown in the entry field.	String	R, W, B, E	The current system date.	any instance of valid date string
* R = read, W = write, B=bound, E =expert, H = hidden					

Events

The date spinner fires the DateChangedEvent. The listener method for this event is dateChanged(com.ibm.spinner.DateChangedEvent)

User interface

The currently selected value is displayed in an input field. The following picture shows what this may look like:



14-Dec-98

You can change the date value by clicking on different sub fields within the input field and using the arrow buttons to spin them. You can also set the values by typing a date string into the input field. If the input is invalid, the DateSpinner restores the previous value after you press the "Enter" or "Tab" key or when you change the focus to another component.

Time spinner

The time spinner displays and spins the time. You can change the time value by clicking on different subfields and spinning on them. You can also type in a time string to set the current value.

Properties

Property	Description	Data Type	Flags*	Default Value	Valid Value
wrapAssociated	Marks whether the different field values are associated.	boolean	R, W, B	True	True - different field values are associated False - different field values are not associated
timeZone	Represents a time zone offset. It is also makes changes for daylight savings time.	String	R, W, B	system's local time zone	valid time zone string
hour	The hour	int	R, W, B, E	The current system hour.	valid hour
minute	The minute	int	R, W, B, E	The current system minute.	valid minute
second	The second	int	R, W, B, E	The current system second.	valid second
formatString	The user-defined pattern string for formatting and parsing time.	String	R, W, B	"h:mm:ss a"	FULL - " h:mm:ss 'o'clock' a z" LONG - "h:mm:ss a z" MEDIUM - "h:mm:ss a" SHORT - "h:mm a"
formattingStyle	The ID of format string. **Note**:This property is the same as the formatString property. However, to keep compatible with Webrunner API, it cannot be eliminated. Therefore, it is a hidden property and can only be manipulated by the set/get methods.	int	R, W, B, H	2	0 - FULL 1 - LONG 2 - MEDIUM 3 - SHORT

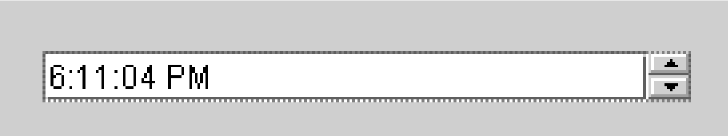
caretPos	The caret position which represents the current field to be changed. It can be one of: HOUR, MINUTE, and SECOND. **Note**:The "caretPos" property of TimeSpinner is not similar to the caret position defined in the TextField. Therefore, when you manipulate TimeSpinner, the caret position displayed in the input field may not be consistent with the value of the caret position displayed in the property sheet.	int	R, W, B	3	3 - HOUR 4 - MINUTE 5 - SECOND
value	The calendar value in long.	long	R, W, B, E	The current system calendar in long.	minimum<value< maximum
calendar (display name:Time)	The calendar value in calendar.	java.util.Calendar	R, W, B	The current system calendar.	any instance of Calendar
maximum	The maximum value.	java.util.Calendar	R, W, B	11:59:59 PM	maximum>minimum
minimum	The minimum value.	java.util.Calendar	R, W, B	00:00:00 AM	any instance of Calendar
dateString	The time shown in the entry field.	String	R, W, B, E	The current system time.	any instance of valid time string
* R = read, W = write, B = bound, E = expert, H = hidden					

Events

The time spinner fires the DateChangedEvent. The listener method for this event is dateChanged(com.ibm.spinner.DateChangedEvent)

User interface

The currently selected value is displayed in an input field. The following picture shows what this may look like:



You can change the time value by clicking on different subfields within the input field and using the arrow buttons to spin them. You can also set the values by typing a time string into the input field. If the input is invalid, the TimeSpinner restores the previous value after you press the "Enter" or "Tab" key or when you change the focus to another component.

Numeric spinner

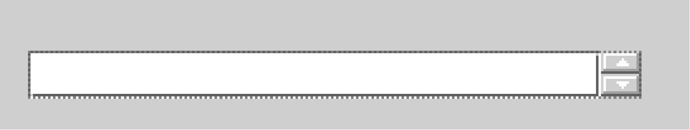
The numeric spinner scrolls through a list of integers within a bounded range. The current selected value is displayed in a text field. You can also enter an integer value as the current value.

Properties

Property	Description	Data Type	Flags*	Default Value	Valid Value
model	The data model used in the Numeric Spinner.	com.sun.java.swing.BoundedRangeModel	R, W, B, E	a new instance of com.sun.java.swing. DefaultBoundedRangeModel (0,0,0,100)	any instance of BoundedRangeModel
increment	The step value by which the value is changed every time.	int	R, W, B	1	increment>0
value	The current value.	int	R, W, B	0	maximum>=value>=minimum
minimum	The minimum value.	int	R, W, B	0	minimum<=maximum
maximum	The maximum value.	int	R, W, B	100	maximum>=minimum
* R = read, W = write, B = bound, E = expert					

User interface

The currently selected value is displayed in an input field. The following picture shows what this may look like:



You can change the numeric value by clicking on different subfields within the input field and using the arrow buttons to spin them. You can also set the values by typing a number into the input field. If the input is invalid, the NumericSpinner restores the previous value after you press the "Enter" or "Tab" key or when you change the focus to another component.

Using the Graphical Toolbox in a browser

You can use the Graphical Toolbox to build panels for Java applets that run in a web browser. This section describes how to convert the simple panel from the Graphical Toolbox Example to run in a browser. The minimum browser levels supported are Netscape 4.05 and Internet Explorer 4.0. In order to avoid having to deal with the idiosyncrasies of individual browsers, we recommend that your applets run using Sun's Java Plug-in. Otherwise, you will need to construct signed JAR files for Netscape, and separate signed CAB files for Internet Explorer.

Constructing the applet

The code to display a panel in an applet is nearly identical to the code used in the Java application example, but first, the code must be repackaged in the `init` method of a **JApplet** subclass. Also, we must add some code to ensure that the applet panel is sized to the dimensions specified in the panel's PDML definition. Here is the source code for our example applet, **SampleApplet.java**.

```
import com.ibm.as400.ui.framework.java.*;

import javax.swing.*;
import java.awt.*;
import java.applet.*;
import java.util.*;

public class SampleApplet extends JApplet
{
    // The following are needed to maintain the panel's size
    private PanelManager        m_pm;
    private Dimension           m_panelSize;

    // Define an exception to throw in case something goes wrong
    class SampleAppletException extends RuntimeException {}

    public void init()
    {
        System.out.println("In init!");

        // Trace applet parameters
        System.out.println("SampleApplet code base=" + getCodeBase());
        System.out.println("SampleApplet document base=" + getDocumentBase());

        // Do a check to make sure we're running a Java virtual machine that's compatible with Swing 1.1
        if (System.getProperty("java.version").compareTo("1.1.5") < 0)
            throw new IllegalStateException("SampleApplet cannot run on Java VM version " +
                                            System.getProperty("java.version") + " - requires 1.1.5 or higher");

        // Instantiate the bean object that supplies data to the panel
        SampleBean bean = new SampleBean();

        // Initialize the object
        bean.load();

        // Set up to pass the bean to the panel manager
        DataBean[] beans = { bean };

        // Update the status bar
        showStatus("Loading the panel definition...");

        // Create the panel manager. Parameters:
        // 1. PDML file as a resource name
        // 2. Name of panel to display
        // 3. List of data objects that supply panel data
        // 4. The content pane of the applet

        try { m_pm = new PanelManager("MyGUI", "PANEL_1", beans, getContentPane()); }
        catch (DisplayManagerException e)
        {
            // Something didn't work, so display a message and exit
            e.displayUserMessage(null);
            throw new SampleAppletException();
        }

        // Identify the directory where the online help resides
        m_pm.setHelpPath("http://MyDomain/MyDirectory/");

        // Display the panel
        m_pm.setVisible(true);
    }

    public void start()
    {
        System.out.println("In start!");

        // Size the panel to its predefined size
        m_panelSize = m_pm.getPreferredSize();
        if (m_panelSize != null)
        {
            System.out.println("Resizing to " + m_panelSize);
            resize(m_panelSize);
        }
        else
            System.err.println("Error: getPreferredSize returned null");
    }

    public void stop()
    {
        System.out.println("In stop!");
    }

    public void destroy()
    {
        System.out.println("In destroy!");
    }

    public void paint(Graphics g)
    {
        // Call the parent first
        super.paint(g);

        // Preserve the panel's predefined size on a repaint
        if (m_panelSize != null)
            resize(m_panelSize);
    }
}
```

The applet's content pane is passed to the Graphical Toolbox as the container to be laid out. In the **start** method, we size the applet pane to its correct size, and we override the **paint** method in order to preserve the panel's size when the browser window is resized.

When running the Graphical Toolbox in a browser, the HTML files for your panel's online help cannot be accessed from a JAR file. They must reside as separate files in the directory where your applet resides. The call to **`PanelManager.setHelpPath`** identifies this directory to the Graphical Toolbox, so that your help files can be located.

HTML tags

Because we recommend the use of Sun's Java Plug-in to provide the correct level of the Java runtime environment, the HTML for identifying a Graphical Toolbox applet is not as straightforward as we would like. Fortunately, the same HTML template may be reused, with only slight changes, for other applets. The markup is designed to be interpreted in both Netscape Navigator and Internet Explorer, and it generates a prompt for downloading the Java Plug-in from Sun's web site if it's not already installed on the user's machine. For detailed information on the workings of the Java Plug-in see the Java Plug-in HTML Specification.

Here is our HTML for the sample applet, in the file **MyGUI.html**:

```
<html>

<head>
<title>Graphical Toolbox Demo</title>
</head>

<body>
<h1>Graphical Toolbox Demo Using Java(tm) Plug-in</h1>
<p>

<!-- BEGIN JAVA(TM) PLUG-IN APPLET TAGS -->

<!-- The following tags use a special syntax which allows both Netscape and Internet Explorer to load   -->
<!-- the Java Plug-in and run the applet in the Plug-in's JRE.  Do not modify this syntax.              -->
<!-- For more information see http://java.sun.com/products/jfc/tsc/swingdoc-current/java_plug_in.html.  -->

<OBJECT classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
        width="400"
        height="200"
        align="left"
        codebase="http://java.sun.com/products/plugin/1.1.3/jinstall-113-win32.cab#Version=1,1,3,0">
    <PARAM name="code"     value="SampleApplet">
    <PARAM name="codebase" value="http://w3.rchland.ibm.com/~dpetty/applets/">
    <PARAM name="archive"  value="MyGUI.jar,jui400.jar,util400.jar,x4j400.jar">
    <PARAM name="type"     value="application/x-java-applet;version=1.1">

    <COMMENT>
    <EMBED type="application/x-java-applet;version=1.1"
           width="400"
           height=200"
           align="left"
           code="SampleApplet"
           codebase="http://w3.rchland.ibm.com/~dpetty/applets/"
           archive="MyGUI.jar,jui400.jar,util400.jar,x4j400.jar"
           pluginspage="http://java.sun.com/products/plugin/1.1.3/plugin-install.html">
        <NOEMBED>
    </COMMENT>
        No support for JDK 1.1 applets found!
        </NOEMBED>
    </EMBED>
</OBJECT>

<!-- END JAVA(TM) PLUG-IN APPLET TAGS -->

<p>
</body>
</html>
```

It is important that the version information be set for 1.1.3.

Note: In this example, we have chosen to store the XML parser JAR file, **x4j400.jar**, on the web server. This is required only when you include your PDML file as part of your applet's installation. For performance reasons, you would normally *serialize* your panel definitions so that the Graphical Toolbox does not have to interpret the PDML at runtime. This greatly improves the performance of your user interface by creating compact binary representations of your panels. For more information see the description of files generated by the tools.

Installing and running the applet

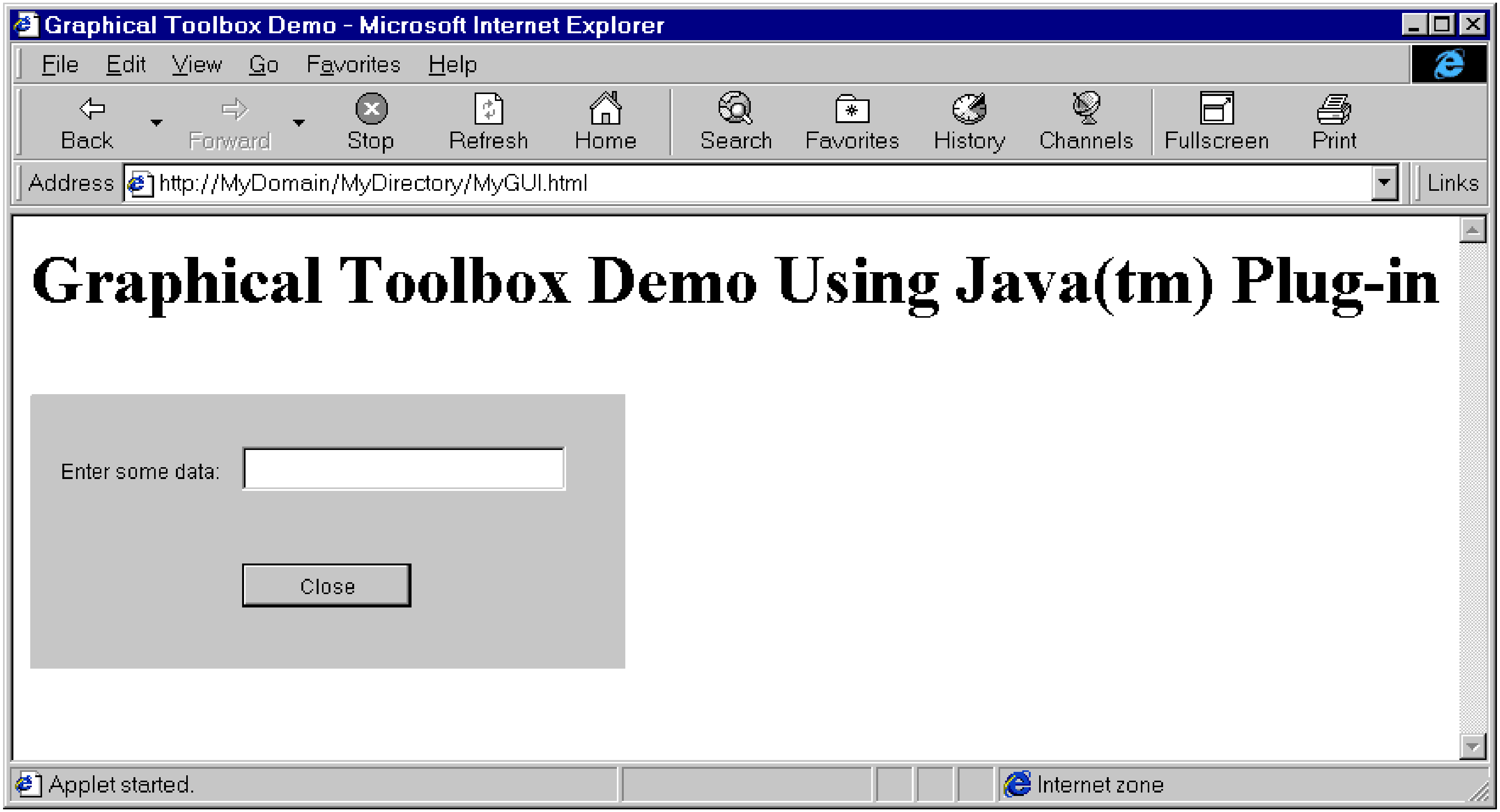
Install the applet on your favorite web server by performing the following steps:

- Compile **SampleApplet.java**.
- Create a JAR file named **MyGUI.jar** to contain the applet binaries. These include the class files produced when you compiled **SampleApplet.java** and **SampleBean.java**, the PDML file **MyGUI.pdml**, and the resource bundle **MyGUI.properties**.
- Copy your new JAR file to a directory of your choice on your web server. Copy the HTML files containing your online help into the server directory.
- Copy the Graphical Toolbox JAR files into the server directory.
- Finally, copy the HTML file **MyGUI.html** containing the imbedded applet into the server directory.

Tip: When testing your applets, ensure that you have removed the Graphical Toolbox jars from the CLASSPATH environment variable on your workstation. Otherwise, you will see error messages saying that the resources for your applet cannot be located on the server.

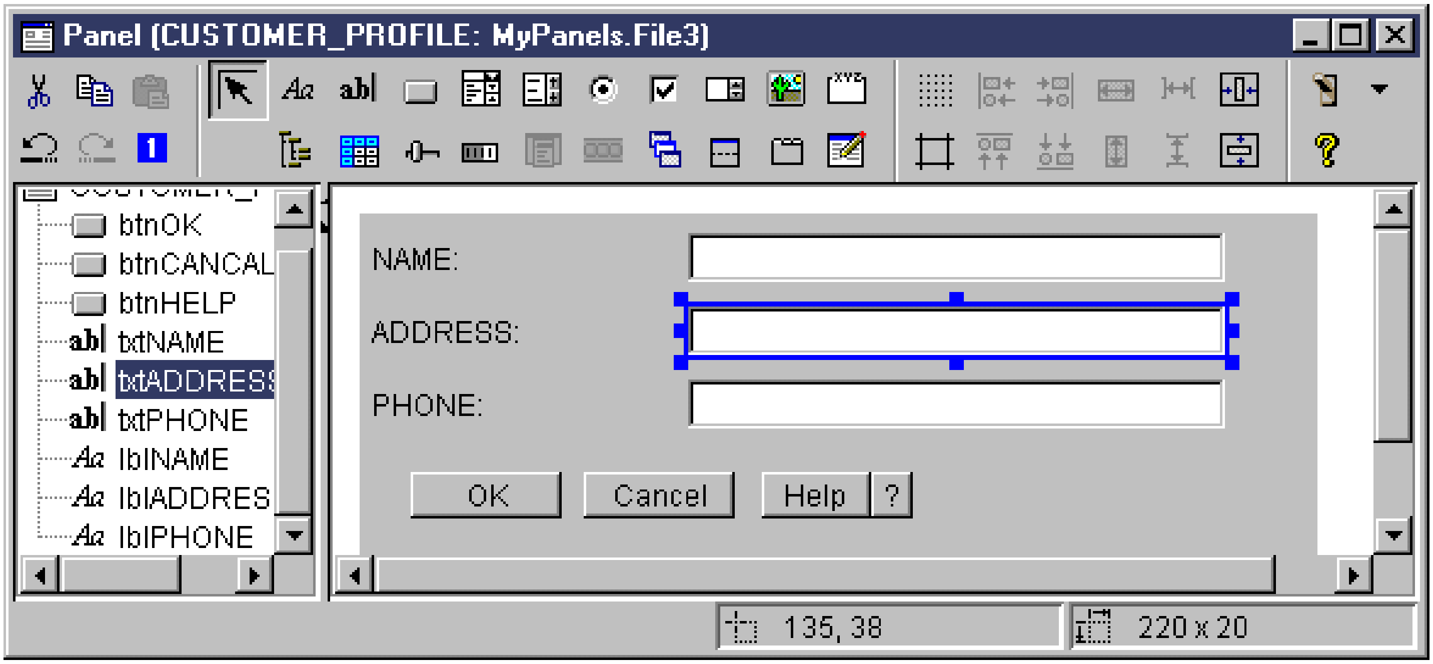
Now you are ready to run the applet. Point your web browser to **MyGUI.html** on the server. If you do not already have the Java Plug-in installed, you will be asked if you want to install it. Once the Plug-in is installed and the applet is started, your browser display should look similar to the following:

Running the sample applet in a browser


Graphical Toolbox Demo - Microsoft Internet Explorer
File
Edit
View
Go
Favorites
Help
Back
Forward
Stop
Refresh
Home
Search
Favorites
History
Channels
Fullscreen
Print
Address
http://MyDomain/MyDirectory/MyGUI.html
Links
Graphical Toolbox Demo Using Java(tm) Plug-in
Enter some data:
Close
Applet started.
Internet zone

Explanation of the Toolbox Widgets

Below is the Java GUI Editor's toolbox and an explanation of what each tool icon does when selected:



Click Pointer to move and resize a component on a panel.

Click Label to insert a static label on a panel.

Click Text to insert a text box on a panel.

Click Button to insert a button on a panel.

Click Combo Box to insert a drop down list box on a panel.

Click List Box to insert a list box on a panel.

Click Radio Button to insert a radio button on a panel.

Click Checkbox to insert a check box on a panel.

Click Spinner to insert a spinner on a panel.

Click Image to insert an image on a panel.

Click Menu Bar to insert a menu bar on a panel.

Click Group Box to insert a labeled group box on a panel.

Click Tree to insert an hierarchical tree on a panel.

Click Table to insert a table on a panel.

Click Slider to insert an adjustable slider on a panel.

Click Progress Bar to insert a progress bar on a panel.

Click Deck Pane to insert a deck pane on a panel. A deck pane contains a stack of panels. The user can select any of the panels, but only the selected panel is fully visible.

Click Split Pane to insert a split pane on a panel. A split pane is one pane divided into two horizontal or vertical panes.

Click Tabbed Pane to insert a tabbed pane on a panel. A tabbed pane contains a collection of panels with tabs at the top. The user clicks a tab to display the contents of a panel. The title of the panel is used as the text for a tab.

Click Custom to insert a custom-defined user interface component on a panel.

Click Toolbar to insert a toolbar on a panel.

Click Toggle Grid to enable a grid on a panel.

Click Align Top to align multiple components on a panel with the top edge of a specific, or primary, component.

Click Align Bottom to align multiple components on a panel with the bottom edge of a specific, or primary, component.

Click Equalize Height to equalize the height of multiple components with the height of a specific, or primary, component.

Click Center Vertically to center a selected component vertically relative to the panel.

Click Toggle Margins to view the margins of the panel.

Click Align Left to align multiple components on a panel with the left edge of a specific, or primary, component.

Click Align Right to align multiple components on a panel with the left edge of a specific, or primary, component.

Click Equalize Width to equalize the width of multiple components with the width of a specific, or primary, component.

Click Center Horizontally to center a selected component horizontally relative to the panel.

Click Cut to cut panel components.

Click Copy button to copy panel components.

Click Paste to paste panel components between different panels or files.

Click Undo to undo the last action.

Click Redo to redo the last action.

Click Tab Order to control the selection order of each panel component when the user presses TAB to navigate through the panel.

Click Preview to display a preview of what a panel will look like.

Click Help to get more specific information on the Graphical Toolbox.

Graphical user interface classes

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

Java programs that use the IBM Toolbox for Java GUI (graphical user interface) classes need Swing 1.1. You get Swing 1.1 either by running Java 2 or by downloading Swing 1.1 from Sun Microsystems, Inc.. In the past, IBM Toolbox for Java has required Swing 1.0.3, and V4R5 is the first release that Swing 1.1 is supported. To move to Swing 1.1, some programming changes were made; therefore, you may have to make some programming changes as well.See the Sun Microsystems, Inc. JFC page for more information about Swing.

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

Use the AS400 panes classes to display iSeries data.

APIs are available to access the following iSeries resources and their tools:

- Command call
- Data queues
- Error events*
- Integrated file system
- JavaApplicationCall
- JDBC
- Jobs*
- Messages*
- Permission
- Print* including the spooled file viewer
- Program call
- Record-level access
- »Resource lists«
- System status
- System values
- Users and Groups

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

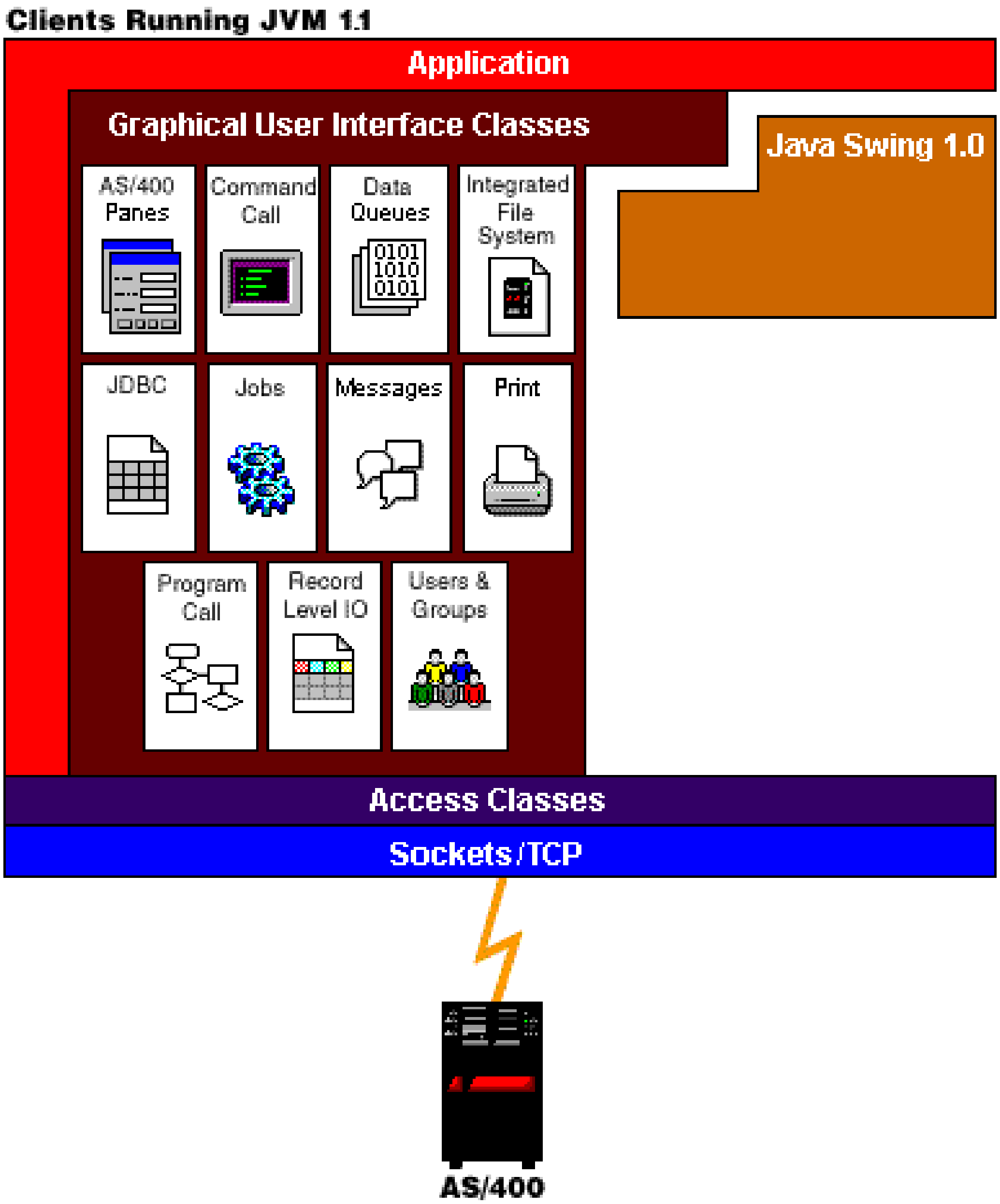
When programming with the IBM Toolbox for Java graphical user interface components, use the Error events classes to report and handle error events to the user.

See Access classes for more information about accessing iSeries data.

Graphical user interface classes

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

Graphical user interface classes



AS/400 Panes

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

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

The following AS/400 panes are available:

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

AS/400 resources

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

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

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

All resources are implementations of the VNode interface.

Setting the root

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

- AS400ListPane presents all of the root's children in its list.
- AS400DetailsPane presents all of the root's children in its table.
- AS400TreePane uses the root as the root of its tree.
- AS400ExplorerPane uses the root as the root of its tree.

Any combination of panes and roots is possible.

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

```
                  // Create the AS/400 resource
                  // representing a list of users.
                  // Assume that "system" is an AS400
                  // object created and initialized
                  // elsewhere.
VUserList userList = new VUserList (system);

                  // Create the AS400DetailsPane object
                  // and set its root to be the user
                  // list.
AS400DetailsPane detailsPane = new AS400DetailsPane ();
detailsPane.setRoot (userList);

                  // Add the details pane to a frame.
                  // Assume that "frame" is a JFrame
```

```
                     // created elsewhere.
frame.getContentPane ().add (detailsPane);
```

Loading the contents

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

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

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

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

```
                     // Load the contents of the details
                     // pane. Assume that the detailsPane
                     // was created and initialized
                     // elsewhere.
detailsPane.load ();

                     // Add the details pane to a frame.
                     // Assume that "frame" is a JFrame
                     // created elsewhere.
frame.getContentPane ().add (detailsPane);
```

Actions and properties panes

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

In some cases, the pop-up menu also presents an item that allows the user to view a properties pane. A properties pane shows various details about the resource and may allow the user to change those details.

The application can control whether actions and properties panes are available by using the setAllowActions() method on the pane.

Models

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

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

The following models are available:

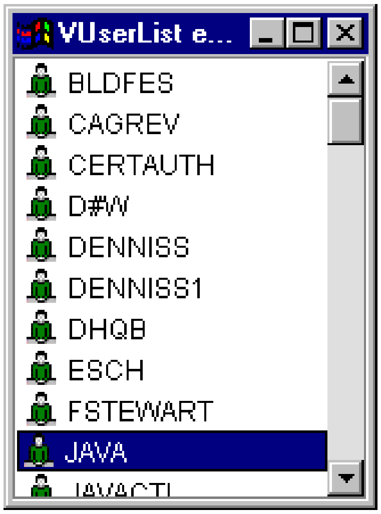
- AS400ListModel implements the JFC ListModel interface as a list of AS/400 resources. This can be used with a JFC JList object.
- AS400DetailsModel implements the JFC TableModel interface as a table of AS/400 resources where each row contains various details about a single resource. This can be used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

Examples

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

  The following image shows the finished product:

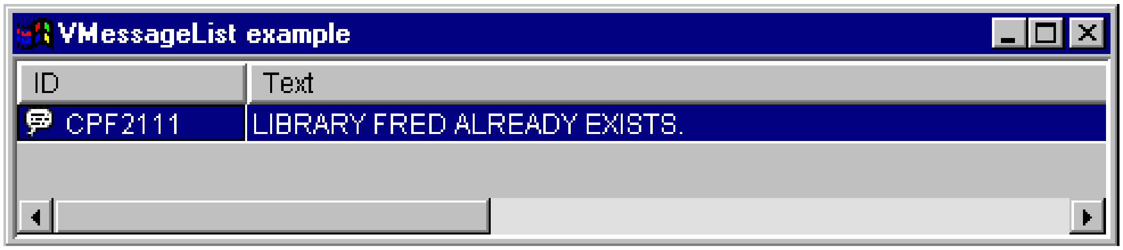
  Using AS400ListPane with a VUserList object



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

  The following image shows the finished product:

  Using AS400DetailsPane with a VMessageList object

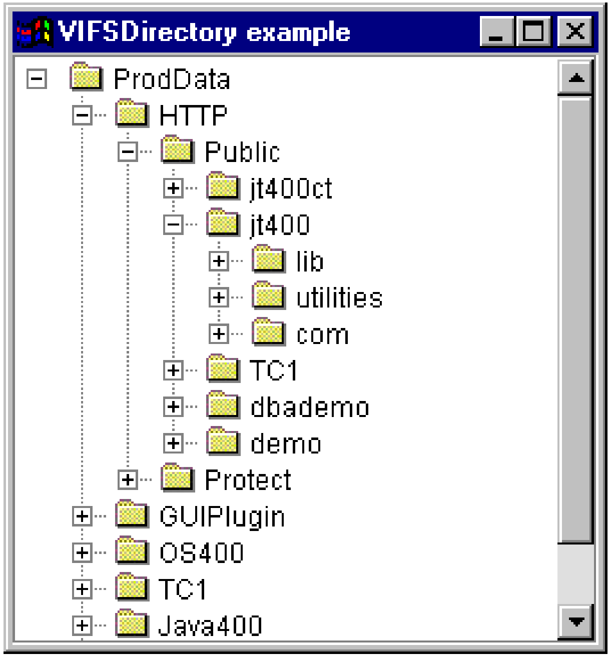


ID	Text
CPF2111	LIBRARY FRED ALREADY EXISTS.

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

  The following image shows the finished product:

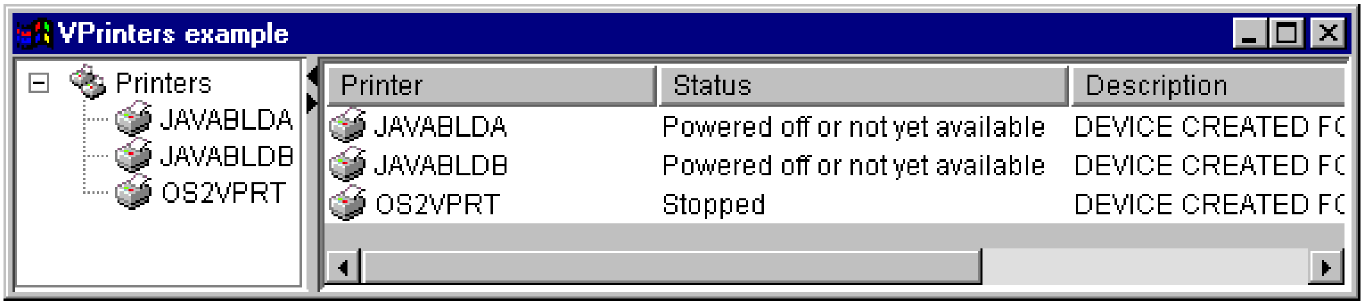
  Using AS400TreePane with a VIFSDirectory object



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

  The following image shows the finished product:

  Using AS400ExplorerPane with a VPrinters object



Printer	Status	Description
JAVABLDA	Powered off or not yet available	DEVICE CREATED FC
JAVABLDB	Powered off or not yet available	DEVICE CREATED FC
OS2VPRT	Stopped	DEVICE CREATED FC

Command Call

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

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

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

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

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

```
                     // Create the CommandCallButton
                     // object. Assume that "system" is
                     // an AS400 object created and
                     // initialized elsewhere.  The button
                     // text says "Press Me", and there is
                     // no icon.
CommandCallButton button = new CommandCallButton ("Press Me", null, system);

                     // Set the command that the button will run.
button.setCommand ("CRTLIB FRED");

                     // Add the button to a frame. Assume
                     // that "frame" is a JFrame created
                     // elsewhere.
frame.getContentPane ().add (button);
```

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

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

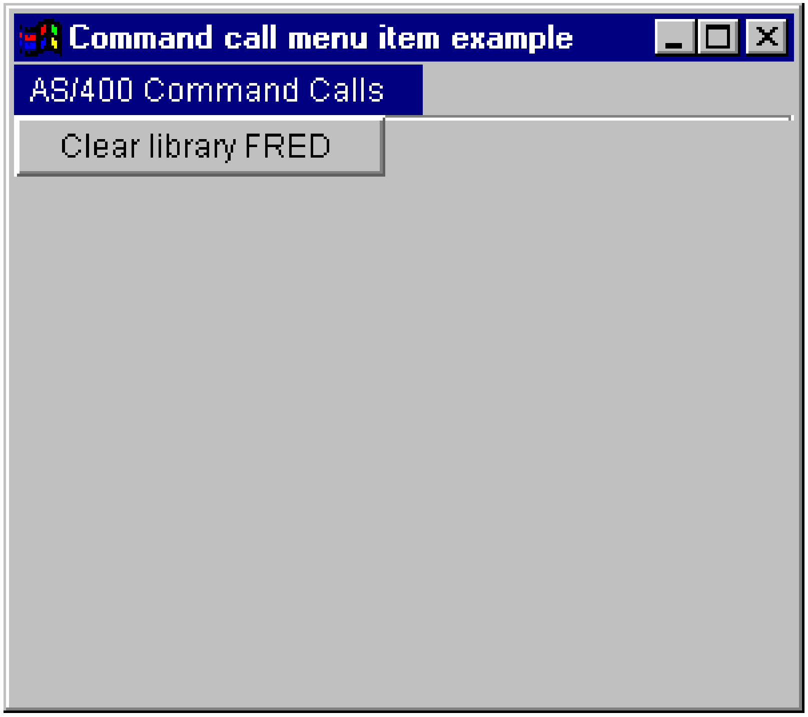
```
                     // Add an ActionCompletedListener that
                     // is implemented using an anonymous
                     // inner class. This is a convenient
                     // way to specify simple event
                     // listeners.
   button.addActionCompletedListener (new ActionCompletedListener ()
   {
        public void actionCompleted (ActionCompletedEvent event)
        {
                     // Cast the source of the event to a
                     // CommandCallButton.
             CommandCallButton sourceButton = (CommandCallButton) event.getSource ();

                     // Get the list of AS/400 messages
                     // that the command generated.
             AS400Message[] messageList = sourceButton.getMessageList ();

                     // ... Process the message list.
        }
   });
```

Examples

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

The image below shows the CommandCall graphical user interface component:

CommandCall GUI component



Data queues

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

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

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

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

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

```
                       // Create the DataQueueDocument
                       // object. Assume that "system" is
                       // an AS400 object created and
                       // initialized elsewhere.
     DataQueueDocument dqDocument = new DataQueueDocument (system, "/QSYS.LIB/MYLIB.LIB/MYQUEUE.DTAQ");

                       // Create a text area to present the
                       // document.
     JTextArea textArea = new JTextArea (dqDocument);

                       // Add the text area to a frame.
                       // Assume that "frame" is a JFrame
                       // created elsewhere.
     frame.getContentPane ().add (textArea);
```

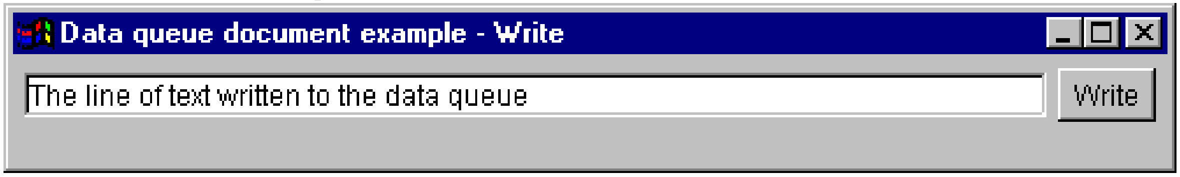
Initially, the contents of the text component are empty. Use read() or peek() to fill the contents with the next entry on the queue. Use write() to write the contents of the text component to the data queue. Note that these documents only work with String data queue entries.

Examples

Example of using a DataQueueDocument in an application.

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

DataQueueDocument GUI component



The line of text written to the data queue

Error events

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

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

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

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

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

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

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

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

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

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

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

Integrated file system

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

The following components are available:

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

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

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

File dialogs

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

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

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

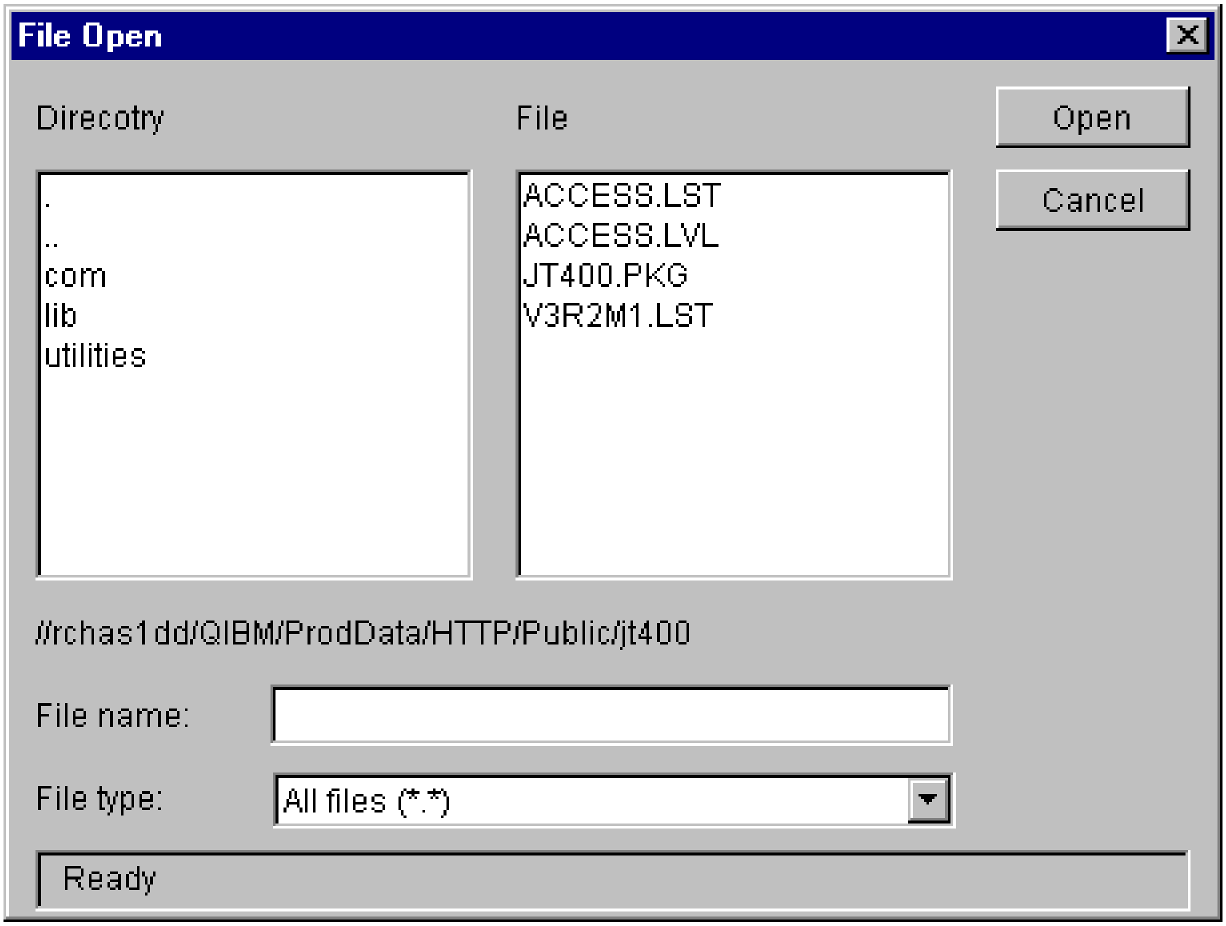
```
                 // Create a IFSFileDialog object
                 // setting the text of the title bar.
                 // Assume that "system" is an AS400
                 // object and "frame" is a JFrame
                 // created and initialized elsewhere.
IFSFileDialog dialog = new IFSFileDialog (frame, "Select a file", system);

                 // Set a list of filters for the dialog.
                 // The first filter will be used
                 // when the dialog is first displayed.
FileFilter[] filterList = {new FileFilter ("All files (*.*)", "*.*"),
                           new FileFilter ("HTML files (*.HTML", "*.HTM")};
         // Then, set the filters in the dialog.
dialog.setFileFilter (filterList, 0);

                 // Set the text on the buttons.
dialog.setOkButtonText ("Open");
dialog.setCancelButtonText ("Cancel");

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

Example

Present an IFSFileDialog and print the selection, if any.

The following image shows the IFSFileDialog graphical user interface component:

IFSFileDialog GUI component



```
.
..
com
lib
utilities
```

```
ACCESS.LST
ACCESS.LVL
JT400.PKG
V3R2M1.LST
```

//rchas1dd/QIBM/ProdData/HTTP/Public/jt400

File type: All files (*.*)

Ready

Directories in AS/400 panes

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

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

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

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

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

The following actions are available for directories:

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

The following actions are available for files:

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

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

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

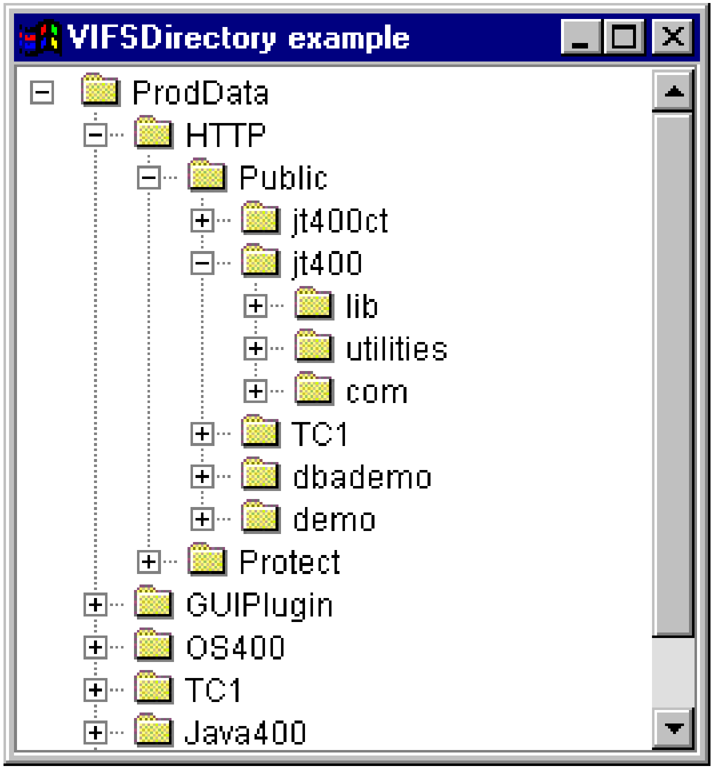
```
                  // Create the VIFSDirectory object.
                  // Assume that "system" in an AS400
                  // object created and initialized
                  // elsewhere.
VIFSDirectory root = new VIFSDirectory (system, "/DirectoryA/DirectoryB");

                  // Create and load an AS400ExplorerPane object.
AS400ExplorerPane explorerPane = new AS400ExplorerPane (root);
explorerPane.load ();

                  // Add the explorer pane to a frame.
                  // Assume that "frame" is a JFrame
                  // created elsewhere.
frame.getContentPane ().add (explorerPane);
```

Example

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

The following image shows the VIFSDirectory graphical user interface component:

VIFSDirectory GUI component



IFSTextFileDocument

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

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

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

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

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

The following example creates and loads an IFSTextFileDocument:

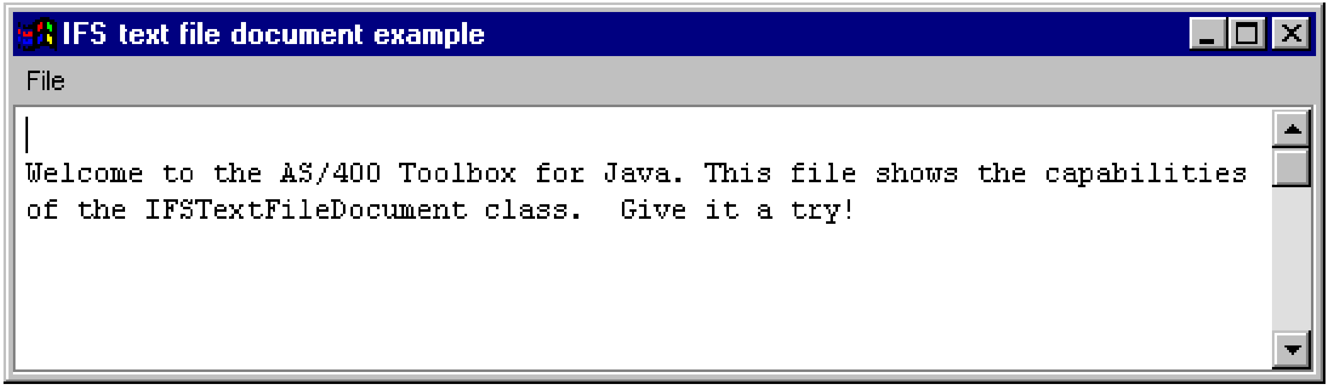
```
                     // Create and load the
                     // IFSTextFileDocument object. Assume
                     // that "system" is an AS400 object
                     // created and initialized elsewhere.
IFSTextFileDocument ifsDocument = new IFSTextFileDocument (system, "/DirectoryA/MyFile.txt");
ifsDocument.load ();

                     // Create a text area to present the
                     // document.
JTextArea textArea = new JTextArea (ifsDocument);

                     // Add the text area to a frame.
                     // Assume that "frame" is a JFrame
                     // created elsewhere.
frame.getContentPane ().add (textArea);
```

Example

Present an IFSTextFileDocument in a JTextPane.

The following image shows the IFSTextFileDocument graphical user interface component:

IFS text file document example



Welcome to the AS/400 Toolbox for Java. This file shows the capabilities of the IFSTextFileDocument class. Give it a try!

VJavaApplicationCall

The VJavaApplicationCall class allows you to run a Java application on the AS/400 from a client by using a graphical user interface (GUI).

The GUI is a panel with two sections. The top section is an output window that displays output that the Java program writes to standard output and standard error. The bottom section is an input field where the user enters the Java environment, the Java program to run with parameters and input the Java program receives via standard input. Refer to the Java command options for more information.

For example, this code would create the following GUI for your Java program.

VJavaApplicationCall is a class that you call from your Java program. However, the IBM Toolbox for Java also provides a utility that is a complete Java application that can be used to call your Java program from a workstation. Refer to the RunJavaApplication class for more information.

JDBC

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

The following components are available:

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

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

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

SQL connections

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

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

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

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

The following example creates and initializes an SQLConnection object:

```
                      // Create an SQLConnection object.
    SQLConnection connection = new SQLConnection ();

                      // Set the URL and user name properties of the connection.
    connection.setURL ("jdbc:as400://MySystem");
    connection.setUserName ("Lisa");
```

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

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

Buttons and menu items

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

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

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

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

```
                       // Create an SQLStatementButton object.
                       // The button text says "Delete All",
                       // and there is no icon.
     SQLStatementButton button = new SQLStatementButton ("Delete All");

                       // Set the connection and SQLStatement
                       // properties.  Assume that "connection"
                       // is an SQLConnection object that is
                       // created and initialized elsewhere.
     button.setConnection (connection);
     button.setSQLStatement ("DELETE FROM MYTABLE");

                       // Add the button to a frame. Assume
                       // that "frame" is a JFrame created
                       // elsewhere.
     frame.getContentPane ().add (button);
```

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

Documents

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

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

The following example creates an SQLStatementDocument in a JTextField:

```
                       // Create an SQLStatementDocument
                       // object. Assume that "connection"
                       // is an SQLConnection object that is
                       // created and initialized elsewhere.
                       // The text of the document is
                       // initialized to a generic query.
     SQLStatementDocument document = new SQLStatementDocument (connection, "SELECT * FROM QIWS.QCUSTCDT");

                       // Create a text field to present the
                       // document.
     JTextField textField = new JTextField ();
     textField.setDocument (document);

                       // Add the text field to a frame.
                       // Assume that "frame" is a JFrame
                       // created elsewhere.
     frame.getContentPane ().add (textField);

                       // Run the SQL statement that is in
                       // the text field.
     document.execute ();
```

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

Result set form panes

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

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

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

```
                       // Create an SQLResultSetFormPane
                       // object. Assume that "connection"
                       // is an SQLConnection object that is
                       // created and initialized elsewhere.
     SQLResultSetFormPane formPane = new SQLResultSetFormPane (connection, "SELECT * FROM QIWS.QCUSTCDT");

                       // Load the results.
     formPane.load ();

                       // Add the form pane to a frame.
                       // Assume that "frame" is a JFrame
                       // created elsewhere.
     frame.getContentPane ().add (formPane);
```

Result set table panes

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

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

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

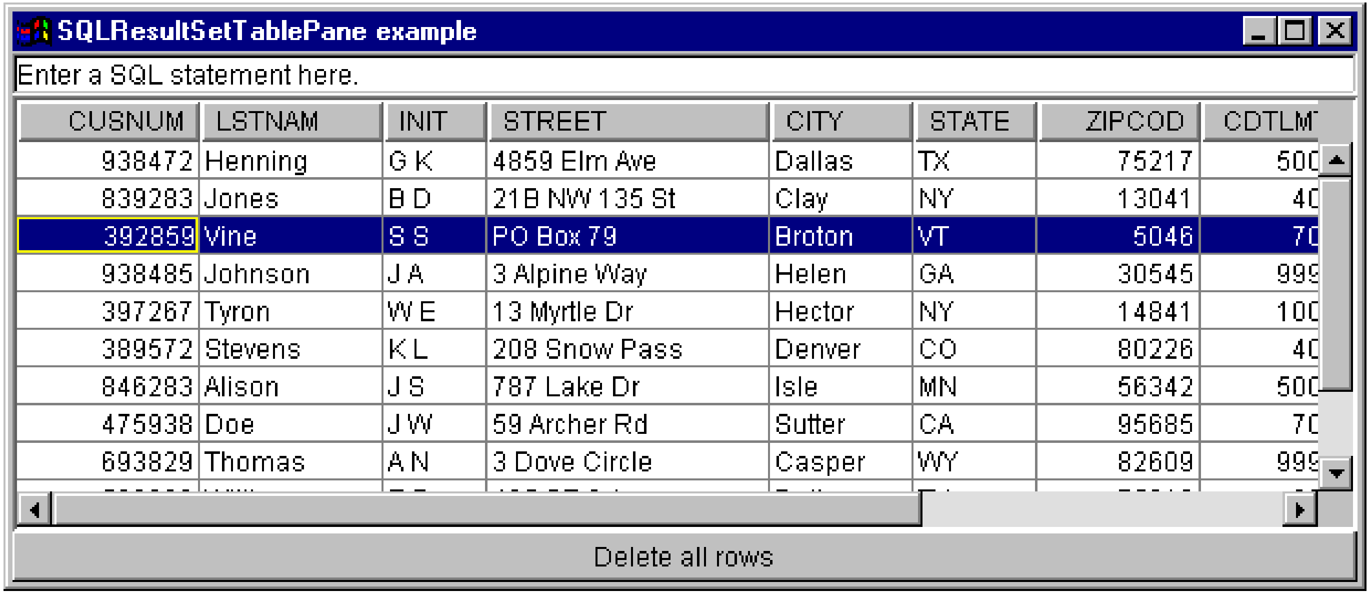
```
                  // Create an SQLResultSetTablePane
                  // object. Assume that "connection"
                  // is an SQLConnection object that is
                  // created and initialized elsewhere.
SQLResultSetTablePane tablePane = new SQLResultSetTablePane (connection, "SELECT * FROM QIWS.QCUSTCDT");

                  // Load the results.
tablePane.load ();

                  // Add the table pane to a frame.
                  // Assume that "frame" is a JFrame
                  // created elsewhere.
frame.getContentPane ().add (tablePane);
```

Example

Present an SQLResultSetTablePane that displays the contents of a table. This example uses an SQLStatementDocument (denoted in the following image by the text, "Enter a SQL statement here") that allows the user to type in any SQL statement, and an SQLStatementButton (denoted by the text, "Delete all rows") that allows the user to delete all rows from the table.

The following image shows the SQLResultSetTablePane graphical user interface component.

SQLResultSetTablePane GUI component



CUSNUM	LSTNAM	INIT	STREET	CITY	STATE	ZIPCOD	CDTLM
938472	Henning	G K	4859 Elm Ave	Dallas	TX	75217	500
839283	Jones	B D	21B NW 135 St	Clay	NY	13041	40
392859	Vine	S S	PO Box 79	Broton	VT	5046	70
938485	Johnson	J A	3 Alpine Way	Helen	GA	30545	999
397267	Tyron	W E	13 Myrtle Dr	Hector	NY	14841	100
389572	Stevens	K L	208 Snow Pass	Denver	CO	80226	40
846283	Alison	J S	787 Lake Dr	Isle	MN	56342	500
475938	Doe	J W	59 Archer Rd	Sutter	CA	95685	70
693829	Thomas	A N	3 Dove Circle	Casper	WY	82609	999

Result set table models

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

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

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

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

```
                  // Create an SQLResultSetTableModel
                  // object. Assume that "connection"
                  // is an SQLConnection object that is
                  // created and initialized elsewhere.
SQLResultSetTableModel tableModel = new SQLResultSetTableModel (connection, "SELECT * FROM QIWS.QCUSTCDT");

                  // Load the results.
tableModel.load ();

                  // Create a JTable for the model.
JTable table = new JTable (tableModel);

                  // Add the table to a frame.  Assume
                  // that "frame" is a JFrame created
                  // elsewhere.
frame.getContentPane ().add (table);
```

SQL query builders

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

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

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

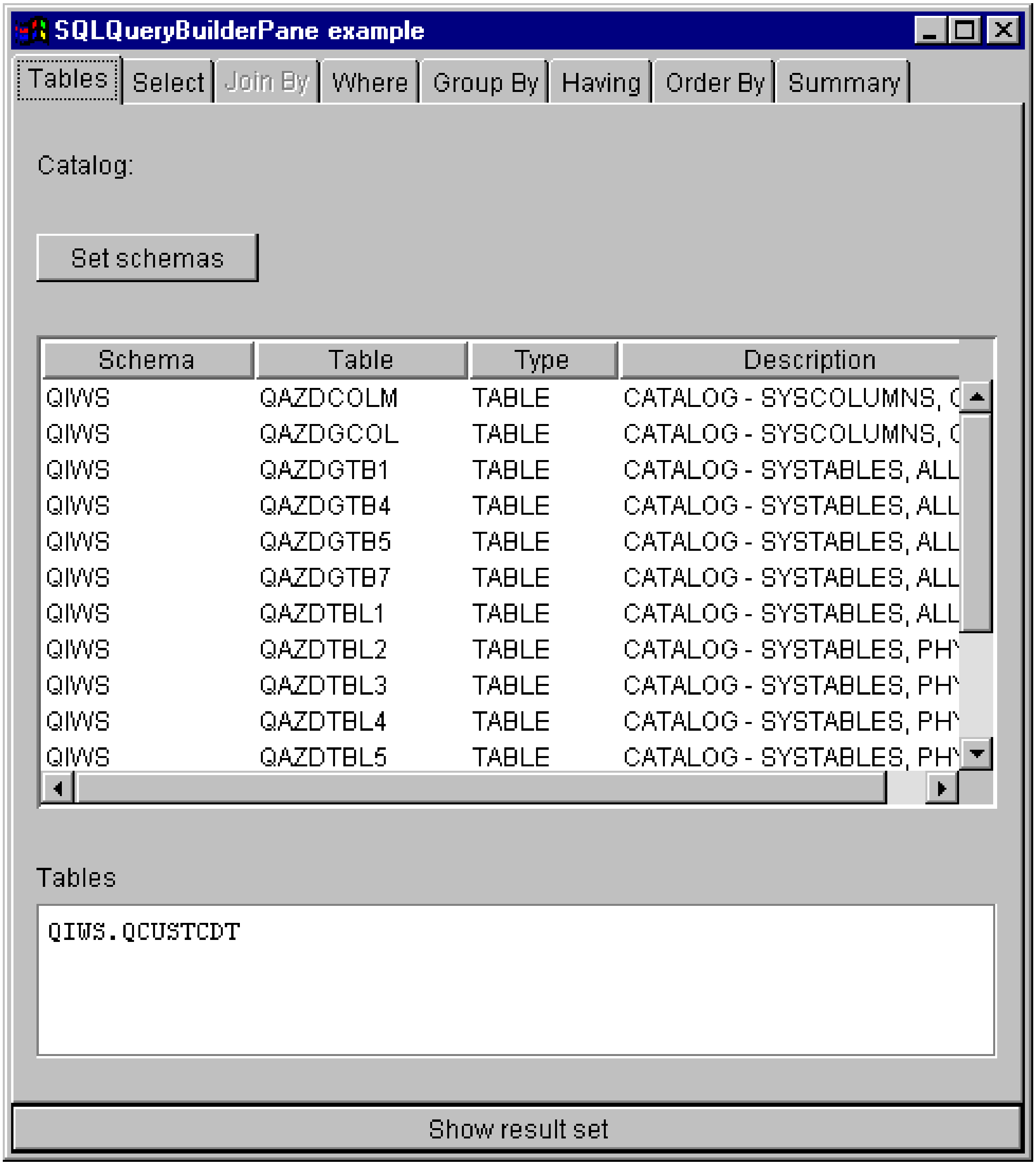
```
                  // Create an SQLQueryBuilderPane
                  // object. Assume that "connection"
                  // is an SQLConnection object that is
                  // created and initialized elsewhere.
SQLQueryBuilderPane queryBuilder = new SQLQueryBuilderPane (connection);

                  // Load the data needed for the query
                  // builder.
queryBuilder.load ();

                  // Add the query builder pane to a
                  // frame. Assume that "frame" is a
                  // JFrame created elsewhere.
frame.getContentPane ().add (queryBuilder);
```

Example

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

The following image shows the SQLQueryBuilderPane graphical user interface component:

SQLQueryBuilderPane GUI component



Schema	Table	Type	Description
QIWS	QAZDCOLM	TABLE	CATALOG - SYSCOLUMNS, C
QIWS	QAZDGCOL	TABLE	CATALOG - SYSCOLUMNS, C
QIWS	QAZDGTB1	TABLE	CATALOG - SYSTABLES, ALL
QIWS	QAZDGTB4	TABLE	CATALOG - SYSTABLES, ALL
QIWS	QAZDGTB5	TABLE	CATALOG - SYSTABLES, ALL
QIWS	QAZDGTB7	TABLE	CATALOG - SYSTABLES, ALL
QIWS	QAZDTBL1	TABLE	CATALOG - SYSTABLES, ALL
QIWS	QAZDTBL2	TABLE	CATALOG - SYSTABLES, PHY
QIWS	QAZDTBL3	TABLE	CATALOG - SYSTABLES, PHY
QIWS	QAZDTBL4	TABLE	CATALOG - SYSTABLES, PHY
QIWS	QAZDTBL5	TABLE	CATALOG - SYSTABLES, PHY

Tables

QIWS.QCUSTCDT

Jobs

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

The following components are available:

- A VJobList object is a resource that represents a list of iSeries jobs for use in AS/400 panes.
- A VJob object is a resource that represents the list of messages in a job log for use in AS400Panes.

You can use AS400Panes, VJobList objects, and VJob objects together to present many views of a job list or job log.

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

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

Either the VJobList or VJob object is then "plugged" into the AS400Pane as the root, using the pane's constructor or setRoot() method.

VJobList has some other useful properties for defining the set of jobs that are presented in AS400Panes. Use setName() to specify that only jobs with a certain name should appear. Use setNumber() to specify that only jobs with a certain number should appear. Similarly, use setUser() to specify that only jobs for a certain user should appear.

When AS400Pane, VJobList, and VJob objects are created, they are initialized to a default state. The list of jobs or job log messages are not loaded at creation time. To load the contents, the caller must explicitly call the load() method on either object. This will initiate communication to the AS/400 system to gather the contents of the list.

At run-time, right-click a job, job list, or job log message to display the shortcut menu. Select **Properties** from the shortcut menu to perform actions on the selected object:

- »Job - Work with properties, such as the type and status. You can also change the value of some of the properties.«
- Job list - Work with the properties, such as name, number, and user properties. You can also change the contents of the list.
- Job log message - Display properties, such as the full text, severity, and time sent.

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

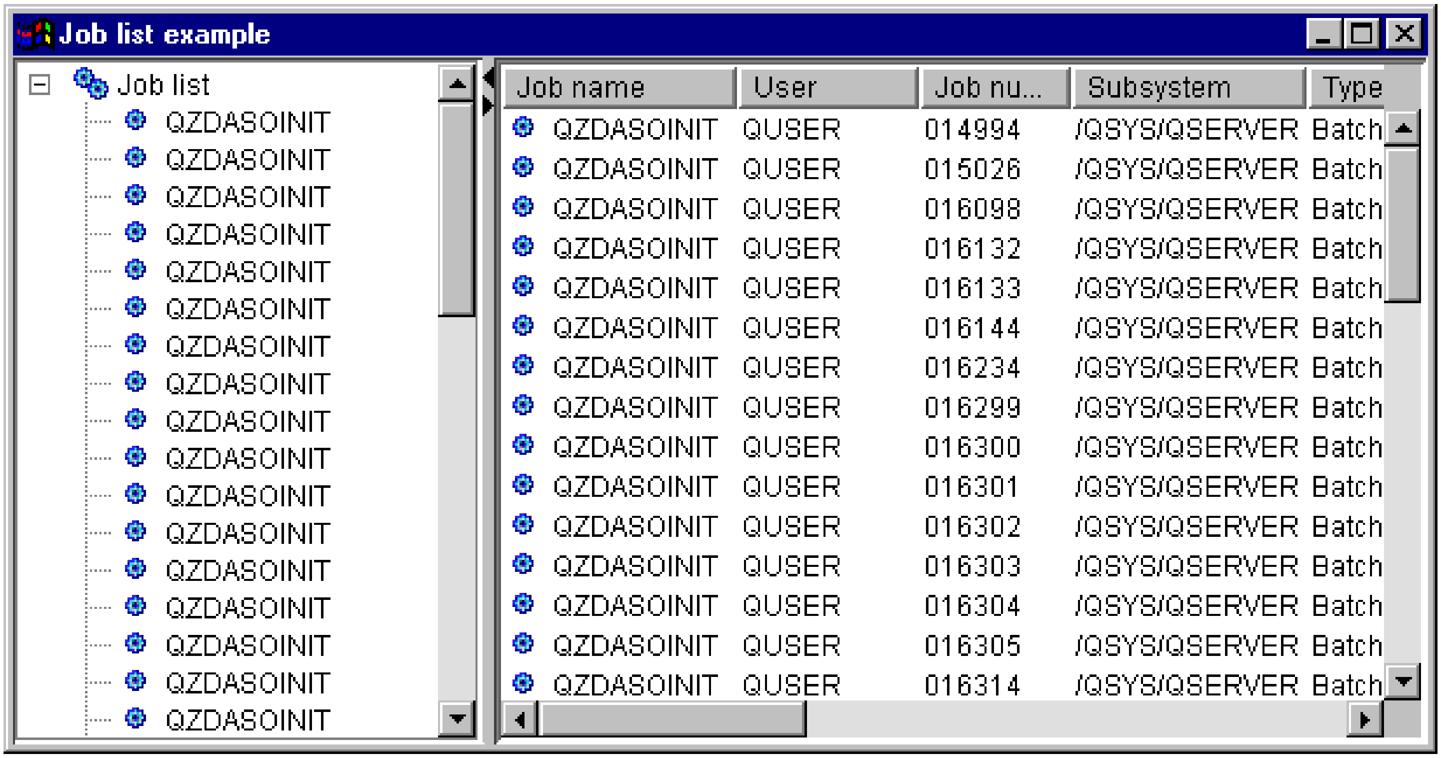
The following example creates a VJobList and presents it in an AS400ExplorerPane:

```
                  // Create the VJobList object. Assume
                  // that "system" is an AS400 object
                  // created and initialized elsewhere.
VJobList root = new VJobList (system);

                  // Create and load an
                  // AS400ExplorerPane object.
AS400ExplorerPane explorerPane = new AS400ExplorerPane (root);
explorerPane.load ();

                  // Add the explorer pane to a frame.
                  // Assume that "frame" is a JFrame
                  // created elsewhere.
frame.getContentPane ().add (explorerPane);
```

Examples

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

The following image shows the VJobList graphical user interface component:



- Job list
  - QZDASOINIT
  - QZDASOINIT
  - QZDASOINIT
  - QZDASOINIT
  - QZDASOINIT
  - QZDASOINIT
  - QZDASOINIT
  - QZDASOINIT
  - QZDASOINIT
  - QZDASOINIT
  - QZDASOINIT
  - QZDASOINIT
  - QZDASOINIT
  - QZDASOINIT
  - QZDASOINIT
  - QZDASOINIT
  - QZDASOINIT

Job name	User	Job nu...	Subsystem	Type
QZDASOINIT	QUSER	014994	/QSYS/QSERVER	Batch
QZDASOINIT	QUSER	015026	/QSYS/QSERVER	Batch
QZDASOINIT	QUSER	016098	/QSYS/QSERVER	Batch
QZDASOINIT	QUSER	016132	/QSYS/QSERVER	Batch
QZDASOINIT	QUSER	016133	/QSYS/QSERVER	Batch
QZDASOINIT	QUSER	016144	/QSYS/QSERVER	Batch
QZDASOINIT	QUSER	016234	/QSYS/QSERVER	Batch
QZDASOINIT	QUSER	016299	/QSYS/QSERVER	Batch
QZDASOINIT	QUSER	016300	/QSYS/QSERVER	Batch
QZDASOINIT	QUSER	016301	/QSYS/QSERVER	Batch
QZDASOINIT	QUSER	016302	/QSYS/QSERVER	Batch
QZDASOINIT	QUSER	016303	/QSYS/QSERVER	Batch
QZDASOINIT	QUSER	016304	/QSYS/QSERVER	Batch
QZDASOINIT	QUSER	016305	/QSYS/QSERVER	Batch
QZDASOINIT	QUSER	016314	/QSYS/QSERVER	Batch

Messages

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

The following components are available:

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

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

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

Message lists

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

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

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

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

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

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

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

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

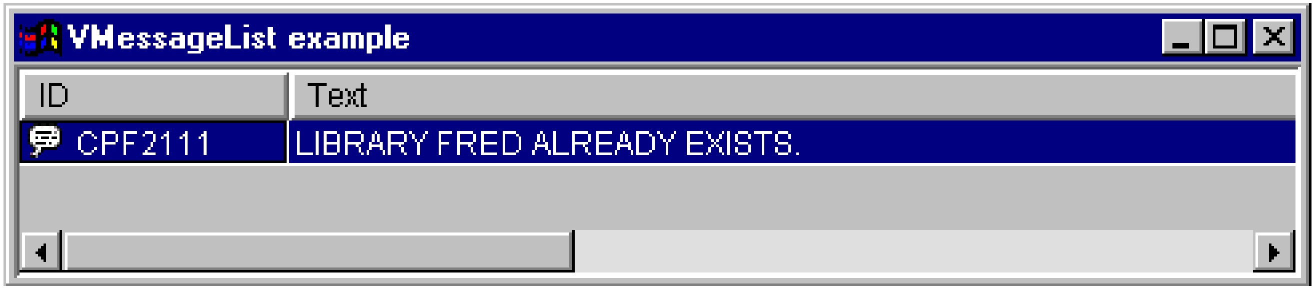
```
                              // Create the VMessageList object.
                              // Assume that "command" is a
                              // CommandCall object created and run
                              // elsewhere.
     VMessageList root = new VMessageList (command.getMessageList ());

                              // Create and load an AS400DetailsPane
                              // object.
     AS400DetailsPane detailsPane = new AS400DetailsPane (root);
     detailsPane.load ();

                              // Add the details pane to a frame.
                              // Assume that "frame" is a JFrame
                              // created elsewhere.
     frame.getContentPane ().add (detailsPane);
```

Example

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

The following image shows the VMessageList graphical user interface component:

VMessageList GUI component



VMessageList example

ID	Text
CPF2111	LIBRARY FRED ALREADY EXISTS.

Message queues

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

To use a VMessageQueue, set the system and path properties. These properties can be set using a constructor or through the setSystem() and setPath() methods. The VMessageQueue object is then "plugged" into the AS/400 pane as the root, using the pane's constructor or setRoot() method.

VMessageQueue has some other useful properties for defining the set of messages that are presented in AS/400 panes. Use setSeverity() to specify the severity of messages that should appear. Use setSelection() to specify the type of messages that should appear.

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

At run-time, a user can perform actions on any message queue or message through the pop-up menu. The following actions are available for message queues:

- Clear - clears the message queue.
- Properties - allows the user to set the severity and selection properties. This may be used to change the contents of the list.

The following action is available for messages on a message queue:

- Remove - removes the message from the message queue.
- Reply - replies to an inquiry message.
- Properties - displays properties such as the severity, type, and date.

Of course, users can only access message queues to which they are authorized. In addition, the caller can prevent the user from performing actions by using the setAllowActions() method on the pane.

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

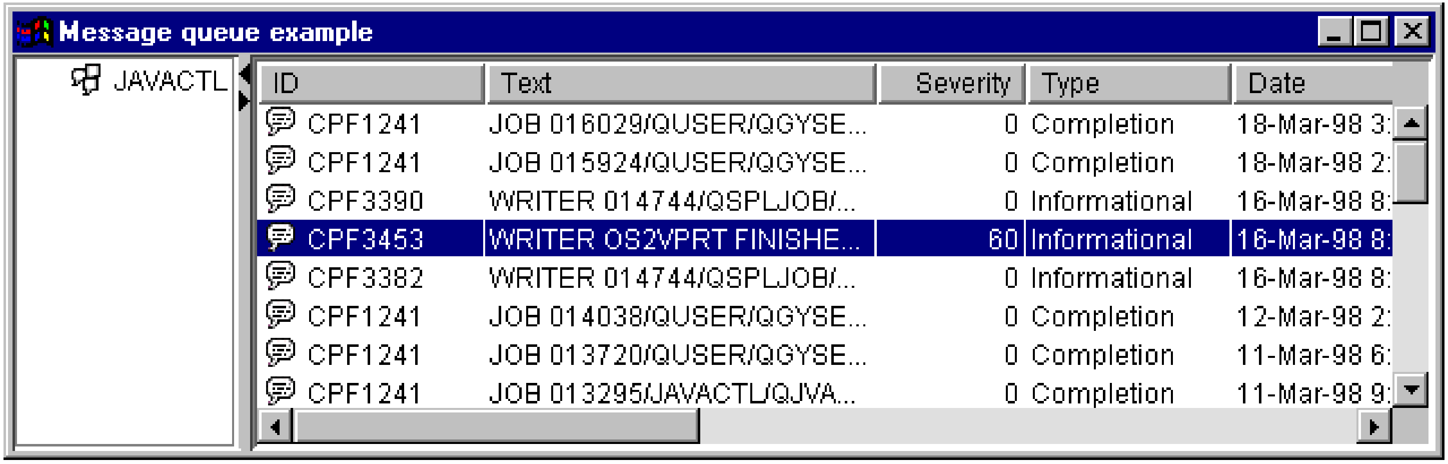
```
                  // Create the VMessageQueue object.
                  // Assume that "system" is an AS400
                  // object created and initialized
                  // elsewhere.
VMessageQueue root = new VMessageQueue (system, "/QSYS.LIB/MYLIB.LIB/MYMSGQ.MSGQ");

                  // Create and load an
                  // AS400ExplorerPane object.
AS400ExplorerPane explorerPane = new AS400ExplorerPane (root);
explorerPane.load ();

                  // Add the explorer pane to a frame.
                  // Assume that "frame" is a JFrame
                  // created elsewhere.
frame.getContentPane ().add (explorerPane);
```

Example

Present the list of messages in a message queue using an AS400ExplorerPane with a VMessageQueue object.

The following image shows the VMessageQueue graphical user interface component:

VMessageQueue GUI component



ID	Text	Severity	Type	Date
CPF1241	JOB 016029/QUSER/QGYSE...	0	Completion	18-Mar-98 3:
CPF1241	JOB 015924/QUSER/QGYSE...	0	Completion	18-Mar-98 2:
CPF3390	WRITER 014744/QSPLJOB/...	0	Informational	16-Mar-98 8:
CPF3453	WRITER OS2VPRT FINISHE...	60	Informational	16-Mar-98 8:
CPF3382	WRITER 014744/QSPLJOB/...	0	Informational	16-Mar-98 8:
CPF1241	JOB 014038/QUSER/QGYSE...	0	Completion	12-Mar-98 2:
CPF1241	JOB 013720/QUSER/QGYSE...	0	Completion	11-Mar-98 6:
CPF1241	JOB 013295/JAVACTL/QJVA...	0	Completion	11-Mar-98 9:

Permission

The Permission classes information can be used in a graphical user interface (GUI) through the VIFSFile and VIFSDirectory classes. Permission has been added as an action in each of these classes.

The following example shows how to use Permission with the VIFSDirectory class:

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

// Create an IFSDirectory using the system name
// and the full path of a QSYS object
VIFSDirectory directory = new VIFSDirectory(as400,
                              "/QSYS.LID/testlib1.lib");

// Create as explorer Pane
AS400ExplorerPane pane = new AS400ExplorerPane((VNode)directory);

// Load the information
pane.load();
```

Print

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

The following components are available:

- A VPrinters object is a resource that represents a list of printers for use in AS/400 panes.
- A VPrinter object is a resource that represents a printer and its spooled files for use in AS/400 panes.
- A VPrinterOutput object is a resource that represents a list of spooled files for use in AS/400 panes.
- A SpooledFileViewer object is a resource that visually represents spooled files.

AS/400 panes are graphical user interface components that present and allow manipulation of one or more AS/400 resources. VPrinters, VPrinter, and VPrinterOutput objects are resources that represent lists of AS/400 print resources in AS/400 panes.

AS/400 pane, VPrinters, VPrinter, and VPrinterOutput objects can be used together to present many views of print resources and to allow the user to select and perform operations on them.

VPrinters

A VPrinters object is a resource that represents a list of printers for use in AS/400 panes.

To use a VPrinters object, set the system property. Set this property by using a constructor or through the setSystem() method. The VPrinters object is then "plugged" into the AS/400 pane as the root, using the pane's constructor or setRoot() method.

A VPrinters object has another useful property for defining the set of printers that is presented in AS/400 panes. Use setPrinterFilter() to specify a filter that defines which printers should appear.

When AS/400 pane and VPrinters objects are created, they are initialized to a default state. The list of printers has not been loaded. To load the contents, the caller must explicitly call the load() method on either object.

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

- Properties - allows the user to set the printer filter property. This may be used to change the contents of the list.

The following actions are available for printers in a printer list:

- Hold - holds the printer.
- Release - releases the printer.
- Start - starts the printer.
- Stop - stops the printer.
- Make available - makes the printer available.
- Make unavailable - makes the printer unavailable.
- Properties - displays properties of the printer and allows the user to set filters.

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

The following example creates a VPrinters object and presents it in an AS400TreePane

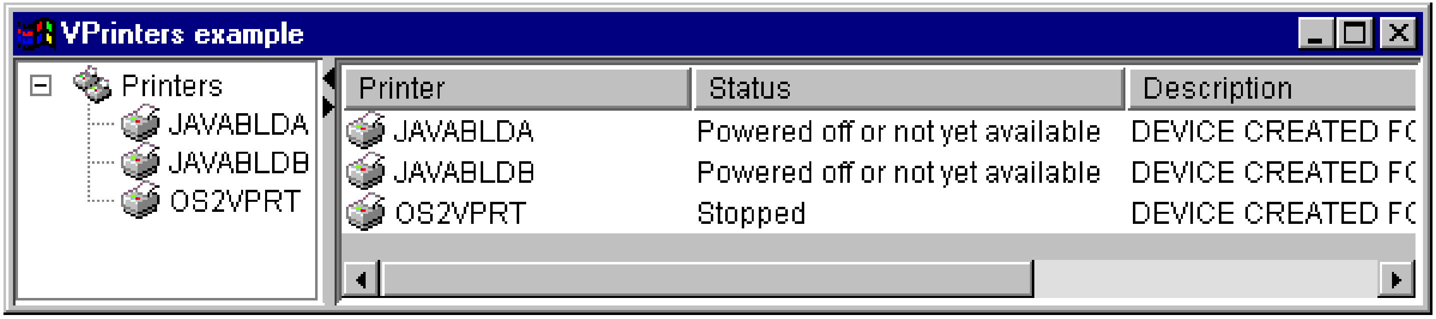
```
                  // Create the VPrinters object.
                  // Assume that "system" is an AS400
                  // object created and initialized
                  // elsewhere.
VPrinters root = new VPrinters (system);

                  // Create and load an AS400TreePane
                  // object.
AS400TreePane treePane = new AS400TreePane (root);
treePane.load ();

                  // Add the tree pane to a frame.
                  // Assume that "frame" is a JFrame
                  // created elsewhere.
frame.getContentPane ().add (treePane);
```

Example

Present print resources using an AS400ExplorerPane with a VPrinters object.

The following image shows the VPrinters graphical user interface component:

VPrinters GUI component



Printer	Status	Description
JAVABLDA	Powered off or not yet available	DEVICE CREATED F(
JAVABLDB	Powered off or not yet available	DEVICE CREATED F(
OS2VPRT	Stopped	DEVICE CREATED F(

VPrinter

A VPrinter object is a resource that represents an AS/400 printer and its spooled files for use in AS/400 panes.

To use a VPrinter, set the printer property. Set this property by using a constructor or through the setPrinter() method. The VPrinter object is then "plugged" into the AS/400 pane as the root, using the pane's constructor or setRoot() method.

When AS/400 pane and VPrinter objects are created, they are initialized to a default state. The printer's attributes and list of spooled files are not loaded at creation time.

To load the contents, the caller must explicitly call the load() method on either object. This will initiate communication to the AS/400 system to gather the contents of the list.

At run-time, a user can perform actions on any printer or spooled file through the pop-up menu. The following actions are available for printers:

- Hold - holds the printer.
- Release - releases the printer.
- Start - starts the printer.
- Stop - stops the printer.
- Make available - makes the printer available.
- Make unavailable - makes the printer unavailable.
- Properties - displays properties of the printer and allows the user to set filters.

The following actions are available for spooled files listed for a printer:

- Reply - replies to the spooled file.
- Hold - holds the spooled file.
- Release - releases the spooled file.
- Print next - prints the next spooled file.
- Send - sends the spooled file.
- Move - moves the spooled file.
- Delete - deletes the spooled file.
- Properties - displays many properties of the spooled file and allows the user to change some of them.

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

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

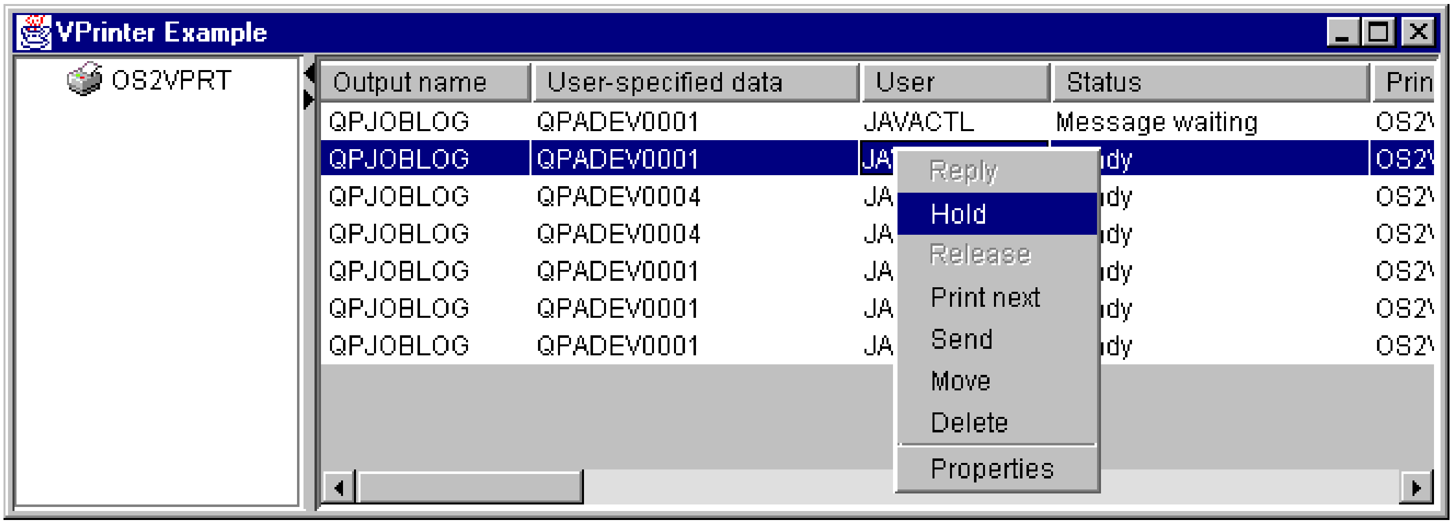
```
                        // Create the VPrinter object.
                        // Assume that "system" is an AS400
                        // object created and initialized
                        // elsewhere.
     VPrinter root = new VPrinter (new Printer (system, "MYPRINTER"));

                        // Create and load an
                        // AS400ExplorerPane object.
     AS400ExplorerPane explorerPane = new AS400ExplorerPane (root);
     explorerPane.load ();

                        // Add the explorer pane to a frame.
                        // Assume that "frame" is a JFrame
                        // created elsewhere.
     frame.getContentPane ().add (explorerPane);
```

Example

Present print resources using an AS400ExplorerPane with a VPrinter object.

The following image shows the VPrinter graphical user interface component:

VPrinter GUI component



Output name	User-specified data	User	Status	Prin
QPJOBLOG	QPADEV0001	JAVACTL	Message waiting	OS2V
QPJOBLOG	QPADEV0001	JA	dy	OS2V
QPJOBLOG	QPADEV0004	JA	dy	OS2V
QPJOBLOG	QPADEV0004	JA	dy	OS2V
QPJOBLOG	QPADEV0001	JA	dy	OS2V
QPJOBLOG	QPADEV0001	JA	dy	OS2V
QPJOBLOG	QPADEV0001	JA	dy	OS2V

VPrinterOutput

A VPrinterOutput object is a resource that represents a list of spooled files on an AS/400 for use in AS/400 panes.

To use a VPrinterOutput object, set the system property. This property can be set using a constructor or through the setSystem() method. The VPrinterOutput object is then "plugged" into the AS/400 pane as the root, using the pane's constructor or setRoot() method.

A VPrinterOutput object has other useful properties for defining the set of spooled files that is presented in AS/400 panes. Use setFormTypeFilter() to specify which types of forms should appear. Use setUserDataFilter() to specify which user data should appear. Finally, use setUserFilter() to specify which users spooled files should appear.

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

At run-time, a user can perform actions on any spooled file or the spooled file list through the pop-up menu. The following action is available for spooled file lists:

- Properties - Allows the user to set the filter properties. This may be used to change the contents of the list.

The following actions are available for spooled files:

- Reply - replies to the spooled file.
- Hold - holds the spooled file.
- Release - releases the spooled file.
- Print next - prints the next spooled file.
- Send - sends the spooled file.
- Move - moves the spooled file.
- Delete - deletes the spooled file.
- Properties - displays many properties of the spooled file and allows the user to change some of them.

Of course, users can only access spooled files to which they are authorized. In addition, the caller can prevent the user from performing actions by using the setAllowActions() method on the pane.

The following example creates a VPrinterOutput and presents it in an AS400ListPane:

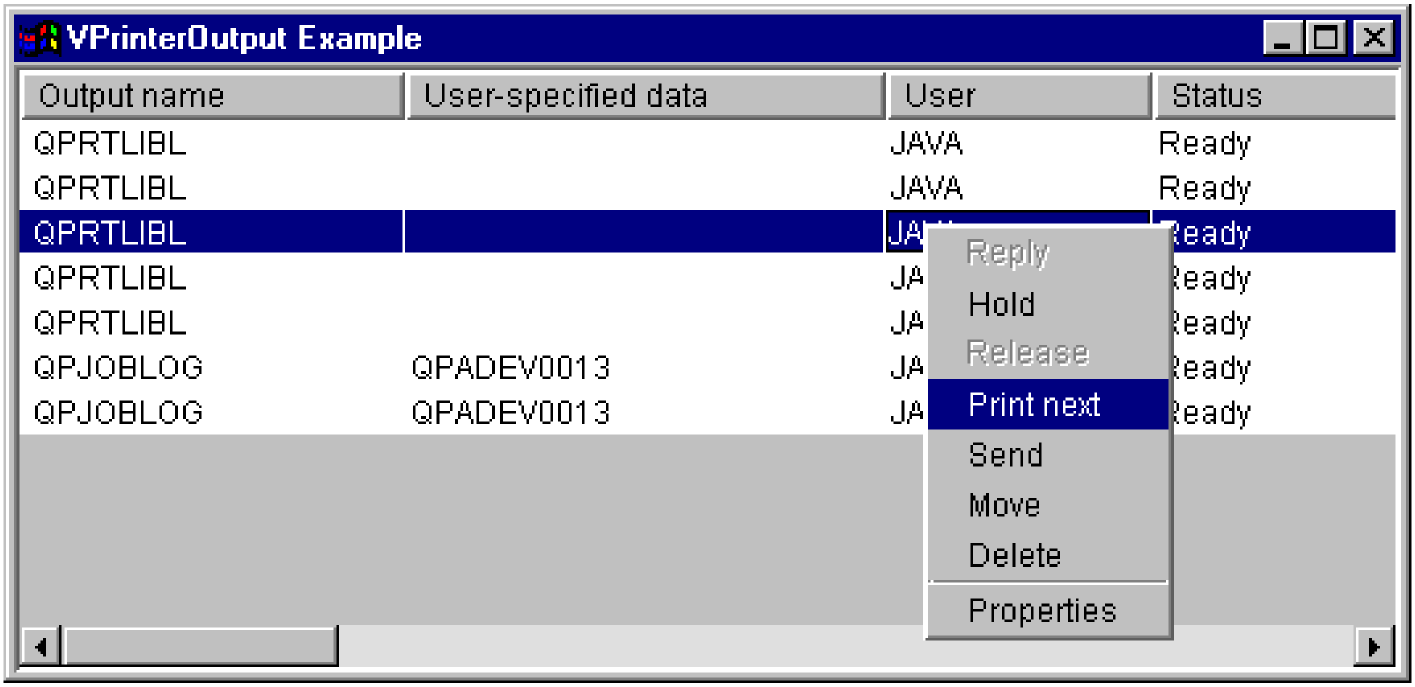
```
                      // Create the VPrinterOutput object.
                      // Assume that "system" is an AS400
                      // object created and initialized
                      // elsewhere.
    VPrinterOutput root = new VPrinterOutput (system);

                      // Create and load an AS400ListPane
                      // object.
    AS400ListPane listPane = new AS400ListPane (root);
    listPane.load ();

                      // Add the list pane to a frame.
                      // Assume that "frame" is a JFrame
                      // created elsewhere.
    frame.getContentPane ().add (listPane);
```

Example

Present a list of spooled files by using the print resource, VPrinterOutput object.

The following image shows the VPrinterOutput graphical user interface component:

VPrinterOutput GUI component



Output name	User-specified data	User	Status
QPRTLIBL		JAVA	Ready
QPRTLIBL		JAVA	Ready
QPRTLIBL		JA	Ready
QPRTLIBL		JA	Ready
QPRTLIBL		JA	Ready
QPJOBLOG	QPADEV0013	JA	Ready
QPJOBLOG	QPADEV0013	JA	Ready

SpooledFileViewer class

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

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

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

Using the SpooledFileViewer class

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

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

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

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

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

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

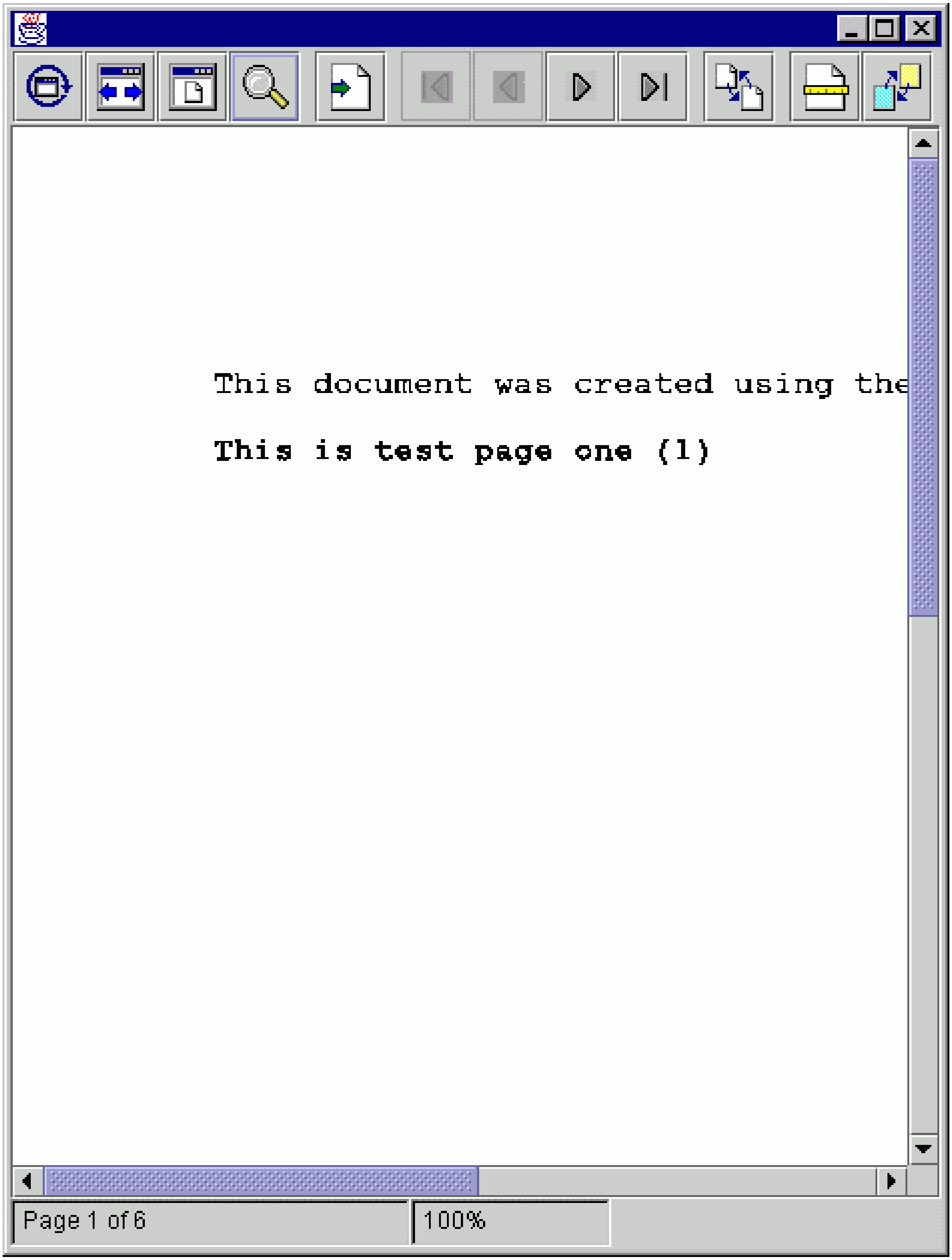
Using the SpooledFileViewer

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

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

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

SpooledFileViewer



This document was created using the

This is test page one (1)

Program call

The program call graphical user interface components allow a Java program to present a button or menu item that calls an AS/400 program. Input, output, and input/output parameters can be specified using ProgramParameter objects. When the program runs, the output and input/output parameters contain data returned by the AS/400 program.

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

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

To use a program call graphical user interface component, set both the system and program properties. Set these properties by using a constructor or through the setSystem() and setProgram() methods.

The following example creates a ProgramCallMenuItem. At run time, when the menu item is selected, it calls a program:

```
                       // Create the ProgramCallMenuItem
                       // object. Assume that "system" is
                       // an AS400 object created and
                       // initialized elsewhere. The menu
                       // item text says "Select Me", and
                       // there is no icon.
     ProgramCallMenuItem menuItem = new ProgramCallMenuItem ("Select Me", null, system);

                       // Create a path name object that
                       // represents program MYPROG in
                       // library MYLIB
     QSYSObjectPathName programName = new QSYSObjectPathName("MYLIB", "MYPROG", "PGM");

                       // Set the name of the program.
     menuItem.setProgram (programName.getPath());

                       // Add the menu item to a menu.
                       // Assume that the menu was created
                       // elsewhere.
     menu.add (menuItem);
```

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

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

```
                       // Add an ActionCompletedListener
                       // that is implemented by using an
                       // anonymous inner class. This is a
                       // convenient way to specify simple
                       // event listeners.
     menuItem.addActionCompletedListener (new ActionCompletedListener ()
     {
          public void actionCompleted (ActionCompletedEvent event)
          {
                       // Cast the source of the event to a
                       // ProgramCallMenuItem.
               ProgramCallMenuItem sourceMenuItem = (ProgramCallMenuItem) event.getSource ();

                       // Get the list of AS/400 messages
                       // that the program generated.
               AS400Message[] messageList = sourceMenuItem.getMessageList ();

                       // ... Process the message list.
          }
     });
```

Parameters

ProgramParameter objects are used to pass parameter data between the Java program and the AS/400 program. Input data is set with the setInputData() method. After the program is run, output data is retrieved with the getOutputData() method.

Each parameter is a byte array. It is up to the Java program to convert the byte array between Java and AS/400 formats. The data conversion classes provide methods for converting data.

You can add parameters to a program call graphical user interface component one at a time using the addParameter() method or all at once using the setParameterList() method.

For more information about using ProgramParameter objects, see the ProgramCall access class.

The following example adds two parameters:

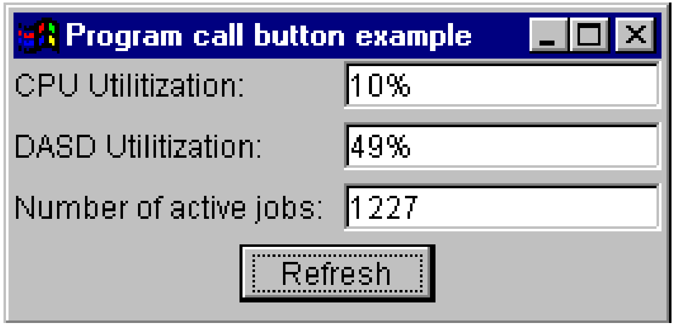
```
                                   // The first parameter is a String
                                   // name of up to 100 characters.
                                   // This is an input parameter.
                                   // Assume that "name" is a String
                                   // created and initialized elsewhere.
     AS400Text parm1Converter = new AS400Text (100, system.getCcsid (), system);
     ProgramParameter parm1 = new ProgramParameter (parm1Converter.toBytes (name));
     menuItem.addParameter (parm1);

                                   // The second parameter is an Integer
                                   // output parameter.
     AS400Bin4 parm2Converter = new AS400Bin4 ();
     ProgramParameter parm2 = new ProgramParameter (parm2Converter.getByteLength ());
     menuItem.addParameter (parm2);

                                   // ... after the program is called,
                                   // get the value returned as the
                                   // second parameter.
     int result = parm2Converter.toInt (parm2.getOutputData ());
```

Examples

Example of using a ProgramCallButton in an application.

The following image shows how the ProgramCallButton looks:

Using ProgramCallButton in an application



CPU Utilitization: 10%

DASD Utilitization: 49%

Number of active jobs: 1227

Record-Level Access

The record-level access graphical user interface components allow a Java program to present various views of AS/400 files.

The following components are available:

- RecordListFormPane presents a list of records from an AS/400 file in a form.
- RecordListTablePane presents a list of records from an AS/400 file in a table.
- RecordListTableModel manages the list of records from an AS/400 file for a table.

Keyed access

You can use the record-level access graphical user interface components with keyed access to an AS/400 file. Keyed access means that the Java program can access the records of a file by specifying a key.

Keyed access works the same for each record-level access graphical user interface component. Use setKeyed() to specify keyed access instead of sequential access. Specify a key using the constructor or the setKey() method. See Specifying the key for more information about how to specify the key.

By default, only records whose keys are equal to the specified key are displayed. To change this, specify the searchType property using the constructor or setSearchType() method. Possible choices are as follows:

- KEY_EQ - Display records whose keys are equal to the specified key.
- KEY_GE - Display records whose keys are greater than or equal to the specified key.
- KEY_GT - Display records whose keys are greater than the specified key.
- KEY_LE - Display records whose keys are less than or equal to the specified key.
- KEY_LT - Display records whose keys are less than the specified key.

The following example creates a RecordListTablePane object to display all records less than or equal to a key.

```
                  // Create a key that contains a
                  // single element, the Integer 5.
Object[] key = new Object[1];
key[0] = new Integer (5);

                  // Create a RecordListTablePane
                  // object. Assume that "system" is an
                  // AS400 object that is created and
                  // initialized elsewhere. Specify
                  // the key and search type.
RecordListTablePane tablePane = new RecordListTablePane (system,
     "/QSYS.LIB/QGPL.LIB/PARTS.FILE", key, RecordListTablePane.KEY_LE);

                  // Load the file contents.
tablePane.load ();

                  // Add the table pane to a frame.
                  // Assume that "frame" is a JFrame
                  // created elsewhere.
frame.getContentPane ().add (tablePane);
```

Record list form panes

A RecordListFormPane presents the contents of an AS/400 file in a form. The form displays one record at a time and provides buttons that allow the user to scroll forward, backward, to the first or last record, or refresh the view of the file contents.

To use a RecordListFormPane, set the system and fileName properties. Set these properties by using the constructor or the setSystem() and setFileName() methods. Use load() to retrieve the file contents and present the first record. When the file contents are no longer needed, call close() to ensure that the file is closed.

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

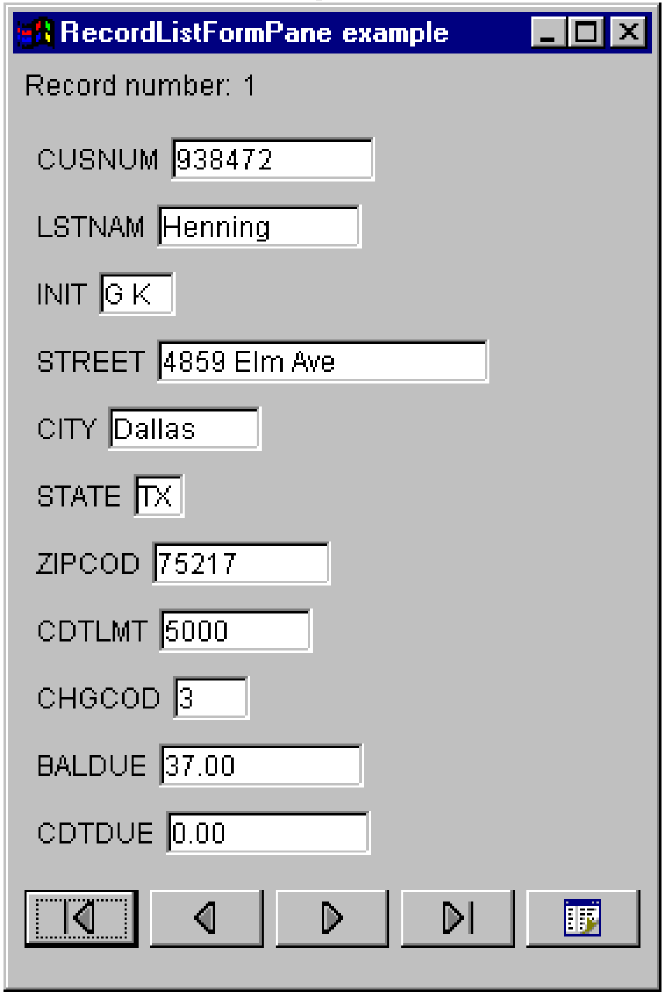
```
                  // Create a RecordListFormPane
                  // object. Assume that "system" is
                  // an AS400 object that is created
                  // and initialized elsewhere.
RecordListFormPane formPane = new RecordListFormPane (system, "/QSYS.LIB/QIWS.LIB/QCUSTCDT.FILE");

                  // Load the file contents.
formPane.load ();

                  // Add the form pane to a frame.
                  // Assume that "frame" is a JFrame
                  // created elsewhere.
frame.getContentPane ().add (formPane);
```

Example

Present an RecordListFormPane which displays the contents of a file.

The following image shows the RecordListFormPane graphical user interface component:

RecordListFormPane GUI component



RecordListFormPane example

Record number: 1

CUSNUM 938472

LSTNAM Henning

INIT G K

STREET 4859 Elm Ave

CITY Dallas

STATE TX

ZIPCOD 75217

CDTLMT 5000

CHGCOD 3

BALDUE 37.00

CDTDUE 0.00

Record list table panes

A RecordListTablePane presents the contents of an AS/400 file in a table. Each row in the table displays a record from the file and each column displays a field.

To use a RecordListTablePane, set the system and fileName properties. Set these properties by using the constructor or the setSystem() and setFileName() methods. Use load() to retrieve the file contents and present the records in the table. When the file contents are no longer needed, call close() to ensure that the file is closed.

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

```
                  // Create an RecordListTablePane
                  // object. Assume that "system" is
                  // an AS400 object that is created
                  // and initialized elsewhere.
RecordListTablePane tablePane = new RecordListTablePane (system, "/QSYS.LIB/QIWS.LIB/QCUSTCDT.FILE");

                  // Load the file contents.
tablePane.load ();

                  // Add the table pane to a frame.
                  // Assume that "frame" is a JFrame
                  // created elsewhere.
frame.getContentPane ().add (tablePane);
```

Record list table models

RecordListTablePane is implemented using the model-view-controller paradigm, in which the data and the user interface are separated into different classes. The implementation integrates RecordListTableModel with Java Foundation Classes' (JFC) JTable. The RecordListTableModel class retrieves and manages the contents of the file and JTable displays the file contents graphically and handles user interaction.

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

To use a RecordListTableModel, set the system and fileName properties. Set these properties by using the constructor or the setSystem() and setFileName() methods. Use load() to retrieve the file contents. When the file contents are no longer needed, call close() to ensure that the file is closed.

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

```
                  // Create a RecordListTableModel
                  // object. Assume that "system" is
                  // an AS400 object that is created
                  // and initialized elsewhere.
RecordListTableModel tableModel = new RecordListTableModel (system, "/QSYS.LIB/QIWS.LIB/QCUSTCDT.FILE");

                  // Load the file contents.
tableModel.load ();

                  // Create a JTable for the model.
JTable table = new JTable (tableModel);

                  // Add the table to a frame. Assume
                  // that "frame" is a JFrame
                  // created elsewhere.
frame.getContentPane ().add (table);
```

»

ResourceListPane and ResourceListDetailsPane

Use the ResourceListPane and ResourceListDetailsPane classes to present a resource list in a graphical user interface (GUI).

- ResourceListPane displays the contents of the resource list in a graphical javax.swing.JList. Every item displayed in the list represents a resource object from the resource list.
- ResourceListDetailsPane displays the contents of the resource list in a graphical javax.swing.JTable. Every row in the table represents a resource object from the resource list.

The table columns for a ResourceListDetailsPane are specified as an array of column attribute IDs. The table contains a column for each element of the array and a row for each resource object.

Pop-up menus are enabled by default for both ResourceListPane and ResourceListDetailsPane.

Most errors are reported as com.ibm.as400.vaccess.ErrorEvents rather than thrown exceptions. Listen for ErrorEvents in order to diagnose and recover from error conditions.

Example: Displaying a resource list in a GUI

This example creates a ResourceList of all users on a system and displays it in a GUI (details pane):

```
// Create the resource list.
AS400 system = new AS400("MYSYSTEM", "MYUSERID", "MYPASSWORD");
RUserList userList = new RUserList(system);

// Create the ResourceListDetailsPane.  In this example,
// there are two columns in the table.  The first column
// contains the icons and names for each user.  The
// second column contains the text description for each
// user.
Object[] columnAttributeIDs = new Object[] { null, RUser.TEXT_DESCRIPTION };
ResourceListDetailsPane detailsPane = new ResourceListDetailsPane();
detailsPane.setResourceList(userList);
detailsPane.setColumnAttributeIDs(columnAttributeIDs);

// Add the ResourceListDetailsPane to a JFrame and show it.
JFrame frame = new JFrame("My Window");
frame.getContentPane().add(detailsPane);
frame.pack();
frame.show();

// The ResourceListDetailsPane will appear empty until
// we load it.  This gives us control of when the list
// of users is retrieved from the iSeries.
detailsPane.load();«
```

System status

The System Status graphical user interface (GUI) components allow you to create GUIs by using the existing AS/400 Panes. You also have the option to create your own GUIs using the Java Foundation Classes (JFC). The VSystemStatus object represents a system status on the AS/400. The VSystemPool object represents a system pool on the AS/400. The VSystemStatusPane represents a visual pane that displays the system status information.

The VSystemStatus class allows you to get information about the status of an AS/400 session within a graphical user interface (GUI) environment. Some of the possible pieces of information you can get are listed below:

- The getSystem() method returns the AS/400 system where the system status information is contained
- The getText() method returns the description text
- The setSystem() method sets the AS/400 where the system status information is located

In addition to the methods mentioned above, you can also access and change system pool information in a graphic interface.

You use VSystemStatus with VSystemStatusPane. VSystemPane is the visual display pane where information is shown for both system status and system pool.

System values GUI

The system value graphical user interface (GUI) components allow a Java program to create GUIs by using the existing AS400 Panes or by creating your own panes using the Java Foundation Classes(JFC). The VSystemValueList object represents a system value list on the AS/400.

To use the System Value GUI component, set the system name with a constructor or through the setSystem() method.

Example

The following example creates a system value GUI using the AS400Explorer Pane:

```
//Create an AS400 object
AS400 mySystem = newAS400("mySystem.myCompany.com");
VSystemValueList mySystemValueList = new VSystemValueList(mySystem);
as400Panel=new AS400ExplorerPane((VNode)mySystemValueList);
//Create and load an AS400ExplorerPane object
as400Panel.load();
```

Users and groups

The users and groups graphical user interface components allow you to present lists of AS/400 users and groups through the VUser class.

The following components are available:

- AS/400 panes are graphical user interface components that present and allow manipulation of one or more AS/400 resources.
- A VUserList object is a resource that represents a list of AS/400 users and groups for use in AS/400 panes.
- A VUserAndGroup object is a resource for use in AS/400 panes that represents groups of AS/400 users. It allows a Java program to list all users, list all groups, or list users who are not in groups.

AS/400 panes and VUserList objects can be used together to present many views of the list. They can also be used to allow the user to select users and groups.

To use a VUserList, you must first set the system property. Set this property by using a constructor or through the setSystem() method. The VUserList object is then "plugged" into the AS/400 pane as the root, using the pane's constructor or setRoot() method.

VUserList has some other useful properties for defining the set of users and groups that are presented in AS/400 panes:

- Use the setUserInfo() method to specify the types of users that should appear.
- Use the setGroupInfo() method to specify a group name.

You can use the VUserAndGroup object to get information about the Users and Groups on the system. Before you can get information about a particular object, you need to load the information so that it can be accessed. You can display the AS/400 system in which the information is found by using the getSystem method.

When AS/400 pane objects and VUserList or VUserAndGroup objects are created, they are initialized to a default state. The list of users and groups has not been loaded. To load the contents, the Java program must explicitly call the load() method on either object to initiate communication to the AS/400 system to gather the contents of the list.

At run-time, right-click a user, user list, or group to display the shortcut menu. Select **Properties** from the shortcut menu to perform actions on the selected object:

- User - Display a list of user information including the description, user class, status, job description, output information, message information, international information, security information, and group information.
- User list - Work with user information and group information properties. You can also change the contents of the list.
- Users and groups - Display properties, such as the user name and description.

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

The following example creates a VUserList and presents it in an AS400DetailsPane:

```
// Create the VUserList object.
// Assume that "system" is an AS400
// object created and initialized
// elsewhere.
VUserList root = new VUserList (system);

// Create and load an
// AS400DetailsPane object.
AS400DetailsPane detailsPane = new AS400DetailsPane (root);
detailsPane.load ();

// Add the details pane to a frame.
// Assume that "frame" is a JFrame
// created elsewhere.
frame.getContentPane ().add (detailsPane);
```

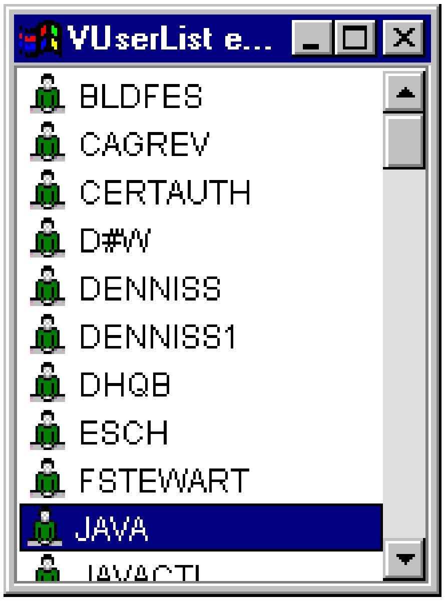
The following example shows how to use the VUserAndGroup object:

```
    // Create the VUserAndGroup object.
    // Assume that "system" is an AS/400 object created and initialized elsewhere.
    VUserAndGroup root = new VUserAndGroup(system);

    // Create and Load an AS/400ExplorerPane
    AS400ExplorerPane explorerPane = new AS400ExplorerPane(root);
    explorerPane.load();

    // Add the explorer pane to a frame
    // Assume that "frame" is a JFrame created elsewhere
    frame.getContentPane().add(explorerPane);
```

Other Examples

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

The following image shows the VUserList graphical user interface component:



HTML Classes

IBM Toolbox for Java HTML classes assist you in:

- Setting up forms and tables for HTML pages
- »Aligning text
- Working with a variety of HTML tags
- Altering the language and direction of text
- Creating ordered and unordered lists
- Creating file lists and HTML hierarchical trees (and the elements within them)
- Adding tag attributes not already defined in the HTML classes (for example, bgcolor and style attributes)«

The HTML classes implement the HTMLTagElement interface. Each class produces an HTML tag for a specific element type. The tag may be retrieved using the getTag() method and can then be imbedded into any HTML document. The tags you generate with the HTML classes are consistent with the HTML 3.2 specification.

The HTML classes can work with servlet classes to get data from the iSeries server. However, they can also be used alone if you supply the table or form data.

The HTML classes make it easier to make HTML forms, tables, and other elements:

- »BidiOrdering class allows you to alter the language and direction of text.«
- »DirFilter class enables you to determine if a File object is a directory.«
- »HTMLAlign class allows you to align blocks of HTML output.«
- »HTMLFileFilter class allows you to determine if a File object is a file.«
- HTMLForm classes help you make forms more easily than CGI scripting.
- »HTMLHeading class allows you to create heading tags for your HTML pages.«
- HTMLHyperlink class helps you create links within your HTML page.
- »HTMLList classes help you create lists for your HTML pages.«
- »HTMLMeta class allows you to create meta tags for your HTML pages.«
- »HTMLParameter class specifies parameters available to the HTMLServlet.«
- »HTMLServlet class allows you to create a server-side include.«
- HTMLTable classes help you make tables for your HTML pages.
- HTMLText class allows you to access the font properties within your HTML page.
- »HTMLTree classes allow you to display an HTML hierarchical tree of HTML elements.«
- URLEncoder class encodes delimiters to use in a URL string.
- »URLParser class allow you to parse a URL string for the URI, properties, and reference.«

NOTE: The jt400Servlet.jar file includes both the HTML and Servlet classes. You must update your CLASSPATH to point to the jt400Servlet.jar file if you want to use the classes in the com.ibm.as400.util.html package.

To find out more information about HTML, see the reference page.

»BidiOrdering class

The BidiOrdering class represents an HTML tag that alters the language and direction of text. An HTML <BDO> string requires two attributes, one for language and the other for the direction of the text.

The BidiOrdering class allows you to:

- Get and set the language attribute
- Get and set the direction of text
- Other?

For more information about using the <BDO> HTML tag, see the W3C Web site.

Example: Using BidiOrdering

The following example creates a BidiOrdering object and sets its language and direction:

```
// Create a BidiOrdering object and set the language and direction.
BidiOrdering bdo = new BidiOrdering();
bdo.setDirection(HTMLConstants.RTL);
bdo.setLanguage("AR");


// Create some text.
HTMLText text = new HTMLText("Some Arabic Text.");
text.setBold(true);

// Add the text to the BidiOrdering and get the HTML tag.
bdo.addItem(text);
bdo.getTag();
```

The print statement produces the following tag:

```
<bdo lang="AR" dir="rtl">
  <b>Some Arabic Text.</b>
</bdo>
```

When you use this tag in an HTML page, browsers that can understand the <BDO> tag display the example like this:

.txeT cibarA emoS

«

HTMLAlign class

The HTMLAlign class enables you to align sections of your HTML document, instead of just aligning individual items, say paragraphs or headings.

The HTMLAlign class represents the <DIV> tag and its associated align attribute. You can use right, left, or center alignment.

You can use this class to perform a variety of actions that include the following:

- Add or remove items from the list of tags you want to align
- Get and set the alignment
- Get and set the direction of the text interpretation
- Get and set the language of the input element
- Get a String representation of the HTMLAlign object

Example: Creating HTMLAlign objects

The following example creates an unordered list, then creates an HTMLAlign object to align the entire list:

```
// Create an unordered list.
UnorderedList uList = new UnorderedList();
uList.setType(HTMLConstants.DISC);
UnorderedListItem uListItem1 = new UnorderedListItem();
uListItem1.setItemData(new HTMLText("Centered unordered list"));
uList.addListItem(uListItem1);
UnorderedListItem uListItem2 = new UnorderedListItem();
uListItem2.setItemData(new HTMLText("Another item"));
uList.addListItem(uListItem2);

// Align the list.
HTMLAlign align = new HTMLAlign(uList, HTMLConstants.CENTER);
System.out.println(align);
```

The previous example produces the following tag:

```
<div align="center">
<ul type="disc">
  <li>Centered unordered list</li>
  <li>Another item</li>
</ul>
```

When you use this tag in an HTML page, it looks like this:

- Centered unordered list
- Another item

HTML form classes

The HTMLForm class represents an HTML form. This class allows you to:

- Add an element, like a button, hyperlink or HTML table to a form
- Remove an element from a form
- Set other form attributes, such as which method to use to send form contents to the server, the hidden parameter list, or the action URL address

The constructor for the HTMLForm object takes a URL address. This address is referred to as an action URL. It is the location of the application on the server that will process the form input. The action URL can be specified on the constructor or by setting the address using the setURL() method. Form attributes are set using various set methods and retrieved using various get methods.

Any HTML tag element may be added to an HTMLForm object using addElement() and removed using removeElement(). The HTML tag element classes that you can add to an HTML form follow:

- Form Input classes: represent input elements for an HTML form
- HTMLText: encapsulates the various text options you can use within an HTML page
- HTMLHyperlink: represents an HTML hyperlink tag
- Layout Form Panel classes: represent a layout of form elements for an HTML form
- TextAreaFormElement: represents a text area element in an HTML form
- LabelFormElement: represents a label for an HTML form element
- SelectFormElement: represents a select input type for an HTML form
- SelectOption: represents an option for a SelectFormElement object in an HTML form
- RadioFormInputGroup: represents a group of radio input objects which allow a user to select one from a group
- HTMLTable: represents an HTML table tag

For more information on creating a form using the HTMLForm class, see this example and the resulting output.

Form Input classes

The FormInput class allows you to:

- Get and set the name of an input element
- Get and set the size of an input element
- Get and set the initial value of an input element

The FormInput class is extended by the classes listed below, classes that provide a way to create specific types of form input elements and allow you to get and set various attributes or retrieve the HTML tag for the input element:

- ButtonFormInput: Represents a button element for an HTML form
- FileFormInput: Represents a file input type, for an HTML form
- HiddenFormInput: Represents a hidden input type for an HTML form
- ImageFormInput: Represents an image input type for an HTML form.
- ResetFormInput: Represents a reset button input for an HTML form
- SubmitFormInput: Represents a submit button input for an HTML form
- TextFormInput: Represents a single line of text input for an HTML form where you define the maximum number of characters in a line. For a password input type, you use PasswordFormInput, which extends TextFormInput and represents a password input type for an HTML form
- ToggleFormInput: Represents a toggle input type for an HTML form. The user can set or get the text label and specify whether the toggle should be checked or selected. The toggle input type can be one of two:
    - RadioFormInput: Represents a radio button input type for an HTML form. Raido buttons may be placed in groups with the RadioFormInputGroup class; this creates a group of radio buttons where the user selects only one of the choices presented.
    - CheckboxFormInput: Represents a checkbox input type for an HTML form where the user may select more than one from the choices presented, and where the checkbox is initialized as either checked or unchecked.

ButtonFormInput class

The ButtonFormInput class represents a button element for an HTML form.

The following example shows you how to create a ButtonFormInput object:

```
ButtonFormInput button = new ButtonFormInput("button1", "Press Me", "test()");
System.out.println(button.getTag());
```

This example produces the following tag:

<input type="button" name="button1" value="Press Me" onclick="test()" />

When you use this tag in an HTML page, it looks like this:

FileFormInput class

The FileFormInput class represents a file input type in an HTML form.

The following code example shows you how to create a new FileFormInput object

```
    FileFormInput file = new FileFormInput("myFile");
    System.out.println(file.getTag());
```

The above code creates the following output:

<input type="file" name="myFile" />

When you use this tag in an HTML page, it looks like this:

HiddenFormInput class

The HiddenFormInput class represents a hidden input type in an HTML form.

The following code example shows you how to create a HiddenFormInput object:

```
    HiddenFormInput hidden = new HiddenFormInput("account", "123456");
    System.out.println(hidden.getTag()):
```

The code above generates the following tag:

<input type="hidden" name="account" value="123456" />

In an HTML page, the HiddenInputType does not display. It simply sends the information (in this case the account number) back to the server.

ImageFormInput class

The ImageFormInput class represents an image input type in an HTML form.

You can retrieve and update many of the attributes for the ImageFormInput class by using the methods provided.

- Get or set the source
- Get or set the alignment
- Get or set the height
- Get or set the width

The following code example shows you how to create an ImageFormInput object:

```
ImageFormInput image = new ImageFormInput("myPicture", "pc_100.gif");
image.setAlignment(HTMLConstants.TOP);
image.setHeight(81);
image.setWidth(100);
```

The above code example generates the following tag:

<input type="image" name="MyPicture" src="pc_100.gif" align="top" height="81" width="100" />

When you use this tag in an HTML form, it looks like this:

ResetFormInput class

The ResetFormInput class represents a reset button input type in an HTML form.

The following code example shows you how to create a ResetFormInput object:

```
ResetFormInput reset = new ResetFormInput();
reset.setValue("Reset");
System.out.println(reset.getTag());
```

The above code example generates the following HTML tag:

<input type="reset" value="Reset" />

When you use this tag in an HTML form, it looks like this:

SubmitFormInput class

The SubmitFormInput class represents a submit button input type in an HTML form.

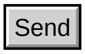
The following code example shows you how to create a SubmitFormInput object:

```
SubmitFormInput submit = new SubmitFormInput();
submit.setValue("Send");
System.out.println(submit.getTag());
```

The code example above generates the following output:

<input type="submit" value="Send" />

When you use this tag in an HTML form, it looks like this:



TextFormInput class

The TextFormInput class represents a single line text input type in an HTML form. The TextFormInput class provides methods that let you get and set the maximum number of characters a user can enter in the text field.

The following example shows you how to create a new TextFormInput object:

```
TextFormInput text = new TextFormInput("userID");
text.setSize(40);
System.out.println(text.getTag());
```

The code example above generates the following tag:

<input type="text" name="userID" size="40" />

When you use this tag in an HTML form, it looks like this:

Name:

PasswordFormInput class

The PasswordFormInput class represents a password input field type in an HTML form.

The following code example shows you how to create a new PasswordFormInput object:

```
    PasswordFormInput pwd = new PasswordFormInput("password");
    pwd.setSize(12);
    System.out.println(pwd.getTag());
```

The code example above generates the following tag:

<input type="password" name="password" size="12" />

When you use this tag in an HTML form, it looks like this:

Password:

RadioFormInput

The RadioFormInput class represents a radio button input type in an HTML form. The radio button may be initialized as selected when constructed.

A set of radio buttons with the same control name make a radio button group. The RadioFormInputGroup class creates radio button groups. Only one radio button within the group may be selected at any time. Also, a specific button may be initialized as selected when the group is constructed.

The following code example shows you how to create a RadioFormInput object:

```
    RadioFormInput radio = new RadioFormInput("age", "twentysomething", "Age 20 - 29", true);
    System.out.println(radio.getTag());
```

The above code example generates the following tag:

<input type="radio" name="age" value="twentysomething" checked="checked" />

When you use this tag in an HTML form, it looks like this:

Age 20-29 ◉

CheckboxFormInput class

The CheckboxFormInput class represents a checkbox input type in an HTML form. The user may select more than one of the choices presented as checkboxes within a form.

The following example shows you how to create a new CheckboxFormInput object:

```
    CheckboxFormInput checkbox = new CheckboxFormInput("uscitizen", "yes", "textLabel", true);
    System.out.println(checkbox.getTag());
```

The code above produces the following output:

```
<input type="checkbox" name="uscitizen" value="yes" checked="checked" /> textLabel
```

When you use this tag in an HTML form, it looks like this:

☑ textLabel

HTML Text class

The HTMLText class allows you to access text properties for your HTML page. Using the HTMLText class, you can get, set and check the status of many text attributes, including:

- Get or set the size of the font
- Set the bold attribute on (true) or off (false) or determine if it is already on
- Set the underscore attribute on (true) or off (false) or determine if it is already on
- Get or set the horizontal alignment of the text

The following example shows you how to create an HTMLText object and set its bold attribute on and its font size to 5.

```
HTMLText text = new HTMLText("IBM");
text.setBold(true);
text.setSize(5);
System.out.println(text.getTag());
```

The print statement produces the following tag:

```
 <font size="5"><b>IBM</b></font>
```

When you use this tag in an HTML page, it looks like this:

IBM

HTMLHyperlink class

The HTMLHyperlink class represents an HTML hyperlink tag. You use the HTMLHyperlink class to create a link within your HTML page. You can get and set many attributes of hyperlinks with this class, including:

- Get or set the Uniform Resource Identifier for the link
- Get or set the title for the link
- Get or set the target frame for the link

The HTMLHyperlink class can print the full hyperlink with defined properties so that you can use the output in your HTML page.

The following is an example for HTMLHyperlink:

```
// Create an HTML hyperlink to the IBM Toolbox for Java home page.
HTMLHyperlink toolbox = new HTMLHyperlink("http://www.ibm.com/as400/toolbox", "IBM Toolbox for Java home page");

// Display the toolbox link tag.
System.out.println(toolbox.toString());
```

The code above produces the following tag:
<a hef="http://www.ibm.com/as400/toolbox">IBM Toolbox for Java home page</a>

When you use this tag in an HTML page, it looks like this:

IBM Toolbox for Java home page

LayoutFormPanel class

The LayoutFormPanel class represents a layout of form elements for an HTML form. You can use the methods that LayoutFormPanel provides to add and remove elements from a panel or to get the number of elements in the layout. You may choose to use one of two layouts:

- GridLayoutFormPanel: Represents a grid layout of form elements for an HTML form.
- LineLayoutFormPanel: Represents a line layout of form elements for an HTML form.

GridLayoutFormPanel

The GridLayoutFormPanel class represents a grid layout of form elements. You use this layout for an HTML form where you specify the number of columns for the grid.

The following example creates a GridLayoutFormPanel object with two columns:

```
         // Create a text form input element for the system.
  LabelFormElement sysPrompt = new LabelFormElement("System:");
  TextFormInput system = new TextFormInput("System");


         // Create a text form input element for the userId.
  LabelFormElement userPrompt = new LabelFormElement("User:");
  TextFormInput user = new TextFormInput("User");


         // Create a password form input element for the password.
  LabelFormElement passwordPrompt = new LabelFormElement("Password:");
  PasswordFormInput password = new PasswordFormInput("Password");


         // Create the GridLayoutFormPanel object with two columns and add the form elements.
  GridLayoutFormPanel panel = new GridLayoutFormPanel(2);
  panel.addElement(sysPrompt);
  panel.addElement(system);
  panel.addElement(userPrompt);
  panel.addElement(user);
  panel.addElement(passwordPrompt);
  panel.addElement(password);


         // Create the submit button to the form.
  SubmitFormInput logonButton = new SubmitFormInput("logon", "Logon");


         // Create HTMLForm object and add the panel to it.
  HTMLForm form = new HTMLForm(servletURI);
  form.addElement(panel);
  form.addElement(logonButton);
```

This example produces the following HTML code:

```
<form action=servletURI method="get">
<table border="0">
<tr>
<td>System:</td>
<td><input type="text" name="System" /></td>
</tr>
<tr>
<td>User:</td>
<td><input type="text" name="User" /></td>
</tr>
<tr>
<td>Password:</td>
<td><input type="password" name="Password" /></td>
</tr>
</table>
<input type="submit" name="logon" value="Logon" />
</form>
```

When you use this HTML code in a webpage, it looks like this:

System:

User:

Password:

Logon

LineLayoutFormPanel class

The LineLayoutFormPanel class represents a line layout of form elements for an HTML form. The form elements are arranged in a single row within a panel.

This example creates a LineLayoutFormPanel object and adds two form elements.

```
CheckboxFormInput privacyCheckbox = new CheckboxFormInput("confidential", "yes", "Confidential", true);
CheckboxFormInput mailCheckbox = new CheckboxFormInput("mailingList", "yes", "Join our mailing list", false);
LineLayoutFormPanel panel = new LineLayoutFormPanel();
panel.addElement(privacyCheckbox);
panel.addElement(mailCheckbox);
String tag = panel.getTag();
```

The code example above generates the following HTML code:

<input type="checkbox" name="confidential" value="yes" checked="checked" /> Confidential <input type="checkbox" name="mailingList" value="yes" /> Join our mailing list <br />

When you use this HTML code in a webpage, it looks like this:

☑ Confidential □ Join our mailing list

TextAreaFormElement class

The TextAreaFormElement class represents a text area element in an HTML form. You determine the size of the text area by setting the number of rows and columns. You can determine the size that a text area element is set for with the getRows() and getColumns() methods.

You set the initial text within the text area with the setText() method. You use the getText() method to see what the initial text has been set to.

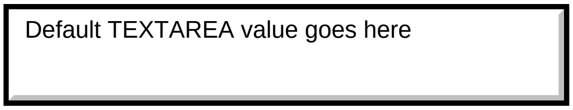
The following example shows you how to create a TextAreaFormElement:

```
    TextAreaFormElement textArea = new TextAreaFormElement("foo", 3, 40);
    textArea.setText("Default TEXTAREA value goes here");
    System.out.println(textArea.getTag());
```

The code example above generates the following HTML code:

```
<form>
<textarea name="foo" rows="3" cols="40">
Default TEXTAREA value goes here
</textarea>
</form>
```

When you use this in an HTML form, it looks like this:



Default TEXTAREA value goes here

LabelFormElement

The LabelFormElement class represents a label for an HTML form element. You use the LabelFormElement class to label elements of an HTML form such as a text area or password form input . The label is one line of text that you set using the setLabel() method. This text does not respond to user input and is there to make the form easier for the user to understand.

The following code example shows you how to create a LabelFormElement object:

```
LabelFormElement label = new LabelFormElement("Account Balance");
System.out.println(label.getTag());
```

This example produces the following output:

```
Account Balance
```

SelectFormElement

The SelectFormElement class represents a select input type for an HTML form. You can add and remove various options within the select element.

SelectFormElement has methods available that allow you to view and change attributes of the select element:

- Use setMultiple() to set whether or not the user can select more than one option
- Use getOptionCount() to determine how many elements are in the option layout
- Use setSize() to set the number of options visible within the select element and use getSize() to determine the number of visible options.

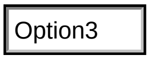
The following example creates a SelectFormElement object with three options. The SelectFormElement object named *list*, is highlighted. The first two options added specify the option text, name, and select attributes. The third option added is defined by a SelectOption object.

```
SelectFormElement list = new SelectFormElement("list1");
SelectOption option1 = list.addOption("Option1", "opt1");
SelectOption option2 = list.addOption("Option2", "opt2", false);
SelectOption option3 = new SelectOption("Option3", "opt3", true);
list.addOption(option3);
System.out.println(list.getTag());
```

The above code example produces the following HTML code:

```
<select name="list1">
<option value="opt1">Option1</option>
<option value="opt2">Option2</option>
<option value="opt3" selected="selected">Option3</option>
</select>
```

When you use this code in an HTML page, it looks like this:



SelectOption

The SelectOption class represents an option in an HTML option form element. You use the option form element in a select form.

Methods are provided that you can use to retrieve and set attributes within a SelectOption. For instance, you can set whether or not the option defaults to being selected. You can also set the input value it will use when the form is submitted.

The following example creates three SelectOption objects within a select form. Each SelectOption object below is highlighted. They are named *option1*, *option2* and *option3*. The *option3* object is initially selected.

```
SelectFormElement list = new SelectFormElement("list1");
SelectOption option1 = list.addOption("Option1", "opt1");
SelectOption option2 = list.addOption("Option2", "opt2", false);
SelectOption option3 = new SelectOption("Option3", "opt3", true);
list.addOption(option3);
System.out.println(list.getTag());
```

The above code example produces the following HTML tag:

```
<select name="list1">
<option value="opt1">Option1</option>
<option value="opt2">Option2</option>
<option value="opt3" selected="selected">Option3</option>
</select>
```

When you use this tag in an HTML form, it looks like this:

Option3

RadioFormInputGroup class

The RadioFormInputGroup class represents a group of RadioFormInput objects. A user can select only one of the RadioFormInput objects from a RadioFormInputGroup.

The RadioFormInputGroup class methods allow you to work with various attributes of a group of radio buttons. With these methods, you can:

- Add a radio button
- Remove a radio button
- Get or set the name of the radio group

The following example creates a radio button group:

```
// Create some radio buttons.
RadioFormInput radio0 = new RadioFormInput("age", "kid", "0-12", true);
RadioFormInput radio1 = new RadioFormInput("age", "teen", "13-19", false);
RadioFormInput radio2 = new RadioFormInput("age", "twentysomething", "20-29", false);
RadioFormInput radio3 = new RadioFormInput("age", "thirtysomething", "30-39", false);
// Create a radio button group and add the radio buttons.
RadioFormInputGroup ageGroup = new RadioFormInputGroup("age");
ageGroup.add(radio0);
ageGroup.add(radio1);
ageGroup.add(radio2);
ageGroup.add(radio3);
System.out.println(ageGroup.getTag());
```

The code example above generates the following HTML code:

<input type="radio" name="age" value="kid" checked="checked" /> 0-12
<input type="radio" name="age" value="teen" /> 13-19
<input type="radio" name="age" value="twentysomething" /> 20-29
<input type="radio" name="age" value="thirtysomething" /> 30-39

When you use this code in an HTML form, it looks like this:

◉ 0-12 ○ 13-19 ○ 20-29 ○ 30-39

HTMLHeading class

The HTMLHeading class represents an HTML header. Each header can have its own alignment and level from 1 (largest font, most importance) to 6.

Methods for the HTMLHeading class include:

- Get and set the text for the header
- Get and set the level of the header
- Get and set the alignment of the header
- Get and set the direction of the text interpretation
- Get and set the language of the input element
- Get a String representation of the HTMLHeader object

Example: Creating HTMLHeading objects

The following example creates three HTMLHeading objects:

```
     // Create and display three HTMLHeading objects.
     HTMLHeading h1 = new HTMLHeading(1, "Heading", HTMLConstants.LEFT);
     HTMLHeading h2 = new HTMLHeading(2, "Subheading", HTMLConstants.CENTER);
     HTMLHeading h3 = new HTMLHeading(3, "Item", HTMLConstants.RIGHT);
     System.out.print(h1 + "\r\n" + h2 + "\r\n" + h3);
```

The previous example produces the following tags:

```
     <h1 align="left">Heading</h1>
     <h2 align="center">Subheading</h2>
     <h3 align="right">Item</h3>
```

When you use these tags in an HTML page, it looks like this:

Heading

Subheading

Item

»HTMLList classes

The HTMLList classes allow you to easily create lists within your HTML pages. These classes provide methods to get and set various attributes of the lists and the items within the lists.

In particular, the parent class HTMLList provides a method to produce a compact list that displays items in as small a vertical space as possible.

- Methods for HTMLList include:
    - Compact the list
    - Add and remove items from the list
    - Add and remove lists from the list (making it possible to nest lists)
- Methods for HTMLListItem include:
    - Get and set the contents of the item
    - Get and set the direction of the text interpretation
    - Get and set the language of the input element

Use the subclasses of HTMLList and HTMLListItem to create your HTML lists:

- OrderedList and OrderedListItem
- UnorderedList and UnorderedListItem

For coding snippets, see the following examples:

- **Example:** Creating ordered lists
- **Example:** Creating unordered lists
- **Example:** Creating nested lists

OrderedList and OrderedListItem

Use the OrderedList and OrderedListItem classes to create ordered lists in your HTML pages.

- Methods for OrderedList include:
    - Get and set the starting number for the first item in the list
    - Get and set the type (or style) for the item numbers
- Methods for OrderedListItem include:
    - Get and set the number for the item
    - Get and set the type (or style) for the item number

By using the methods in OrderedListItem, you can override the numbering and type for a specific item in the list.

See the example for creating ordered lists.

UnorderedList and UnorderedListItem

Use the UnorderedList and UnorderedListItem classes to create unordered lists in your HTML pages.

- Methods for UnorderedList include:
    - Get and set the type (or style) for the items
- Methods for UnorderedListItem include:
    - Get and set the type (or style) for the item

See the example for creating unordered lists.

Examples

The following examples show you how to use the HTMLList classes to create ordered lists, unordered lists, and nested lists.

Example: Creating ordered lists

The following example creates an ordered list:

```
     // Create an OrderedList.
OrderedList oList = new OrderedList(HTMLConstants.SMALL_ROMAN);
     // Create the OrderedListItems.
OrderedListItem listItem1 = new OrderedListItem();
OrderedListItem listItem2 = new OrderedListItem();
     // Set the data in the OrderedListItems.
listItem1.setItemData(new HTMLText("First item"));
listItem2.setItemData(new HTMLText("Second item"));
     // Add the list items to the OrderedList.
oList.addListItem(listItem1);
oList.addListItem(listItem2);
System.out.println(oList.getTag());
```

The previous example produces the following tags:

```
<ol type="i">
<li>First item</li>
<li>Second item</li>
</ol>
```

When you use these tags in an HTML page, it looks like this:

i. First item

ii. Second item

Example: Creating unordered lists

The following example creates an unordered list:

```
     // Create an UnorderedList.
UnorderedList uList = new UnorderedList(HTMLConstants.SQUARE);
     // Create the UnorderedListItems.
UnorderedListItem listItem1 = new UnorderedListItem();
UnorderedListItem listItem2 = new UnorderedListItem();
     // Set the data in the UnorderedListItems.
listItem1.setItemData(new HTMLText("First item"));
listItem2.setItemData(new HTMLText("Second item"));
     // Add the list items to the UnorderedList.
uList.addListItem(listItem1);
uList.addListItem(listItem2);
System.out.println(uList.getTag());
```

The previous example produces the following tags:

```
<ul type="square">
<li>First item</li>
<li>Second item</li>
</ul>
```

When you use these tags in an HTML page, it looks like this:

- First item
- Second item

Example: Creating nested lists

The following example creates a nested list:

```
     // Create an UnorderedList.
UnorderedList uList = new UnorderedList(HTMLConstants.SQUARE);
     // Create and set the data for UnorderedListItems.
UnorderedListItem listItem1 = new UnorderedListItem();
UnorderedListItem listItem2 = new UnorderedListItem();
listItem1.setItemData(new HTMLText("First item"));
listItem2.setItemData(new HTMLText("Second item"));
     // Add the list items to the UnorderedList.
uList.addListItem(listItem1);
uList.addListItem(listItem2);

     // Create an OrderedList.
OrderedList oList = new OrderedList(HTMLConstants.SMALL_ROMAN);
     // Create the OrderedListItems.
OrderedListItem listItem1 = new OrderedListItem();
OrderedListItem listItem2 = new OrderedListItem();
OrderedListItem listItem3 = new OrderedListItem();
     // Set the data in the OrderedListItems.
listItem1.setItemData(new HTMLText("First item"));
listItem2.setItemData(new HTMLText("Second item"));
listItem3.setItemData(new HTMLText("Third item"));
     // Add the list items to the OrderedList.
oList.addListItem(listItem1);
oList.addListItem(listItem2);
     // Add (nest) the unordered list to OrderedListItem2
oList.addList(uList);
     // Add another OrderedListItem to the OrderedList
     // after the nested UnorderedList.
oList.addListItem(listItem3);
System.out.println(oList.getTag());
```

The previous example produces the following tags:

```
<ol type="i">
<li>First item</li>
<li>Second item</li>
<ul type="square">
<li>First item</li>
<li>Second item</li>
</ul>
<li>Third item</li>
</ol>
```

When you use these tags in an HTML page, it looks like this:

i. First item

ii. Second item

  - First item
  - Second item

iii. Third item

«

»HTMLMeta class

The HTMLMeta class represents meta-information used within an HTMLHead tag. Attributes in META tags are used when identifying, indexing, and defining information within the HTML document.

Attributes of the META tag include:

- NAME - the name associated with the contents of the META tag
- CONTENT - values associated with the NAME attribute
- HTTP-EQUIV - information gathered by HTTP servers for response message headers
- LANG - the language
- URL - used to redirect users from the current page to another URL

For example, to help search engines determine the contents of a page, you might use the following META tag:

```
<META name="keywords" lang="en-us" content="games, cards, bridge">
```

You can also use HTMLMeta to redirect a user from one page to another.

Methods for the HTMLMeta class include:

- Get and set the NAME attribute
- Get and set the CONTENT attribute
- Get and set the HTTP-EQUIV attribute
- Get and set the LANG attribute
- Get and set the URL attribute

Example: Creating META tags

The following example creates two META tags:

```
// Create a META tag to help search engines determine page content.
HTMLMeta meta1 = new HTMLMeta();
meta1.setName("keywords");
meta1.setLang("en-us");
meta1.setContent("games, cards, bridge");
// Create a META tag used by caches to determine when to refresh the page.
HTMLMeta meta2 = new HTMLMeta("Expires", "Mon, 01 Jun 2000 12:00:00 GMT");
System.out.print(meta1 + "\r\n" + meta2);
```

The previous example produces the following tags:

```
<meta name="keywords" content="games, cards, bridge">
<meta http-equiv="Expires" content="Mon, 01 Jun 2000 12:00:00 GMT">
```

«

»

HTMLParameter class

The HTMLParameter class represents the parameters you can use with the HTMLServlet class. Each parameter has its own name and value.

Methods for the HTMLParameter class include:

- Get and set the name of the parameter
- Get and set the value of the parameter

Example: Creating HTMLParameter tags

The following example creates an HTMLParameter tag:

```
     // Create an HTMLServletParameter.
HTMLParameter parm = new HTMLParameter ("age", "21");
System.out.println(parm);
```

The previous example produces the following tag:

```
<param name="age" value="21">
```

«

»HTMLServlet class

The HTMLServlet class represents a server-side include. The servlet object specifies the name of the servlet and, optionally, its location. You may also choose to use the default location on the local system.

The HTMLServlet class works with the HTMLParameter class, which specifies the parameters available to the servlet.

Methods for the HTMLServlet class include:

- Add and remove HTMLParameters from the servlet tag
- Get and set the location of the servlet
- Get and set the name of the servlet
- Get and set the alternate text of the servlet

Example: Creating HTMLServlet tags

The following example an HTMLServlet tag:

```
     // Create an HTMLServlet.
HTMLServlet servlet = new HTMLServlet("myServlet", "http://server:port/dir");
     // Create a parameter, then add it to the servlet.
HTMLParameter param = new HTMLParameter("parm1", "value1");
servlet.addParameter(param);
     // Create and add second parameter
HTMLParameter param2 = servlet.add("parm2", "value2");
     // Create the alternate text if the web server does not support the servlet tag.
servlet.setText("The web server providing this page does not support the SERVLET tag.")
System.out.println(servlet);
```

The previous example produces the following tags:

```
<servlet name="myServlet" codebase="http://server:port/dir">
<param name="parm1" value="value1">
<param name="parm2" value="value2">
The web server providing this page does not support the SERVLET tag.
</servlet>
```

«

HTML Table classes

The HTMLTable class allows you to easily set up tables that you can use in HTML pages. This class provides methods to get and set various attributes of the table, including:

- Get and set the width of the border
- Get the number of rows in the table
- Add a column or row to the end of the table
- Remove a column or row at a specified column or row position

The HTMLTable class uses other HTML classes to make creating a table easier. The other HTML classes that work to create tables are:

- HTMLTableCell: Creates a table cell
- HTMLTableRow: Creates a table row
- HTMLTableHeader: Creates a table header cell
- HTMLTableCaption: Creates a table caption

Example

Example: Using HTMLTable classes.

HTMLTableCell class

The HTMLTableCell class takes any HTMLTagElement object as input and creates the table cell tag with the specified element. The element can be set on the constructor or through either of two setElement() methods.

Many cell attributes can be retrieved or updating using methods that are provided in the HTMLTableCell class. Some of the actions you can do with these methods are:

- Get or set the row span
- Get or set the cell height
- Set whether or not the cell data will use normal HTML line breaking conventions

The following example creates an HTMLTableCell object and displays the tag:

```
//Create an HTMLHyperlink object.
HTMLHyperlink link = new HTMLHyperlink("http://www.ibm.com",
                    "IBM Home Page");
HTMLTableCell cell = new HTMLTableCell(link);
cell.setHorizontalAlignment(HTMLConstants.CENTER);
System.out.println(cell.getTag());
```

The getTag() method above gives the output of the example:

```
<td align="center"><a href="http://www.ibm.com">IBM Home Page</a></td>
```

HTMLTableRow class

The HTMLTableRow class creates a row within a table. This class provides various methods for getting and setting attributes of a row. Some things you can do with these methods are:

- Add or remove a column from the row
- Get column data at the specified column Index
- Get column index for the column with the specified cell.
- Get the number of columns in a row
- Set horizontal and vertical alignments

The following is an example for HTMLTableRow:

```
// Create a row and set the alignment.
HTMLTableRow row = new HTMLTableRow();
row.setHorizontalAlignment(HTMLTableRow.CENTER);

// Create and add the column information to the row.
HTMLText account = new HTMLText(customers_[rowIndex].getAccount());
HTMLText name = new HTMLText(customers_[rowIndex].getName());
HTMLText balance = new HTMLText(customers_[rowIndex].getBalance());

row.addColumn(new HTMLTableCell(account));
row.addColumn(new HTMLTableCell(name));
row.addColumn(new HTMLTableCell(balance));

// Add the row to an HTMLTable object (assume that the table already exists).
table.addRow(row);
```

HTMLTableHeader

The HTMLTableHeader class inherits from the HTMLTableCell class. It creates a specific type of cell, the header cell, giving you a **<th>** cell instead of a **<td>** cell. Like the HTMLTableCell class, you call various methods in order to update or retreive attributes of the header cell.

The following is an example for HTMLTableHeader:

```
// Create the table headers.
HTMLTableHeader account_header = new HTMLTableHeader(new HTMLText("ACCOUNT"));
HTMLTableHeader name_header = new HTMLTableHeader(new HTMLText("NAME"));
HTMLTableHeader balance_header = new HTMLTableHeader();
HTMLText balance = new HTMLText("BALANCE");
balance_header.setElement(balance);

// Add the table headers to an HTMLTable object (assume that the table already exists).
table.addColumnHeader(account_header);
table.addColumnHeader(name_header);
table.addColumnHeader(balance_header);
```

HTMLTableCaption

The HTMLTableCaption class creates a caption for your HTML table. The class provides methods for updating and retrieving the attributes of the caption. For example, you can use the setAlignment() method to specify to which part of the table the caption should be aligned. The following is an example for HTMLTableCaption:

```
// Create a default HTMLTableCaption object and set the caption text.
HTMLTableCaption caption = new HTMLTableCaption();
caption.setElement("Customer Account Balances - January 1, 2000");

// Add the table caption to an HTMLTable object (assume that the table already exists).
table.setCaption(caption);
```

»

HTMLTree classes

The HTMLTree class allows you to easily set up a hierarchical tree of HTML elements that you can use in HTML pages. This class provides methods to get and set various attributes of the tree, in addition to methods allowing you to:

- Get and set the HTTP Servlet Request
- Add an HTMLTreeElement or FileTreeElement to the tree
- Remove an HTMLTreeElement or FileTreeElement from the tree

The HTMLTree class uses other HTML classes that make it easier to create a hierarchical tree:

- HTMLTreeElement: Creates a tree element
- FileTreeElement: Creates a file tree element
- FileListElement: Creates a file list element

Examples

The following examples show different ways to use the HTMLTree classes:

- **Example:** Using the HTMLTree classes
- **Example:** Creating a traversable IFS tree

«

HTMLTreeElement class

The HTMLTreeElement class represents an hierarchical element within an HTMLTree or other HTMLTreeElements.

Many tree element attributes can be retrieved or updating using methods that are provided in the HTMLTreeElement class. Some of the actions you can do with these methods are:

- Get or set the visible text of the tree element
- Get or set the URL for the expanded and collapsed icon
- Set whether or not the tree element will be expanded

The following example creates an HTMLTreeElement object and displays the tag:

```
// Create an HTMLTree.
HTMLTree tree = new HTMLTree();

// Create parent HTMLTreeElement.
HTMLTreeElement parentElement = new HTMLTreeElement();
parentElement.setTextUrl(new HTMLHyperlink("http://myWebPage", "My Web Page"));


// Create HTMLTreeElement Child.
HTMLTreeElement childElement = new HTMLTreeElement();
childElement.setTextUrl(new HTMLHyperlink("http://anotherWebPage", "Another Web Page"));
parentElement.addElement(childElement);

// Add the tree element to the tree.
tree.addElement(parentElement);
System.out.println(tree.getTag());
```

The getTag() method in the above example generates HTML tags that look like those displayed below. The example below is for display purposes only:

- My Web Page

  - Another Web Page

FileTreeElement class

The FileTreeElement class represents the Integrated File System within an HTMLTree view.

Many tree element attributes can be retrieved or updating using methods that are provided in the HTMLTreeElement class. Some of the actions you can do with these methods are:

- Get or set the URL for the expanded and collapsed icon
- Set whether or not the tree element will be expanded

The following example creates a FileTreeElement object and displays the tag:

```
// Create an HTMLTree.
HTMLTree tree = new HTMLTree();

// Create a URLParser object.
URLParser urlParser = new URLParser(httpServletRequest.getRequestURI());

// Create an AS400 object.
AS400 system = new AS400(mySystem, myUserId, myPassword);

// Create an IFS object.
IFSJavaFile root = new IFSJavaFile(system, "/QIBM");

// Create a DirFilter object and get the directories.
DirFilter filter = new DirFilter();
File[] dirList = root.listFiles(filter);

for (int i=0; i < dirList.length; i++)
{

    // Create a FileTreeElement.
    FileTreeElement node = new FileTreeElement(dirList[i]);

    // Set the Icon URL.
    ServletHyperlink sl = new ServletHyperlink(urlParser.getURI());
    sl.setHttpServletResponse(resp);
    element.setIconUrl(sl);

    // Add the FileTreeElement to the tree.
    tree.addElement(element);
}

System.out.println(tree.getTag());
```

The getTag() method above gives the output of the example:

+ ProdData
+ Test Folder
+ UserData
+ XML
+ include
+ locales

»

FileListElement class

The FileListElement class allows you to create a file list element, which represents the contents of an Integrated File System (IFS) directory.

You can use a FileListRenderer with a FileListElement object to specify how you want to display the list of files.

The FileListElement class provides methods that allow you to:

- List and sort the elements of the file list
- Get and set the HTTP Servlet Request
- Get and set the FileListRenderer
- Get and set the HTMLTable with which to display the file list

You can use this class with the FileTreeElement class to create a traversable list of IFS files.

The FileListElement javadoc shows how to create and display a FileListElement object.

Examples

The following example shows how you can use the FileListElement class with HTMLTree classes (FileTreeElement and HTMLTreeElement) to create a traversable IFS tree:

- **Example:** Creating a traversable IFS tree«

IBM Toolbox for Java beans

JavaBeans(TM) are reuseable software components that are written in Java. The component is a piece of program code that provides a well-defined, functional unit, which can be as small as a label for a button on a window or as large as an entire application.

JavaBeans can be either visual or nonvisual components. Non-visual JavaBeans still have a visual representation, such as an icon or a name, to allow visual manipulation.

All IBM Toolbox for Java public classes are also JavaBeans. These classes were built to Javasoft JavaBean standards; they function as reuseable components. The properties and methods for an IBM Toolbox for Java bean are the same as the properties and methods of the class.

JavaBeans can be used within an application program or they can be visually manipulated in builder tools, such as the IBM VisualAge for Java product.

Examples

- Example: The IBM Toolbox for Java bean example show one way to use JavaBeans in your program.
- Example: The Visual bean builder code example shows one way to create a program from JavaBeans by using a visual bean builder such as IBM Visual Age for Java.

Example: IBM Toolbox for Java bean code

The following example creates an AS400 object and a CommandCall object, and then registers listeners on the objects. The listeners on the objects print a comment when the AS/400 system connects or disconnects and when the CommandCall object completes the running of a command.

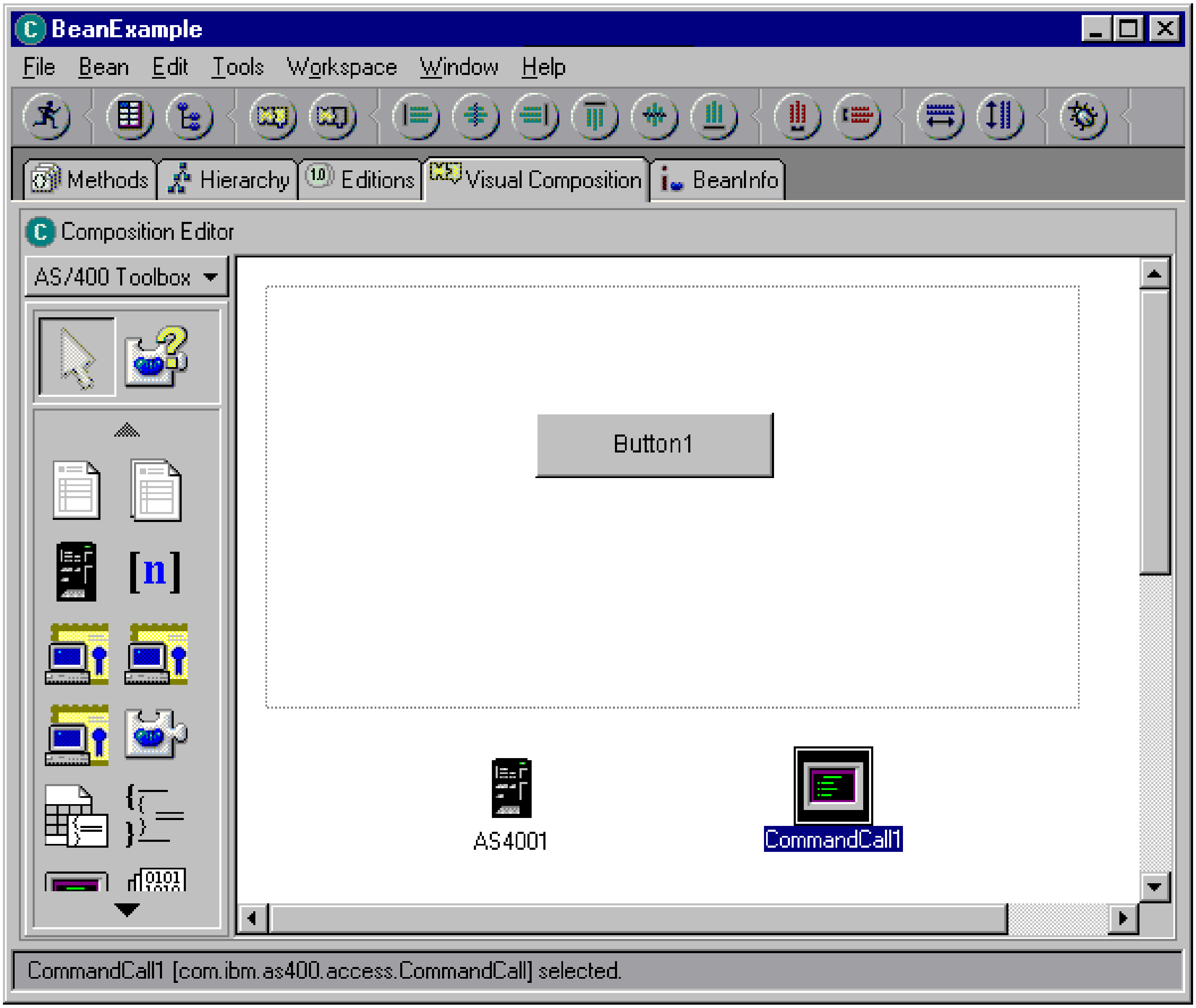
```
//////////////////////////////////////////////////////////////////////////
//
// Beans example.  This program uses the JavaBeans support in the
// IBM Toolbox for Java classes.
//
// Command syntax:
//    BeanExample
//
//////////////////////////////////////////////////////////////////////////
//
// This source is an example of JavaBeans in the IBM Toolbox for Java.
// IBM grants you a nonexclusive license to use this as an example from
// which you can generate a similar function tailored to your own
// specific needs.
//
// This sample code is provided by IBM for illustrative purposes only.
// These examples have not been thoroughly tested under all conditions.
// IBM, therefore, cannot guarantee or imply reliability, serviceability,
// or function of these programs.
//
// All programs contained herein are provided to you "AS IS" without any
// warranties of any kind. The implied warranties of merchantability and
// fitness for a particular purpose are expressly disclaimed.
//
// IBM Toolbox for Java
// (C) Copyright IBM Corp. 1997
// All rights reserved.
// US Government Users Restricted Rights -
// Use, duplication, or disclosure restricted
// by GSA ADP Schedule Contract with IBM Corp.
//
//////////////////////////////////////////////////////////////////////////


import com.ibm.as400.access.AS400;
import com.ibm.as400.access.CommandCall;
import com.ibm.as400.access.ConnectionListener;
import com.ibm.as400.access.ConnectionEvent;
import com.ibm.as400.access.ActionCompletedListener;
import com.ibm.as400.access.ActionCompletedEvent;

class BeanExample
{
    AS400       as400_  = new AS400();
    CommandCall cmd_    = new CommandCall( as400_ );

    BeanExample()
    {
        // Whenever the system is connected or disconnected print a
        // comment. Do this by adding a listener to the AS400 object.
        // When a system is connected or disconnected, the AS400 object
        // will call this code.

        as400_.addConnectionListener
        (new ConnectionListener()
          {
             public void connected(ConnectionEvent event)
             {
                 System.out.println( "System connected." );
             }
             public void disconnected(ConnectionEvent event)
             {
                 System.out.println( "System disconnected." );
             }
          }
        );

        // Whenever a command runs to completion print a comment. Do this
        // by adding a listener to the commandCall object. The commandCall
        // object will call this code when it runs a command.

        cmd_.addActionCompletedListener(
            new ActionCompletedListener()
            {
                public void actionCompleted(ActionCompletedEvent event)
                {
                    System.out.println( "Command completed." );
                }
            }
        );
    }

    void runCommand()
    {
        try
        {
            // Run a command. The listeners will print comments when the
            // system is connected and when the command has run to
            // completion.
            cmd_.run( "TESTCMD PARMS" );
        }
        catch (Exception ex)
        {
            System.out.println( ex );
        }
    }

    public static void main(String[] parameters)
    {
       BeanExample be = new BeanExample();

       be.runCommand();

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

Visual bean builder code example

This example uses the IBM VisualAge for Java Enterprise Edition V2.0 Composition Editor, but other visual bean builders are similar. This example creates an applet for a button that, when pressed, runs a command on an AS/400.

- Drag and drop a Button on the applet. (The Button can be found in the bean builder on the left side of the Visual Composition tab in the figure below.)
- Drop a CommandCall bean and an AS400 bean outside the applet. (The beans can be found in the bean builder on the left side of the Visual Composition tab in the figure below.)

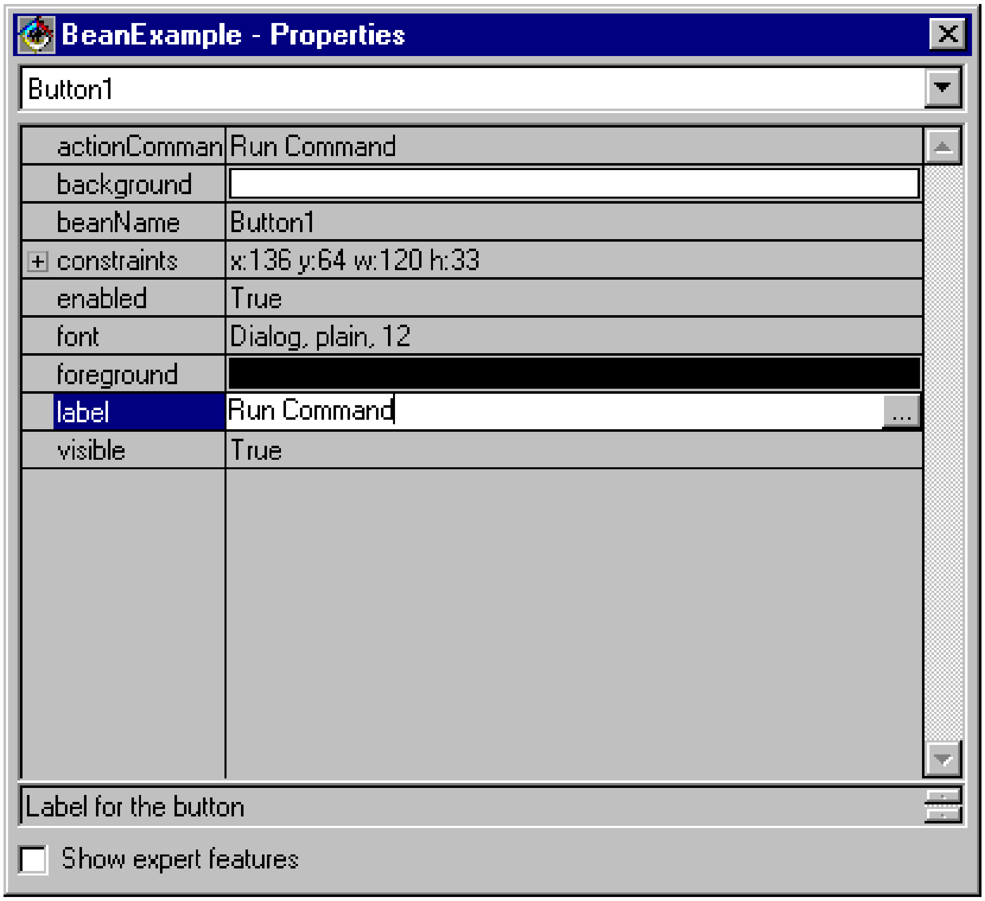
VisualAge Visual Composition Editor window - gui.BeanExample.



CommandCall1 [com.ibm.as400.access.CommandCall] selected.

- Edit the bean properties. (To edit, select the bean and then right-click to display a pop-up window, which has Properties as an option.)
  - Change the label of the Button to **Run command**, as shown in the figure below.

    Changing the label of the button to Run command.



BeanExample - Properties

Button1

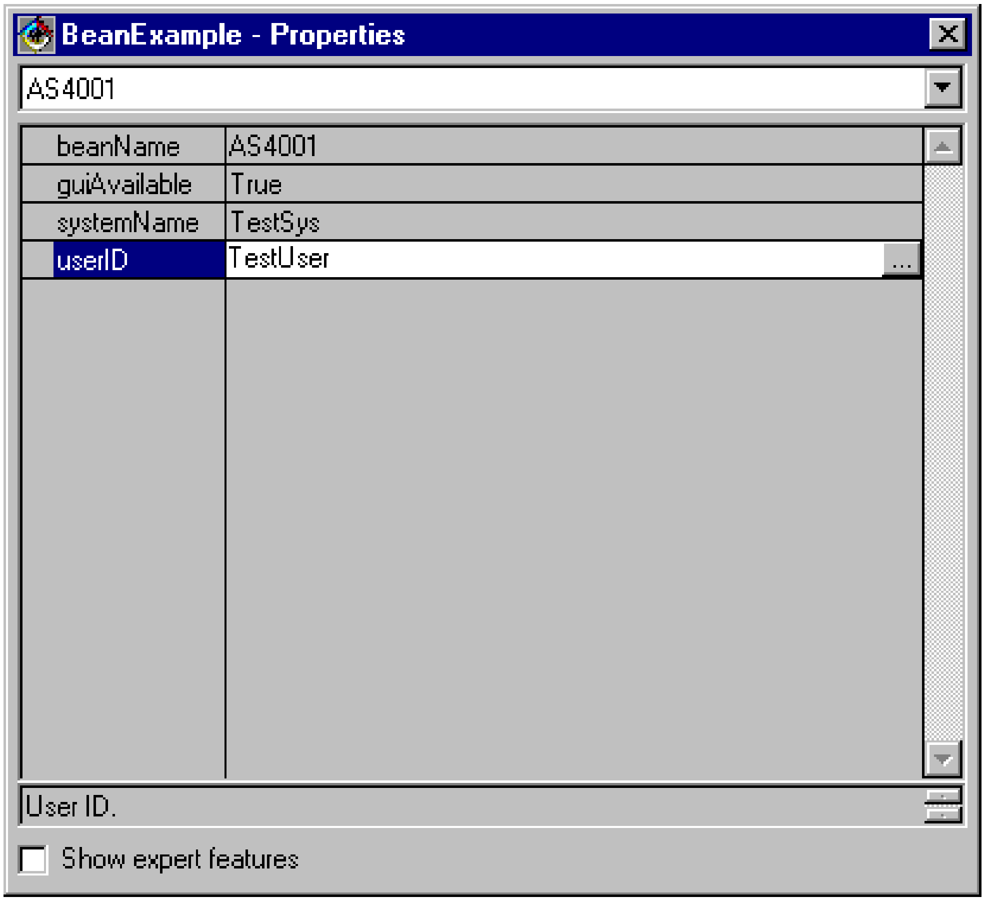
actionComman	Run Command
background	□
beanName	Button1
⊞ constraints	x:136 y:64 w:120 h:33
enabled	True
font	Dialog, plain, 12
foreground	■
label	Run Command
visible	True

Label for the button

□ Show expert features

- Change the system name of the AS400 bean to **TestSys**.
- Change the user ID of the AS400 bean to **TestUser**, as shown in the figure below.

Changing the name of the user ID to TestUser.



BeanExample - Properties

AS4001

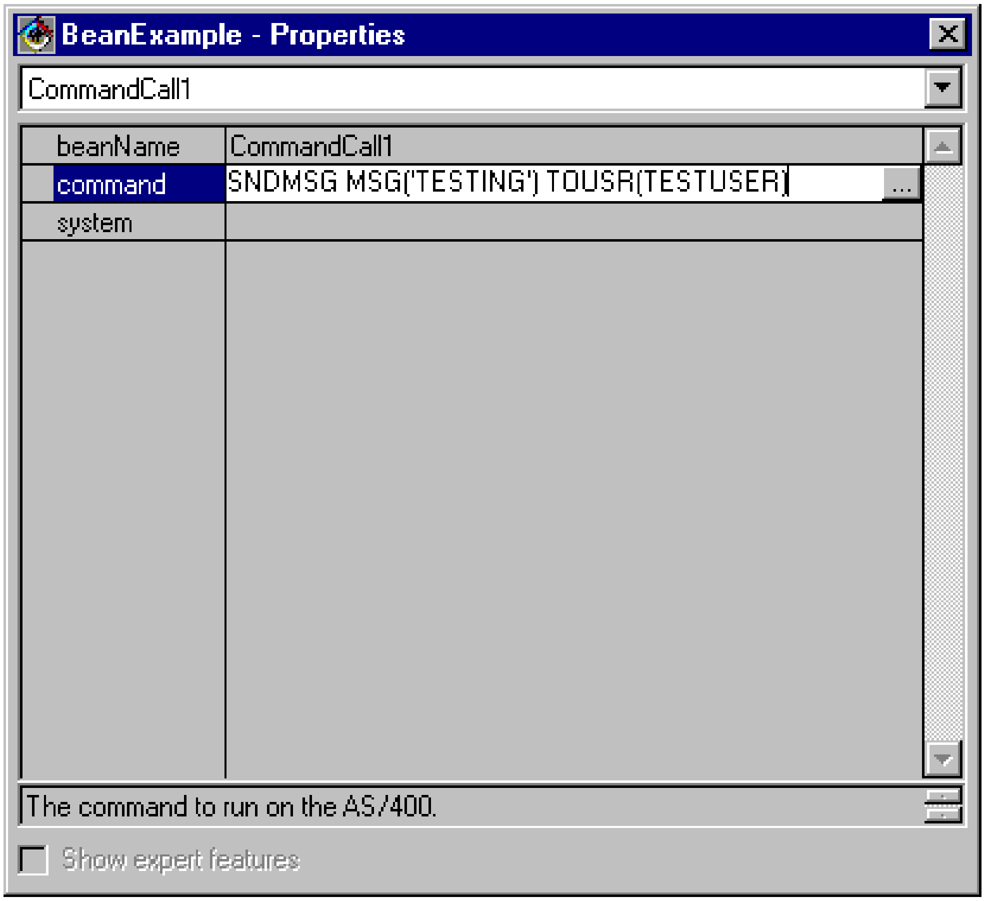
beanName	AS4001
guiAvailable	True
systemName	TestSys
userID	TestUser

User ID.

□ Show expert features

- Change the command of the CommandCall bean to **SNDMSG MSG('Testing') TOUSR('TESTUSER')**, as shown in the figure below.

Changing the command of the CommandCall bean.



BeanExample - Properties

CommandCall1

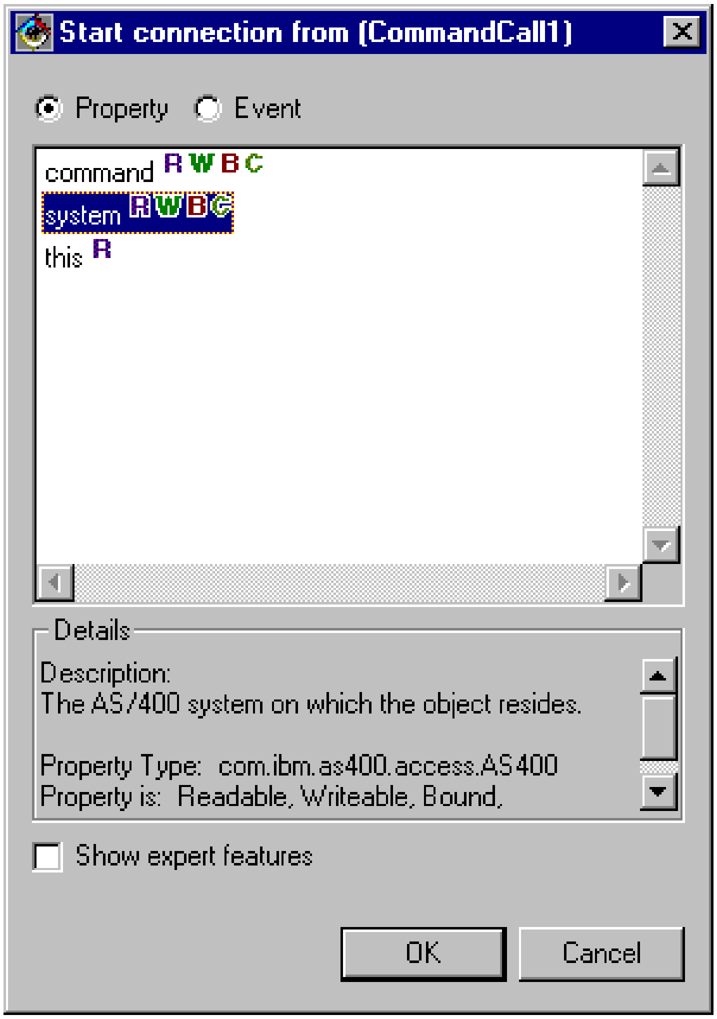
beanName	CommandCall1
command	SNDMSG MSG('TESTING') TOUSR(TESTUSER)
system	

The command to run on the AS/400.

□ Show expert features

- Connect the AS400 bean to the CommandCall bean. The method you use to do this varies between bean builders. For this example, do the following:
  - Select the CommandCall bean and then click the right mouse button
  - Select **Connect**
  - Select **Connectable Features**
  - Select **system** from the list of features as shown in the figure below.
  - Select the AS400 bean
  - Select **this** from the pop-up menu that appears over the AS400 bean

Connecting AS400 bean to CommandCall bean.

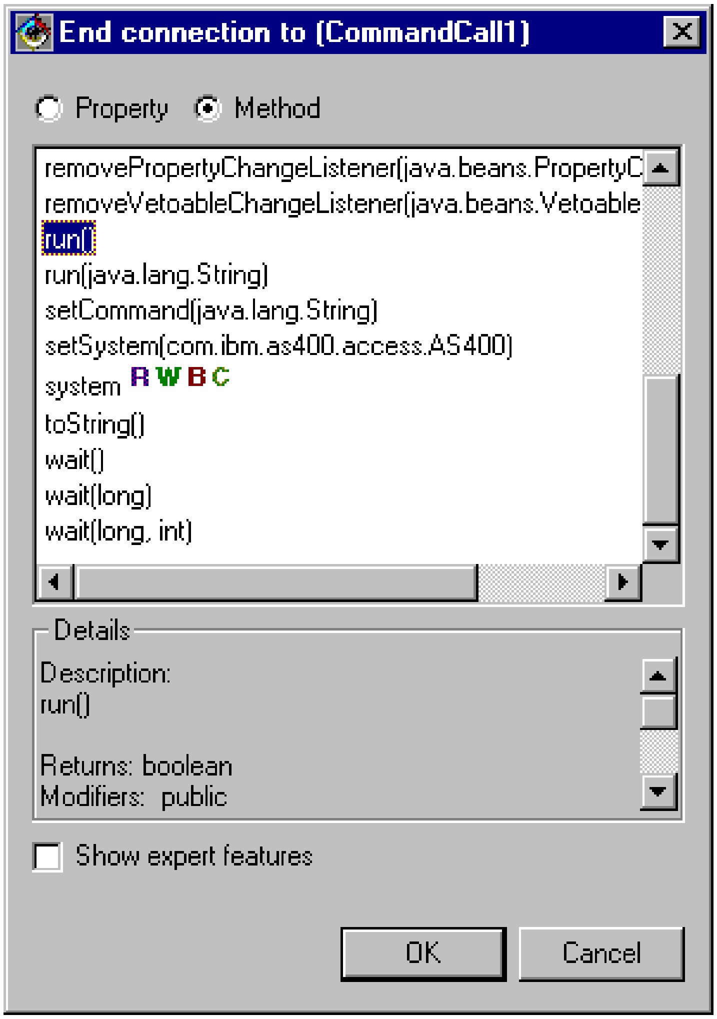


Description:
The AS/400 system on which the object resides.

Property Type: com.ibm.as400.access.AS400
Property is: Readable, Writeable, Bound.

- Connect the button to the CommandCall bean.
  - Select the Button bean and then click the right mouse button
  - Select **Connect**
  - Select **actionPerformed**
  - Select the CommandCall bean
  - Select **Connectable Features** from the pop-up menu that appears
  - Select **run()** from the list of methods as shown in the figure below.

Connecting a method to a button.



○ Property ● Method

```
removePropertyChangeListener(java.beans.PropertyC
removeVetoableChangeListener(java.beans.Vetoable
run()
run(java.lang.String)
setCommand(java.lang.String)
setSystem(com.ibm.as400.access.AS400)
system R W B C
toString()
wait()
wait(long)
wait(long, int)
```

Details

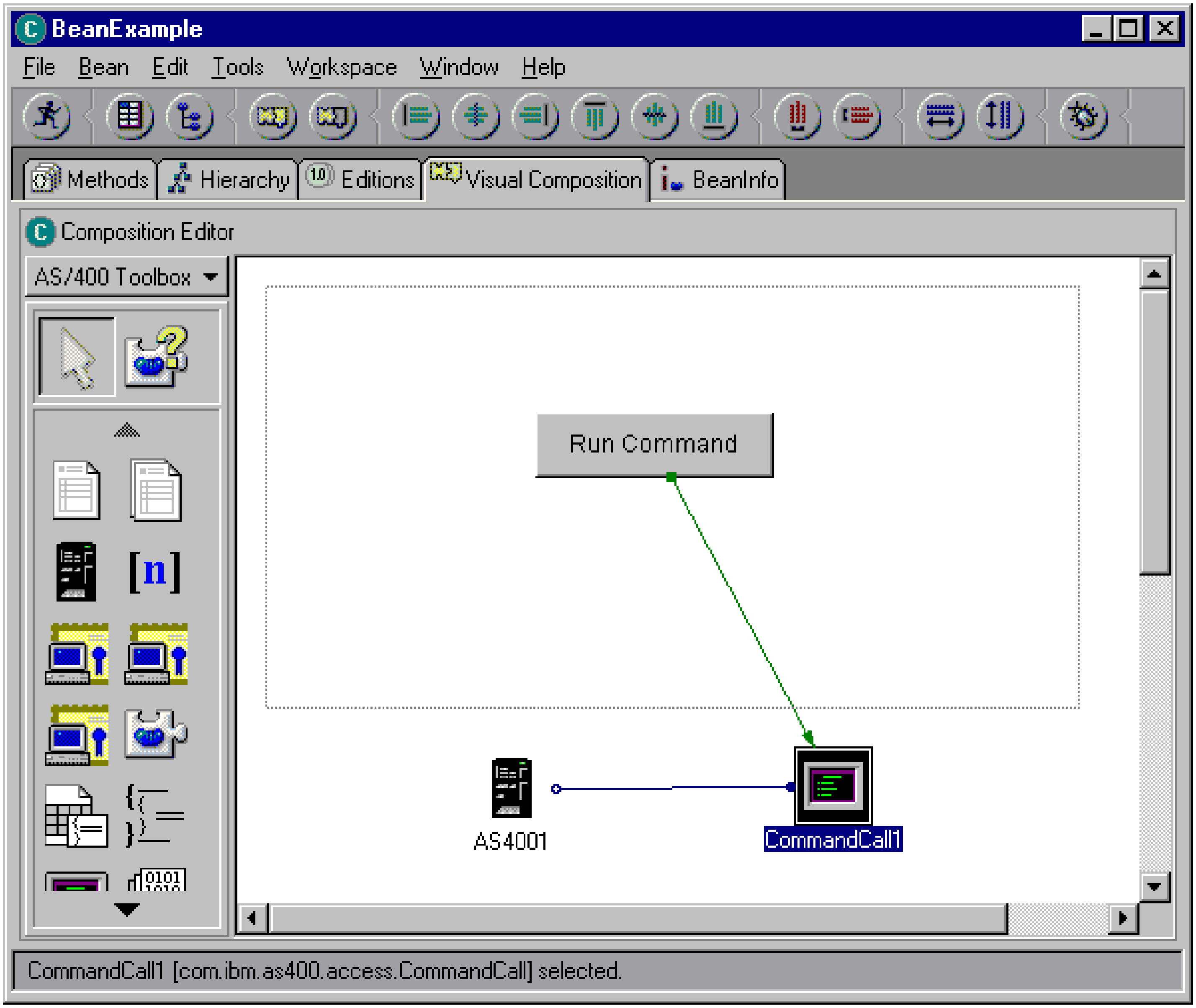
Description:
run()

Returns: boolean
Modifiers: public

□ Show expert features

When you are finished, the VisualAge Visual Composition Editor window should look like the figure below.

VisualAge Visual Composition Editor window - Finished bean example.



CommandCall1 [com.ibm.as400.access.CommandCall] selected.

Program Call Markup Language

Overview

Program Call Markup Language (PCML) is a tag language that helps you call AS/400 programs, with less Java code. PCML is based upon the Extensible Markup Language (XML), a tag syntax you use to describe the input and output parameters for AS/400 programs. PCML enables you to define tags that fully describe AS/400 programs called by your Java application. For more information about XML, see the XML reference section.

A huge benefit of PCML is that it allows you to write less code. Ordinarily, extra code is needed to connect, retrieve, and translate data between an AS/400 and IBM Toolbox for Java objects. However, by using PCML, your calls to the AS/400 with the IBM Toolbox for Java classes are automatically handled. PCML class objects are generated from the PCML tags and help minimize the amount of code you need to write in order to call AS/400 programs from your application.

Platform requirements

Although PCML was designed to support distributed program calls to AS/400 program objects from a Java platform, you can also use PCML to make calls to an AS/400 program from within an AS/400 environment as well.

Topics for more information

Refer to the following topics on how to use PCML:

- Call programs with the help of PCML
- Build program calls with PCML tags
- A PCML example

Building iSeries program calls with PCML

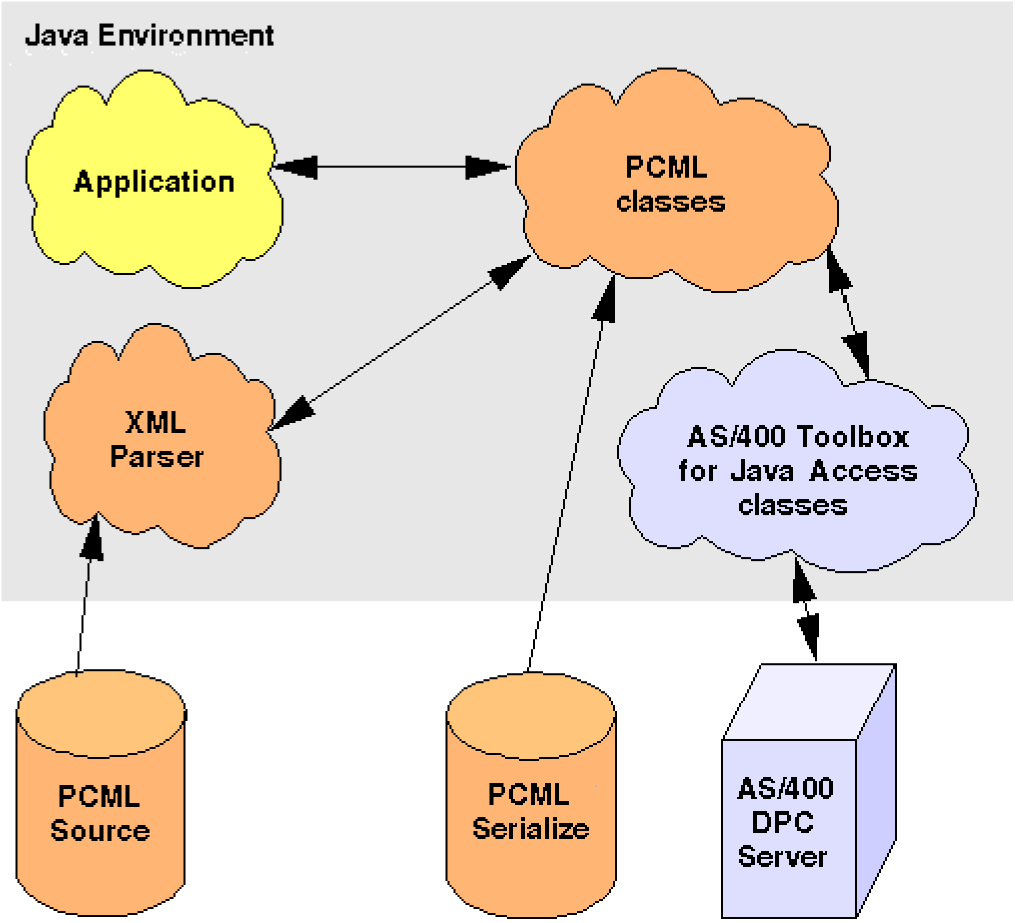
To build iSeries program calls with PCML, you must start by creating the following:

- Java application
- PCML source file

Depending upon your design process, you must write one or more PCML source files where you describe the interfaces to the iSeries programs that will be called by your Java application. Refer to PCML syntax for a detailed description of the language.

Then, your Java application, shown in yellow in Figure 1 below, interacts with the ProgramCallDocument class. The ProgramCallDocument class uses your PCML source file to pass information between your Java application and the iSeries programs.

Figure 1. Making program calls to the AS/400 using PCML.



When your application constructs the ProgramCallDocument object, the IBM XML parser reads and parses the PCML source file.

After the ProgramCallDocument class has been created, the application program uses the ProgramCallDocument class's methods to retrieve the necessary information from the server through the iSeries distributed program call (DPC) server.

To increase run-time performance, the ProgramCallDocument class can be serialized during your product build time. The ProgramCallDocument is then constructed using the serialized file. In this case, the IBM XML parser is not used at run-time. Refer to Using serialized PCML files.

Using PCML source files

Your Java application uses PCML by constructing a ProgramCallDocument object with a reference to the PCML source file. The ProgramCallDocument object considers the PCML source file to be a Java resource. Consequently, the PCML source file is found using the Java CLASSPATH.

The following Java code constructs a ProgramCallDocument object:

```
    AS400 as400 = new AS400();
    ProgramCallDocument pcmlDoc = new ProgramCallDocument(as400, "myPcmlDoc");
```

The ProgramCallDocument object will look for your PCML source in a file called `myPcmlDoc.pcml`. Notice that the `.pcml` extension is not specified on the constructor.

If you are developing a Java application in a Java "package," you can package-qualify the name of the PCML resource:

```
    AS400 as400 = new AS400();
    ProgramCallDocument pcmlDoc = new ProgramCallDocument(as400, "com.company.package.myPcmlDoc");
```

Using serialized PCML files

To increase run-time performance, you can use a serialized PCML file. A serialized PCML file contains serialized Java objects representing the PCML. The objects that are serialized are the same objects that are created when you construct the ProgramCallDocument from a source file as described above.

Using serialized PCML files gives you better performance because the IBM XML parser is not needed at run-time to process the PCML tags.

The PCML can be serialized using either of the following methods:

- From the command line:

  ```
  »java com.ibm.as400.data.ProgramCallDocument -serialize mypcml«
  ```

  This method is helpful for having batch processes to build your application.
- From within a Java program:

  ```
  ProgramCallDocument pcmlDoc; // Initialized elsewhere

  pcmlDoc.serialize();
  ```

If your PCML is in a source file named `myDoc.pcml`, the result of serialization is a file named `myDoc.pcml.ser`.

PCML source files vs. serialized PCML files

Consider the following code to construct a ProgramCallDocument:

```
    AS400 as400 = new AS400();
    ProgramCallDocument pcmlDoc = new ProgramCallDocument(as400, "com.mycompany.mypackage.myPcmlDoc");
```

The ProgramCallDocument constructor will first try to find a serialized PCML file named **`myPcmlDoc.pcml.ser`** in the **`com.mycompany.mypackage`** package in the Java CLASSPATH. If a serialized PCML file does not exist, the constructor will then try to find a PCML source file named **`myPcmlDoc.pcml`** in the **`com.mycompany.mypackage`** package in the Java CLASSPATH. If a PCML source file does not exist, an exception is thrown.

Qualified names

Your Java application uses the **ProgramCallDocument.setValue()** method to set input values for the iSeries program being called. Likewise, your application uses the **ProgramCallDocument.getValue()** method to retrieve output values from the iSeries program.

When accessing values from the ProgramCallDocument class, you must specify the fully qualified name of the document element or <**data**> tag. The qualified name is a concatenation of the names of all the containing tags with each name separated by a period.

For example, given the following PCML source, the qualified name for the **"nbrPolygons"** item is **"polytest.parm1.nbrPolygons"**. The qualified name for accessing the **"x"** value for one of the points in one of the polygons is **"polytest.parm1.polygon.point.x"**.

If any one of the elements needed to make the qualified name is unnamed, all descendants of that element do not have a qualified name. Any elements that do not have a qualified name cannot be accessed from your Java program.

```
<pcml version="1.0">
  <program name="polytest" path="/QSYS.lib/MYLIB.lib/POLYTEST.pgm">
    <!-- Parameter 1 contains a count of polygons along with an array of polygons -->
    <struct name="parm1" usage="inputoutput">
      <data name="nbrPolygons" type="int" length="4" init="5" />
      <!-- Each polygon contains a count of the number of points along with an array of points -->
      <struct name="polygon" count="nbrPolygons">
        <data name="nbrPoints" type="int" length="4" init="3" />
        <struct name="point" count="nbrPoints" >
          <data name="x" type="int" length="4" init="100" />
          <data name="y" type="int" length="4" init="200" />
        </struct>
      </struct>
    </struct>
  </program>
</pcml>
```

Accessing data in arrays

Any <**data**> or <**struct**> element can be defined as an array using the **count** attribute. Or, a <**data**> or <**struct**> element can be contained within another <**struct**> element that is defined as an array.

Furthermore, a <**data**> or <**struct**> element can be in a multidimensional array if more than one containing element has a **count** attribute specified.

In order for your application to set or get values defined as an array or defined within an array, you must specify the array index for each dimension of the array. The array indices are passed as an array of **int** values. Given the source for the array of polygons shown above, the following Java code can be used to retrieve the information about the polygons:

```
    ProgramCallDocument polytest; // Initialized elsewhere
    Integer nbrPolygons, nbrPoints, pointX, pointY;
    nbrPolygons = (Integer) polytest.getValue("polytest.parm1.nbrPolygons");
    System.out.println("Number of polygons:" + nbrPolygons);
    indices = new int[2];
    for (int polygon = 0; polygon < nbrPolygons.intValue(); polygon++)
    {
        indices[0] = polygon;
```

```
    nbrPoints = (Integer) polytest.getValue("polytest.parm1.polygon.nbrPoints", indices );
    System.out.println("  Number of points:" + nbrPoints);

    for (int point = 0; point < nbrPoints.intValue(); point++)
    {
        indices[1] = point;
        pointX = (Integer) polytest.getValue("polytest.parm1.polygon.point.x", indices );
        pointY = (Integer) polytest.getValue("polytest.parm1.polygon.point.y", indices );
        System.out.println("    X:" + pointX + " Y:" + pointY);
    }
}
```

Debugging

When you use PCML to call programs with complex data structures, it is easy to have errors in your PCML that result in exceptions from the ProgramCallDocument class. If the errors are related to incorrectly describing offsets and lengths of data, the exceptions can be difficult to debug.

The com.ibm.as400.data.PcmlMessageLog class allows you to turn on a tracing function that prints to the standard output stream information that can be helpful in problem determination. You can call the following method to turn the tracing function on:

```
com.ibm.as400.data.PcmlMessageLog.setTraceEnabled(true);
```

When the tracing function is turned on, the following types of information are printed to the standard output stream:

- A dump of the hexadecimal data being transferred between the Java application and the iSeries program. This shows the program input parameters after character data is converted to EBCDIC and integers are converted to big-endian. It also shows the output parameters before they are converted to the Java environment.

  The data is shown in a typical hexadecimal dump format with hexadecimal digits on the left and a character interpretation on the right. The following is an example of this dump format:

```
qgyolobj[6]
Offset : 0....... 4....... 8....... C....... 0....... 4....... 8....... C.......   0...4...8...C...0...4...8...C...
     0 : 5CE4E2D9 D7D9C640 4040                                                   **USRPRF                         *
```

  In the above example, the dump shows the seventh parameter has 10 bytes of data set to "*USRPRF ".

- For output parameters, following the hexadecimal dump is a description of how the data has been interpreted for the document.

```
/QSYS.lib/QGY.lib/QGYOLOBJ.pgm[2]
Offset : 0....... 4....... 8....... C....... 0....... 4....... 8....... C.......   0...4...8...C...0...4...8...C...
     0 : 0000000A 0000000A 00000001 00000068 D7F0F9F9 F0F1F1F5 F1F4F2F6 F2F5F400  *................P09901151426254.*
    20 : 00000410 00000001 00000000 00000000 00000000 00000000 00000000 00000000  *................................*
    40 : 00000000 00000000 00000000 00000000                                      *................                *
Reading data -- Offset: 0   Length: 4   Name: "qgyolobj.listInfo.totalRcds"   Byte data: 0000000A
Reading data -- Offset: 4   Length: 4   Name: "qgyolobj.listInfo.rcdsReturned"    Byte data: 0000000A
Reading data -- Offset: 8   Length: 4   Name: "qgyolobj.listInfo.rqsHandle"   Byte data: 00000001
Reading data -- Offset: c   Length: 4   Name: "qgyolobj.listInfo.rcdLength"   Byte data: 00000068
Reading data -- Offset: 10  Length: 1   Name: "qgyolobj.listInfo.infoComplete"    Byte data: D7
Reading data -- Offset: 11  Length: 7   Name: "qgyolobj.listInfo.dateCreated" Byte data: F0F9F9F0F1F1F5
Reading data -- Offset: 18  Length: 6   Name: "qgyolobj.listInfo.timeCreated" Byte data: F1F4F2F6F2F5
Reading data -- Offset: 1e  Length: 1   Name: "qgyolobj.listInfo.listStatus"  Byte data: F4
Reading data -- Offset: 1f  Length: 1   Name: "qgyolobj.listInfo.[8]" Byte data: 00
Reading data -- Offset: 20  Length: 4   Name: "qgyolobj.listInfo.lengthOfInfo"    Byte data: 00000410
Reading data -- Offset: 24  Length: 4   Name: "qgyolobj.listInfo.firstRecord" Byte data: 00000001
Reading data -- Offset: 28  Length: 40  Name: "qgyolobj.listInfo.[11]"    Byte data:
00000000000000000000000000000000000000000000000000000000000000000000000000000000
```

  The above messages can be very helpful in diagnosing cases where the output data coming from the iSeries program does not match the PCML source. This can easily occur when you are using dynamic lengths and offsets.

PCML syntax

PCML consists of the following tags, each of which has its own attribute tags:

- The program tag begins and ends code that describes one program
- The struct tag defines a named structure which can be specified as an argument to a program or as a field within another named structure. A structure tag contains a data or a structure tag for each field in the structure.
- The data tag defines a field within a program or structure.

For example, below, the PCML syntax describes one program with one category of data and some isolated data.

```
<program>

    <struct>
        <data> </data>
    </struct>

    <data> </data>

</program>
```

The program tag

The program tag can be expanded with the following elements:

```
<program name="name"
    [ entrypoint="entry-point-name" ]
    [ path="path-name" ]
    [ parseorder="name-list" ] >
    [ returnvalue="{ void | integer }" ]
»   [ threadsafe="{ true | false }" ]«</program>
```

Attribute	Value	Description
entrypoint=	*entry-point-name*	Specifies the name of the entry point within a service program object that is the target of this program call.
name=	*name*	Specifies the name of the program.
path=	*path-name*	Specifies the path to the program object. The default value is to assume the program is in the QSYS library. The path must be a valid IFS path name to a *PGM or *SRVPGM object. If a *SRVPGM object is called, the entrypoint attribute must be specified to indicate the name of the entrypoint to be called. If the entrypoint attribute is not specified, the default value for this attribute is assumed to be a *PGM object from the QSYS library. If the entrypoint attribute is specified, the default value for this attribute is assumed to be a *SRVPGM object in the QSYS library. The path name should be specified as all uppercase characters.
parseorder=	*name-list*	Specifies the order in which output parameters will be processed. The value specified is a blank separated list of parameter names in the order in which the parameters are to be processed. The names in the list must be identical to the names specified on the **name** attribute of tags belonging to the <**program**>. The default value is to process output parameters in the order the tags appear in the document. Some programs return information in one parameter that describes information in a previous parameter. For example, assume a program returns an array of structures in the first parameter and the number of entries in the array in the second parameter. In this case, the second parameter must be processed in order for the ProgramCallDocument to determine the number of structures to process in the first parameter.
returnvalue=	*void* The program does not return a value. *integer* The program returns a 4-byte signed integer.	Specifies the type of value, if any, that is returned from a service program call. This attribute is not allowed for *PGM object calls.
»**threadsafe=**	*true* The program is considered to be thread-safe. *false* The program is not thread-safe.	When you call a Java program and an iSeries program that are on the same server, use this property to specify whether you want to call the iSeries program in the same job and on the same thread as the Java program. If you know your program is thread-safe, setting the property to *true* results in better performance. To keep the environment safe, the default is to call programs in separate server jobs. The default value is *false*.«

The struct tag

The structure tag can be expanded with the following elements:

```
<struct name="name"
    [ count="{number | data-name }"]
    [ maxvrm="version-string" ]
    [ minvrm="version-string" ]
    [ offset="{number | data-name }" ]
    [ offsetfrom="{number | data-name | struct-name }" ]
    [ outputsize="{number | data-name }" ]
    [ usage="{ inherit | input | output | inputoutput }" ]>
</struct>
```

Attribute	Value	Description
name=	*name*	Specifies the name of the **<struct> element**
count=	*number* where *number* defines a fixed, never-changing sized array. *data-name* where *data-name* defines the name of a **<data>** element within the PCML document that will contain, at runtime, the number of elements in the array. The *data-name* specified can be a fully qualified name or a name that is relative to the current element. In either case, the name must reference a **<data>** element that is defined with **type=***"int"*. See Resolving Relative Names for more information on how relative names are resolved.	Specifies that the element is an array and identifies the number of entries in the array. If this attribute is omitted, the element is not defined as an array, although it may be contained within another element that is defined as an array.
maxvrm=	*version-string*	Specifies the highest AS/400 version on which the element exists. If the AS/400 version is greater than the version specified on the attribute, the element and its children, if any exist, will not be processed during a call to a program. The **maxvrm** element is helpful for defining program interfaces which differ between releases of AS/400. The syntax of the version string must be "VvRrMm," where the capitals letters "V," "R," and "M" are literal characters and "v," "r," and "m" are one or more digits representing the version, release and modification level, respectively. The value for "v" must be from 1 to 255 inclusively. The value for "r" and "m" must be from 0 to 255, inclusively.
minvrm=	*version-string*	Specifies the lowest AS/400 version on which this element exists. If the AS/400 version is less than the version specified on this attribute, this element and its children, if any exist, will not be processed during a call to a program. This attribute is helpful for defining program interfaces which differ between releases of AS/400. The syntax of the version string must be "VvRrMm," where the capitals letters "V," "R," and "M" are literal characters and "v," "r," and "m" are one or more digits representing the version, release and modification level, respectively. The value for "v" must be from 1 to 255, inclusively. The value for "r" and "m" must be from 0 to 255, inclusively.

offset=	*number* where *number* defines a fixed, never-changing offset. *data-name* where *data-name*defines the name of a **<data>** element within the PCML document that will contain, at runtime, the offset to the element. The **data-name** specified can be a fully qualified name or a name that is relative to the current element. In either case, the name must reference a **<data>** element that is defined with **type=***"int"*. See Resolving Relative Names for more information on how relative names are resolved.	Specifies the offset to the **<struct>** element within an output parameter. Some programs return information with a fixed structure followed by one or more variable length fields or structures. In this case, the location of a variable length element is usually specified as an offset or displacement within the parameter. The **offset** attribute is used to describe the offset to this **<struct>** element. **Offset** is used in conjunction with the **offsetfrom** attribute. If the **offsetfrom** attribute is not specified, the base location for the offset specified on the **offset** attribute is the parent of the element. See Specifying Offsets for more information on how to use the **offset** and **offsetfrom** attributes. The **offset** and **offsetfrom** attributes are only used to process output data from a program. These attributes do not control the offset or displacement of input data. If the attribute is omitted, the location of the data for the element is immediately following the preceding element in the parameter, if any.
offsetfrom=	*number* where *number* defines a fixed, never-changing base location. A *number* attribute is most typically used to specify **number=***"0"* indicating that the offset is an absolute offset from the beginning of the parameter. *data-name* where *data-name* defines the name of a **<data>** element to be used as a base location for the offset. The element name specified must be the parent or an ancestor of this element. The value from the **offset** attribute will be relative to the location of the element specified on this attribute. The *data-name* specified can be a fully qualified name or a name that is relative to the current element. In either case, the name must reference an ancestor of this element. See Resolving Relative Names for more information on how relative names are resolved. *struct-name* where *struct-name* defines the name of a **<struct>** element to be used as a base location for the offset. The element name specified must be the parent or an ancestor of this element. The value from the **offset** attribute will be relative to the location of the element specified on this attribute. The *struct-name* specified can be a fully qualified name or a name that is relative to the current element. In either case, the name must reference an ancestor of this element. See Resolving Relative Names for more information on how relative names are resolved.	Specifies the base location from which the **offset** attribute is relative. If the **offsetfrom** attribute is not specified, the base location for the offset specified on the **offset** attribute is the parent of this element. See Specifying Offsets for more information on how to use the **offset** and **offsetfrom** attributes. The **offset** and **offsetfrom** attributes are only used to process output data from a program. These attributes do not control the offset or displacement of input data.
outputsize=	*number* where *number* defines a fixed,never-changing number of bytes to reserve. *data-name* where *data-name* defines the name of a **<data>** element within the PCML document that will contain, at runtime, the number of bytes to reserve for output data. The *data-name* specified can be a fully qualified name or a name that is relative to the current element. In either case, the name must reference a **<data>** element that is defined with **type=***"int"*. See Resolving Relative Names for more information on how relative names are resolved.	Specifies the number of bytes to reserve for output data for the element. For output parameters which are variable in length, the **outputsize** attribute is needed to specify how many bytes should be reserved for data to be returned from the AS/400 program. **Outputsize** can be specified on all variable length fields and variable sized arrays, or it can be specified for an entire parameter that contains one or more variable length fields. **Outputsize** is not necessary and should not be specified for fixed-size output parameters. The value specified on the attribute is used as the total size for the element including all children of the element. Therefore, the **outputsize** attribute is ignored on any children or descendants of the element.

		If the attribute is omitted, the number of bytes to reserve for output data is determined at runtime by adding the number of bytes to reserve for all of the children of the **<struct>** element.
usage=	*inherit*	Usage is inherited from the parent element. If the structure does not have a parent, usage is assumed to be **inputoutput**.
	input	The structure is an input value to the host program. For character and numeric types, the appropriate conversion is performed.
	output	The structure is an output value from the host program. For character and numeric types, the appropriate conversion is performed.
	inputoutput	The structure is both and input and an output value.

Resolving relative names

Several attributes allow you to specify the name of another element, or tag, within the document as the attribute value. The name specified can be a name that is relative to the current tag.

Names are resolved by seeing if the name can be resolved as a child or descendent of the tag containing the current tag. If the name cannot be resolved at this level, the search continues with the next highest containing tag. This resolution must eventually result in a match of a tag that is contained by the <**pcml**> tag, in which case the name is considered to be an absolute name, not a relative name.

```
<pcml version="1.0">
  <program name="polytest" path="/QSYS.lib/MYLIB.lib/POLYTEST.pgm">
    <!-- Parameter 1 contains a count of polygoins along with an array of polygons -->
    <struct name="parm1" usage="inputoutput">
      <data name="nbrPolygons" type="int" length="4" init="5" />
      <!-- Each polygon contains a count of the number of points along with an array of points -->
      <struct name="polygon" count="nbrPolygons">
        <data name="nbrPoints" type="int" length="4" init="3" />
        <struct name="point" count="nbrPoints" >
          <data name="x" type="int" length="4" init="100" />
          <data name="y" type="int" length="4" init="200" />
        </struct>
      </struct>
    </struct>
  </program>
</pcml>
```

Specifying offsets

Some programs return information with a fixed structure followed by one or more variable length fields or structures. In this case, the location of a variable length element is usually specified as an offset or displacement within the parameter.

An offset is the distance in bytes from a the beginning of the parameters to the beginning of a field or structure. A displacement is the distance in bytes from the beginning of one structure to the beginning of another structure.

For offsets, since the distance is from the beginning of the parameter, you should specify **offsetfrom="0"**. The following is an example of an offset from the beginning of the parameter:

```
<pcml version="1.0">
  <program name="myprog" path="/QSYS.lib/MYLIB.lib/MYPROG.pgm">
    <!-- receiver variable contains a path -->
    <struct name="reciever" usage="output" outputsize="2048">
      <data name="pathType"         type="int"  length="4" />
      <data name="offsetToPathName" type="int"  length="4" />
      <data name="lengthOfPathName" type="int"  length="4" />
      <data name="pathName"         type="char" length="lengthOfPathName"
               offset="offsetToPathName"  offsetfrom="0"/>
    </struct>
  </program>
</pcml>
```

For displacements, since the distance is from the beginning of another structure, you specify the name of the structure to which the offset is relative. The following is an example of an displacement from the beginning of a named structure:

```
<pcml ="1.0">
  <program name="myprog" path="/QSYS.lib/MYLIB.lib/MYPROG.pgm">
    <!-- receiver variable contains an object -->
    <struct name="reciever" usage="output" >
      <data name="objectName"       type="char"  length="10" />
      <data name="libraryName"      type="char"  length="10" />
      <data name="objectType"       type="char"  length="10" />
      <struct name="pathInfo" usage="output" outputsize="2048" >
        <data name="pathType"         type="int"  length="4" />
        <data name="offsetToPathName" type="int"  length="4" />
        <data name="lengthOfPathName" type="int"  length="4" />
        <data name="pathName"         type="char" length="lengthOfPathName"
                 offset="offsetToPathName"  offsetfrom="pathInfo"/>
      </struct>
    </struct>
  </program>
</pcml>
```

The data tag

The data tag can have the following attributes. Attributes enclosed in brackets, [], indicate that the attribute is optional. If you specify an optional attribute, do not include the brackets in your source. Some attribute values are shown as a list of choices enclosed in braces, {}, with possible choices separated by vertical bars, |. When you specify one of these attributes, do not include the braces in your source and only specify one of the choices shown.

```
<data type="{ char | int | packed | zoned | float | byte | struct }"
»    [ bidistringtype="{ ST4 | ST5 | ST6 | ST7 | ST8 | ST9 | ST10 | ST11 | DEFAULT }"]«
    [ ccsid="{ number | data-name }" ]
    [ count="{ number | data-name }" ]
    [ init="string" ]
    [ length="{ number | data-name }" ]
    [ maxvrm="version-string" ]
    [ minvrm="version-string" ]
    [ name="name" ]
    [ offset="{ number | data-name }" ]
    [ offsetfrom="{ number | data-name | struct-name }" ]
    [ outputsize="{ number | data-name | struct-name }" ]
    [ passby= "{ reference | value }" ]
    [ precision="number" ]
    [ struct="struct-name" ]
    [ usage="{ inherit | input | output | inputoutput }" ]>
</data>
```

Attribute	Value	Description
type=	*char* where *char* indicates a character value. The **length** attribute specifies the number of bytes of data which may be different than the number of characters. A *char* data value is returned as a *java.lang.String*. *int* where *int* is an integer value. The **length** attribute specifies the number of bytes, "2" or "4". The **precision** attribute specifies the number of bits of precision. For example: **length=***"2"* **precision=***"15"* Specifies a 16-bit signed integer. An *int* data value with these specifications is returned as a *java.lang.Short*. **length=***"2"* **precision=***"16"* Specifies a 16-bit unsigned integer. An *int* data value with these specifications is returned as a *java.lang.Integer*. **length=***"4"* **precision=***"31"* Specifies a 32-bit signed integer. An *int* data value with these specifications is returned as a *java.lang.Integer*. **length=***"4"* **precision=***"32"* Specifies a 32-bit unsigned integer. An *int* data value is returned as a *java.lang.Long*. »**length=***"8"* **precision=***"63"* Specifies a 64-bit signed integer. An *int* data value is returned as a *java.lang.Long*.« For **length=***"2"*, the default precision is "15". For **length=***"4"*, the default precision is "31". *packed* where *packed* is a packed decimal value. The **length** attribute specifies the number of digits. The **precision** attribute specifies the number of decimal positions. A	Indicates the type of data being used (character, integer, packed, zoned, floating point, byte, or struct).

	packed data value is returned as a *java.math.BigDecimal*. *zoned* where *zoned* is a zoned decimal value. The **length** attribute specifies the number of digits. The **precision** attribute specifies the number of decimal positions. A *zoned* data value is returned as a *java.math.BigDecimal*. *float* where *float* is a floating point value. The **length** attribute specifies the number of bytes, "4" or "8". For **length=***"4"*, the *float* data value is returned as a *java.lang.Float*. For **length=***"8"*, the *float* data value is returned as a *java.lang.Double*. *byte* where *byte* is a byte value. The **length** attribute specifies the number of bytes. No conversion is performed on the data. A *byte* data value is returned as an array of *byte* values (*byte[]*). *struct* where *struct* specifies the name of the **<struct>** element. A *struct* allows you to define a structure once and reuse it multiple times within the document. When **type=***"struct"*, it is as if the structure specified appeared at this location in the document.	
»**bidistringtype=**	*DEFAULT* where DEFAULT is the default string type for non-bidirectional data (LTR). *ST4* where *ST4* is String Type 4. *ST5* where *ST5* is String Type 5. *ST6* where *ST6* is String Type 6. *ST7* where *ST7* is String Type 7. *ST8* where *ST8* is String Type 8. *ST9* where *ST9* is String Type 9. *ST10* where *ST10* is String Type 10. *ST11* where *ST11* is String Type 11.	Specifies the bidirectional string type for **<data>** elements with **type=***"char"*. If this attribute is omitted, string type for this element is implied by the CCSID (whether explicitly specified or the default CCSID of the host environment). String types are defined in the javadoc for the BidiStringType class.«
ccsid=	*number* where *number* defines a fixed, never-changing CCSID. *data-name* where *data-name* defines the name that will contain, at runtime, the CCSID of the character data. The *data-name* specified can be a fully qualified name or a name that is relative to the current element. In either case, the name must reference a **<data>** element that is defined with **type=***"int"*. See Resolving Relative Names for more information on how relative names are resolved.	Specifies the host Coded Character Set ID (CCSID) for character data for the **<data>** element. The **ccsid** attribute can be specified only for **<data>** elements with **type=***"char"*. If this attribute is omitted, character data for this element is assumed to be in the default CCSID of the host environment.

<table>
<tr>
<td><b>count=</b></td>
<td><i>number</i>
where <i>number</i> defines a fixed, never-changing number of elements in a sized array.

<i>data-name</i>
where <i>data-name</i> defines the name of a <b>&lt;data&gt;</b> element within the PCML document that will contain, at runtime, the number of elements in the array. The <i>data-name</i> specified can be a fully qualified name or a name that is relative to the current element. In either case, the name must reference a <b>&lt;data&gt;</b> element that is defined with <b>type=</b><i>"int"</i>. See Resolving Relative Names for more information on how relative names are resolved.</td>
<td>Specifies that the element is an array and identifies the number of entries in the array.

If the <i>count</i> attribute is omitted, the element is not defined as an array, although it may be contained within another element that is defined as an array.</td>
</tr>
<tr>
<td><b>init=</b></td>
<td><i>string</i></td>
<td>Specifies an initial value for the <b>&lt;data&gt;</b> element. The <i>init</i> value is used if an initial value is not explicitly set by the application program when <b>&lt;data&gt;</b> elements with <b>usage=</b><i>"input"</i> or <b>usage=</b><i>"inputoutput"</i> are used.

The initial value specified is used to initialize scalar values. If the element is defined as an array or is contained within a structure defined as an array, the initial value specified is used as an initial value for all entries in the array.</td>
</tr>
<tr>
<td><b>length=</b></td>
<td><i>number</i>
where <i>number</i> defines a fixed, never-changing length.

<i>data-name</i>
where <i>data-name</i> defines the name of a <b>&lt;data&gt;</b> element within the PCML document that will contain, at runtime, the length. A <i>data-name</i> can be specified only for <b>&lt;data&gt;</b> elements with <b>type=</b><i>"char"</i> or <b>type=</b><i>"byte"</i>. The <i>data-name</i> specified can be a fully qualified name or a name that is relative to the current element. In either case, the name must reference a <b>&lt;data&gt;</b> element that is defined with <b>type=</b><i>"int"</i>. See Resolving Relative Names for more information on how relative names are resolved.</td>
<td>Specifies the length of the data element. Usage of this attribute varies depending on the data type.

<table>
<tr><th>Data Type</th><th>Description</th></tr>
<tr><td><b>type=</b><i>"char"</i></td><td>The <b>length</b> attribute specifies the number of bytes, of data for this element. Note that this is not necessarily the number of characters. A literal <i>number</i> or <i>data-name</i> must be specified.</td></tr>
<tr><td><b>type=</b><i>"int"</i></td><td>The <b>length</b> attribute specifies the number of bytes of data for this element: "2", "4", or »"8". « The <b>precision</b> attribute is used to specify the number of bits of precision and indicates whether the integer is signed or unsigned. A literal <i>number</i> must be specified.</td></tr>
<tr><td><b>type=</b><i>"packed"</i></td><td>The <b>length</b> attribute specifies the number of numeric digits of data for this element. The <b>precision</b> attribute is used to specify the number of decimal digits. A literal <i>number</i> must be specified.</td></tr>
<tr><td><b>type=</b><i>"zoned"</i></td><td>The <b>length</b> attribute specifies the number of numeric digits of data for this element. The precision attribute is used to specify the number of decimal digits. A literal <i>number</i> must be specified.</td></tr>
<tr><td><b>type=</b><i>"float"</i></td><td>The <b>length</b> attribute specifies the number of bytes, 4 or 8, of data for this element. A literal <i>number</i> must be specified.</td></tr>
<tr><td><b>type=</b><i>"byte"</i></td><td>The <b>length</b> attribute specifies the number of bytes of data for this element. A literal <i>number</i> or <i>data-name</i> must be specified.</td></tr>
<tr><td><b>type=</b><i>"struct"</i></td><td>The <b>length</b> attribute is not allowed.</td></tr>
</table>
</td>
</tr>
<tr>
<td><b>maxvrm=</b></td>
<td><i>version-string</i></td>
<td>Specifies the highest version of iSeries on which this element exists. If the iSeries version is greater than the version specified on this attribute, this element and its children, if any exist, will not be processed during a call to a program. This attribute is helpful for defining program interfaces which differ between releases of iSeries.

The syntax of the version string must be "VvRrMm", where the capitals letters "V," "R," and "M" are literal characters and "v," "r," and "m" are one or more digits representing the version, release and modification level, respectively. The value for "v" must be from 1 to 255 inclusively. The value for "r" and "m" must be from 0 to 255, inclusively.</td>
</tr>
</table>

minvrm=	*version-string*	Specifies the lowest version of iSeries on which this element exists. If the iSeries version is less than the version specified on this attribute, this element and its children, if any exist, will not be processed during a call to a program. This attribute is helpful for defining program interfaces which differ between releases of iSeries. The syntax of the version string must be "VvRrMm," where the capitals letters "V," "R," and "M" are literal characters and "v," "r," and "m" are one or more digits representing the version, release and modification level, respectively. The value for "v" must be from 1 to 255 inclusively. The value for "r" and "m" must be from 0 to 255, inclusively.
name=	*name*	Specifies the name of the **<data>** element.
offset=	*number* where *number* defines a fixed, never-changing offset. *data-name* where *data-name* defines the name of a **<data>** element within the PCML document that will contain, at runtime, the offset to this element. The *data-name* specified can be a fully qualified name or a name that is relative to the current element. In either case, the name must reference a **<data>** element that is defined with **type=***"int"*. See Resolving Relative Names for more information on how relative names are resolved.	Specifies the offset to the **<data>** element within an output parameter. Some programs return information with a fixed structure followed by one or more variable length fields or structures. In this case, the location of a variable length element is usually specified as an offset or displacement within the parameter. An **offset** attribute is used in conjunction with the **offsetfrom** attribute. If the **offsetfrom** attribute is not specified, the base location for the offset specified on the **offset** attribute is the parent of this element. See Specifying Offsets for more information on how to use the **offset** and **offsetfrom** attributes. The **offset** and **offsetfrom** attributes are only used to process output data from a program. These attributes do not control the offset or displacement of input data. If this attribute is omitted, the location of the data for this element is immediately following the preceding element in the parameter, if any.
offsetfrom=	*number* where *number* defines a fixed, never-changing base location. *Number* is most typically used to specify **number=***"0"* indicating that the offset is an absolute offset from the beginning of the parameter. *data-name* where *data-name* defines the name of a **<data>** element used as a base location for the offset. The element name specified must be the parent or an ancestor of this element. The value from the **offset** attribute will be relative to the location of the element specified on this attribute. The *data-name* specified can be a fully qualified name or a name that is relative to the current element. In either case, the name must reference an ancestor of this element. See Resolving Relative Names for more information on how relative names are resolved. *struct-name* where *struct-name* defines the name of a **<struct>** element used as a base location for the offset. The element name specified must be the parent or an ancestor of this element. The value from the **offset** attribute will be relative to the location of the element specified on this attribute. The *struct-name* specified can be a fully qualified name or a name that is relative to the current element. In either case, the name must reference an ancestor of this element. See Resolving Relative Names for more information on how relative names are resolved.	Specifies the base location from which the **offset** attribute is relative. If the **offsetfrom** attribute is not specified, the base location for the offset specified on the **offset** attribute is the parent of this element. See Specifying Offsets for more information on how to use the **offset** and **offsetfrom** attributes. The **offset** and **offsetfrom** attributes are only used to process output data from a program. These attributes do not control the offset or displacement of input data.

outputsize=	*number* where a *number* defines a fixed, never-changing number of bytes to reserve. *data-name* where *data-name* defines the name of a **<data>** element within the PCML document that will contain, at runtime, the number of bytes to reserve for output data. The *data-name* specified can be a fully qualified name or a name that is relative to the current element. In either case, the name must reference a **<data>** element that is defined with **type=***"int"*. See Resolving Relative Names for more information on how relative names are resolved.	Specifies the number of bytes to reserve for output data for the element. For output parameters which are variable in length, the **outputsize** attribute is needed to specify how many bytes should be reserved for data to be returned from the iSeries program. An **outputsize** attribute can be specified on all variable length fields and variable sized arrays, or it can be specified for an entire parameter that contains one or more variable length fields. **Outputsize** is not necessary and should not be specified for fixed-size output parameters. The value specified on this attribute is used as the total size for the element including all the children of the element. Therefore, the **outputsize** attribute is ignored on any children or descendants of the element. If **outputsize** is omitted, the number of bytes to reserve for output data is determined at runtime by adding the number of bytes to reserve for all of the children of the **<struct>** element.
passby=	*reference* where *reference* indicates that the parameter will be passed by reference. When the program is called, the program will be passed a pointer to the parameter value. *value* where *value* indicates an integer value. This value is allowed only when **type=** *"int"* and **length=***"4"* is specified.	Specifies whether the parameter is passed by reference or passed by value. This attribute is allowed only when this element is a child of a **<program>** element defining a service program call.
precision=	*number*	Specifies the number of bytes of precision for some numeric data types. **Data Type** — **Description** **type=***"int"* **length=***"2"* — Use **precision=***"15"* for a signed 2-byte integer. Use **precision=***"16"* for an unsigned 2-byte integer. The default value is "15". **type=***"int"* **length=***"4"* — Use **precision=***"31"* for a signed 4-byte integer. Use **precision=***"32"* for an unsigned 4-byte integer. »**type=***"int"* **length=***"8"* — Use **precision=***"63"* for a signed 8-byte integer. « **type=***"zoned"* — The precision specifies the number of decimal digits. The number specified must be greater than or equal to zero and less than or equal to the total number of digits specified on the *length* attribute. **type=***"zoned"* — The precision specifies the number of decimal digits. The number specified must be greater than or equal to zero and less than or equal to the total number of digits specified on the *length* attribute.
struct=	*name*	Specifies the name of a **<struct>** element for the **<data>** element. A **struct** attribute can be specified only for **<data>** elements with **type=***"struct"*.
usage=	*inherit*	Usage is inherited from the parent element. If the structure does not have a parent, usage is assumed to be *inputoutput*.
	input	Defines an input value to the host program. For character and numeric types, the appropriate conversion is performed.
	output	Defines an output value from the host program. For character and numeric types, the appropriate conversion is performed.
	inputoutput	Defines both and input and an output value.

Resolving relative names

Several attributes allow you to specify the name of another element, or tag, within the document as the attribute value. The name specified can be a name that is relative to the current tag.

Names are resolved by seeing if the name can be resolved as a child or descendent of the tag containing the current tag. If the name cannot be resolved at this level, the search continues with the next highest containing tag. This resolution must eventually result in a match of a tag that is contained by the <**pcml**> tag, in which case the name is considered to be an absolute name, not a relative name.

```
<pcml version="1.0">
  <program name="polytest" path="/QSYS.lib/MYLIB.lib/POLYTEST.pgm">
    <!-- Parameter 1 contains a count of polygoins along with an array of polygons -->
    <struct name="parm1" usage="inputoutput">
      <data name="nbrPolygons" type="int" length="4" init="5" />
      <!-- Each polygon contains a count of the number of points along with an array of points -->
      <struct name="polygon" count="nbrPolygons">
        <data name="nbrPoints" type="int" length="4" init="3" />
        <struct name="point" count="nbrPoints" >
          <data name="x" type="int" length="4" init="100" />
          <data name="y" type="int" length="4" init="200" />
        </struct>
      </struct>
    </struct>
  </program>
</pcml>
```

Specifying offsets

Some programs return information with a fixed structure followed by one or more variable length fields or structures. In this case, the location of a variable length element is usually specified as an offset or displacement within the parameter.

An offset is the distance in bytes from the beginning of the parameters to the beginnings of a field or structure. A displacement is the distance in bytes from the beginning of one structure to the beginning of another structure.

For offsets, since the distance is from the beginning of the parameter, you should specify **offsetfrom="0"**. The following is an example of an offset from the beginning of the parameter:

```
<pcml version="1.0">
  <program name="myprog" path="/QSYS.lib/MYLIB.lib/MYPROG.pgm">
    <!-- receiver variable contains a path -->
    <struct name="receiver" usage="output" outputsize="2048">
      <data name="pathType"         type="int"  length="4" />
      <data name="offsetToPathName" type="int"  length="4" />
      <data name="lengthOfPathName" type="int"  length="4" />
      <data name="pathName"         type="char" length="lengthOfPathName"
               offset="offsetToPathName"  offsetfrom="0"/>
    </struct>
  </program>
</pcml>
```

For displacements, since the distance is from the beginning of another structure, you specify the name of the structure to which the offset is relative. The following is an example of an displacement from the beginning of a named structure:

```
<pcml version="1.0">
  <program name="myprog" path="/QSYS.lib/MYLIB.lib/MYPROG.pgm">
    <!-- receiver variable contains an object -->
    <struct name="receiver" usage="output" >
      <data name="objectName"       type="char"  length="10" />
      <data name="libraryName"      type="char"  length="10" />
      <data name="objectType"       type="char"  length="10" />
      <struct name="pathInfo" usage="output" outputsize="2048" >
        <data name="pathType"         type="int"  length="4" />
        <data name="offsetToPathName" type="int"  length="4" />
        <data name="lengthOfPathName" type="int"  length="4" />
        <data name="pathName"         type="char" length="lengthOfPathName"
                 offset="offsetToPathName"  offsetfrom="pathInfo"/>
      </struct>
    </struct>
  </program>
```

```
</pcml>
```

Program Call Markup Language (PCML) examples

Some examples of using Program Call Markup Language to call OS/400 APIs are listed below. The explanation of each example links to a page that shows the PCML source followed by a Java program.

- Simple example of retrieving data: Shows the PCML source and Java program needed to retrieve information about a user profile on the AS/400. The API being called is the *Retrieve User Information* (**QSYRSURI**) API.
- Retrieving a list of information: Shows the PCML source and Java program needed to retrieve a list of authorized users on an AS/400. The API being called is the *Open List of Authorized Users* (**QGYOLAUS**) API. This example illustrates how to access an array of structures returned by an AS/400 program.
- Retrieving multidimensional data Shows the PCML source and Java program needed to retrieve a list Network File System (NFS) exports from an AS/400. The API being called is the *Retrieve NFS Exports* (**QZNFRTVE**) API. This example illustrates how to access arrays of structures within an array of structures.

Note: The proper authority for each example varies but may include specific object authorities and special authorities. In order to run these examples, you must sign on with a user profile that has authority to do the following:

- Call the OS/400 API in the example
- Access the information being requested

License information

IBM grants you a nonexclusive license to use these as examples from which you can generate similar function tailored to your own specific needs. These samples are provided in the form of source material which you may change and use.

DISCLAIMER

This sample code is provided by IBM for illustrative purposes only. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. All programs contained herein are provided to you "AS IS" without any warranties of any kind. ALL WARRANTIES, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE EXPRESSLY DISCLAIMED.

Your license to this sample code provides you no right or licenses to any IBM patents. IBM has no obligation to defend or indemnify against any claim of infringement, including but not limited to: patents, copyright, trade secret, or intellectual property rights of any kind.

COPYRIGHT

Simple example of retrieving data

PCML source for calling QSYRUSRI

```
<pcml version="1.0">

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

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

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

</pcml>
```

Java program source for calling QSYRUSRI

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

// Example program to call "Retrieve User Information" (QSYRUSRI) API
public class qsyrusri {

    public qsyrusri() {
    }

    public static void main(String[] argv)
    {
        AS400 as400System;          // com.ibm.as400.access.AS400
        ProgramCallDocument pcml;   // com.ibm.as400.data.ProgramCallDocument
        boolean rc = false;         // Return code from ProgramCallDocument.callProgram()
        String msgId, msgText;      // Messages returned from AS/400
        Object value;               // Return value from ProgramCallDocument.getValue()

        System.setErr(System.out);

        // Construct AS400 without parameters, user will be prompted
        as400System = new AS400();

        try
        {
            // Uncomment the following to get debugging information
            //com.ibm.as400.data.PcmlMessageLog.setTraceEnabled(true);

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

            // Construct ProgramCallDocument
            // First parameter is system to connect to
            // Second parameter is pcml resource name. In this example,
            // serialized PCML file "qsyrusri.pcml.ser" or
            // PCML source file "qsyrusri.pcml" must be found in the classpath.
            pcml = new ProgramCallDocument(as400System, "qsyrusri");

            // Set input parameters. Several parameters have default values
            // specified in the PCML source. Do not need to set them using Java code.
            System.out.println("    Setting input parameters...");
            pcml.setValue("qsyrusri.receiverLength", new Integer((pcml.getOutputsize("qsyrusri.receiver"))));

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

            // If return code is false, we received messages from the AS/400
            if(rc == false)
            {
                // Retrieve list of AS/400 messages
                AS400Message[] msgs = pcml.getMessageList("qsyrusri");

                // Iterate through messages and write them to standard output
                for (int m = 0; m < msgs.length; m++)
                {
                    msgId = msgs[m].getID();
                    msgText = msgs[m].getText();
                    System.out.println("    " + msgId + " - " + msgText);
                }
                System.out.println("** Call to QSYRUSRI failed. See messages above **");
                System.exit(0);
            }
            // Return code was true, call to QSYRUSRI succeeded
            // Write some of the results to standard output
            else
            {
                value = pcml.getValue("qsyrusri.receiver.bytesReturned");
                System.out.println("        Bytes returned:      " + value);
                value = pcml.getValue("qsyrusri.receiver.bytesAvailable");
                System.out.println("        Bytes available:     " + value);
                value = pcml.getValue("qsyrusri.receiver.userProfile");
                System.out.println("        Profile name:        " + value);
                value = pcml.getValue("qsyrusri.receiver.previousSignonDate");
                System.out.println("        Previous signon date:" + value);
                value = pcml.getValue("qsyrusri.receiver.previousSignonTime");
                System.out.println("        Previous signon time:" + value);
            }
        }
        catch (PcmlException e)
        {
            System.out.println(e.getLocalizedMessage());
            e.printStackTrace();
            System.out.println("*** Call to QSYRUSRI failed. ***");
            System.exit(0);
        }

        System.exit(0);
    } // End main()

}
```

Example of retrieving a list of information

PCML source for calling QGYOLAUS

```
<pcml version="1.0">

<!-- PCML source for calling "Open List of Authorized Users" (QGYOLAUS) API -->

  <!-- Format AUTU0150 - Other formats are available -->
  <struct name="autu0150">
    <data name="name"         type="char" length="10" />
    <data name="userOrGroup"  type="char" length="1"  />
    <data name="groupMembers" type="char" length="1"  />
    <data name="description"  type="char" length="50" />
  </struct>

  <!-- List information structure (common for "Open List" type APIs) -->
  <struct name="listInfo">
    <data name="totalRcds"    type="int"  length="4" />
    <data name="rcdsReturned" type="int"  length="4" />
    <data name="rqsHandle"    type="byte" length="4" />
    <data name="rcdLength"    type="int"  length="4" />
    <data name="infoComplete" type="char" length="1" />
    <data name="dateCreated"  type="char" length="7" />
    <data name="timeCreated"  type="char" length="6" />
    <data name="listStatus"   type="char" length="1" />
    <data                     type="byte" length="1" />
    <data name="lengthOfInfo" type="int"  length="4" />
    <data name="firstRecord"  type="int"  length="4" />
    <data                     type="byte" length="40" />
  </struct>

  <!-- Program QGYOLAUS and its parameter list for retrieving AUTU0150 format -->
  <program name="qgyolaus" path="/QSYS.lib/QGY.lib/QGYOLAUS.pgm" parseorder="listInfo receiver">
    <data   name="receiver"       type="struct" struct="autu0150" usage="output"
              count="listInfo.rcdsReturned" outputsize="receiverLength" />
    <data   name="receiverLength" type="int"    length="4"  usage="input" init="16384" />
    <data   name="listInfo"       type="struct" struct="listInfo" usage="output" />
    <data   name="rcdsToReturn"   type="int"    length="4"  usage="input" init="264" />
    <data   name="format"         type="char"   length="10" usage="input" init="AUTU0150" />
    <data   name="selection"      type="char"   length="10" usage="input" init="*USER" />
    <data   name="member"         type="char"   length="10" usage="input" init="*NONE" />
    <data   name="errorCode"      type="int"    length="4"  usage="input" init="0" />

  </program>

  <!-- Program QGYGTLE returned additional "records" from the list
       created by QGYOLAUS. -->
  <program name="qgygtle" path="/QSYS.lib/QGY.lib/QGYGTLE.pgm" parseorder="listInfo receiver">
    <data   name="receiver"       type="struct" struct="autu0150" usage="output"
            count="listInfo.rcdsReturned" outputsize="receiverLength" />
    <data   name="receiverLength" type="int"    length="4" usage="input" init="16384" />
    <data   name="requestHandle"  type="byte"   length="4" usage="input" />
    <data   name="listInfo"       type="struct" struct="listInfo" usage="output" />
    <data   name="rcdsToReturn"   type="int"    length="4" usage="input" init="264" />
    <data   name="startingRcd"    type="int"    length="4" usage="input" />
    <data   name="errorCode"      type="int"    length="4" usage="input" init="0" />
  </program>

  <!-- Program QGYCLST closes the list, freeing resources on the AS/400 -->
  <program name="qgyclst" path="/QSYS.lib/QGY.lib/QGYCLST.pgm" >
    <data   name="requestHandle"  type="byte"   length="4" usage="input" />
    <data   name="errorCode"      type="int"    length="4" usage="input" init="0" />
  </program>
</pcml>
```

Java program source for calling QGYOLAUS

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

// Example program to call "Retrieve List of Authorized Users" (QGYOLAUS) API
public class qgyolaus
{

  public static void main(String[] argv)
  {
    AS400 as400System;         // com.ibm.as400.access.AS400
    ProgramCallDocument pcml;  // com.ibm.as400.data.ProgramCallDocument
    boolean rc = false;        // Return code from ProgramCallDocument.callProgram()
    String msgId, msgText;     // Messages returned from AS/400
    Object value;              // Return value from ProgramCallDocument.getValue()

    int[] indices = new int[1]; // Indices for access array value
    int nbrRcds,                // Number of records returned from QGYOLAUS and QGYGTLE
        nbrUsers;               // Total number of users retrieved
    String listStatus;          // Status of list on AS/400
    byte[] requestHandle = new byte[4];

    System.setErr(System.out);

    // Construct AS400 without parameters, user will be prompted
    as400System = new AS400();

    try
    {
      // Uncomment the following to get debugging information
      //com.ibm.as400.data.PcmlMessageLog.setTraceEnabled(true);

      System.out.println("Beginning PCML Example..");
      System.out.println("    Constructing ProgramCallDocument for QGYOLAUS API...");

      // Construct ProgramCallDocument
      // First parameter is system to connect to
      // Second parameter is pcml resource name. In this example,
      // serialized PCML file "qgyolaus.pcml.ser" or
      // PCML source file "qgyolaus.pcml" must be found in the classpath.
      pcml = new ProgramCallDocument(as400System, "qgyolaus");

      // All input parameters have default values specified in the PCML source.
      // Do not need to set them using Java code.

      // Request to call the API
      // User will be prompted to sign on to the system
      System.out.println("    Calling QGYOLAUS API requesting information for the sign-on user.");
      rc = pcml.callProgram("qgyolaus");

      // If return code is false, we received messages from the AS/400
      if(rc == false)
      {
        // Retrieve list of AS/400 messages
        AS400Message[] msgs = pcml.getMessageList("qgyolaus");

        // Iterate through messages and write them to standard output
        for (int m = 0; m < msgs.length; m++)
        {
            msgId = msgs[m].getID();
            msgText = msgs[m].getText();
            System.out.println("    " + msgId + " - " + msgText);
        }
        System.out.println("** Call to QGYOLAUS failed. See messages above **");
        System.exit(0);
      }
      // Return code was true, call to QGYOLAUS succeeded
      // Write some of the results to standard output
      else
      {
        boolean doneProcessingList = false;
        String programName = "qgyolaus";
        nbrUsers = 0;
        while (!doneProcessingList)
        {
          nbrRcds = pcml.getIntValue(programName + ".listInfo.rcdsReturned");
          requestHandle = (byte[]) pcml.getValue(programName + ".listInfo.rqsHandle");

          // Iterate through list of users
          for (indices[0] = 0; indices[0] < nbrRcds; indices[0]++)
          {
               value = pcml.getValue(programName + ".receiver.name", indices);
               System.out.println("User:  " + value);

               value = pcml.getValue(programName + ".receiver.description", indices);
               System.out.println("\t\t" + value);
          }

          nbrUsers += nbrRcds;

          // See if we retrieved all the users.
          // If not, subsequent calls to "Get List Entries" (QGYGTLE)
          // would need to be made to retrieve the remaining users in the list.
          listStatus = (String) pcml.getValue(programName + ".listInfo.listStatus");
          if ( listStatus.equals("2")   // List is marked as "Complete"
            || listStatus.equals("3") ) // Or list is marked "Error building"
          {
            doneProcessingList = true;
          }
          else
          {
            programName = "qgygtle";

            // Set input parameters for QGYGTLE
            pcml.setValue("qgygtle.requestHandle", requestHandle);
            pcml.setIntValue("qgygtle.startingRcd", nbrUsers + 1);

            // Call "Get List Entries" (QGYGTLE) to get more users from list
            rc = pcml.callProgram("qgygtle");

            // If return code is false, we received messages from the AS/400
            if(rc == false)
            {
              // Retrieve list of AS/400 messages
              AS400Message[] msgs = pcml.getMessageList("qgygtle");

              // Iterate through messages and write them to standard output
              for (int m = 0; m < msgs.length; m++)
              {
                  msgId = msgs[m].getID();
                  msgText = msgs[m].getText();
                  System.out.println("    " + msgId + " - " + msgText);
              }
              System.out.println("** Call to QGYGTLE failed. See messages above **");
              System.exit(0);
            }
            // Return code was true, call to QGYGTLE succeeded

          }
        }
        System.out.println("Number of users returned:  " + nbrUsers);

        // Call the "Close List" (QGYCLST) API
        pcml.setValue("qgyclst.requestHandle", requestHandle);
        rc = pcml.callProgram("qgyclst");
      }
    }
    catch(PcmlException e)
    {
      System.out.println(e.getLocalizedMessage());
      e.printStackTrace();
      System.out.println("*** Call to QGYOLAUS failed. ***");
      System.exit(0);
    }
    System.exit(0);
  }
}
```

Example of retrieving multidimensional data

PCML source for calling QZNFRTVE

```
<pcml version="1.0">

  <struct name="receiver">
    <data name="lengthOfEntry"            type="int"  length="4" />
    <data name="dispToObjectPathName"     type="int"  length="4" />
    <data name="lengthOfObjectPathName"   type="int"  length="4" />
    <data name="ccsidOfObjectPathName"    type="int"  length="4" />
    <data name="readOnlyFlag"             type="int"  length="4" />
    <data name="nosuidFlag"               type="int"  length="4" />
    <data name="dispToReadWriteHostNames" type="int"  length="4" />
    <data name="nbrOfReadWriteHostNames"  type="int"  length="4" />
    <data name="dispToRootHostNames"      type="int"  length="4" />
    <data name="nbrOfRootHostNames"       type="int"  length="4" />
    <data name="dispToAccessHostNames"    type="int"  length="4" />
    <data name="nbrOfAccessHostNames"     type="int"  length="4" />
    <data name="dispToHostOptions"        type="int"  length="4" />
    <data name="nbrOfHostOptions"         type="int"  length="4" />
    <data name="anonUserID"               type="int"  length="4" />
    <data name="anonUsrPrf"               type="char" length="10" />
    <data name="pathName"                 type="char" length="lengthOfObjectPathName"
          offset="dispToObjectPathName" offsetfrom="receiver" />

    <struct name="rwAccessList" count="nbrOfReadWriteHostNames"
            offset="dispToReadWriteHostNames" offsetfrom="receiver">
      <data name="lengthOfEntry"          type="int"  length="4" />
      <data name="lengthOfHostName"       type="int"  length="4" />
      <data name="hostName"               type="char" length="lengthOfHostName" />
      <data                               type="byte" length="0"
            offset="lengthOfEntry" />
    </struct>

    <struct name="rootAccessList" count="nbrOfRootHostNames"
            offset="dispToRootHostNames" offsetfrom="receiver">
      <data name="lengthOfEntry"          type="int"  length="4" />
      <data name="lengthOfHostName"       type="int"  length="4" />
      <data name="hostName"               type="char" length="lengthOfHostName" />
      <data                               type="byte" length="0"
            offset="lengthOfEntry" />
    </struct>

    <struct name="accessHostNames" count="nbrOfAccessHostNames"
            offset="dispToAccessHostNames" offsetfrom="receiver" >
      <data name="lengthOfEntry"          type="int"  length="4" />
      <data name="lengthOfHostName"       type="int"  length="4" />
      <data name="hostName"               type="char" length="lengthOfHostName" />
      <data                               type="byte" length="0"
            offset="lengthOfEntry" />
    </struct>

    <struct name="hostOptions" offset="dispToHostOptions" offsetfrom="receiver" count="nbrOfHostOptions">
      <data name="lengthOfEntry"          type="int"  length="4" />
      <data name="dataFileCodepage"       type="int"  length="4" />
      <data name="pathNameCodepage"       type="int"  length="4" />
      <data name="writeModeFlag"          type="int"  length="4" />
      <data name="lengthOfHostName"       type="int"  length="4" />
      <data name="hostName"               type="char" length="lengthOfHostName" />
      <data                               type="byte" length="0"
            offset="lengthOfEntry" />
    </struct>

    <data type="byte" length="0" offset="lengthOfEntry" />
  </struct>

  <struct name="returnedRcdsFdbkInfo">
    <data name="bytesReturned"            type="int" length="4" />
    <data name="bytesAvailable"           type="int" length="4" />
    <data name="nbrOfNFSExportEntries"    type="int" length="4" />
    <data name="handle"                   type="int" length="4" />
  </struct>

  <program name="qznfrtve" path="/QSYS.lib/QZNFRTVE.pgm" parseorder="returnedRcdsFdbkInfo receiver" >
    <data name="receiver"             type="struct" struct="receiver" usage="output"
              count="returnedRcdsFdbkInfo.nbrOfNFSExportEntries" outputsize="receiverLength"/>
    <data name="receiverLength"       type="int"    length="4" usage="input" init="4096" />
    <data name="returnedRcdsFdbkInfo" type="struct" struct="returnedRcdsFdbkInfo" usage="output" />
    <data name="formatName"           type="char"   length="8" usage="input" init="EXPE0100" />
    <data name="objectPathName"       type="char"   length="lengthObjPathName" usage="input" init="*FIRST" />
    <data name="lengthObjPathName"    type="int"    length="4" usage="input" init="6" />
    <data name="ccsidObjectPathName"  type="int"    length="4" usage="input" init="0" />
    <data name="desiredCCSID"         type="int"    length="4" usage="input" init="0" />
    <data name="handle"               type="int"    length="4" usage="input" init="0" />
    <data name="errorCode"            type="int"    length="4" usage="input" init="0" />
  </program>

</pcml>
```

Java program source for calling QZNFRTVE

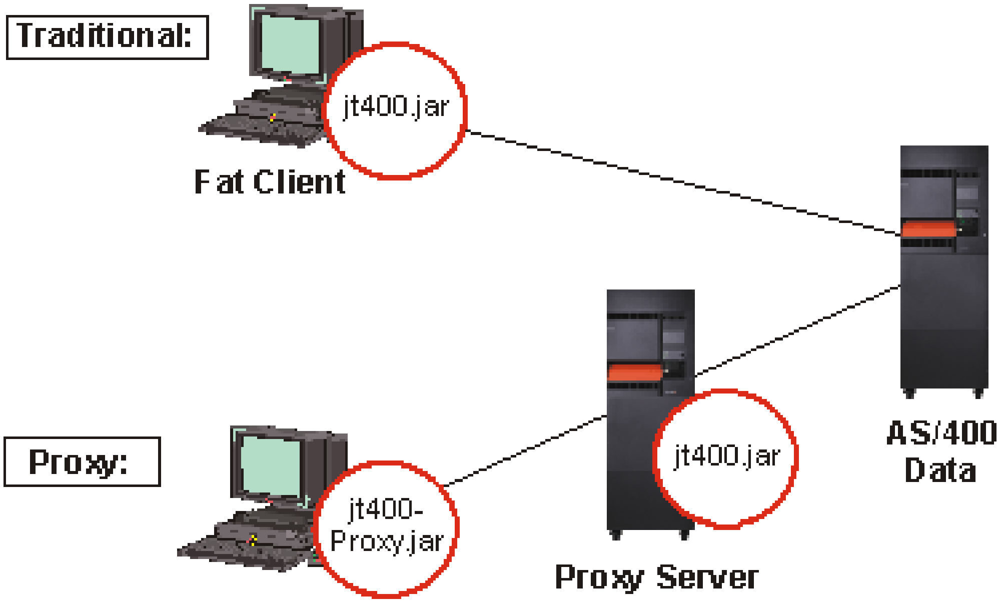
```
import com.ibm.as400.data.ProgramCallDocument;
import com.ibm.as400.data.PcmlException;
import com.ibm.as400.access.AS400;
import com.ibm.as400.access.AS400Message;

// Example program to call "Retrieve NFS Exports" (QZNFRTVE) API
public class qznfrtve
{
  public static void main(String[] argv)
  {
    AS400 as400System;         // com.ibm.as400.access.AS400
    ProgramCallDocument pcml;  // com.ibm.as400.data.ProgramCallDocument
    boolean rc = false;        // Return code from ProgramCallDocument.callProgram()
    String msgId, msgText;     // Messages returned from AS/400
    Object value;              // Return value from ProgramCallDocument.getValue()

    System.setErr(System.out);

    // Construct AS400 without parameters, user will be prompted
    as400System = new AS400();

    int[] indices = new int[2]; // Indices for access array value
    int nbrExports;             // Number of exports returned
    int nbrOfReadWriteHostNames, nbrOfRWHostNames,
        nbrOfRootHostNames,      nbrOfAccessHostnames, nbrOfHostOpts;

    try
    {
      // Uncomment the following to get debugging information
      // com.ibm.as400.data.PcmlMessageLog.setTraceEnabled(true);

      System.out.println("Beginning PCML Example..");
      System.out.println("    Constructing ProgramCallDocument for QZNFRTVE API...");

      // Construct ProgramCallDocument
      // First parameter is system to connect to
      // Second parameter is pcml resource name. In this example,
      // serialized PCML file "qznfrtve.pcml.ser" or
      // PCML source file "qznfrtve.pcml" must be found in the classpath.
      pcml = new ProgramCallDocument(as400System, "qznfrtve");

      // Set input parameters. Several parameters have default values
      // specified in the PCML source. Do not need to set them using Java code.
      System.out.println("    Setting input parameters...");
      pcml.setValue("qznfrtve.receiverLength", new Integer( ( pcml.getOutputsize("qznfrtve.receiver"))));

      // Request to call the API
      // User will be prompted to sign on to the system
      System.out.println("    Calling QZNFRTVE API requesting NFS exports.");
      rc = pcml.callProgram("qznfrtve");

      if (rc == false)
      {
        // Retrieve list of AS/400 messages
        AS400Message[] msgs = pcml.getMessageList("qznfrtve");

        // Iterate through messages and write them to standard output
        for (int m = 0; m < msgs.length; m++)
        {
            msgId = msgs[m].getID();
            msgText = msgs[m].getText();
            System.out.println("    " + msgId + " - " + msgText);
        }
        System.out.println("** Call to QZNFRTVE failed. See messages above **");
        System.exit(0);
      }
      // Return code was true, call to QZNFRTVE succeeded
      // Write some of the results to standard output
      else
      {
        nbrExports = pcml.getIntValue("qznfrtve.returnedRcdsFdbkInfo.nbrOfNFSExportEntries");
        // Iterate through list of exports
        for (indices[0] = 0; indices[0] < nbrExports; indices[0]++)
        {
          value = pcml.getValue("qznfrtve.receiver.pathName", indices);
          System.out.println("Path name = " + value);

          // Iterate and write out Read Write Host Names for this export
          nbrOfReadWriteHostNames = pcml.getIntValue("qznfrtve.receiver.nbrOfReadWriteHostNames", indices);
          for(indices[1] = 0; indices[1] < nbrOfReadWriteHostNames; indices[1]++)
          {
            value = pcml.getValue("qznfrtve.receiver.rwAccessList.hostName", indices);
            System.out.println("    Read/write access host name = " + value);
          }

          // Iterate and write out Root Host Names for this export
          nbrOfRootHostNames = pcml.getIntValue("qznfrtve.receiver.nbrOfRootHostNames", indices);
          for(indices[1] = 0; indices[1] < nbrOfRootHostNames; indices[1]++)
          {
            value = pcml.getValue("qznfrtve.receiver.rootAccessList.hostName", indices);
            System.out.println("    Root access host name = " + value);
          }

          // Iterate and write out Access Host Names for this export
          nbrOfAccessHostnames = pcml.getIntValue("qznfrtve.receiver.nbrOfAccessHostNames", indices);
          for(indices[1] = 0; indices[1] < nbrOfAccessHostnames; indices[1]++)
          {
            value = pcml.getValue("qznfrtve.receiver.accessHostNames.hostName", indices);
            System.out.println("    Access host name = " + value);
          }

          // Iterate and write out Host Options for this export
          nbrOfHostOpts = pcml.getIntValue("qznfrtve.receiver.nbrOfHostOptions", indices);
          for(indices[1] = 0; indices[1] < nbrOfHostOpts; indices[1]++)
          {
            System.out.println("    Host options:");
            value = pcml.getValue("qznfrtve.receiver.hostOptions.dataFileCodepage", indices);
            System.out.println("        Data file code page = " + value);
            value = pcml.getValue("qznfrtve.receiver.hostOptions.pathNameCodepage", indices);
            System.out.println("        Path name code page = " + value);
            value = pcml.getValue("qznfrtve.receiver.hostOptions.writeModeFlag", indices);
            System.out.println("        Write mode flag = " + value);
            value = pcml.getValue("qznfrtve.receiver.hostOptions.hostName", indices);
            System.out.println("        Host name = " + value);
          }
        } // end for loop iterating list of exports
      } // end call to QZNFRTVE succeeded
    }
    catch(PcmlException e)
    {
      System.out.println(e.getLocalizedMessage());
      e.printStackTrace();
      System.exit(-1);
    }

    System.exit(0);
  } // end main()
}
```

Proxy Support

IBM Toolbox for Java includes proxy support for some classes. Proxy support is the processing that IBM Toolbox for Java needs to carry out a task on a Java virtual machine (JVM) when the application is on a different JVM. »Proxy support includes using the Secure Sockets Layer (SSL) protocol to encrypt data.«

The proxy classes reside in jt400Proxy.jar, which ships with the rest of the IBM Toolbox for Java. The proxy classes, like the other classes in the IBM Toolbox for Java, comprise a set of platform independent Java classes that can run on any computer with a Java virtual machine. The proxy classes dispatch all method calls to a server application, or proxy server. The full IBM Toolbox for Java classes are on the proxy server. When a client uses a proxy class, the request is transferred to the proxy server which creates and administers the real IBM Toolbox for Java objects.

The following picture shows how the standard and proxy client connect to the AS/400. The proxy server can be the iSeries that contains the data.



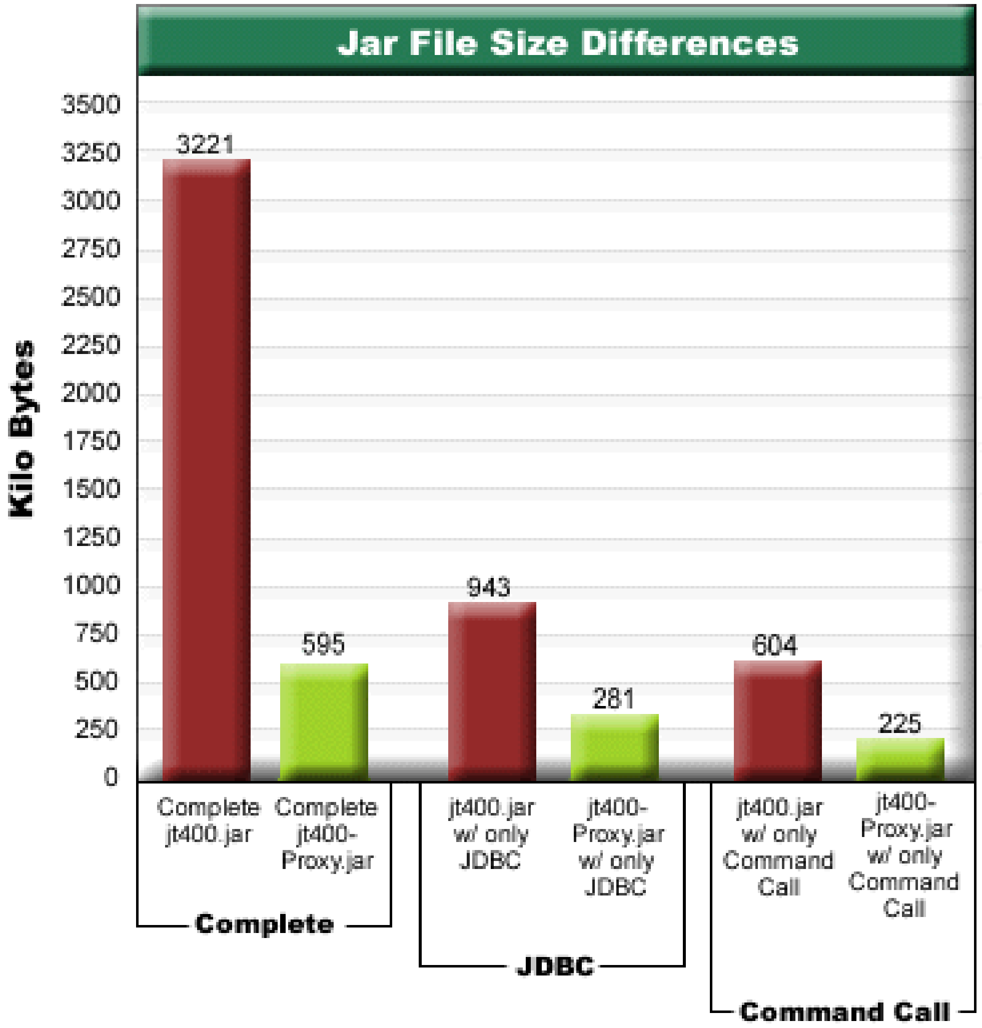
An application that uses proxy support performs more slowly than if it uses standard IBM Toolbox for Java classes due to the extra communication needed to support the smaller proxy classes. Applications that make fewer method calls have less performance degradation.

Before proxy support, the classes containing the public interface, all the classes needed to process a request, and the application itself ran on the same JVM. When using proxy support, the public interface must be with the application, but classes for processing requests can run on a different JVM. Proxy support does not change the public interface. The same program can run with either the proxy version of IBM Toolbox for Java or the standard version.

Using the jt400Proxy.jar file

The goal of the multiple-tier, proxy scenario is to make the public interface jar file as small as possible, so that downloading it from an applet takes less time. When you use the proxy classes, you don't need to install the entire IBM Toolbox for Java on the client. Instead, use AS400JarMaker on the jt400Proxy.jar file to include only the required components, which makes the jar file as small as possible.

The chart below compares the size of the proxy jar files with the standard jar files:



An additional benefit is that proxy support requires you have to have fewer ports open through a firewall. With standard IBM Toolbox for Java, you must have multiple ports open. »This is because each IBM Toolbox for Java service uses a different port to communicate with the server.« For example, Command call uses a different port than JDBC, which uses a different port than print, and so on. You must allow each of these ports through the firewall. However, when using proxy support, all the data flows through the same port.

Standard proxy and HTTP tunneling

»Two options are available for running via a proxy: standard proxy and HTTP tunneling:

- Standard proxy is where the proxy client and proxy server communicate by using a socket over a port. The default port is 3470. Change the default port by using the setPort() method on the ProxyServer class, or by using the -port option when starting the proxy server. For example:

```
java com.ibm.as400.access.ProxyServer -port 1234
```

- HTTP tunneling is where the proxy client and proxy server communicate by way of the HTTP server. The IBM Toolbox for Java provides a servlet that handles the proxy request. The proxy client calls the servlet by way of the HTTP server. The advantage of tunneling is that you do not have to open an additional port through the firewalls, because communication is by way of the HTTP port. The disadvantage of tunneling is that it is slower than standard proxy.

IBM Toolbox for Java uses the proxy server name to determine if standard proxy or tunneling proxy is being used:

- For standard proxy, just use the server name. For example:

```
com.ibm.as400.access.AS400.proxyServer=myServer
```

- For tunneling, use a URL to force the proxy client to use tunneling. For example:

  ```
  com.ibm.as400.access.AS400.proxyServer=http://myServer
  ```

When running standard proxy, a socket connection exists between the client and server. If that connection fails, the server cleans up resources associated with that client.

When using HTTP tunneling, using the HTTP protocol makes proxy connectionless. That is, a new connection is made for each data flow. Because the protocol is connectionless, the server does not know if the client application is no longer active. Consequently, the server does not know when to clean up resources. The tunneling server solves this problem by using a thread to clean up resources at a predetermined interval (which is based on a timeout value).

At the end of the predetermined interval, the thread runs and cleans up resources that have not been used lately. Two system properties govern the thread:

- com.ibm.as400.access.TunnelProxyServer.clientCleanupInterval is how often, in seconds, the cleanup thread runs. The default is every two hours.
- com.ibm.as400.access.TunnelProxyServer.clientLifetime is how long, in seconds, a resource can be idle before it is cleaned up. The default is 30 minutes.«

Using proxy server

To use the proxy server implementation of the IBM Toolbox for Java classes, complete the following steps:

1. Run AS400ToolboxJarMaker on jt400Proxy.jar to discard classes that you do not need. This step is optional but recommended.
2. Determine how to get jt400Proxy.jar to the client.
   - For Java programs, use the AS400ToolboxInstaller class or another method to get it to the client.
   - For Java applets, you may be able to download the jar file from the HTML server.
3. Determine what server you will use for the proxy server.
   - For Java applications, the proxy server can be any computer.
   - For Java applets, the proxy server must be running on the same computer as the HTTP server.
4. Ensure that you have put jt400.jar in the CLASSPATH on the server.
5. Start the proxy server or use the proxy servlet:
   - For standard proxy, start the proxy server by using the following command:

     ```
     java com.ibm.as400.access.ProxyServer
     ```
   - »For tunneling proxy, configure your HTTP server to use the proxy servlet. The servlet class name is com.ibm.as400.access.TunnelProxyServer and it is contained in jt400.jar.
6. On the client, set a system property to identify the proxy server. IBM Toolbox for Java uses this system property to determine if standard proxy or tunneling proxy is being used.
   - For standard proxy, the property value is the name of the machine that runs the proxy server. For example:

     ```
     com.ibm.as400.access.AS400.proxyServer=myServer
     ```
   - For tunneling proxy, use a URL to force the proxy client to use tunneling. For example:

     ```
     com.ibm.as400.access.AS400.proxyServer=http://myServer
     ```

   «
7. Run the client program.

When you want to work with both the proxy classes and classes not in jt400Proxy.jar, you can refer to jt400.jar instead of jt400Proxy.jar. jt400Proxy.jar is a subset of the jt400.jar and, therefore, all of the proxy classes are contained in the jt400.jar file.

Using SSL

»When using proxy, three options are available for encrypting data as it flows from the proxy client to the target iSeries server. SSL algorithms are used to encrypt data.

1. The data flows between the proxy client and proxy server can be encrypted.
2. The data flows between the proxy server and target iSeries server can be encrypted.
3. Both one and two. The data flow between proxy client and proxy server, and the flow between the proxy server and the target iSeries can be encrypted.

See Secure Sockets Layer for more information.«

Examples: Using proxy servers

We have provided »three« specific examples for using a proxy server with the steps listed above.

- Running a Java application using proxy support
- Running a Java applet using proxy support
- »Running a Java application using tunneling proxy support.«

Classes enabled to work with proxy server

Some IBM Toolbox for Java classes are enabled to work with the proxy server application. These include the following:

- JDBC
- Record-level access
- Integrated file system
- Print
- Data Queues
- Command Call
- Program Call
- Service Program Call
- User space
- Data area
- AS400 class
- SecureAS400 class

Other classes are not supported at this time by jt400Proxy. Also, integrated file system permissions are not functional using only the proxy jar file. However, you can use the JarMaker class to include these classes from the jt400.jar file.

Example: Running a Java application using Proxy Support

The following example shows you the steps to run a Java application using proxy support.

1. Choose a machine to act as the proxy server. The Java environment and CLASSPATH on the proxy server machine should include the `jt400.jar` file. This machine must be able to connect to the iSeries system.
2. Start the proxy server on this machine by typing:
   ```
   java com.ibm.as400.access.ProxyServer -verbose
   ```
   Specifying verbose will allow you to monitor when the client connects and disconnects.
3. Choose a machine to act as the client. The Java environment and CLASSPATH on the client machine should include the `jt400Proxy.jar` file and your application classes. This machine must be able to connect to the proxy server but does not need a connection to the iSeries system.
4. Set the value of the `com.ibm.as400.access.AS400.proxyServer` system property to be the name of your proxy server, and run the application. An easy way to do this is by using the -D option on most Java Virtual Machine invocations:
   ```
   java -Dcom.ibm.as400.access.AS400.proxyServer=psMachineName YourApplication
   ```
5. As your application runs, you should see (if you set verbose in step 2) the application make at least one connection to the proxy server.

Example: Running a Java applet using proxy support

The following example shows you the steps to run a Java applet using proxy support.

1. Choose a machine to act as the proxy server. Applets can initiate network connections only to the machine from which they were originally downloaded; therefore, it works best to run the proxy server on the same machine as the HTTP server. The Java environment and CLASSPATH on the proxy server machine should include the `jt400.jar` file.
2. Start the proxy server on this machine by typing:
   ```
   java com.ibm.as400.access.ProxyServer -verbose
   ```
   Specifying verbose will allow you to monitor when the client connects and disconnects.
3. Applet code needs to be downloaded before it runs so it is best to reduce the size of the code as much as possible. The AS400ToolboxJarMaker can reduce the `jt400Proxy.jar` significantly by including only the code for the components that your applet uses. For instance, if an applet uses only JDBC, we can reduce the `jt400Proxy.jar` file to include the minimal amount of code by running:
   ```
   java utilities.AS400ToolboxJarMaker -source jt400Proxy.jar -destination
   jt400ProxySmall.jar -component JDBC
   ```
4. The applet must set the value of the `com.ibm.as400.access.AS400.proxyServer` system property to be the name of your proxy server. A convenient way to do this for applets is using a compiled Properties class (Example). Compile this class and place the generated Properties.class file in the `com/ibm/as400/access directory` (the same path your html file is coming from). For example, if the html file is `/mystuff/HelloWorld.html`, then Properties.class should be in `/mystuff/com/ibm/as400/access`.
5. Put the `jt400ProxySmall.jar` in the same directory as as the html file (/mystuff/ in step 4).
6. Refer to the applet like this in your HTML file:
   ```
   <APPLET archive="jt400Proxy.jar, Properties.class" code="YourApplet.class"
   width=300 height=100> </APPLET>
   ```

Example: Running a Java application using Tunneling Proxy Support

The following example shows you the steps to run a Java application using tunneling proxy support.

1. Choose the HTTP server that you want to run the proxy server, then configure it to run servlet com.ibm.as400.access.TunnelProxyServer (in jt400.jar).
   Note: Ensure that the HTTP server has a connection to the iSeries system that contains the data or resource that the application uses because the servlet connects to that iSeries to carry out requests.
2. Choose a machine to act as the client and ensure that the CLASSPATH on the client machine includes the jt400Proxy.jar file and your application classes. The client must be able to connect to the HTTP server but does not need a connection to the iSeries system.
3. Set the value of the com.ibm.as400.access.AS400.proxyServer property to be the name of your HTTP server in URL format.
4. Run the application, setting the value of the com.ibm.as400.access.AS400.proxyServer property to be the name of your HTTP server in URL format.. An easy way to do this is by using the -D option found on most JVMs:

   ```
   java -Dcom.ibm.as400.access.AS400.proxyServer=http://psMachineName YourApplication
   ```

   Note: The proxy client code creates a proper servlet URL by concatenating "servlet" and the servlet name to the server name. In this example, it converts http://psMachineName to http://psMachineName/servlet/TunnelProxyServer

Resource classes

The com.ibm.as400.resource package provides a generic framework for working with various AS400 objects and lists. This framework provides a consistent programming interface to all such objects and lists.

The resource package includes the following classes:

- Resource - an object that represents an iSeries resource, such as a user, printer, job, message, or file. Concrete subclasses of resource include:
    - RIFSFile
    - RJavaProgram
    - RJob
    - RPrinter
    - RQueuedMessage
    - RSoftwareResource
    - RUser

  Note: The NetServer classes in the access package are also concrete subclasses of Resource.
- ResourceList - an object that represents a list of iSeries resources, such as a list of users, printers, jobs, messages, or files. Concrete subclasses of resource include:
    - RIFSFileList
    - RJobList
    - RJobLog
    - RMessageQueue
    - RPrinterList
    - RUserList
- Presentation - an object that allows you to present information about resource objects, resource lists, attributes, selections, and sorts to end users

»

Resource and ChangeableResource classes

The com.ibm.as400.resource.Resource and com.ibm.as400.resource.ChangeableResource abstract classes represent an iSeries resource.

Resource

Resource is an abstract class that provides generic access to the attributes of any resource. Every attribute is identified using an attribute ID, and any given subclass of Resource will normally document the attribute IDs that it supports.

Resource provides only read access to the attribute values.

IBM Toolbox for Java provides the following resource objects:

- RIFSFile - represents a file or directory in the iSeries integrated file system
- RJavaProgram - represents a Java program on the iSeries
- RJob - represents an iSeries job
- RPrinter - represents an iSeries printer
- RQueuedMessage - represents a message in an iSeries message queue or job log
- RSoftwareResource - represents a licensed program on the iSeries
- RUser - represents an iSeries user

ChangeableResource

The ChangeableResource abstract class, a subclass of Resource, adds the ability to change attribute values of an iSeries resource. Attribute changes are cached internally until they are committed or canceled. This allows you to change many attribute values at once.

Note: The NetServer classes in the access package are also concrete subclasses of Resource and ChangeableResource.

Examples

The following examples show how you can directly use concrete subclasses of Resource and ChangeableResource, and also how generic code can work with any Resource or ChangeableResource subclass.

- Retrieving an attribute value from RUser, a concrete subclass of Resource
- Setting attribute values for RJob, a concrete subclass of ChangeableResource
- Using generic code to access resources«

Resource lists

The com.ibm.as400.resource.ResourceList class represents a list of iSeries resources. This is an abstract class which provides generic access to the contents of the list.

IBM Toolbox for Java provides the following resource lists:

- RIFSFileList - represents a list of files and directories in the iSeries integrated file system
- RJobList - represents a list of iSeries jobs
- RJobLog - represents a list of messages in an iSeries job log
- RMessageQueue - represents a list of messages in an iSeries message queue
- RPrinterList - represents a list of iSeries printers
- RUserList - represents a list of iSeries users

A resource list is always either open or closed. The resource list must be open in order to access its contents. In order to provide immediate access to the contents of the list and manage memory efficiently, most resource lists are loaded incrementally.

Resource lists allow you to:

- Open the list
- Close the list
- Access a specific Resource from the list
- Wait for a particular resource to load
- Wait for the complete resource list to load

You can also filter resource lists by using selection values. Every selection value is identified using a selection ID. Similarly, resource lists can be sorted using sort values. Every sort value is identified using a sort ID. Any given subclass of ResourceList will normally document the selection IDs and sort IDs that it supports.

Examples

The following examples show various ways of working with resource lists:

- Example: Getting and printing the contents of a ResourceList
- Example: Using generic code to access a ResourceList
- Example: Displaying a resource list in a servlet (HTML table)

»

Presentation class

Every resource object, resource list, and meta data object has an associated com.ibm.as400.resource.Presentation object that provides translated information, such as the name, full name, and icon.

Example: Printing a resource list and its sort values using their Presentations

You can use the Presentation information to present resource objects, resource lists, attributes, selections, and sorts to end users in text format.

```
void printCurrentSort(ResourceList resourceList) throws ResourceException
{
    // Get the presentation for the ResourceList and print its full name.
    Presentation resourceListPresentation = resourceList.getPresentation();
    System.out.println(resourceListPresentation.getFullName());

    // Get the current sort value.
    Object[] sortIDs = resourceList.getSortValue();

    // Print each sort ID.
    for(int i = 0; i < sortIDs.length; ++i)
    {
        ResourceMetaData sortMetaData = resourceList.getSortMetaData(sortIDs[i]);
        System.out.println("Sorting by " + sortMetaData.getName());
    }
}
```

«

Security classes

The IBM Toolbox for Java gives you security classes. You use the security classes to provide secured connections to an AS/400 system, verify a user's identity, and associate a user with the operating system thread when running on the local AS/400 system. The security services included are:

- Secure Sockets Layer (SSL): Provides secure connections both by encrypting the data exchanged between a client and an AS/400 server session and by performing server authentication.
- Authentication Services: Provide the ability to:
  - Authenticate a user identity and password against the OS/400 user registry.
  - Ability to assign an identity to the current OS/400 thread.

Secure Sockets Layer

Secure Sockets Layer (SSL) provides secure connections by:

- Encrypting the data exchanged between a client and server session
- Performing server authentication

Using SSL negatively affects performance because SSL connections perform slower than connections that do not have encryption. Use SSL connections when the sensitivity of the data transferred merits the increased cost in performance, for example, when transferring credit card or bank statement information.

Using SSL with an iSeries server requires OS/400 V4R4 or later.

Before you begin using SSL with IBM Toolbox for Java, you must understand your legal responsibilities.

SSL algorithms

IBM Toolbox for Java does not contain the algorithms needed to encrypt and decrypt data. »These algorithms are shipped with the IBM iSeries Client Encryption licensed programs 5722-CE2 and 5722-CE3. You need to order one of the 5722-CEx product versions of SSL depending on the country in which you live. Contact your IBM representative for more information or to order:

- IBM iSeries Client Encryption (56-bit), 5722-CE2, which is used in countries other than the United States or Canada
- IBM iSeries Client Encryption (128-bit), 5722-CE3, which is used only in the United States and Canada

Setting up your SSL environment

IBM Toolbox for Java provides for two environments for using SSL to encrypt your data, which you must properly set up.

- Using encryption between the IBM Toolbox for Java classes and the OS/400 servers
- Using encryption between the proxy client and the proxy server

Compatibility with earlier versions of IBM Toolbox for Java

The encryption algorithms and keyring class files are updated in V5R1 and using them requires that you:

- Use the V5R1 Client Encryption licensed programs
- Update the KeyRing.class file

The V5R1 version of IBM Toolbox for Java requires that you use the V5R1 version of client encryption.

When you have compatible V5R1 versions of IBM Toolbox for Java and client encryption on your client, you can connect to V4R4 and newer versions of OS/400.«

SSL legal responsibilities

IBM iSeries Client Encryption (56-bit) and IBM iSeries Client Encryption (128-bit) licensed products provide SSL Version 3.0 encryption support using 56-bit and 128-bit encryption algorithms.

These programs contain data encryption technology that is subject to special export licensing requirements of the US Department of Commerce. Other countries also may have export and import licensing requirements.

You are hereby notified that use by, or transfer to, users in the same or different countries of the same program may be prohibited or subject to:

- Special import laws, regulations, or policies of the user's national government
- Special export laws, regulations, or policies from your national government

You assume all responsibilities for ensuring that the program is used or transferred in accordance with all applicable import and export laws, regulations, and policies from now and beyond the expiration of the license.

You and your users must comply with other country's import/export laws.

»Using SSL to encrypt data between IBM Toolbox for Java and OS/400 servers

You can use SSL to encrypt data exchanged between IBM Toolbox for Java classes and OS/400 servers. On the client side, you use the files that come with the IBM iSeries Client Encryption licensed program (5722-CE2 or 5722-CE3) to encrypt the data. On the server side, you must use the OS/400 digital certificate manager to configure the OS/400 servers to exchange encrypted data.

Setting up your client and server to use SSL

To encrypt data flowing between the IBM Toolbox for Java classes and OS/400 servers, complete the following tasks:

1. Set up OS/400 to exchange encrypted data.
2. Set up the client (the IBM Toolbox for Java classes) to exchange encrypted data. The procedure for this step depends on the kind of certificate you used when setting up SSL on your server:
    - Set up the client when using a server certificate from a trusted authority
    - Set up the client when using a self-signed certificate

      Note: Setting up your client by using a certificate from a trusted authority is significantly easier and faster than using a self-signed certificate.
3. Use the SecureAS400 object to force IBM Toolbox for Java to encrypt data.

   Note: Completing the first two steps above only creates a secure path between the client and the server. Your application must use the SecureAS400 object to tell the IBM Toolbox for Java which data to encrypt. Data that flows through the SecureAS400 object is the only data that is encrypted. If you use an AS400 object, data is not encrypted and the normal path to the server is used.«

Setting up iSeries servers to use SSL

To set up your iSeries servers to use SSL with IBM Toolbox for Java, complete the following steps:

1. »Install the IBM Cryptographic Access Provider for iSeries licensed program (5722-AC2 or 5722-AC3) on your iSeries servers to provide server-side encryption.
   - 5722-AC2 provides 56-bit encryption
   - 5722-AC3 provides 128-bit encryption.
2. Install the IBM iSeries Client Encryption licensed program (5722-CE2 or 5722-CE3) on your iSeries server.« Client Encryption provides the Java classes and utilities used by the IBM Toolbox for Java classes on the client side.
3. Change the authority of the directory that contains the client encryption files.
4. Get and configure the server certificate.
5. Apply the certificate to the following iSeries servers that are used by IBM Toolbox for Java:
   - QIBM_OS400_QZBS_SVR_CENTRAL
   - QIBM_OS400_QZBS_SVR_DATABASE
   - QIBM_OS400_QZBS_SVR_DTAQ
   - QIBM_OS400_QZBS_SVR_NETPRT
   - QIBM_OS400_QZBS_SVR_RMTCMD
   - QIBM_OS400_QZBS_SVR_SIGNON
   - QIBM_OS400_QZBS_SVR_FILE
   - QIBM_OS400_QRW_SVR_DDM_DRDA

Changing the authority of the directory that contains the client encryption files

To help you meet the SSL legal responsibilities required when using cryptography algorithms, the directory that contains the files is shipped with public authority *EXCLUDE. You must change the authority of the directory to allow access by only those users authorized to use encryption algorithms.

Use OS/400 object security to control access to the client encryption files by completing the following steps:

1. On your server, enter the following command:

   ```
   wrklnk '/QIBM/ProdData/HTTP/Public/jt400/*'
   ```
2. Select option 9 in the SSL56 or SSL128 directory.
3. Ensure that *PUBLIC has *EXCLUDE authority.
4. Give *RX authority to the directory to individual or groups of users who need access to the SSL files.

   Note: You can not deny access to the SSL files to users that have *ALLOBJ special authority.

Getting and configuring server certificates

Before you get and configure your server certificate, you need to install the following products:

- IBM HTTP Server for iSeries » (5722-DG1)« licensed program
- Base operating system option 34 (Digital Certificate Manager)

The process you follow to get and configure your server certificate depends on the kind of certificate you use:

- If you get a certificate from a trusted authority (such as VeriSign, Inc., or RSA Data Security, Inc.), install the certificate on iSeries then apply it to the host servers.
- If you choose not to use a certificate from a trusted authority, you can build your own certificate to be used on iSeries.

Build the certificate by using Digital Certificate Manager.

1. Create the certificate authority on the iSeries server. See the Information Center topic, Acting as your own CA.
2. Assign which host servers will trust the certificate authority that you created.
3. Create a system certificate from the certificate authority that you created.
4. Assign which host servers will use the system certificate that you created.

Using a certificate from a trusted authority

IBM Toolbox for Java ships a keyring file that supports server certificates from a set of trusted authorities, represented by the following companies:

- IBM World Registry
- Integrion Financial Network
- RSA Data Security, Inc.
- Thawte Consulting
- VeriSign, Inc.

»The keyring file already supports certificates that you get from one of these trusted authorities. All you need to do is get the zip files that contain the encryption algorithms and add it to your CLASSPATH statement.

To use the certificate, complete the following steps:

1. Select the directory where you want to put the zip files.
2. Download the version of SSL that you want to use by copying the zip files into the selected directory:
    - For 56-bit encryption (used with licensed program 5722-CE2), copy /QIBM/ProdData/HTTP/Public/jt400/SSL56/sslightx.zip.
    - For 128-bit encryption (used with licensed program 5722-CE3), copy /QIBM/ProdData/HTTP/Public/jt400/SSL128/sslightu.zip.
3. Add the zip file to your CLASSPATH statement.«

Using a self-signed certificate

When you choose not to use a certificate from a trusted authority, you must download the server certificate (to each server that has a self-signed certificate) so that the IBM Toolbox for Java classes can use it. You also have to get the zip files that contain the encryption algorithms and add it to your CLASSPATH statement.

To use the self-signed certificate, complete the following steps:

1. Select the directory where you want to put the zip files.
2. Download the version of SSL that you want to use by copying both the encryption algorithms and the utilties you need to work with a self-signed certificate:
   - »For 56-bit encryption (used with the licensed programs 5722-CE2) copy /QIBM/ProdData/HTTP/Public/jt400/SSL56/sslightx.zip, cfwk.zip, and ssltools.jar
   - For 128-bit encryption (used with the licensed programs 5722-CE3) copy /QIBM/ProdData/HTTP/Public/jt400/SSL128/sslightu.zip, cfwk.zip, and ssltools.jar.
3. Add ssltools.jar and the zip files to your CLASSPATH statement.«
4. Create a directory on your client named <SSL>\com\ibm\as400\access where <SSL> is the directory where you copied the jar and zip files.
5. »From a command prompt within the <SSL> directory on your client, run the following command:

   ```
   java utilities.KeyringDB com.ibm.as400.access.KeyRing -connect <systemname>:<port>
   ```
   «

   where <port> is the server port of any of the host servers. For example, you can use 9476, which is the default port for the secure sign-on server on the iSeries.

   Note: You must use com.ibm.as400.access.KeyRing because it is the only location where the IBM Toolbox for Java looks for your certificates.
6. Type the number of the Certificate Authority (CA) certificate that you want to add to your server. Be sure to add the CA certificate and not the site certificate.
7. When you are prompted to enter a certificate name, you can type any alphanumeric string.

   »**Note:** You need to run KeyringDB to each server that has a self-signed certificate to add each certificate to the KeyRing class. On each iSeries that you wish to use SSL connections, run the following command to add the certificates:

   ```
   java utilities.KeyringDB com.ibm.as400.access.KeyRing connect <systemname>:<port>
   ```
   «

After completing the above steps, you have finished setting up the self-certificates. You can run the application, after you ensure the following are in your CLASSPATH statement:

- the directory that contains com\ibm\as400\access\KeyRing.class
- jt400.jar
- sslightx.zip or sslightu.zip (depending on which file you downloaded)
- cfwk.zip

Because jt400.jar contains the default copy of KeyRing.class, the directory that contains com\ibm\as400\access\KeyRing.class must be in the CLASSPATH before jt400.jar.

Note: Instead of adding the directory that contains the KeyRing.class file to your CLASSPATH statement, you can replace the old class in jt400.jar with the new KeyRing.class.

»Using SSL to encrypt data between the proxy client and the proxy server

You can use SSL to encrypt data exchanged between the proxy client and proxy server. The files that come with the IBM iSeries Client Encryption licensed program (5722-CE2 or 5722-CE3) are used to encrypt the data. Like IBM Toolbox for Java, these files are platform-independent Java classes that enable the proxy client and proxy server to run on any platform with a Java virtual machine.

Perform the following tasks to encrypt data flowing between the proxy client and the proxy server:

1. Set up the proxy server to handle encrypted data.
2. Set up the proxy client to handle encrypted data.
3. Use the SecureAS400 object to force the IBM Toolbox for Java to encrypt data.

Note: The first two steps only create a secure path between proxy client and the proxy server. The application must use the SecureAS400 object to tell the IBM Toolbox for Java to flow data across the secure path. Using an AS400 object does not encrypt the data, and instead uses the normal path to the server.

If you want to encrypt data flowing between the proxy client and the proxy server, you need only the Java classes that come with the Client Encryption licensed program (5722-CE2 or 5722-CE3). If you want to encrypt data flowing between the proxy server and the iSeries servers, you must also set up encryption between the proxy server and the iSeries server. «

»Setting up SSL on the proxy server

To enable SSL, the proxy server must have a server certificate. Use the IKeyman graphical user interface (GUI) to create the server certificate that the proxy server will use. IKeyman is a GUI tool, so you must run it on a client machine. Once you have created the certificate, you can copy it to the iSeries if your proxy server runs on iSeries.

To set up the proxy server to handle encrypted data, perform the following tasks:

1. Set up your client to run the IKeyman GUI.
2. Create a server certificate for the proxy server.
3. Start the proxy server using the certificate you just created.

Setting up your client to run the IKeyman GUI

The IKeyman GUI is a Java program based on Java Swing 1.1 interfaces. To use IKeyman, your client must be running the Java 1.1.8 JVM (and the Swing 1.1 plug-in) or the Java 2 JVM.

The IKeyman GUI is part of the IBM iSeries Client Encryption licensed program (5722-CE2 or 5722-CE3) in ssltools.jar. The procedure you use to set up your client to use SSL (and to run IKeyman) depends on which version of the licensed program you are running.

Set up your client to use SSL by completing the following steps:

1. Select the directory on your workstation where you want to put the necessary jar and zip files.
2. Copy the necessary files to the selected directory:
    - When using 56-bit encryption, after loading licensed program 5722-CE2 on your iSeries, copy the following files to your workstation:
        - /QIBM/ProdData/http/public/jt400/ssl56/sslightx.zip
        - /QIBM/ProdData/http/public/jt400/ssl56/ssltools.jar
        - /QIBM/ProdData/http/public/jt400/ssl56/cfwk.zip
        - /QIBM/ProdData/http/public/jt400/ssl56/cfwk.sec
    - When using 128-bit encryption, after loading licenced program 5722-CE3 on your iSeries, copy the following files to your workstation:
        - /QIBM/ProdData/http/public/jt400/ssl128/sslightu.zip
        - /QIBM/ProdData/http/public/jt400/ssl128/ssltools.jar
        - /QIBM/ProdData/http/public/jt400/ssl128/cfwk.zip
        - /QIBM/ProdData/http/public/jt400/ssl128/cfwk.sec
3. Add the jar file and the zip files to your CLASSPATH statement. Do not add the .sec file to your CLASSPATH.

    Note: cfwk.zip must be the first item in your classpath.

Creating a server certificate

You use the IKeyman GUI to create a self-signed certificate.

Note: If the IKeyman GUI stops running, check to make sure that cfwk.zip is the first item in your CLASSPATH and that cfwk.sec is in the same directory as the cfwk.zip.

Create a server certificate for the proxy server by completing the following steps:

1. Start the IKeyman GUI by using the following command:

    ```
    java -Dkeyman.javaOnly=true com.ibm.gsk.ikeyman.Ikeyman
    ```
2. From the IKeyman **Key Database File** menu, select **New**.
3. In the **New** dialog, do not alter the the **Key Database Type**, which should be **SSLight key database class**.
4. Type a **File Name** (ProxyServerKeyring.class, for example) or click **Browse** to locate the class file you want to use for the keyring.

    Note: Remember the keyring file name, because you need it to start the secure proxy server.
5. Type a **Location** (path) or accept the default location, which is the current working directory, then click **OK**.
6. In the **Password Prompt** dialog, type a **Password** and **Confirm Password**, then click **OK**. (**Set expiration time** is optional, and you do not need to select it.)

Note: Remember your password, which you need to start the secure proxy server. The key icons in this dialog represent the relative strength of your password. A strong password requires a mix of uppercase and lowercase alphanumeric characters.

7. From the IKeyman **Create** file menu, select **New Self-Signed Certificate**.
8. In the **Create New Self-Signed Certificate** dialog, type a **Key Label** (for example, MyCertificate) and **Organization**.
9. Click the **Country** list to select a country, type a **Validity Period** or accept the default value, then click **OK**.
10. From the IKeyman **Key Database File** menu, select **Close**, then (from the same menu) click **Exit**.

You should now be able to see the keyring that you just created in your current directory.

Starting the proxy server using the new certificate

Before starting the proxy server, make sure that the CLASSPATH for the proxy server contains jt400.jar, sslightx.zip, and the location of the proxy server keyring.

Start the Proxy Server using the certificate you just created. Use the -keyringName and -keyringPassword parameters to pass this information to the proxy server. For example:

```
java com.ibm.as400.access.ProxyServer -keyringName ProxyServerKeyring -keyringPassword pxypswrd
```

«

»Setting up SSL on the proxy client

The following procedure leads you through adding the server certificate to the certificate database on the client, which is stored in a Java .class file. Adding the server certificate to the client is necesssary because the server uses a self-signed certificate.

Set up the proxy client to exchange encrypted data by completing the following tasks:

1. Set up your proxy server to handle encrypted data, then start the proxy server.
2. Set up your client to use SSL.
3. Use KeyringDB to get the server certificate of the proxy server.
4. Set up the client to use the updated KeyRing.class file.
5. Set the secure proxy settings on the client.

Setting up your client to use SSL

The tool that downloads the certificate (KeyringDB) is a Java program. To use this program, you must be running Java 1.1.8 or Java 2 JVM on your client. KeyringDB is part of the IBM iSeries Client Encryption licensed program (5722-CE2 or 5722-CE3) in ssltools.jar. The procedure you use to set up your client to use SSL depends on which version of the licensed program you are running.

After setting up your proxy server, set up your client to use SSL by completing the following steps:

1. Select the directory on your workstation where you want to put the necessary jar and zip files.
2. Copy the necessary files to the selected directory:
   - When using 56-bit encryption, after loading licenced program 5722-CE2 on your iSeries, copy the following files to your workstation:
     - /QIBM/ProdData/http/public/jt400/ssl56/sslightx.zip
     - /QIBM/ProdData/http/public/jt400/ssl56/ssltools.jar
     - /QIBM/ProdData/http/public/jt400/ssl56/cfwk.zip
   - When using 128-bit encryption, after loading licenced program 5722-CE3 on your AS/400, copy the following files to your workstation:
     - /QIBM/ProdData/http/public/jt400/ssl128/sslightu.zip
     - /QIBM/ProdData/http/public/jt400/ssl128/ssltools.jar
     - /QIBM/ProdData/http/public/jt400/ssl128/cfwk.zip
3. Add the jar file and the zip files to your CLASSPATH statement.
4. Create a directory on your client named <SSL>\com\ibm\as400\access where <SSL> is the directory where you copied the jar and zip files.

Adding the server certificate for the proxy server

KeyringDB creates a new KeyRing.class file that contains the server certificate and puts it in the com\ibm\as400\access subdirectory off the current directory.

Use the KeyringDB tool to add the server certificate to KeyRing.class by completing the following steps:

1. From the directory where you put the jar and zip files, run the following command:

   ```
   java utilities.KeyringDB com.ibm.as400.access.KeyRing -connect proxyServerName:port
   ```

   where:

   - *proxyServerName* is the name of the machine running the proxy server
   - *port* is the port that the secure proxy server is listening to (3471 by default)

     For example:

     ```
     java utilities.KeyringDB com.ibm.as400.access.KeyRing -connect myProxyServer:3471
     ```
2. When asked which certificate to use, choose site certificate 0.
3. When you are prompted to enter a certificate name, you can type any alphanumeric string.

Setting up the client to use the updated KeyRing.class file

The jt400Proxy.jar file contains KeyRing.class. To set up the client to use the updated KeyRing.class file, ensure the following are in your CLASSPATH statement:

- the directory that contains com\ibm\as400\access\KeyRing.class
- jt400Proxy.jar
- sslightx.zip or sslightu.zip (depending on which file you downloaded)
- cfwk.zip

Because jt400Proxy.jar contains the default copy of KeyRing.class, the directory that contains com\ibm\as400\access\KeyRing.class must be in the CLASSPATH before jt400Proxy.jar.

Note: Instead of adding the directory that contains the KeyRing.class file to your CLASSPATH statement, you can add the new KeyRing.class to your jt400Proxy.jar file. Adding the new KeyRing.class file to jt400Proxy.jar overwrites the old version.

Setting the secure proxy settings on the client

To tell the proxy client to communicate with the proxy server across a secure connection, set the following system properties:

```
com.ibm.as400.access.AS400.proxyServer=proxyServer
```

where *proxyServer* is the name of the machine that is running the proxy server

```
com.ibm.as400.access.SecureAS400.proxyEncryptionMode=mode
```

where *mode* is one of the following integers:

- 1 to encrypt between proxy client and proxy server
- 2 to encrypt between proxy server and the iSeries server
- 3 to encrypt between proxy client and proxy server, and between proxy server and the iSeries server

  For example, the following command starts an application using SSL:

```
java -Dcom.ibm.as400.access.AS400.proxyServer=myProxyServer
   -Dcom.ibm.as400.access.SecureAS400.proxyEncryptionMode=1   myApplication
```

«

Authentication Services

Classes are provided by the IBM Toolbox for Java that interact with the security services provided by OS/400. Specifically, support is provided to authenticate a user identity, someitmes referred to as a *principal*, and password against the OS/400 user registry. A credential representing the authenticated user can then be established. You can use the credential to alter the identity of the current OS/400 thread to perform work under the authorities and permissions of the authenticated user. In effect, this swap of identity results in the thread acting as if a signon was performed by the authenticated user.

Note: The services to establish and swap credentials are only supported for AS/400 systems at release V5R1M0 or greater.

Overview of support provided

The AS400 object now provides authentication for a given user profile and password against the AS/400 system. You can also retrieve credentials representing authenticated user profiles and passwords for the system. To do this, you use the getProfileToken() methods to retrieve instances of the ProfileTokenCredential class. Think of profile tokens as a representation of an authenticated user profile and password for a specific AS/400 system. Profile tokens expire based on time, up to one hour, but can be refreshed in certain cases to provide an extended life span.

Setting thread identities

You can establish a credential on either a remote or local context. Once created, you can serialize or distribute the credential as required by the calling application. When passed to a running process on the associated AS/400, a credential can be used to modify or *swap* the OS/400 thread identity and perform work on behalf of the previously authenticated user.

A practical application of this support might be in a two tier application, with authentication of a user profile and password being performed by a graphical user interface on the first tier (i.e. a PC) and work being performed for that user on the second tier (the AS/400). By utilizing ProfileTokenCredentials, the application can avoid directly passing user IDs and passwords over the network. The profile token can then be distributed to the program on the second tier, which can perform the *swap()* and operate under the OS/400 authorities and permissions assigned to the user.

Note: While inherently more secure than passing a user profile and password due to limited life span, profile tokens should still be considered sensitive information by the application and handled accordingly. Since the token represents an authenticated user and password, it could potentially be exploited by a hostile application to perform work on behalf of that user. It is ultimately the responsibility of the application to ensure that credentials are accessed in a secure manner.

Example

Refer to this code for an example of how to use a profile token credential to swap the OS/400 thread identity and perform work on behalf of a specific user.

Security example

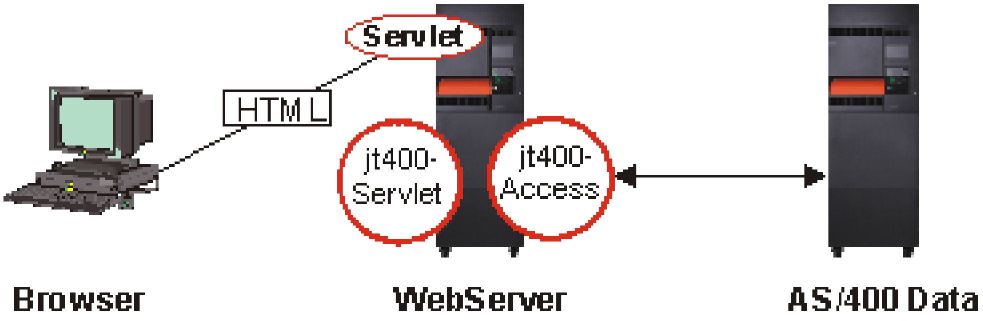
The following code example shows you how to use a profile token credential to swap the OS/400 thread identity and perform work on behalf of a specific user:

```
// Prepare to work with the local AS/400 system.
AS400 system = new AS400("localhost", "*CURRENT", "*CURRENT");

// Create a single-use ProfileTokenCredential with a 60 second timeout.
// A valid user ID and password must be substituted.
ProfileTokenCredential pt = new ProfileTokenCredential();
pt.setSystem(system);
pt.setTimeoutInterval(60);
pt.setTokenType(ProfileTokenCredential.TYPE_SINGLE_USE);
pt.setToken("USERID", "PASSWORD");

// Swap the OS/400 thread identity, retrieving a credential to
// swap back to the original identity later.
AS400Credential cr = pt.swap(true);

// Perform work under the swapped identity at this point.

// Swap back to the original OS/400 thread identity.
cr.swap();

// Clean up the credentials.
cr.destroy();
pt.destroy();
```

Servlet classes

The servlet classes that are provided with IBM Toolbox for Java work with the access classes, which are located on the webserver, to give you access to information located on the iSeries server. You decide how to use the servlet classes to assist you with your own servlet projects.

The following diagram shows how the servlet classes work between the browser, webserver, and iSeries data. A browser connects to the webserver that is running the servlet. jt400Servlet.jar and jt400Access.jar files reside on the webserver because the servlet classes use some of the access classes to retrieve the data and the HTML classes to present the data. The webserver is connected to the iSeries system where the data is.



There are »four« types of servlet classes included with IBM Toolbox for Java:

- »Authentication classes «
- RowData classes
- RowMetaData classes
- Converter classes

Note: The jt400Servlet.jar file includes both the HTML and Servlet classes. You must update your CLASSPATH to point to both the jt400Servlet.jar and the jt400Access.jar if you want to use classes in the com.ibm.as400.util.html and com.ibm.as400.util.servlet packages.

For more information about servlets in general, see the reference section.

Authentication classes

Two classes in the servlet package perform authentication for servlets: AuthenticationServlet and AS400Servlet.

AuthenticationServlet class

AuthenticationServlet is an HttpServlet implementation that performs basic authentication for servlets. Subclasses of AuthenticationServlet should override one or more of the following methods:

- Override the validateAuthority() method to perform the authentication (required)
- Override the bypassAuthentication() method so that the subclass authenticates only certain requests
- Override the postValidation() method to allow additional processing of the request after authentication

The AuthenticationServlet class provides methods that allow you to:

- Initialize the servlet
- Get the authenticated user ID
- Set a user ID after bypassing authentication
- Log exceptions and messages

AS400Servlet class

The AS400Servlet class is an abstract subclass of AuthenticationServlet that represents an HTML servlet. You can use a connection pool to share connections and manage the number of connections a servlet user can have to the server.

The AS400Servlet class provides methods that allow you to:

- Validate user authority (by overriding the validateAuthority() method of the AuthenticationServlet class)
- Connect to a system
- Get and return connection pool objects to and from the pool
- Close a connection pool
- Get and set the HTML document head tags
- Get and set the HTML document end tags

For more information about servlets in general, see the reference section.

RowData class

The RowData class is an abstract class that provides a way to describe and access a list of data.

The RowData classes allow you to:

- Get and set the current position
- Get the row data at a given column using the getObject() method
- Get the meta data for the row
- Get or set the properties for an object at a given column
- Get the number of rows in the list using the length() method.

There are »four « main classes within the RowData class that have many of the same properties:

- ListRowData
- RecordListRowData
- »ResourceListRowData«
- SQLResultSetRowData

RowData position

There are several methods that allow you to get and set the current position within a list

Set methods		Get methods
absolute()	next()	getCurrentPosition()
afterLast()	previous()	isAfterLast()
beforeFirst()	relative()	isBeforeFirst()
first()		isFirst()
last()		isLast()

ListRowData class

The ListRowData class allows you to do the following:

- Add and remove rows to and from the result list.
- Get and set the row
- Get information about the list's columns with the getMetaData() method
- Set column information with the setMetaData() method

The ListRowData class represents a list of data. ListRowData can represent many types of information, including the following, through IBM Toolbox for Java access classes:

- A directory in the integrated file system
- A list of jobs
- A list of messages in a message queue
- A list of users
- A list of printers
- A list of spooled files

This example shows you how ListRowData and HTMLTableConverter work. It shows the java code, HTML code, and HTML look and feel.

RecordListRowData class

The RecordListRowData class allows you to do the following:

- Add and remove rows to and from the record list.
- Get and set the row
- Set the record format with the setRecordFormat method
- Get the record format.

The RecordListRowData class represents a list of records. A record can be obtained from the AS/400 system in different formats, including:

- A record to be written to or read from an AS/400 file
- An entry in a data queue
- The parameter data from a program call
- Any data return that needs to be converted between AS/400 format and Java format

This example shows you how RecordListRowData and HTMLTableConverter work. It shows the java code, HTML code, and HTML look and feel.

SQLResultSetRowData class

The SQLResultSetRowData class represents an SQL result set as a list of data. This data is generated by an SQL statement through JDBC. With methods provided, you can get and set the result set metadata.

This example shows you how ListRowData and HTMLTableConverter work. It shows the java code, HTML code, and HTML look and feel.

RowMetaData classes

The RowMetaData class defines an interface that you use to find out information about the columns of a RowData object.

With the RowMetaData classes you can do the following:

- Get the number of columns
- Get the name, type, or size of the column
- Get or set the column label
- Get the precision or scale of the column data
- Determine if the column data is text data

There are three main classes that implement the RowMetaData class. These classes provide all the RowMetaData functions listed above in addition to having their own specific functions:

- ListMetaData
- RecordFormatMetaData
- SQLResultSetMetaData

ListMetaData class

The ListMetaData lets you get information about and change settings for the columns in a ListRowData class. It uses the setColumns() method to set the number of columns, clearing any previous column information. Alternatively, you can also pass the number of columns when you set the constructor's parameters.

This example shows you how ListMetaData, ListRowData and HTMLTableConverter work. It shows the java code, HTML code, and HTML look and feel.

RecordFormatMetaData class

The RecordFormatMetaData makes use of the IBM Toolbox for Java RecordFormat class. It allows you to provide the record format when you set the constructor's parameters or use the get and set methods to access the record format.

The following example shows you how to create a RecordFormatMetaData object:

```
// Create a RecordFormatMetaData object from a sequential file's record format.
RecordFormat recordFormat = sequentialFile.getRecordFormat();
RecordFormatMetaData metadata = new RecordFormatMetaData(recordFormat);

// Display the file's column names.
int numberOfColumns = metadata.getColumnCount();
for (int column=0; column < numberOfColumns; column++)
{
   System.out.println(metadata.getColumnName(column));
}
```

SQLResultSetMetaData

The SQLResultSetMetaData returns information about the columns of an SQLResultSetRowData object. You can either provide the result set when you set the constructor's parameters or use the get and set methods to access the result set meta data.

The following example shows you how to create an SQLResultSetMetaData object:

```
// Create an SQLResultSetMetaData object from the result set's metadata.
SQLResultSetRowData rowdata = new SQLResultSetRowData(resultSet);
SQLResultSetMetaData sqlMetadata  = rowdata.getMetaData();

// Display the column precision for non-text columns.
String name = null;
int numberOfColumns = sqlMetadata.getColumnCount();
for (int column=0; column < numberOfColumns; column++)
{
   name = sqlMetadata.getColumnName(column);
   if (sqlMetadata.isTextData(column))
   {
      System.out.println("Column: " + name + " contains text data.");
   }
   else
   {
      System.out.println("Column: " + name + " has a precision of " + sqlMetadata.getPrecision(column));
   }
}
```

Converter classes

You use the converter classes to convert row data into formatted string arrays. The result is in HTML format and ready for presentation on your HTML page. The following classes take care of the conversion for you:

- StringConverter
- HTMLFormConverter
- HTMLTableConverter

StringConverter class

The StringConverter class is an abstract class that represents a row data string converter. It provides a convert() method to convert row data. This returns a string array representation of that row's data.

HTMLFormConverter class

The HTMLFormConverter classes extends StringConverter by providing an additional convert method called convertToForms(). This method converts row data into an array of single-row HTML tables. You can use these table tags to display the formatted information on a browser.

You can tailor the appearance of the HTML form by using the various get and set methods to view or change the attributes of the form. For example, some of the attributes that you can set include:

- Alignment
- Cell spacing
- Header hyperlinks
- Width

Examples

Example: Using HTMLFormConverter. (Compile and run this example with a webserver running)

»Example: Presenting ResourceList in a servlet.«

HTMLTableConverter class

The HTMLTableConverter class extends StringConverter by providing a convertToTables() method. This method converts row data into an array of HTML tables that a servlet can use to display the list on a browser.

You can use the getTable() and setTable() methods to choose a default table that will be used during conversion. You can set table headers within the HTML table object or you can use the meta data for the header information by setting setUseMetaData() to true.

The setMaximumTableSize() method allows you to limit the number of rows in a single table. If the row data does not all fit within the specified size of table, the converter will produce another HTML table object in the output array. This will continue until all row data has been converted.

Examples

Below are several examples that illustrate how to use the HTMLTableConverter class:

- Example: Using ListRowData
- Example: Using RecordListRowData
- Example: Using SQLResultSetRowData
- »Example: Presenting ResourceList in a servlet«

Utility classes

Utility classes are classes that help you do specific tasks. IBM Toolbox for Java offers the following utilities:

- AS400ToolboxInstaller - Allows you to install and update IBM Toolbox for Java classes on the client. This function is available both as a Java program and has an application programming interface(API).
- AS400ToolboxJarMaker - Generates a faster loading IBM Toolbox for Java JAR file by creating a smaller JAR file from a larger one, or by selectively unzipping a JAR file to gain access to the individual content files.
- JavaApplicationCall - Allows you to run a Java program on iSeries from a command line prompt or graphical user interface.
- »JPing - Allows you to query a server to find out which services are active. You can also specify if you want to ping the SSL ports.«

Client installation and update classes

The IBM Toolbox for Java classes can be referenced at their location in the integrated file system on the AS/400. Because program temporary fixes (PTFs) are applied to this location, Java programs that access these classes directly on the AS/400 system automatically receive these updates. Accessing the classes from the AS/400 does not work for every situation, specifically those listed below:

- If a low-speed communication link connects AS/400 and the client, the performance of loading the classes from the AS/400 may be unacceptable.
- If Java applications use the CLASSPATH environment variable to access the classes on the client file system, you need AS/400 Client Access to redirect file system calls to the AS/400. It may not be possible for AS/400 Client Access to reside on the client.

In these cases, installing the classes on the client is a better solution. The AS400ToolboxInstaller class provides client installation and update functions to manage IBM Toolbox for Java classes when they reside on a client.

Using the AS400ToolboxInstaller

You can invoke the AS400ToolboxInstaller object in the following ways:

- From within your program
- From a command line

If your Java program uses IBM Toolbox for Java functions, you can include the AS400ToolboxInstaller class as a part of your program. When the Java program is installed or first run, it can use the AS400ToolboxInstaller class to install the IBM Toolbox for Java classes on the client. When the Java program is restarted, it can use the AS400ToolboxInstaller to update the classes on the client.

Note: If you are using the V3R2 or V3R2M1 version of the IBM Toolbox for Java and you want to upgrade to V4R2 or a later version, you must use a V4R2 or later level of the AS400ToolboxInstaller class. You must use this level to ensure that machine readable information (MRI) stored in .property files in earlier releases of the IBM Toolbox for Java is properly replaced by .class files used in later releases.

The AS400ToolboxInstaller class copies files to the client's local file system. This class may not work in an applet; many browsers do not allow a Java program to write to the local file system.

Embedding the AS400ToolboxInstaller class in your program

The AS400ToolboxInstaller class provides the application programming interfaces (APIs) that are necessary to install, uninstall, and update IBM Toolbox for Java classes from within the program on the client.

Use the install() method to install or update the IBM Toolbox for Java classes. To install or update, provide the source and target path, and the name of the package of classes in your Java program. The source URL points to the location of the control files on the server. The directory structure is copied from the server to the client.

The install() method only copies files; it **does not** update the CLASSPATH environment variable. If the install() method is successful, the Java program can call the getClasspathAdditions() method to determine what must be added to the CLASSPATH environment variable.

The following example shows how to use the AS400ToolboxInstaller class to install files from an AS/400 called "mySystem" to directory "jt400" on drive d:, and then how to determine what must be added to the CLASSPATH environment variable:

```
                      // Install the IBM Toolbox for Java
                      // classes on the client.
     URL sourceURL = new URL("http://mySystem.myCompany.com/QIBM/ProdData/HTTP/Public/jt400/");

     if (AS400ToolboxInstaller.install(
             "ACCESS",
             "d:\\jt400",
             sourceURL))

     {
                      // If the IBM Toolbox for Java classes were installed
                      // or updated, find out what must be added to the
                      // CLASSPATH environment variable.
        Vector additions = AS400ToolboxInstaller.getClasspathAdditions();

                      // If updates must be made to CLASSPATH
        if (additions.size() > 0)
        {
```

```
                        // ... Process each classpath addition
          }
      }

                        // ... Else no updates were needed.
```

Use the isInstalled() method to determine if the IBM Toolbox for Java classes are already installed on the client. Using the isInstalled() method allows you to determine if you want to complete the install now or postpone it to a more convenient time.

The install() method both installs and updates files on the client. A Java program can call the isUpdateNeeded() method to determine if an update is needed before calling install().

Use the unInstall() method to remove the IBM Toolbox for Java classes from the client. The unInstall method only removes files; the CLASSPATH environment variable is not changed. Call the getClasspathRemovals() method to determine what can be removed from the CLASSPATH environment variable.

For more examples of how to install and update the AS400ToolboxInstaller class within a program on the client workstation, refer to the Install/Update example.

Running the AS400ToolboxInstaller class from the command line

The AS400ToolboxInstaller class can be used as a stand-alone program, run from the command line. Running the AS400ToolboxInstaller from the command line means you do not have to write a program. Instead, you run it as a Java application to install, uninstall, or update the IBM Toolbox for Java classes.

Specifying the appropriate install, uninstall, or compare option, invoke the AS400ToolboxInstaller class with the following command:

```
java utilities.AS400ToolboxInstaller [options]
```

The **-source** option indicates where the IBM Toolbox for Java classes can be found and **-target** indicates where the IBM Toolbox for Java classes are to be stored on the client.

Options are also available to install the entire toolbox or just certain functions. For instance, an option exists to install just the proxy classes of IBM Toolbox for Java.

AS400ToolboxJarMaker

While the JAR file format was designed to speed up the downloading of Java program files, the AS400ToolboxJarMaker class generates an even faster loading IBM Toolbox for Java JAR file through its ability to create a smaller JAR file from a larger one.

Also, the AS400ToolboxJarMaker class can unzip a JAR file for you to gain access to the individual content files for basic use.

Flexibility of AS400ToolboxJarMaker

All of the AS400ToolboxJarMaker functions are performed with the JarMaker class and the AS400ToolboxJarMaker subclass:

- The generic JarMaker tool operates on any JAR or Zip file; it splits a jar file or reduces the size of a jar file by removing classes that are not used.
- The AS400ToolboxJarMaker customizes and extends JarMaker functions for easier use with IBM Toolbox for Java JAR files.

According to your needs, you can invoke the AS400ToolboxJarMaker methods from within your own Java program or from a command line. Call AS400ToolboxJarMaker from the command line by using the following syntax:

```
java utilities.JarMaker [options]
```

where

- options = one or more of the available options

For a complete set of options available to run at a command line prompt, see the following:

- Options for the JarMaker base class
- Extended options for the AS/400ToolboxJarMaker subclass

Using AS400ToolboxJarMaker

Uncompressing a JAR file

Suppose you wanted to uncompress just one file bundled within a JAR file. AS400ToolboxJarMaker allows you to expand the file into one of the following:

- Current directory (extract(jarFile))
- Another directory (extract(jarFile, outputDirectory))

For example, with the following code, you are extracting AS400.class and all of its dependent classes from jt400.jar:

```
java utilities.AS400ToolboxJarMaker -source jt400.jar
    -extract outputDir
    -requiredFile com/ibm/as400/access/AS400.class
```

Splitting up a single JAR file into multiple, smaller JAR files

Suppose you wanted to split up a large JAR file into smaller JAR files, according to your preference for maximum JAR file size. AS400ToolboxJarMaker, accordingly, provides you with the split(jarFile, splitSize) function.

In the following code, jt400.jar is split into a set of smaller JAR files, none larger than 300K:

```
java utilities.AS400ToolboxJarMaker -split 300
```

Removing unused files from a JAR file

With AS400ToolboxJarMaker, you can exclude any IBM Toolbox for Java files not needed by your application by selecting only the IBM Toolbox for Java components, languages, and CCSIDs that you need to make your application run. AS400ToolboxJarMaker also provides you with the option of including or excluding the JavaBean files associated with the components you select.

For example, the following command creates a JAR file that contains only those IBM Toolbox for Java classes needed to make the CommandCall and ProgramCall components of the IBM Toolbox for Java work:

```
java utilities.AS400ToolboxJarMaker -component CommandCall,ProgramCall
```

Additionally, if it is unnecessary to convert text strings between Unicode and the double byte character set (DBCS) conversion tables, you can create a 400K byte smaller JAR file by omitting the unneeded conversion tables with the -ccsid option:

```
java utilities.AS400ToolboxJarMaker -component CommandCall,ProgramCall -ccsid 61952
```

Note: Conversion classes are not included with the program call classes. When including program call classes, you must also explicitly include the conversion classes used by your program by using the -ccsid option.

RunJavaApplication

The RunJavaApplication and VRunJavaApplication classes are utilities to run Java programs on the Java virtual machine for AS/400. Unlike JavaApplicationCall and VJavaApplicationCall classes that you call from your Java program, RunJavaApplication and VRunJavaApplication are complete programs.

The RunJavaApplication class is a command line utility. It lets you set the environment (CLASSPATH and properties, for example) for the Java program. You specify the name of the Java program and its parameters, then you start the program. Once started, you can send input to the Java program which it receives via standard input. The Java program writes output to standard output and standard error.

The VRunJavaApplication utility has the same capabilities. The difference is VJavaApplicationCall uses a graphical user interface while JavaApplicationCall is a command line interface.

»JPing

The JPing class is a command line utility that allows you to query your servers to see which services are running and which ports are in service. To query your servers from within a Java application, use the AS400JPing class.

See the JPing javadoc for more information about using JPing from within your Java application.

Call JPing from the command line by using the following syntax:

```
java utilities.JPing System [options]
```

where:

- System = the iSeries server that you want to query
- [options] = one or more of the available options

Options

You can use one or more of the following options. For options that have abbreviations, the abbreviation is listed in parenthesis.

-help (-h or **-?)**

Displays the help text.

-service *OS/400_Service* **(-s** *OS/400_Service***)**

Specifies one specific service to ping. The default action is to ping all services. You can use this option to specify one of the following services: as-file, as-netprt, as-rmtcmd, as-dtaq, as-database, as-ddm, as-central, and as-signon.

-ssl

Specifies whether or not to ping the ssl ports. The default action is not to ping the ssl ports.

-timeout (-t)

Specifies the timeout period in milliseconds. The default setting is 20000, or 20 seconds.

Example: Using JPing from the command line

For example, use the following command to ping the as-dtaq service, including ssl ports, with a timeout of 5 seconds:

```
java utilities.JPing myServer -s as-dtaq -ssl -t 5000«
```

Tutorial: Detailed code examples

Use the tutorials provided for ideas on how to code your own Java programs using the IBM Toolbox for Java classes. The examples in this section are similar to the other examples provided in the IBM Toolbox for Java information. However, each example in this section contains additional information explaining the key lines in the code. Click the Notes image to display the detailed explanation. (You may have to page down in the example before you see the Notes image.)

Note: To take advantage of the JavaScript coding that enables the pop-up windows containing the detailed description, your browser must support JavaScript (Internet Explorer 4.0 and Netscape 3.0 and 4.0 support JavaScript). If you do not have this level of browser, the detailed descriptions are included in footnotes at the bottom of the page.

The following classes have a tutorial provided:

- Message queue
- GUI
- Record-level access
- JDBC

Code Examples

The following table is the entry point for all of the **examples** used throughout the IBM Toolbox for Java information. The examples from the Tutorial section are not included below.

Access Classes	GUI Classes
Utility Classes	Beans
PCML	Graphical Toolbox
Servlet classes	HTML classes
Security classes	Resource classes
Tips for Programming	

Code examples from the access classes

This section lists the code examples that are provided throughout the documentation of the access classes.

»AS400JPing

- Example: Using AS400JPing within a Java program to ping the iSeries Remote Command Service«

»BidiTransform

- Example: Using the AS400BidiTransform class to transform bidirectional text«

Command call

- Example: Using the CommandCall class to run a command on iSeries
- Example: Using CommandCall to prompt for the name of the iSeries, command to run, and print the result

»Connection pooling

- Example: Using an AS400ConnectionPool to create connections to iSeries«

Data area

- Example: Creating and writing to a decimal data area

Data conversion and description

- Example: How to use RecordFormat and Record with the data queue classes
- »Example: Using the FieldDescription, RecordFormat, and Record classes«

Data queues

- Example: Create a DataQueue object, read data, and disconnect

Digital certificate

- Example: List the digital certificates that belong to a user

»EnvironmentVariable

- Example: Creating, setting, and getting environment variables«

Exceptions

- Example: Catching a thrown exception, retrieving the return code, and displaying the exception text

FTP

- Example: Copy a set of files from a directory on a server with FTP class
- Example: Copy a set of files from a directory on a server with AS400FTP subclass

Integrated file system

- Example: Using the integrated file system classes to copy a file from one directory to another on the iSeries
- Example: How to use IFSJavaFile instead of java.io.File
- Example: Using the integrated file system classes to list the contents of a directory on the iSeries
- »Example: Using the IFSFile listFiles() method to list the contents of a directory«

JavaApplicationCall

- Example: Running a program on the iSeries from the client that outputs "Hello World!"

JDBC

- Example: Using the JDBC driver to create and populate a table
- Example: Using the JDBC driver to query a table and output its contents
- »Example: Using the JDBC 2.0 Optional Package extensions to create and update a table«

Jobs

- Example: Retrieving and changing job information using the cache
- Example: Listing all active jobs
- Example: Printing all of the messages in the job log for a specific user
- Example: Listing the job identification information for a specific user
- Example: Getting a list of jobs on the iSeries and output the job's status followed by a job identifier
- Example: Displaying messages in the job log for a job that belongs to the current user

Message queue

- Example: How to use the message queue object
- Example: Printing the contents of the message queue
- Example: How to retrieve and print a message
- Example: Listing the contents of the message queue

»NetServer

- Example: Using a NetServer object to change the name of the NetServer«

Print

- Example: Creating a spooled file on an iSeries from an input stream
- Example: Generating an SCS data stream using the SCS3812Writer class
- Example: Reading an existing iSeries spooled file
- Example: Asynchronously listing all spooled files on a system and how to use the PrintObjectListListener interface to get feedback as the list is being built
- Example: Asynchronously listing all spooled files on a system *without* using the PrintObjectListListener interface
- Example: Synchronously listing all spooled files on a system

Permission

- Example: Set the authority of an AS400 object

Program call

- Example: Using the ProgramCall class
- Example: Passing parameter data with a Programparameter object

QSYSObjectPathName

- Example: Building an integrated file system name
- Example: Using toPath() to build an AS400 object name
- Example: How to use the QSYSObjectPathName class to parse the integrated file system path name

Record-level access

- Example: Accessing an iSeries file sequentially
- Example: Using the record-level access classes to read an iSeries file
- Example: Using the record-level access classes to read records by key from an iSeries file
- »Example: Using the LineDataRecordWriter class«

Service program call

- Example: Using ServiceProgramCall to call a procedure.

System Status

- Example: Use caching with the SystemStatus class

System Pool

- Example: Set the maximum faults size for a system pool

System Values

- Example: How to use the SystemValue and SystemValueList classes

Trace

- Example: Using the setTraceOn() method
- Example: Preferred way of using trace

User Groups

- Example: Retrieving a list of users
- Example: Listing all the users of a group

User Space

- Example: How to create a user space

Code examples using the GUI classes

This section lists the code examples that are provided throughout the documentation of the graphical user interface (GUI) classes.

AS400Panes

- Example: Creating an AS400DetailsPane to present the list of users defined on the systemAS400DetailsPane
- Example: Loading the contents of a details pane before adding it to a frame
- Example: Using an AS400 ListPane to present a list of users
- Example: Using an AS400DetailsPane to display messages returned from a command call
- Example: Using an AS400TreePane to display a tree view of a directory
- Example: Using an AS400ExplorerPane to present various print resources

Command call

- Example: Creating a CommandCallButton
- Example: Adding the ActionCompletedListener to process all iSeries messages that a command generates
- Example: Using the CommandCallMenuItem

Data queues

- Example: Creating a DataQueueDocument
- Example: Using a DataQueueDocument

Error events

- Example: Handling error events
- Example: Defining an error listener
- Example: Using a customized handler to handle error events

JDBC

- Example: Using the JDBC driver to create and populate a table
- Example: Using the JDBC driver to query a table and output its contents
- »Example: Creating an AS400JDBCDataSourcePane«

Jobs

- Example: Creating a VJobList and presenting the list in an AS400ExplorerPane
- Example: Presenting a list of jobs in an explorer pane

Program call

- Example: Creating a ProgramCallMenuItem
- Example: Processing all program generated iSeries messages
- Example: Adding two parameters
- Example: Using a ProgramCallButton in an application

Record-level access

- Example: Creating a RecordListTablePane object to display all records less than or equal to a key

SpooledFileViewer

- Example: Creating a Spooled File Viewer to view a spooled file previously created on an iSeries

System Values

- Example: Creating a system value GUI using the AS400Explorer pane

Users and groups

- Example: Creating a VUserList with the AS400DetailsPane
- Example: Using an AS400ListPane to create a list of users for selection

Code examples from the utility classes

This section lists the code examples that are provided throughout the documentation of the utility classes.

IBM Toolbox Installer

- Example: Using the AS400ToolboxInstaller class
- Example: Installing the AS/400Toolbox with the AS400ToolboxInstaller
- Example: Installing the ACCESS package from the command line.
- Example: Working with the Graphical Toolbox class from the command line.

JarMaker

- Example: Extracting AS400.class and all its dependent classes from jt400.jar
- Example: Splitting jt400.jar into a set of 300K files
- Example: Removing unused files from a JAR file
- Example: Creating a 400K byte smaller JAR file by omitting the conversion tables with the -ccsid parameter

Examples: JavaBeans

This section lists the code examples that are provided throughout the documentation of the bean topics.

- Example: Using listeners to print a comment when you connect and disconnect to the system and run commands
- Example: Using applets and IBM VisualAge for Java to create buttons that run commands

Examples from the servlet classes

The following examples show you some of the ways that you can use the servlet classes:

- Example: Using the ListRowData class
- Example: Using the RecordListRowData class
- Example: Using the SQLResultSetRowData class
- Example: Using the HTMLFormConverter class
- Example: Using the ListMetaData class
- Example: Using the SQLResultSetMetaData class
- ≫Example: Presenting a resource list in a servlet≪

You can also use the servlet and HTML classes together, like in this example.

Examples from the HTML classes

The following examples show you some of the ways that you can use the HTML classes:

- »Example: Using the BidiOrdering class«
- »Example Creating HTMLAlign objects«
- Example: Using the HTML form classes
- Form input class examples:
    - Example: Creating a ButtonFormInput object
    - Example: Creating a FileFormInput object
    - Example: Creating a HiddenFormInput object
    - Example: Creating an ImageFormInput object
    - Example: Creating a ResetFormInput object
    - Example: Creating a SubmitFormInput object
    - Example: Creating a TextFormInput object
    - Example: Creating a PasswordFormInput object
    - Example: Creating a RadioFormInput object
    - Example: Creating a CheckboxFormInput object
- »Example Creating HTMLHeading objects«
- Example: Using the HTMLHyperlink class
- »HTMLList examples
    - Example: Creating ordered lists
    - Example: Creating unordered lists
    - Example: Creating nested lists«
- »Example: Creating HTMLMeta tags«
- »Example: Creating HTMLParameter tags«
- »Example: Creating HTMLServlet tags«
- Example: Using the HTMLText class
- »HTMLTree examples
    - Example: Using the HTMLTree class
    - Example: Creating a traversable IFS tree«
- Layout form classes:
    - Example: Using the GridLayoutFormPanel class
    - Example: Using the LineLayoutFormPanel class
- Example: Using the TextAreaFormElement class
- Example: Using the LabelFormOutput class
- Example: Using the SelectFormElement class
- Example: Using the SelectOption class
- Example: Using the RadioFormInputGroup class
- Example: Using the RadioFormInput class

- Example: Using the HTMLTable classes
  - Example: Using the HTMLTableCell class
  - Example: Using the HTMLTableRow class
  - Example: Using the HTMLTableHeader class
  - Example: Using the HTMLTableCaption class

You can also use the HTML and servlet classes together, like in this example.

Tips for programming examples

This section lists the code examples that are provided throughout the documentation of the managing connections topic.

Managing connections

- Example: Making a connection to the iSeries server with a CommandCall object
- Example: Making two connections to the iSeries server with a CommandCall object
- Example: Creating CommandCall and IFSFileInputStream objects with an AS400 object
- » Example: Using AS400ConnectionPool to preconnect to the iSeries server«
- » Example: Using AS400ConnectionPool to preconnect to a specific service on the iSeries server, then reuse the connection«

Starting and ending connections

- Example: How a Java program preconnects to an iSeries server
- Example: How a Java program disconnects from an iSeries server
- Example: How a Java program disconnects and reconnects to the iSeries server with disconnectService() and run()
- Example: How a Java program disconnects from the iSeries server and fails to reconnect

Exceptions

- Example: Using exceptions

Error events

- Example: Handling error events
- Example: Defining an error listener
- Example: Using a customized handler to handle error events

Trace

- Example: Using trace
- Example: Using setTraceOn()
- Example: Using component trace

Optimization

- Example: How to create two AS400 objects
- Example: How an AS400 object is used to represent a second AS/400 system

Install and update

- Example: Using the AS400Toolbox Installer class

Tips for programming

This section features pointers for programming with the AS/400 Toolbox for Java. Select the links below to view the tips.

1. Find out how to properly shut down your Java program.
2. Use integrated file system path names in your programs. This section covers integrated file system path names, parameters, and special values.
3. Follow these tips on managing connections to an AS/400. See how to use the AS400 class to start and end socket connections.

   NOTE: If your application is an Enterprise JavaBean, consider turning off IBM Toolbox for Java thread support. Your application will be slower but will be compliant with the Enterprise JavaBean specification.
4. Read about running IBM Toolbox for Java classes on the Java virtual machine for AS/400. This section covers how to best access AS/400 resources, how to run the classes, and what sign-on factors to consider.
5. When programming with the IBM Toolbox for Java access classes, use the exception classes to handle errors.
6. When programming with the graphical user interface (GUI) classes, use the error events classes to handle errors.
7. See how to use the Trace class in your programs.
8. Find out how to optimize your program for better performance.
9. Read about the performance improvements you can get when using the Java virtual machine for AS/400.
10. Use the install and update section for information about the AS400ToolboxInstaller class and managing IBM Toolbox for Java classes on a client.
11. Read about IBM Toolbox for Java and Java national language support.
12. See these resources for IBM Toolbox for Java Service and support.
13. Use the JarMaker class and create faster loading IBM Toolbox for Java JAR files.

Shutting down your Java program

To ensure that your program shuts down properly, issue System.exit(0) as the last instruction before your Java program ends.

Note: Avoid using System.exit(0) with servlets because doing so shuts down the entire Java virtual machine.

IBM Toolbox for Java connects to the server with user threads. Because of this, a failure to issue System.exit(0) may keep your Java program from properly shutting down.

Using System.exit(0) is not a requirement, but a precaution. There are times that you must use this command to exit a Java program and it is not problematic to use System.exit(0) when it is not necessary.

Integrated file system path names for AS/400 objects

Your Java program must use integrated file system names to refer to AS/400 objects, such as programs, libraries, commands, or spooled files. The integrated file system name is the name of an AS/400 object as it would be accessed in the library file system of the integrated file system on the AS/400.

The path name may consist of the following pieces:

library	The library in which the object resides. The library is a **required** portion of an integrated file system path name. The library name must be 10 or fewer characters and be followed by **.lib**.
object	The name of the object that the integrated file system path name represents. The object is a **required** portion of an integrated file system path name. The object name must be 10 or fewer characters and be followed by **.type**, where **type** is the type of the object. Types can be found by prompting for the OBJTYPE parameter on commands, such as WRKOBJ.
type	The type of the object. The type of the object must be specified when specifying the **object**. (See **object** above.) The type name must be 6 or fewer characters.
member	The name of the member that this integrated file system path name represents. The member is an **optional** portion of an integrated file system path name. It can be specified only when the **object type** is **FILE**. The member name must be 10 or fewer characters and followed by **.mbr**.

Follow these conditions when determining and specifying the integrated file system name:

- The forward slash (/) is the path separator character.
- The root-level directory, called QSYS.LIB, contains the AS/400 library structure.
- Objects that reside in the AS/400 library QSYS have the following format:

```
/QSYS.LIB/object.type
```

- Objects that reside in other libraries have the following format:

```
/QSYS.LIB/library.LIB/object.type
```

- The object type extension is the AS/400 abbreviation used for that type of object.

To see a list of these types, enter an AS/400 command that has object type as a parameter and press F4 (Prompt) for the type. For example, the AS/400 command **Work with Objects** (WRKOBJ) has an object type parameter.

Below are some commonly used types:

Abbreviation	Object
.CMD	command
.DTAQ	data queue
.FILE	file
.FNTRSC	font resource
.FORMDF	form definition
.LIB	library
.MBR	member
.OVL	overlay
.PAGDFN	page definition
.PAGSET	page segment

.PGM	program
.OUTQ	output queue
.SPLF	spooled file

Use these examples to determine how to specify integrated file system path names:

Description	Integrated file system name
Program MY_PROG in library MY_LIB on the AS/400	/QSYS.LIB/MY_LIB.LIB/MY_PROG.PGM
Data queue MY_QUEUE in library MY_LIB on the AS/400	/QSYS.LIB/MY_LIB.LIB/MY_QUEUE.DTAQ
Member JULY in file MONTH in library YEAR1998 on the AS/400	/QSYS.LIB/YEAR1998.LIB/MONTH.FILE/JULY.MBR

Special values that the IBM Toolbox for Java recognizes in the integrated file system

In an integrated file system path name, special values that normally begin with an asterisk, such as ***ALL**, are depicted without the asterisk. Instead, use leading and trailing percent signs (**%ALL%**). In the integrated file system, an asterisk is a wildcard character.

The IBM Toolbox for Java classes recognize the following special values:

With	Use	(Instead Of)
Library name	%ALL%	(*ALL)
	%ALLUSR%	(*ALLUSR)
	%CURLIB%	(*CURLIB)
	%LIBL%	(*LIBL)
	%USRLIBL%	(*USRLIBL)
Object name	%ALL%	(*ALL)
Member name	%ALL%	(*ALL)
	%FILE%	(*FILE)
	%FIRST%	(*FIRST)
	%LAST%	(*LAST)

See the QSYSObjectPathName class for information about building and parsing integrated file system names.

Managing connections

Creating, starting, and ending a connection to an iSeries are discussed below, and some code examples are provided as well.

To connect to an iSeries system, your Java program must create an AS400 object. The AS400 object contains up to one socket connection for each iSeries server type. A service corresponds to a job on the server and is the interface to the data on the server.

Note: If you are creating Enterprise JavaBeans, you need to comply with the EJB specification of not allowing IBM Toolbox for Java threads during your connection.

Every connection to each server has its own job on the iSeries. A different server supports each of the following:

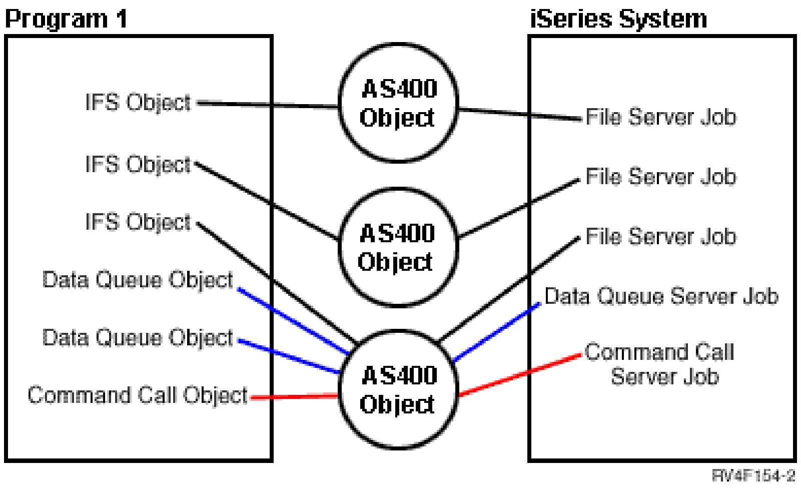
- JDBC
- Program call and command call
- Integrated file system
- Print
- Data queue
- Record-level access

Notes:

- The print classes use one socket connection per AS400 object if the application does not try to do two things that require the network print server at the same time.
- A print class creates additional socket connections to the network print server if needed. The extra conversations are disconnected if they are not used for 5 minutes.

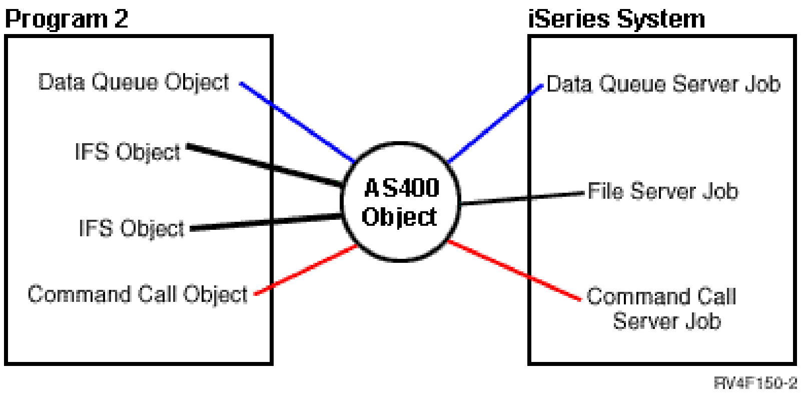
The Java program can control the number of connections to iSeries. To optimize communications performance, a Java program can create multiple AS400 objects for the same system as shown in the figure below. This creates multiple socket connections to the iSeries.

Java program creating multiple AS400 objects and socket connections for the same iSeries system



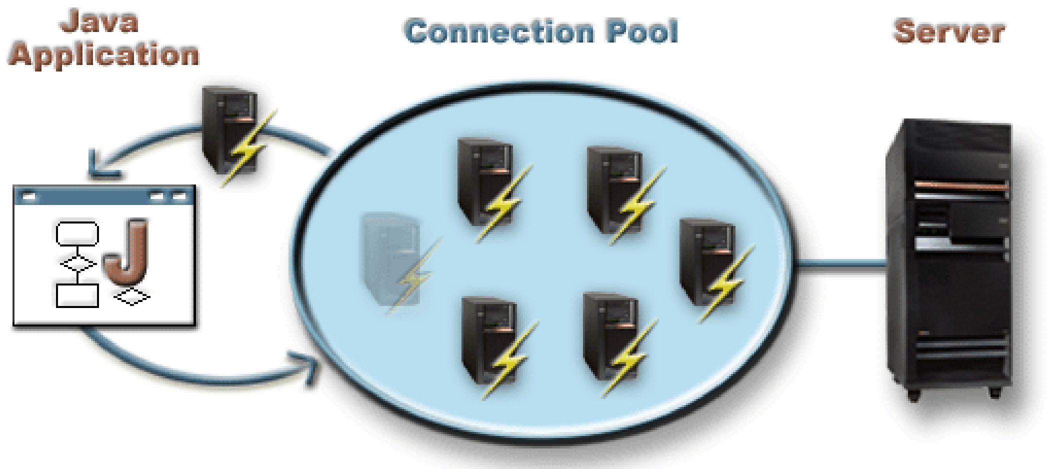
To conserve iSeries system resources, create only one AS400 object as shown in the figure below. This approach reduces the number of connections, which reduces the amount of resource used on the system.

Java program creating a single AS400 object and socket connection for the same iSeries system



»You can also choose to use a connection pool to manage connections as shown in the figure below. This approach reduces the amount of time it takes to connect to iSeries by reusing a connection previously established for the user.

Java program getting a connection from an AS400ConnectionPool to an iSeries server



«

The following examples show how to create and use AS400 objects:

Example 1: In the following example, two CommandCall objects are created that send commands to the same iSeries system. Because the CommandCall objects use the same AS400 object, only one connection to the system is created.

```
                       // Create an AS400 object.
     AS400 sys = new AS400("mySystem.myCompany.com");

                       // Create two command call objects that use
                       // the same AS400 object.
     CommandCall cmd1 = new CommandCall(sys,"myCommand1");
     CommandCall cmd2 = new CommandCall(sys,"myCommand2");

                       // Run the commands.  A connection is made when the
                       // first command is run.  Since they use the same
                       // AS400 object the second command object will use
                       // the connection established by the first command.
     cmd1.run();
     cmd2.run();
```

Example 2: In the following example, two CommandCall objects are created that send commands to the same iSeries system. Because the CommandCall objects use different AS400 objects, two connections to the system are created.

```
                       // Create two AS400 objects to the same AS/400 system.
     AS400 sys1 = new AS400("mySystem.myCompany.com");
     AS400 sys2 = new AS400("mySystem.myCompany.com");

                       // Create two command call objects.  They use
                       // different AS400 objects.
     CommandCall cmd1 = new CommandCall(sys1,"myCommand1");
     CommandCall cmd2 = new CommandCall(sys2,"myCommand2");

                       // Run the commands.  A connection is made when the
                       // first command is run.  Since the second command
                       // object uses a different AS400 object, a second
                       // connection is made when the second command is run.
     cmd1.run();
     cmd2.run();
```

Example 3: In the following example, a CommandCall object and an IFSFileInputStream object are created using the same AS400 object. Because the CommandCall object and the IFSFileInput Stream object use different services on the iSeries system, two connections are created.

```
                       // Create an AS400 object.
     AS400 newConn1 = new AS400("mySystem.myCompany.com");

                       // Create a command call object.
     CommandCall cmd = new CommandCall(newConn1,"myCommand1");

                       // Create the file object.  Creating it causes the
                       // AS400 object to connect to the file service.
     IFSFileInputStream file = new IFSFileInputStream(newConn1,"/myfile");

                       // Run the command.  A connection is made to the
                       // command service when the command is run.
     cmd.run();
```

»**Example 4:** In the following example, an AS400ConnectionPool is used to get an iSeries connection. This example (like Example 3 above) does not specify a service, so the connection to the command service is made when the command is run.

```
                       // Create an AS400ConnectionPool.
```

```
AS400ConnectionPool testPool1 = new AS400ConnectionPool();
                  // Create a connection.
AS400 newConn1 = testPool1.getConnection("myAS400", "myUserID", "myPassword");
                  // Create a command call object that uses the AS400 object.
CommandCall cmd = new CommandCall(newConn1,"myCommand1");
                  // Run the command.  A connection is made to the
                  // command service when the command is run.
cmd.run();
                  // Return connection to pool.
testPool1.returnConnectionToPool(newConn1);
```

Example 5: The following example uses AS400ConnectionPool to connect to a particular service when requesting the connection from the pool. This eliminates the time required to connect to the service when the command is run (see Example 4 above). If the connection is returned to the pool, the next call to get a connection can return the same connection object. This means that no extra connection time is needed, either on creation or use.

```
                  // Create an AS400ConnectionPool.
AS400ConnectionPool testPool1 = new AS400ConnectionPool();
                  // Create a connection to the AS400.COMMAND service. (Use the service number constants
                  // defined in the AS400 class (FILE, PRINT, COMMAND, DATAQUEUE, etc.))
AS400 newConn1 = testPool1.getConnection("myAS400", "myUserID", "myPassword", AS400.COMMAND);
                  // Create a command call object that uses the AS400 object.
CommandCall cmd = new CommandCall(newConn1,"myCommand1");
                  // Run the command.  A connection has already been made
                  // to the command service.
cmd.run();
                  // Return connection to pool.
testPool1.returnConnectionToPool(newConn1);
                  // Get another connection to command service.  In this case, it will return the same
                  // connection as above, meaning no extra connection time will be needed either now or when the
                  // command service is used.
AS400 newConn2 = testPool1.getConnection("myAS400", "myUserID", "myPassword", AS400.COMMAND);
```

«

Starting and ending connections

The Java program can control when a connection is started and ended. By default, a connection is started when information is needed from the server. You can control exactly when the connection is made by calling the connectService() method on the AS400 object to preconnect to the server.

»Using an AS400ConnectionPool, you can create a connection preconnected to a service without calling the connectService() method, as in Example 5 above.«

The following examples show Java programs connecting and disconnecting to iSeries.

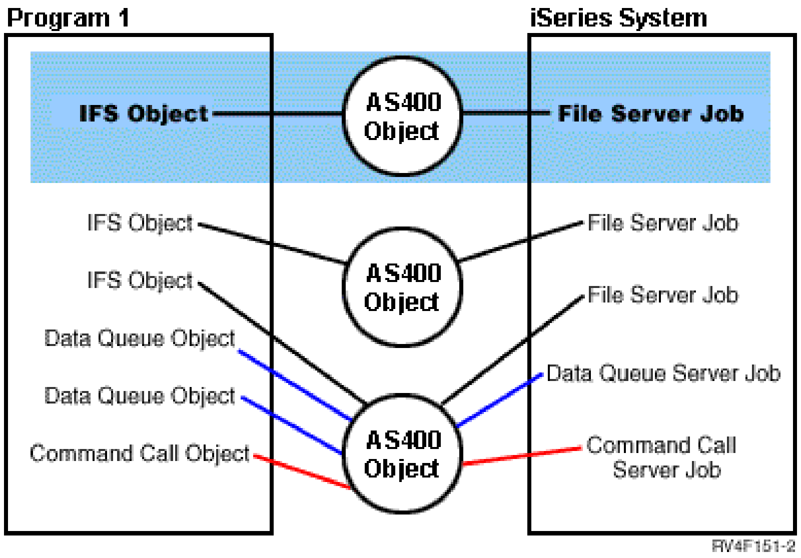
Example 1: This example shows how to preconnect to iSeries:

```
                  // Create an AS400 object.
AS400 system1 = new AS400("mySystem.myCompany.com");

                  // Connect to the command service.  Do it now
                  // instead of when data is first sent to the
                  // command service.  This is optional since the
                  // AS400 object will connect when necessary.
system1.connectService(AS400.COMMAND);
```

Example 2: Once a connection is started, the Java program is responsible for disconnecting, which is done either implicitly by the AS400 object, or explicitly by the Java program. A Java program disconnects by calling the disconnectService() method on the AS400 object. To improve performance, the Java program should disconnect only when the program is finished with a service. If the Java program disconnects before it is finished with a service, the AS400 object reconnects, if it is possible to reconnect, when data is needed from the service.

The figure below shows how disconnecting the connection for the first integrated file system object connection ends only that single instance of the AS400 object connection, not all of the integrated file system object connections.

Single object using its own service for an instance of an AS400 object is disconnected

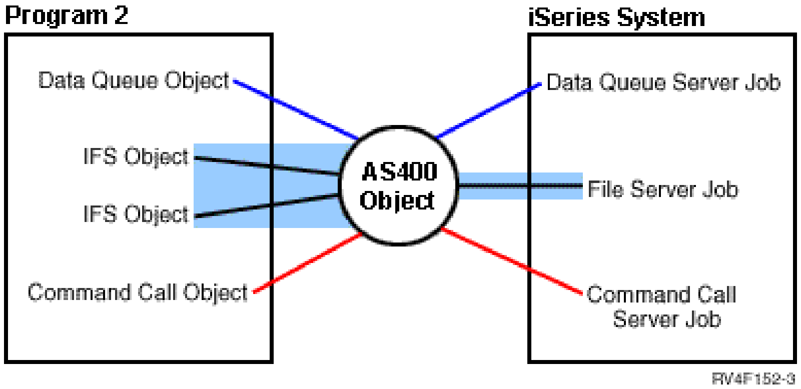


This example shows how the Java program disconnects a connection:

```
                      // Create an AS400 object.
AS400 system1 = new AS400("mySystem.myCompany.com");

                      // ... use command call to send several commands
                      // to the AS/400.  Since connectService() was not
                      // called, the AS400 object automatically
                      // connects when the first command is run.

                      // All done sending commands so disconnect the
                      // connection.
system1.disconnectService(AS400.COMMAND);
```

Example 3: Multiple objects that use the same service and share the same AS400 object share a connection. Disconnecting ends the connection for all objects that are using the same service for each instance of an AS400 object as is shown in the figure below.

All objects using the same service for an instance of an AS400 object are disconnected



For example, two CommandCall objects use the same AS400 object. When disconnectService() is called, the connection is ended for both CommandCall objects. When the run() method for the second CommandCall object is called, the AS400 object must reconnect to the service:

```
                      // Create an AS400 object.
AS400 sys = new AS400("mySystem.myCompany.com");

                      // Create two command call objects.
CommandCall cmd1 = new CommandCall(sys,"myCommand1");
CommandCall cmd2 = new CommandCall(sys,"myCommand2");

                      // Run the first command
cmd1.run();

                      // Disconnect from the command service.
sys.disconnectService(AS400.COMMAND);

                      // Run the second command.  The AS400 object
                      // must reconnect to the AS/400.
cmd2.run();
```

```
                  // Disconnect from the command service.  This
                  // is the correct place to disconnect.
sys.disconnectService(AS400.COMMAND);
```

Example 4: Not all IBM Toolbox for Java classes automatically reconnect. Some method calls in the integrated file system classes do not reconnect because the file may have changed. While the file was disconnected, some other process may have deleted the file or changed its contents. In the following example, two file objects use the same AS400 object. When disconnectService() is called, the connection is ended for both file objects. The read() for the second IFSFileInputStream object fails because it no longer has a connection to the server.

```
                  // Create an AS400 object.
AS400 sys = new AS400("mySystem.myCompany.com");

                  // Create two file objects.  A connection to the
                  // AS/400 is created when the first object is
                  // created.  The second object uses the connection
                  // created by the first object.
IFSFileInputStream file1 = new IFSFileInputStream(sys,"/file1");
IFSFileInputStream file2 = new IFSFileInputStream(sys,"/file2");

                  // Read from the first file, then close it.
int i1 = file1.read();
file1.close();

                  // Disconnect from the file service.
sys.disconnectService(AS400.FILE);

                  // Attempt to read from the second file.  This
                  // fails because the connection to the file service
                  // no longer exists.  The program must either
                  // disconnect later or have the second file use a
                  // different AS400 object (which causes it to
                  // have its own connection).
int i2 = file2.read();

                  // Close the second file.
file2.close();

                  // Disconnect from the file service.  This
                  // is the correct place to disconnect.
sys.disconnectService(AS400.FILE);
```

Java virtual machine for iSeries

The IBM Toolbox for Java classes run on the iSeries Java virtual machine. »In fact, the classes run on any platform that supports the Java Development Kit (JDK) 1.1.x and the Java 2 Software Development Kit (J2SDK) specifications.«

When you run IBM Toolbox for Java classes on the iSeries JVM, do the following:

- Choose whether to use the iSeries JVM or the IBM Toolbox for Java classes to access iSeries resources when running in the iSeries JVM.
- Check out Running IBM Toolbox for Java classes on the iSeries JVM.
- Read about setting system name, user ID, and password in the iSeries JVM.

For more information about iSeries support for different Java platforms, see the Support for multiple JDKs in IBM Developer Kit for Java.

Comparing the iSeries Java virtual machine and the IBM Toolbox for Java classes

You always have at least two ways to access an iSeries resource when your Java program is running on the iSeries Java virtual machine (JVM). You can use either of the following interfaces:

- Facilities built into Java
- An IBM Toolbox for Java class

When deciding which interface to use, consider the following factors:

- **Location** - Where a program runs is the most important factor in deciding which interface set to use. Does the program do the following:
  - Run only on the client?
  - Run only on the server?
  - Run on both client and server, but in both cases the resource is an iSeries resource?
  - Run on one iSeries JVM and access resources on another iSeries server?
  - Run on different kinds of servers?

  If the program runs on both client and server (including an iSeries server as a client to a second iSeries server) and accesses only iSeries resources, it may be best to use the IBM Toolbox for Java interfaces.

  If the program must access data on many types of servers, it may be best to use Java native interfaces.
- **Consistency / Portability** - The ability to run IBM Toolbox for Java classes on iSeries servers means that the same interfaces can be used for both client programs and server programs. When you have only one interface to learn for both client programs and server programs, you can be more productive.

  Writing to IBM Toolbox for Java interfaces makes your program less **server** portable, however.

  If your program must run to an iSeries server as well as other servers, you may find it better to use the facilities that are built into Java.
- **Complexity** - The IBM Toolbox for Java interface is built especially for easy access to an iSeries resource. Often, the only alternative to using the IBM Toolbox for Java interface is to write a program that accesses the resource and communicates with that program through Java Native Interface (JNI).

  You must decide whether it is more important to have better Java neutrality and write a program to access the resource, or to use the IBM Toolbox for Java interface, which is less portable.
- **Function** - The IBM Toolbox for Java interface often provides more function than the Java interface. For example, the IFSFileOutputStream class of the IBM Toolbox for Java licensed program has more function than the FileOutputStream class of java.io. Using IFSFileOutputStream makes your program specific to the iSeries, however. You lose **server** portability by using the IBM Toolbox for Java class.

  You must decide whether portability is more important or whether you want to take advantage of the additional function.
- **Resource** - When running on the iSeries JVM, many of the IBM Toolbox for Java classes still make requests through the host servers. Therefore, a second job (the server job) carries out the request to access a resource.

  This request may take more resource than a Java native interface that runs under the job of the Java program.
- **iSeries server as a client** - If your program runs on one iSeries server and accesses data on a second iSeries server, your best choice may be to use IBM Toolbox for Java classes. These classes provide easy access to the resource on the second iSeries server.

  An example of this is Data Queue access. The Data Queue interfaces of the IBM Toolbox for Java licensed program provide easy access to the data queue resource.

  Using the IBM Toolbox for Java also means your program works on both a client and server to access an iSeries data queue. It also works when running on one iSeries server to access a data queue on another iSeries server.

  The alternative is to write a separate program (in C, for example) that accesses the data queue. The Java program calls this program when it needs to access the data queue.

This method is more server-portable; you can have one Java program that handles data queue access and different versions of the program for each server you support.

Running IBM Toolbox for Java classes on the iSeries Java virtual machine

Below are special considerations for running the IBM Toolbox for Java classes on the iSeries Java virtual machine (JVM):

JDBC

Two IBM-supplied JDBC drivers are available to programs running on the iSeries JVM:

- The IBM Toolbox for Java JDBC driver
- The IBM Developer Kit for Java JDBC driver

The IBM Toolbox for Java JDBC driver is best to use when the program is running in a client/server environment.

The IBM Developer Kit for Java JDBC driver is best to use when the program is running on an iSeries server.

If the same program runs on both the workstation and the server, you should load the correct driver through a system property instead of coding the driver name into your program.

Program call

Two common ways to call a program are as follows:

- The ProgramCall class of the IBM Toolbox for Java
- Through a Java Native Interface (JNI) call

The ProgramCall class of the IBM Toolbox for Java licensed program has the advantage that it can call any iSeries program.

You may not be able to call your iSeries program through JNI. An advantage of JNI is that it is more portable across server platforms.

Command call

Two common ways to call a command are as follows:

- The CommandCall class of the IBM Toolbox for Java
- java.lang.runtime.exec()

The CommandCall class generates a list of messages that are available to the Java program once the command completes. This list of messages is not available through java.lang.runtime.exec().

java.lang.runtime.exec() is portable across many platforms, so if your program must access files on different types of servers, java.lang.runtime.exec() is a better solution.

Integrated file system

Listed below are two common ways to access a file in the integrated file system of the iSeries server:

- The IFSFile classes of the IBM Toolbox for Java licensed program
- The file classes that are a part of java.io

The IBM Toolbox for Java integrated file system classes have the advantage of providing more function than the java.io classes. The IBM Toolbox for Java classes also work in applets, and they do not need a method of redirection (such as IBM iSeries Client Access Family for Windows) to get from a workstation to the server.

The java.io classes are portable across many platforms, which is an advantage. If your program must access files on different types of servers, java.io is a better solution.

If you use java.io classes on a client, you need a method of redirection (such as the IBM iSeries Client Access Family for Windows) to get to the AS/400 file system.

Setting system name, user ID, and password with an AS400 object in the iSeries Java virtual machine

The AS400 object allows special values for system name, user ID, and password when the Java program is running on the iSeries Java virtual machine (JVM).

When you run a program on the iSeries JVM, be aware of some special values and other considerations:

- If system name, user ID, or password is not set on the AS400 object, the AS400 object connects to the current server by using the user ID and password of the job that started the Java program. **A password must be supplied when using record-level access while connecting to v4r3 and earlier machines. When connecting to a v4r4 or later machine, it can propagate the signed-on user's password like the rest of the IBM Toolbox for Java components.**
- The special value, **localhost**, can be used as the system name. In this case, the AS400 object connects to the current server.
- The special value, ***current**[1], can be used as the user ID or password on the AS400 object. In this case, the user ID or password (or both) of the job that started the Java program is used.
- The special value, ***current**[1], can be used as the user ID or password on the AS400 object when the Java program is running on the iSeries JVM of one iSeries server, and the program is accessing resources on another iSeries server. In this case, the user ID and password of the job that started the Java program on the source system are used when connecting to the target system.

  [1] The Java program cannot set the password to "*current" if you are using record-level access and V4R3 or earlier. When you use record-level access, "localhost" is valid for system name and "*current" is valid for user ID; however, the Java program must supply the password.

  *current works only on systems running at Version 4 Release 3 (V4R3) and later. Password and user ID must be specified on system running on V4R2 systems.

- User ID and password prompting is disabled when the program runs on the server.

  For more information about user ID and password values in the server environment, see Summary of user ID and password values on an AS400 object.

The following examples show how to use the AS400 object with the iSeries JVM.

Example 1: When a Java program is running in the iSeries JVM, the program does not have to supply a system name, user ID, or password.

A password must be supplied when using record-level access.

If these values are not supplied, the AS400 object connects to the local system by using the user ID and password of the job that started the Java program.

When the program is running on the Java virtual machine for AS/400, setting the system name to **localhost** is the same as not setting the system name. The following example shows how to connect to the current AS/400:

```
// Create two AS400 objects.  If the Java program is running in the
// iSeries JVM, the behavior of the two objects is the same.
// They will connect to the current server using the user ID and
// password of the job that started the Java program.
AS400 sys  = new AS400()
AS400 sys2 = new AS400("localhost")
```

Example 2: The Java program can set a user ID and password even when the program is running on the iSeries JVM. These values override the user ID and password of the job that started the Java program.

In the following example, the Java program connects to the current server, but the program uses a user ID and password that differs from those of the job that started the Java program.

```
// Create an AS400 object.  Connect to the current server but do
// not use the user ID and password of the job that started the
// program. The supplied values are used.
AS400 sys = new AS400("localhost", "USR2", "PSWRD2")
```

Example 3: A Java program that is running on one server can connect to and use the resources of other iSeries systems.

If ***current** is used for user ID and password, the user ID and password of the job that started the Java program is used when the Java program connects to the target server.

In the following example, the Java program is running on one server, but uses resources from another server. The user ID and password of the job that started the Java program are used when the program connects to the second server.

```
// Create an AS400 object. This program will run on one server
// but will connect to a second server (called "target").
// Because *current is used for user ID and password, the user
// ID and password of the job that started the program will be
// used when connecting to the second server.
AS400 target = new AS400("target", "*current", "*current")
```

OS/400 optimization

The IBM Toolbox for Java licensed program is »written in Java,« so it runs on any platform with a certified Java virtual machine (JVM). The IBM Toolbox for Java classes function in the same way no matter where they run.

Additional classes come with OS/400 that enhance the behavior of the IBM Toolbox for Java when it is running on the iSeries JVM. Sign-on behavior and performance are improved when running on the iSeries JVM and connecting to the same iSeries. OS/400 incorporated the additional classes starting at Version 4 Release 3.

»Enabling the Optimizations

IBM Toolbox for Java comes in two packages: as a separate licensed program and with OS/400.

- Licensed Program 5722-JC1. The licensed program version of IBM Toolbox for Java ships files in the following directory:

  ```
  /QIBM/ProdData/http/public/jt400/lib
  ```

  These files do not contain OS/400 optimizations. Use these files if you want behavior consistent with running the IBM Toolbox for Java on a client.
- OS/400. IBM Toolbox for Java is also shipped with OS/400 in directory

  ```
  /QIBM/ProdData/OS400/jt400/lib
  ```

  These files do contain the classes that optimize the IBM Toolbox for Java when running on the iSeries JVM.

For more information see Note 1 in the information about Jar files.«

Sign-on considerations

With the additional classes provided with OS/400, Java programs have additional options for providing system name, user ID and password information to the IBM Toolbox for Java.

When accessing an iSeries resource, the IBM Toolbox for Java classes must have a system name, user ID and password.

- **When running on a client**, the system name, user ID and password are provided by the Java program, or the IBM Toolbox for Java retrieves these values from the user through a sign-on dialog.
- **When running on the iSeries Java virtual machine**, the IBM Toolbox for Java has one more option. It can send requests to the current (local) server using the user ID and password of the job that started the Java program.

With the additional classes, the user ID and password of the current job also can be used when a Java program that is running on one iSeries accesses the resources on another iSeries. In this case, the Java program sets the system name, then uses the special value "*current" for the user ID and password.

The Java program can only set the password to "*current" if you are using record-level access V4R4 or later. Otherwise, when you use record-level access, "localhost" is valid for system name and "*current" is valid for user ID; however, the Java program must supply the password.

A Java program sets system name, user ID, and password values in the AS400 object.

To use the job's user ID and password, the Java program can use "*current" as user ID and password, or it can use the constructor that does not have user ID and password parameters.

To use the current iSeries, the Java program can use "localhost" as the system name or use the default constructor. That is,

```
    AS400 system = new AS400();
```

is the same as

```
    AS400 system = new AS400("localhost", "*current", "*current");
```

Two AS400 objects are created in the following example. The two objects have the same behavior: they both run a command to the current iSeries using the job's user ID and password. One object uses the special value for the user ID and password, while the other uses the default constructor and does not set user ID or password.

```
                  // Create an AS400 object. Since the default
                  // constructor is used and system, user ID and
                  // password are never set, the AS400 object sends
                  // requests to the local iSeries using the job's
                  // user ID and password. If this program were run
                  // on a client, the user would be prompted for
                  // system, user ID and password.
AS400 sys1 = new AS400();

                  // Create an AS400 object. This object sends
                  // requests to the local iSeries using the job's
                  // user ID and password. This object will not work
                  // on a client.
AS400 sys2 = new AS400("localhost", "*current", "*current");

                  // Create two command call objects that use the
                  // AS400 objects.
CommandCall cmd1 = new CommandCall(sys1,"myCommand1");
CommandCall cmd2 = new CommandCall(sys2,"myCommand2");

                  // Run the commands.
cmd1.run();
cmd2.run();
```

In the following example an AS400 object is created that represents a second iSeries system. Since "*current" is used, the job's user ID and password from the iSeries running the Java program are used on the second (target) iSeries.

```
                  // Create an AS400 object. This object sends
                  // requests to a second iSeries using the user ID
                  // and password from the job on the current iSeries.
AS400 sys = new AS400("mySystem.myCompany.com", "*current", "*current");


                  // Create a command call object to run a command
                  // on the target iSeries.
CommandCall cmd = new CommandCall(sys,"myCommand1");


                  // Run the command.
cmd.run();
```

Performance improvements

With the additional classes provided by OS/400, Java programs running on the Java virtual machine for iSeries experience improved performance. Performance is improved in some cases because less communication function is used, and in other cases, an iSeries API is used instead of calling the server program.

Shorter download time

In order to download the minimum number of IBM Toolbox for Java class files, use the proxy server in combination with the AS400ToolboxJarMaker tool.

Faster communication

For all IBM Toolbox for Java functions except JDBC and integrated file system access, Java programs running on the Java virtual machine for iSeries will run faster. The programs run faster because less communication code is used when communicating between the Java program and the server program on the server that does the request.

JDBC and integrated file system access were not optimized because facilities already exist that make these functions run faster. When running on the iSeries, you can use the JDBC driver for iSeries instead of the JDBC driver that comes with the IBM Toolbox for Java. To access files on the server, you can use java.io instead of the integrated file system access classes that come with the IBM Toolbox for Java.

Directly calling iSeries APIs

Performance of the following classes of the IBM Toolbox for Java is improved because these classes directly call iSeries APIs instead of calling a server program to carry out the request:

- AS400Certificate classes
- CommandCall
- DataQueue
- ProgramCall
- Record-level database access classes
- ServiceProgramCall
- UserSpace

APIs are directly called only if the user ID and password match the user ID and password of the job running the Java program. To get the performance improvement, the user ID and password must match the user ID and password of the job that starts the Java program. For best results, use "localhost" for system name, "*current" for user ID, and "*current" for password.

Port mapping changes

The port mapping system has been changed, which makes accessing a port faster. Before this change, a request for a port would be sent to the port mapper. From there, the iSeries server would determine which port was available and return that port to the user to be accepted. Now, you can either tell the server which port to use or specify that the default ports be used. This option eliminates the wasted time of the server determining the port for you. Use the WRKSRVTBLE command to view or change the list of ports for the server.

For the port mapping improvement, a few methods have been added to AS400 class:

- getServicePort
- setServicePort
- setServicePortsToDefault

MRI changes

MRI files are now shipped within the IBM Toolbox for Java program as class files instead of property files. The iSeries server finds messages in class files faster than in property files. ResourceBundle.getString() now runs faster because the MRI files are stored in the first place that the computer searches. Another advantage of changing to class files is that the server can find the translated version of a string faster.

Converters

Two classes allow faster, more efficient conversion between Java and the iSeries:

- Binary Converter: Converts between Java byte arrays and Java simple types.
- Character Converter: Converts between Java string objects and iSeries code packages.

»Also, the IBM Toolbox for Java now incorporates its own conversion tables for over 100 commonly used CCSIDs. Previously, the IBM Toolbox for Java either deferred to Java for nearly all text conversion. If Java did not possess the correct conversion table, IBM Toolbox for Java downloaded the conversion table from the server.

The IBM Toolbox for Java performs all text conversion for any CCSID of which it is aware. When it encounters an unknown CCSID, it attempts to let Java handle the conversion. At no point does the IBM Toolbox for Java attempt to download a conversion table from the server. This technique greatly reduces the amount of time it takes for an IBM Toolbox for Java application to perform text conversion. No action is required by the user to take advantage of this new text conversion; the performance gains all occur in the underlying converter tables.«

Performance tip regarding the Create Java Program (CRTJVAPGM) command

If your Java application runs on the iSeries Java virtual machine (JVM), you can **significantly improve performance** if you create a Java program from an IBM Toolbox for Java zip file or jar file. Enter the **CRTJVAPGM** command on an iSeries command line to create the program. (See the online help information for the **CRTJVAPGM** command for more information.) By using the **CRTJVAPGM** command, you save the Java program that is created (and that contains the IBM Toolbox for Java classes) when your Java application starts. Saving the Java program that is created allows you to save startup processing time. You save startup processing time because the Java program on the server does not have to be re-created each time your Java application is started.

If you are using the V4R2 or V4R3 version of IBM Toolbox for Java, you cannot run the **CRTJVAPGM** command against the jt400.zip or jt400.jar file because it is too big; however, you may be able to run it against the jt400Access.zip file. At V4R3, IBM Toolbox for Java licensed program includes an additional file, jt400Access.zip. jt400Access.zip contains only the access classes, not the visual classes.

When you run Java applications on a V4R5 (or earlier) system, use jt400Access.zip. When you run Java applications on a V5R1 system, use jt400Native.jar. The **CRTJVAPGM** command has already been run against jt400Native.jar.

Java national language support

Java supports a set of national languages, but it is a subset of the languages that the AS/400 system supports.

When a mismatch between languages occurs, for example, if you are running on a local workstation that is using a language that is not supported by Java, the IBM Toolbox for Java licensed program **may issue some error messages in English**.

Service and support for the IBM Toolbox for Java

Use the following resources for service and support:

- Trouble-shooting information on the IBM Toolbox for Java home page
- Support information on the iSeries home page
- IBM software support on the IBM Software Support Services Web site

Support services for the IBM Toolbox for Java, 5722-JC1, are provided under the usual terms and conditions for iSeries software products. Support services include program services, voice support, and consulting services. Contact your local IBM representative for more information.

Resolving IBM Toolbox for Java program defects is supported under program services and voice support, while resolving application programming and debugging issues is supported under consulting services.

IBM Toolbox for Java application program interface (API) calls are supported under consulting services unless any of the following are true:

- It is clearly a Java API defect, as demonstrated by re-creation in a relatively simple program.
- It is a question asking for documentation clarification.
- It is a question about the location of samples or documentation.

All programming assistance is supported under consulting services including those program samples provided in the IBM Toolbox for Java licensed program. Additional samples may be made available on the Internet at the iSeries home page on an unsupported basis.

Problem solving information is provided with the IBM Toolbox for Java Licensed Program Product. If you believe there is a potential defect in the IBM Toolbox for Java API, a simple program that demonstrates the error will be required.

IBM Toolbox for Java reference links

While we do expect that most developers who are using AS/400 Toolbox for Java know and understand HTML, XHTML, Java, and Servlets, this section is provided as a brief overview of these topics. These links give you a source for additional learning.

IBM Toolbox for Java

The IBM Toolbox for Java and JTOpen Web site has information about troubleshooting, service packs, performance tips, examples, and links to more information. You can also download a zipped package of this information, including the javadocs.

HTML

HTML (HyperText Markup Language) is the most popular language for publishing documents on the World Wide Web. There are many good sites for learning HTML. Some of these are:

- The World Wide Web Consortium: W3C sets standards for publishing on the web. This site contains some HTML instruction and information about current standards for publishing on the World Wide Web
- HTMLCompendium.org: HTMLCompendium provides information on coding HTML, including definitions of common tags and events
- HTML Tag List: Provides definitions for various HTML tags

XHTML

XHTML is touted as the successor to HTML 4.0. It is based on HTML 4.0, but incorporates the extensibility of XML. Sites that provide information on XHTML and XML are provided below:

- IBM: Provides a site dedicated to the work IBM does with XML and how it works to facilitate e-commerce
- The Web Developer's Virtual Library: Gives a good overview of XHTML and how it works with HTML and XML for web authoring
- The World Wide Web Consortium: Provides an introduction to XHTML and reasons web developers would want to turn to this markup language as a standard
- XML.com: Journal that provides updated information on XML in the computer industry

Java

- IBM Java Home Page: Information about how Java developers are using IBM products for e-commerce
- IBM VisualAge for Java and iSeries: Provides information on the IBM VisualAge for Java product
- "The Source for Java Technology" from Sun Microsystems: Information about the various uses for Java, including new technologies
- Java for iSeries, PartnerWorld for Development iSeries: Provides information about Java and the ways IBM business partners can and are using it

Servlets

- IBM Websphere Application Server: Servlet-based web application server
- Java Servlet API: Information from Sun about servlets

JNDI

- Java Naming and Directory Interface(TM) (JNDI): Information from Sun about JNDI, including a list of available service providers.
- iSeries Directory Services (LDAP): Information about LDAP (Lightweight Directory Access Protocol) on OS/400.

Other references

- IBM HTTP Server for iSeries: Information about the HTTP server IBM can provide
- IBM iSeries Client Access Express home page: Information about Client Access and how it works with various Java products
- IBM Host On-Demand: A browser-based 5250 emulator
- IBM Software Support Services : Gateway to IBM software support