IBM i
7.2

Programming
IBM Developer Kit for Java

IBM

Note

Before using this information and the product it supports, read the information in "Notices" on page 491.

Contents

IBM Developer Kit for Java

IBM Developer Kit for Java™ is optimized for use in the IBM® i environment. It uses the compatibility of Java programming and user interfaces, so you can develop your own IBM i applications.

IBM Developer Kit for Java allows you to create and run Java programs on your IBM i server. IBM Developer Kit for Java is a compatible implementation of the Oracle America, Inc. Java Technology, so we assume that you are familiar with their Java Development Kit (JDK) documentation. To make it easier for you to work with their information and ours, we provide links to Oracle America, Inc.'s information.

If for any reason our links to Oracle America, Inc. Java Development Kit documentation do not work, refer to their HTML reference documentation for the information that you need. You can find this information on the World Wide Web at The Source for Java Technology.

Note: Read the “Code license and disclaimer information” on page 489 for important legal information.

What's new for IBM i 7.3

Read about new or significantly changed information for the IBM Developer Kit for Java topic collection.

The following changes have been made to IBM Developer Kit for Java in IBM i 7.3:

- The licensed program for IBM Developer Kit for Java is 5770-JV1.

  Customers still using Classic Java should refer to “Considerations for using IBM Technology for Java Virtual Machine” on page 4 before upgrading to IBM Technology for Java.
- The following topics have been updated to reflect the supported options of 5770-JV1 at IBM i 7.3:
  - “Support for multiple Java Development Kits (JDKs)” on page 6
  - “Installing Java on your IBM i server” on page 2
  - “List of Java system properties” on page 15
- PASE for i now enforces stack execution disable protection.

  To improve system security, the default behavior for PASE for i programs has changed so that instructions run from memory areas (stack, heap, and shared memory) of a process are blocked. IBM Technology for Java JIT generated code is created in memory areas. A PASE for i program that calls the `JNI_CreateJavaVM()` API needs to follow the instructions in PASE for i What's new for IBM i 7.3 to mark the program as needing to allow program execution from memory areas.
- Additional documentation about IBM Technology for Java can be found on the IBM Center for Java Technology Developer Web site.

  The IBM Technology for Java JVM is based on the AIX® version of the IBM Center for Java Technology Developer Kit. The IBM Center for Java Technology Developer Kits have common documentation that applies to all supported platforms. Within that common documentation there exits sections that document platform differences. When IBM i platform documentation does not exit the AIX version of the documentation should be used. See the IBM Center for Java Technology Developer Diagnostic Guide for additional information.

How to see what's new or changed

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

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

To find other information about what's new or changed this release, see the Memo to users.

PDF file for IBM Developer Kit for Java

You can view and print a PDF file of this information.

To view or download the PDF version of this document, select IBM Developer Kit for Java (about 1925 KB).

Saving PDF files

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

1. Right-click the PDF link in your browser.
2. Click the option that saves the PDF locally.
3. Navigate to the directory in which you want to save the PDF.
4. Click **Save**.

Downloading Adobe Reader

You need Adobe Reader installed on your system to view or print these PDFs. You can download a free copy from the Adobe Web site (www.adobe.com/products/acrobat/readstep.html).

Installing and configuring Java

If you have not yet used Java on your IBM i server, follow these steps to install it, configure it, and practice running a simple Hello World Java program.

"What's new for IBM i 7.3" on page 1
Read about new or significantly changed information for the IBM Developer Kit for Java topic collection.

"Customizing your IBM i server for Java usage" on page 12
After you install Java on your server, you can customize your server.

"Downloading and installing Java packages" on page 7
Use this information to download, install, and use Java packages more effectively on the IBM i platform.

"Release-to-release compatibility" on page 27
This topic describes considerations when you are moving Java applications from an earlier release to the most current release.

Installing Java on your IBM i server

Installing IBM Developer Kit for Java allows you to create and run Java programs on your system. The Java Virtual Machine (JVM) included in IBM Developer Kit for Java is the IBM Technology for Java Virtual Machine and is available in both 32-bit and 64-bit versions.

IBM Technology for Java Virtual Machine is included in licensed program 5770-JV1. Licensed program 5770-JV1 is shipped with the system CDs. To access the IBM Technology for Java option, perform the following steps:

1. Enter the Go Licensed Program (GO LICPGM) command and select Option 10 (Display)
2. If you do not see this licensed program listed, then perform the following steps:
   a) Enter the GO LICPGM command on the command line.
   b) Select Option 11 (Install licensed program).
   c) Choose Option 1 (Install) for licensed program (LP) 5770-JV1 *BASE, and select the option that you want to install.

3. Load the latest Java PTF group. This step is optional although recommended. For more information, see “Applying program temporary fixes” on page 486.
4. Set the `JAVA_HOME` environment variable to the home directory of the Java Development Kit that you want to use. At a command line, enter one of the following commands:
   a) `ADDENVVAR ENVVAR(JAVA_HOME) VALUE('/QOpenSys/QIBM/ProdData/JavaVM/jdk80/32bit')`
   b) `ADDENVVAR ENVVAR(JAVA_HOME) VALUE('/QOpenSys/QIBM/ProdData/JavaVM/jdk80/64bit')`
   c) `ADDENVVAR ENVVAR(JAVA_HOME) VALUE('/QOpenSys/QIBM/ProdData/JavaVM/jdk71/32bit')`
   d) `ADDENVVAR ENVVAR(JAVA_HOME) VALUE('/QOpenSys/QIBM/ProdData/JavaVM/jdk71/64bit')`
   e) `ADDENVVAR ENVVAR(JAVA_HOME) VALUE('/QOpenSys/QIBM/ProdData/JavaVM/jdk70/32bit')`
   f) `ADDENVVAR ENVVAR(JAVA_HOME) VALUE('/QOpenSys/QIBM/ProdData/JavaVM/jdk70/64bit')`

If you are unsure what JVM you currently using, you can check using the following methods. If you see `IBM J9 VM` in the result, you are using IBM Technology for Java.

- Look in the job log for the job containing the JVM. There will be a message that states what JVM you are using.
- As part of the Java command you are using to run your application, add `-showversion`. You will see one additional line that shows the JVM you are using.
- From qsh or qp2term, run `java -version`.

Related concepts

“Customizing your IBM i server for Java usage” on page 12
After you install Java on your server, you can customize your server.

Related tasks

Running your first Hello World Java program
This topic will help you to run your first Java program.

Creating, compiling, and running a HelloWorld Java program
Creating the simple Hello World Java program is a great place to start when becoming familiar with the IBM Developer Kit for Java.

Mapping a network drive to your server
To map a network drive, complete the following steps.

“Running your first Hello World Java program” on page 8
This topic will help you to run your first Java program.

Related reference

Creating and editing Java source files
You can create and edit Java source files in a number of ways: using IBM i Access for Windows, on a workstation, with EDTF, and with SEU.

Related information

Licensed program releases and sizes

Considerations for using IBM Technology for Java Virtual Machine

Be aware of these considerations when using IBM Technology for Java Virtual Machine.

Java Native Interface considerations

If you have integrated language environment (ILE) programs that use Java Native Interface (JNI) functions, you must compile these programs with teraspace storage enabled. Because teraspace storage is not enabled by default, it is likely that you need to recompile. This is necessary because the Java object is in PASE for i storage, which is mapped on top of teraspace storage, and a teraspace storage pointer is returned. Also, the JNI functions, such as GetxxxArrayRegion, have a parameter to a buffer where the data is placed. This pointer must point to teraspace storage to enable the JNI function in PASE for i to copy the data into this storage. If you have not compiled your program with teraspace storage enabled, you will receive the escape message `MCH4443 (Invalid storage model for target program LOADLIB)`.

Adopted authority

Adopted authority for Java programs is not supported by IBM Technology for Java Virtual Machine.

Diagnostic messages and files

When ILE native methods encounter problems, you will see messages in the job log. When IBM Technology for Java Virtual Machine or PASE for i native methods encounter problems, they will dump diagnostic files into the IFS. There are several types of these "core files," including `core.*.dmp`, `javacore.*.txt`, `Snap*.trc`, and `heapdump.*.phd`. The files range in size from tens of KB up to hundreds of MB. In most cases, more severe problems produce larger files. The larger files can quickly and quietly consume large amounts of IFS space. Despite the space these files consume, they are useful for debugging purposes. When possible, you should preserve these files until the underlying problem has been resolved.

For more information, see Advanced control of dump agents in the Java Diagnostics Guide.

Migration considerations

When migrating from the Classic JVM, which was the default 64-bit virtual machine that existed in IBM i 6.1, to the 32-bit version of IBM Technology for Java, consider that there may be limitations when using the 32-bit environment. For example, the amount of addressable memory is much smaller. In 32-bit mode, the Java object heap cannot grow much larger than 3 gigabytes. You will also be limited to running approximately 1000 threads. If your application requires more than 1000 threads or a Java object heap larger than 3 gigabytes use the 64-bit version of IBM Technology for Java. See "Support for multiple Java Development Kits (JDKs)" on page 6 for more information.

Table 1 on page 4 shows the levels of Java Developer Kit, which is also referred to as Classic Java, and the suggested IBM Technology for Java replacement.

Note: IBM recommends Java SE 71 when migrating from Java Developer Kit 1.4 or 5.0.

Table 1. Classic Java levels and the suggested IBM Technology for Java replacement.

If you are using classic Java version:	**Then possible IBM Technology for Java replacements include:**
Java Developer Kit 1.4 (5761-**JV1** option 6)	Java SE 8 32 bit (5770-JV1 option 16) Java SE 8 64 bit (5770-JV1 option 17) Java SE 71 32 bit (5770-JV1 option 14) Java SE 71 64 bit (5770-JV1 option 15) Java SE 7 32 bit (5770-JV1 option 14) Java SE 7 64 bit (5770-JV1 option 15)

Table 1. Classic Java levels and the suggested IBM Technology for Java replacement. (continued)	
If you are using classic Java version:	**Then possible IBM Technology for Java replacements include:**
Java Developer Kit 5.0 (5761-**JV1**option 7)	Java SE 8 32 bit (5770-JV1 option 16) Java SE 8 64 bit (5770-JV1 option 17) Java SE 71 32 bit (5770-JV1 option 14) Java SE 71 64 bit (5770-JV1 option 15) Java SE 7 32 bit (5770-JV1 option 14) Java SE 7 64 bit (5770-JV1 option 15)
Java Developer Kit 6 (5761-**JV1**option 10)	Java SE 8 32 bit (5770-JV1 option 16) Java SE 8 64 bit (5770-JV1 option 17) Java SE 71 32 bit (5770-JV1 option 14) Java SE 71 64 bit (5770-JV1 option 15) Java SE 7 32 bit (5770-JV1 option 14) Java SE 7 64 bit (5770-JV1 option 15)

Related concepts

"Release-to-release compatibility" on page 27

This topic describes considerations when you are moving Java applications from an earlier release to the most current release.

Installing a licensed program with the Restore Licensed Program command

The programs listed in the *Install Licensed Programs* display are those supported by the LICPGM installation when your server was new. Occasionally, new programs become available which are not listed as licensed programs on your server. If this is the case with the program you want to install, you must use the Restore Licensed Program (RSTLICPGM) command to install it.

To install a licensed program with the Restore Licensed Program (RSTLICPGM) command, follow these steps:

1. Put the tape or CD-ROM containing the licensed program in the appropriate drive.
2. On the IBM i command line, type:

   RSTLICPGM

   and press the Enter key.

   The *Restore Licensed Program (RSTLICPGM)* display appears.
3. In the *Product* field, type the ID number of the licensed program you want to install.
4. In the *Device* field, specify your install device.

   Note: If you are installing from a tape drive, the device ID is usually in the format **TAP*xx***, where ***xx*** is a number, like **01**.
5. Keep the default settings for the other parameters in the *Restore Licensed Program* display. Press the Enter key.
6. More parameters appear. Keep these default settings also. Press the Enter key. The program begins installing.

When the licensed program is finished installing, the *Restore Licensed Programs* display appears again.

Support for multiple Java Development Kits (JDKs)

The IBM i platform supports multiple versions of the Java Development Kits (JDKs) and the Java 2 Platform, Standard Edition.

Note: In this documentation, depending on the context, the term JDK refers to any supported version of the JDK or the Java 2 Platform, Standard Edition (J2SE). Usually, the context in which JDK occurs includes a reference to the specific version and release number.

IBM i supports using multiple JDKs simultaneously, but only through multiple Java virtual machines. A single Java virtual machine runs one specified JDK. You can run one Java virtual machine per job.

Find the JDK that you are using or want to use, and select the coordinating option to install. See “Installing Java on your IBM i server” on page 2 to install more than one JDK at one time.

When using IBM Technology for Java, you select which 5770-JV1 option to run (and therefore which JDK/bit mode) by setting the JAVA_HOME environment variable. Once a Java virtual machine is up and running, changing the JAVA_HOME environment variable has no effect.

The following table lists the supported options for this release.

5770-JV1 options	JAVA_HOME
Option 16 - IBM Technology for Java 8 32-bit	/QOpenSys/QIBM/ProdData/JavaVM/jdk80/32bit
Option 17- IBM Technology for Java 8 64-bit	/QOpenSys/QIBM/ProdData/JavaVM/jdk80/64bit
Option 14 - IBM Technology for Java 71 32-bit Option 14 - IBM Technology for Java 7 32-bit	/QOpenSys/QIBM/ProdData/JavaVM/jdk71/32bit /QOpenSys/QIBM/ProdData/JavaVM/jdk70/32bit
Option 15 - IBM Technology for Java 71 64-bit Option 15 - IBM Technology for Java 7 64-bit	/QOpenSys/QIBM/ProdData/JavaVM/jdk71/64bit /QOpenSys/QIBM/ProdData/JavaVM/jdk70/64bit

The default JDK chosen in this multiple JDK environment depends on which 5770-JV1 Options are installed. The following table gives some examples. You can access IBM Technology for Java JDKs by setting the JAVA_HOME environment variable, or by specifying a fully qualified path to the Java tool or utility located in the JDK you want to use.

Install	Enter	Result
All supported options are installed	`java QIBMHello`	8.0 32-bit is used
Option 14 (7&7.1 32-bit) and Option 15(7&7.1 64-bit)	`java QIBMHello`	7.1 32-bit is used

Note: If you install only one JDK, the default JDK is the one you installed. If you install more than one JDK, the following order of precedence determines the default JDK:

1. Option 16 - IBM Technology for Java 8.0 32-bit
2. Option 17 - IBM Technology for Java 8.0 64-bit
3. Option 14 - IBM Technology for Java 7.1 32-bit
4. Option 15 - IBM Technology for Java 7.1 64-bit
5. Option 14 - IBM Technology for Java 7.0 32-bit
6. Option 15 - IBM Technology for Java 7.0 64-bit

Installing Java extensions

Extensions are packages of Java classes that you can use to extend the functionality of the core platform. Extensions are packaged in one or more ZIP files or JAR files, and are loaded into the Java virtual machine by an extension class loader.

The extension mechanism allows the Java virtual machine to use the extension classes in the same way that the virtual machine uses the system classes. The extension mechanism also provides a way for you to retrieve extensions from specified Uniform Resource Locators (URLs) when they are not already installed in the Java 2 Platform, Standard Edition (J2SE).

Some JAR files for extensions are shipped with IBM i. If you would like to install one of these extensions, enter this command:

```
ADDLNK OBJ('/QIBM/ProdData/Java400/ext/extensionToInstall.jar')
     NEWLNK('/QIBM/UserData/Java400/ext/extensionToInstall.jar')
     LNKTYPE(*SYMBOLIC)
```

Where

`extensionToInstall.jar`

is the name of the ZIP or JAR file that contains the extension that you want to install.

Note: JAR files of extensions not provided by IBM may be placed in the /QIBM/UserData/Java400/ext directory.

When you create a link or add a file to an extension in the /QIBM/UserData/Java400/ext directory, the list of files that the extension class loader searches changes for *every Java virtual machine that is running on your server*. If you do not want to impact the extension class loaders for other Java virtual machines on your server, but you still want to create a link to an extension or install an extension not shipped by IBM with the server, follow these steps:

1. Create a directory to install the extensions. Use either the Make Directory (MKDIR) command from the IBM i command line or the `mkdir` command from the Qshell Interpreter.
2. Place the extension JAR file in the directory created.
3. Add the new directory to the java.ext.dirs property. You can add the new directory to the java.ext.dirs property by using the PROP field of the JAVA command from the IBM i command line.

If the name of your new directory is /home/username/ext, the name of your extension file is extensionToInstall.jar, and the name of your Java program is Hello, then the commands that you enter should look like this:

```
MKDIR DIR('/home/username/ext')

CPY OBJ('/productA/extensionToInstall.jar') TODIR('/home/username/ext') or
copy the file to /home/username/ext using FTP (file transfer protocol).

JAVA Hello PROP((java.ext.dirs '/home/username/ext'))
```

Downloading and installing Java packages

Use this information to download, install, and use Java packages more effectively on the IBM i platform.

Packages with graphical user interfaces

Java programs used with graphical user interface (GUI) require the use of a presentation device with graphical display capabilities. For example, you can use a personal computer, technical workstation, or network computer. You can use Native Abstract Windowing Toolkit (NAWT) to provide your Java applications and servlets with the full capability of the Java 2 Platform, Standard Edition (J2SE) Abstract Windowing Toolkit (AWT) graphics functions. For more information, see Native Abstract Windowing Toolkit (NAWT).

Case sensitivity and integrated file system

Integrated file system provides file systems, which are both case-sensitive and those that are not with regard to file names. QOpenSys is an example of a case-sensitive file system within the integrated file system. Root, '/', is an example of a case-insensitive file system. For more information, see the Integrated file system topic.

Even though a JAR or class may be located in a file system which is not case-sensitive, Java is still a case-sensitive language. While `wrklnk '/home/Hello.class'` and `wrklnk '/home/hello.class'` produce the same results, `JAVA CLASS(Hello)` and `JAVA CLASS(hello)` are calling different classes.

ZIP file handling

ZIP files, like JAR files, contain a set of Java classes. ZIP files are treated the same way as JAR files.

Java extensions framework

In J2SE, extensions are packages of Java classes that you can use to extend the functionality of the core platform. An extension or application is packaged in one or more JAR files. The extension mechanism allows the Java virtual machine to use the extension classes in the same way that the virtual machine uses the system classes. The extension mechanism also provides a way for you to retrieve extensions from specified URLs when they are not already installed in the J2SE or Java 2 Runtime Environment, Standard Edition.

See "Installing Java extensions" on page 7 for information about installing extensions.

Running your first Hello World Java program

This topic will help you to run your first Java program.

You can get your Hello World Java program up and running in either of these ways:

1. You can simply run the Hello World Java program that was shipped with the IBM Developer Kit for Java.

   To run the program that is included, perform the following steps:

   a) Check that the IBM Developer Kit for Java is installed by entering the Go Licensed Program (GO LICPGM) command. Then, select option 10 (Displayed installed licensed programs). Verify that licensed program 5770-JV1 *BASE and at least one of the options are listed as installed.
   b) Enter `java QIBMHello` on the IBM i Main Menu command line. Press Enter to run the Hello World Java program.
   c) If the IBM Developer Kit for Java was installed correctly, `QIBMHello` appears in the Java Shell Display. Press F3 (Exit) or F12 (Exit) to return to the command entry display.
   d) If the Hello World class does not run, check to see that the installation was completed successfully, or see "Getting support for Java on IBM i" on page 487 for service information.
2. You can also run your own Hello Java program. For more information about how to create your own Hello Java program, see "Creating, compiling, and running a HelloWorld Java program" on page 9.

Related tasks

Installing Java on your IBM i server
Installing IBM Developer Kit for Java allows you to create and run Java programs on your system. The Java Virtual Machine (JVM) included in IBM Developer Kit for Java is the IBM Technology for Java Virtual Machine and is available in both 32-bit and 64-bit versions.

Creating, compiling, and running a HelloWorld Java program
Creating the simple Hello World Java program is a great place to start when becoming familiar with the IBM Developer Kit for Java.

Mapping a network drive to your server
To map a network drive, complete the following steps.

Related reference

Creating and editing Java source files

You can create and edit Java source files in a number of ways: using IBM i Access for Windows, on a workstation, with EDTF, and with SEU.

Creating, compiling, and running a HelloWorld Java program

Creating the simple Hello World Java program is a great place to start when becoming familiar with the IBM Developer Kit for Java.

To create, compile, and run your own Hello World Java program, perform the following steps:

1. Map a network drive to your system.
2. Create a directory on your server for your Java applications.
   a) On the command line, type:

   ```
   CRTDIR DIR('/mydir')
   ```

   where *mydir* is the name of the directory you are creating.

   Press the **Enter** key.
3. Create the source file as an American Standard Code for Information Interchange (ASCII) text file in the integrated file system. You can either use an integrated development environment (IDE) product or a text-based editor such as Windows Notepad to code your Java application.
   a) Name your text file `HelloWorld.java`.
   b) Make sure that your file contains this source code:

   ```
   class HelloWorld {
        public static void main (String args[]) {
             System.out.println("Hello World");
        }
   }
   ```
4. Compile the source file.
   a) Enter the Start Qshell (STRQSH) command to start the Qshell Interpreter.
   b) Use the change directory (cd) command to change the current directory to the integrated file system directory that contains the `HelloWorld.java` file.
   c) Enter `javac` followed by the name of the file as you have it saved on your disk. For example, enter `javac HelloWorld.java`.
5. Set the file authorities on the class file in the integrated file system.
6. Run the class file.
   a) Ensure that your Java classpath is set up correctly.
   b) On the Qshell command line, type `java` followed by `HelloWorld` to run your `HelloWorld.class` with the Java virtual machine. For example, enter `java HelloWorld`. You can also use the Run Java (RUNJVA) command on your system to run `HelloWorld.class`:
   RUNJVA CLASS(HelloWorld)
   c) `"Hello World"` prints to your screen if everything was entered correctly. If running in the Qshell environment, the shell prompt (by default, a $) appears, indicating that the Qshell is ready for another command.
   d) Press F3 (Exit) or F12 (Disconnect) to return to the command entry display.

Related tasks

Installing Java on your IBM i server
Installing IBM Developer Kit for Java allows you to create and run Java programs on your system. The Java Virtual Machine (JVM) included in IBM Developer Kit for Java is the IBM Technology for Java Virtual Machine and is available in both 32-bit and 64-bit versions.

Running your first Hello World Java program
This topic will help you to run your first Java program.

Mapping a network drive to your server

To map a network drive, complete the following steps.

Related reference

Creating and editing Java source files
You can create and edit Java source files in a number of ways: using IBM i Access for Windows, on a workstation, with EDTF, and with SEU.

"Mapping a network drive to your server" on page 10
To map a network drive, complete the following steps.

"Creating and editing Java source files" on page 11
You can create and edit Java source files in a number of ways: using IBM i Access for Windows, on a workstation, with EDTF, and with SEU.

"Java classpath" on page 12
The Java virtual machine uses the Java classpath to find classes during runtime. Java commands and tools also use the classpath to locate classes. The default system classpath, the CLASSPATH environment variable, and the classpath command parameter all determine what directories are searched when looking for a particular class.

"Java file authorities in the integrated file system" on page 232
To run or debug a Java program, the class file, JAR file, or ZIP file needs to have read authority (*R). Directories need read and execute authorities (*RX).

Run Java (RUNJVA) command

Mapping a network drive to your server

To map a network drive, complete the following steps.

1. Make sure that you have IBM i Access for Windows installed on your server and on your workstation. For more information on how to install and configure IBM i Access for Windows, see Installing IBM i Access for Windows. You must have a connection configured for the server before you can map a network drive.
2. Open Windows Explorer:
   a) Right-click the **Start** button on your Windows taskbar.
   b) Click **Explore** in the menu.
3. Select **Map Network Drive** from the **Tools** menu.
4. Select the drive that you want to use to connect to your server.
5. Type the path name to your server. For example, *MYSERVER* where *MYSERVER* is the name of your server.
6. Check the **Reconnect at logon** box if it is blank.
7. Click **OK** to finish.

Your mapped drive now appears in the **All Folders** section of Windows Explorer.

Related tasks

Installing Java on your IBM i server
Installing IBM Developer Kit for Java allows you to create and run Java programs on your system. The Java Virtual Machine (JVM) included in IBM Developer Kit for Java is the IBM Technology for Java Virtual Machine and is available in both 32-bit and 64-bit versions.

Running your first Hello World Java program
This topic will help you to run your first Java program.

Creating, compiling, and running a HelloWorld Java program
Creating the simple Hello World Java program is a great place to start when becoming familiar with the IBM Developer Kit for Java.

Related reference

Creating and editing Java source files

You can create and edit Java source files in a number of ways: using IBM i Access for Windows, on a workstation, with EDTF, and with SEU.

Creating and editing Java source files

You can create and edit Java source files in a number of ways: using IBM i Access for Windows, on a workstation, with EDTF, and with SEU.

With IBM i Access for Windows

Java source files are American Standard Code for Information Interchange (ASCII) text files in the integrated file system.

You can create and edit a Java source file with IBM i Access for Windows and a workstation-based editor.

On a workstation

You can create a Java source file on a workstation. Then, transfer the file to the integrated file system by using file transfer protocol (FTP).

To create and edit Java source files on a workstation:

1. Create the ASCII file on the workstation by using the editor of your choice.
2. Connect to your server with FTP.
3. Transfer the source file to your directory in the integrated file system as a binary file, so that the file remains in ASCII format.

With EDTF

You can edit files from any file system using the Edit File (EDTF) CL command. It is an editor that is similar to the Source Entry Utility (SEU) for editing stream files or database files. See the Edit File (EDTF) CL command for information.

If you use the EDTF command to create a new stream file, the file will tagged with an extended binary-coded decimal interchange code (EBCDIC) coded character set identifier (CCSID). Java files need to be tagged with an ASCII CCSID. You can use the Qshell utility `touch` to create an empty stream file with the an ASCII CCSID and then use the EDTF command to edit the file. For example, to create an empty stream file `/tmp/Test.java` with an ASCII CCSID of 819, use the following command:

```
QSH CMD('touch -C 819 /tmp/Test.java')
```

With Source Entry Utility

You can create a Java source file as a text file by using source entry utility (SEU).

To create a Java source file as a text file by using SEU, perform the following steps:

1. Create a source file member by using SEU.
2. Use the Copy To Stream File (CPYTOSTMF) command to copy the source file member to an integrated file system stream file, while converting the data to ASCII.

If you need to make changes to the source code, change the database member by using SEU and copy the file again.

For information about storing files, see “Java-related files in the IFS” on page 232.

Related tasks

Installing Java on your IBM i server

Installing IBM Developer Kit for Java allows you to create and run Java programs on your system. The Java Virtual Machine (JVM) included in IBM Developer Kit for Java is the IBM Technology for Java Virtual Machine and is available in both 32-bit and 64-bit versions.

Running your first Hello World Java program
This topic will help you to run your first Java program.

Creating, compiling, and running a HelloWorld Java program
Creating the simple Hello World Java program is a great place to start when becoming familiar with the IBM Developer Kit for Java.

Mapping a network drive to your server
To map a network drive, complete the following steps.

Customizing your IBM i server for Java usage

After you install Java on your server, you can customize your server.

Java classpath

The Java virtual machine uses the Java classpath to find classes during runtime. Java commands and tools also use the classpath to locate classes. The default system classpath, the CLASSPATH environment variable, and the classpath command parameter all determine what directories are searched when looking for a particular class.

The java.ext.dirs property determines the classpath for the extensions that are loaded. See "Installing Java extensions" on page 7 for more information.

The default bootstrap classpath is system-defined, and you should not change it. On your server, the default bootstrap classpath specifies where to find the classes that are part of the IBM Developer Kit for Java, and other system classes.

The java.endorsed.dirs property is a standard way to override endorsed versions of Java classes by adding JAR files to the bootstrap classpath. See Java Endorsed Standards Override Mechanism for more information.

To find any other classes on the system, specify the classpath to search by using the CLASSPATH environment variable or the classpath parameter. The classpath parameter that is used on a tool or command overrides the value that is specified in the CLASSPATH environment variable.

You can work with the CLASSPATH environment variable using the Work with Environment Variable (WRKENVVAR) command. From the WRKENVVAR display, you can add or change the CLASSPATH environment variable. The Add Environment Variable (ADDENVVAR) command and Change Environment Variable (CHGENVVAR) command either add or change the CLASSPATH environment variable.

The value of the CLASSPATH environment variable is a list of path names, separated by colons (:), which are searched to find a particular class. A path name is a sequence of zero or more directory names. These directory names are followed by the name of the directory, the ZIP file, or the JAR file that is to be searched in the integrated file system. The components of the path name are separated by the slash (/) character. Use a period (.) to indicate the current working directory.

You can set the CLASSPATH variable in the Qshell environment by using the export utility that is available using the Qshell Interpreter.

These commands add the CLASSPATH variable to your Qshell environment and set it to the value `".:/myclasses.zip:/Product/classes"`

- This command sets the CLASSPATH variable in the Qshell environment:

```
export -s CLASSPATH=.:/myclasses.zip:/Product/classes
```

- This command sets the CLASSPATH variable from the command line:

```
ADDENVVAR ENVVAR(CLASSPATH) VALUE(".:/myclasses.zip:/Product/classes")
```

The J2SE searches the bootstrap classpath first, then the extension directories, then the classpath. The search order for J2SE, using the previous example above, is:

1. The bootstrap classpath, which is in the sun.boot.class.path property,
2. The extension directories, which is in the java.ext.dirs property,
3. The current working directory,
4. The myclasses.zip file that is located in the "root" (/) file system,
5. The classes directory in the Product directory in the "root" (/) file system.

Some Java tools and commands contain a classpath parameter in which a list of path names can be specified. The parameter has the same syntax as the CLASSPATH environment variable. The following list shows some of the tools and commands for which the classpath parameter can be specified:

- `java` command in Qshell
- `javac` tool
- `javah` tool
- `javap` tool
- `javadoc` tool
- `rmic` tool
- Run Java (RUNJVA) command

For more information about these commands, see "Java commands and tools" on page 355. If you use the classpath parameter with any of these command or tools, it ignores the CLASSPATH environment variable.

You can override the CLASSPATH environment variable by using the java.class.path property. You can change the java.class.path property, as well as other properties, by using the SystemDefault.properties file. The values in the SystemDefault.properties files override the CLASSPATH environment variable. For information about the SystemDefault.properties file, see the "SystemDefault.properties file" on page 14.

The -Xbootclasspath option and the java.endorsed.dirs property also affects what directories the system searches when looking for classes. Using -Xbootclasspath/a:*path* appends *path* to the default bootstrap classpath, /p:*path* prepends *path* to the bootstrap classpath, and :*path* replaces the bootstrap classpath with *path*. JAR files located in the directories specified for the java.endorsed.dirs property are prepended to the bootstrap classpath.

Note: Be careful when you specify -Xbootclasspath because unpredictable results occur when a system class cannot be found or is incorrectly replaced by a user-defined class. Therefore, you should allow the system default classpath to be searched before any user-specified classpath.

See "Java system properties" on page 14 for information about how to determine the environment in which Java programs run.

For more information, see the Program and CL Command APIs or the Integrated file system.

Related concepts

Java system properties
Java system properties determine the environment in which you run your Java programs. They are similar to system values or environment variables in IBM i.

Internationalization

You can customize your Java programs for a specific region of the world by creating internationalized Java program. By using time zones, locales, and character encoding, you can ensure that your Java program reflects the correct time, place, and language.

Java system properties

Java system properties determine the environment in which you run your Java programs. They are similar to system values or environment variables in IBM i.

Starting an instance of a Java virtual machine (JVM) sets the values for the system properties that affect that JVM.

You can choose to use the default values for Java system properties or you can specify values for them by using the following methods:

- Adding parameters to the command line (or the Java Native Interface (JNI) invocation API) when you start the Java program
- Using the QIBM_JAVA_PROPERTIES_FILE job-level environment variable to point to a specific properties file. For example:

```
ADDENVVAR ENVVAR(QIBM_JAVA_PROPERTIES_FILE)
  VALUE(/QIBM/userdata/java400/mySystem.properties)
```

- Creating a SystemDefault.properties file that you create in your user.home directory
- Using the /QIBM/userdata/java400/SystemDefault.properties file

IBM i and the JVM determine the values for Java system properties by using the following order of precedence:

1. Command line or JNI invocation API
2. QIBM_JAVA_PROPERTIES_FILE environment variable
3. user.home SystemDefault.properties file
4. /QIBM/UserData/Java400/SystemDefault.properties
5. Default system property values

Related concepts

Java classpath
The Java virtual machine uses the Java classpath to find classes during runtime. Java commands and tools also use the classpath to locate classes. The default system classpath, the CLASSPATH environment variable, and the classpath command parameter all determine what directories are searched when looking for a particular class.

Internationalization
You can customize your Java programs for a specific region of the world by creating internationalized Java program. By using time zones, locales, and character encoding, you can ensure that your Java program reflects the correct time, place, and language.

SystemDefault.properties file

The SystemDefault.properties file is a standard Java properties file that enables you to specify default properties of your Java environment.

This file may be used to send in both JVM properties and JVM options. Previously only JVM properties were supported. To also allow JVM options, the first line of the file must contain "`#AllowOptions`" or else everything will be treated as a JVM property.

The SystemDefault.properties file that resides in your home directory takes priority over the SystemDefault.properties file that resides in the /QIBM/UserData/Java400 directory.

Properties that you set in the `/YourUserHome/SystemDefault.properties` file affect only the following specific Java virtual machines:

- JVMs that you start without specifying a different user.home property
- JVMs that others users start by specifying the property `user.home = /YourUserHome/`

Example: SystemDefault.properties file

The following example sets several Java properties and options:

```
#AllowOptions
#Comments start with pound sign
prop1=12345
-Dprop2
-Dprop3=abcd
-Xmx200m
prop4=value
-Xnojit
```

The Java properties and options above affects the JVM in the following way:

- There are four properties: `prop1`, `prop2`, `prop3`, and `prop4`.
- The max heap size is 200 MB.
- The JIT will not be used.

If the `#AllowOptions` line is removed from the example above, then the JVM would contain six properties: `prop1`, `-Dprop2`, `-Dprop3`, `-Xms200m`, `prop4`, and `-Xnojit`.

List of Java system properties

Java system properties determine the environment in which the Java programs run. They are like system values or environment variables in IBM i.

Starting a Java virtual machine (JVM) sets the system properties for that instance of the JVM. For more information about how to specify values for Java system properties, see the following pages:

- "Java system properties" on page 14
- "SystemDefault.properties file" on page 14

For more information about Java system properties, see "JSSE for Java system properties" on page 269

The following table lists the Java system properties for the supported IBM Technology for Java (5770-JV1) options. For each property, the table lists the name of the property and either the default values that apply or a brief description. The table indicates which system properties have different values in different versions of the Java 2 Platform, Standard Edition (J2SE). When the column that lists the default values does not indicate different versions of the J2SE, all supported versions of the J2SE use that default value.

Note: Not all properties are listed. Only properties that are set uniquely for IBM i are listed.

Java property	Default value
file.encoding	Defaults based on default language ID and country ID of the job. Maps the coded character set identifier (CCSID) to the corresponding ISO ASCII CCSID. Also, sets the `file.encoding` value to the Java value that represents the ISO ASCII CCSID. The `file.encoding` value must be specified on JVM startup, and should not be changed at runtime. See "File.encoding values and IBM i CCSID" on page 21 for a description of how the default is chosen, and a table that shows the relationship between possible `file.encoding` values and the closest matching CCSID.
i5os.crypto.device	Specifies the cryptographic device to use. If this property is not set, the default device CRP01 is used.
i5os.crypto.keystore	Specifies the CCA keystore file to use. If this property is not set, the keystore file named in the cryptographic device description is used.

Java property	Default value
java.compiler	IBM Technology for Java compiler level. This property is used for output only.
java.ext.dirs	Java SE 8 32bit(default): • `/QOpenSys/QIBM/ProdData/JavaVM/jdk80/32bit/jre/lib/ext` • `/QIBM/UserData/Java400/ext` Java SE 80 64bit: • `/QOpenSys/QIBM/ProdData/JavaVM/jdk80/64bit/jre/lib/ext` • `/QIBM/UserData/Java400/ext` Java SE 71 32bit: • `/QOpenSys/QIBM/ProdData/JavaVM/jdk71/32bit/jre/lib/ext` • `/QIBM/UserData/Java400/ext` Java SE 71 64bit: • `/QOpenSys/QIBM/ProdData/JavaVM/jdk71/64bit/jre/lib/ext` • `/QIBM/UserData/Java400/ext` Java SE 7 32bit: • `/QOpenSys/QIBM/ProdData/JavaVM/jdk70/32bit/jre/lib/ext` • `/QIBM/UserData/Java400/ext` Java SE 7 64bit: • `/QOpenSys/QIBM/ProdData/JavaVM/jdk70/64bit/jre/lib/ext` • `/QIBM/UserData/Java400/ext`
java.home	Java SE 8 32bit: `/QOpenSys/QIBM/ProdData/JavaVM/jdk80/32bit/jre` Java SE 8 64bit: `/QOpenSys/QIBM/ProdData/JavaVM/jdk80/64bit/jre` Java SE 71 32bit: `/QOpenSys/QIBM/ProdData/JavaVM/jdk71/32bit/jre` Java SE 71 64bit: `/QOpenSys/QIBM/ProdData/JavaVM/jdk71/64bit/jre` Java SE 7 32bit: `/QOpenSys/QIBM/ProdData/JavaVM/jdk70/32bit/jre` Java SE 7 64bit: `/QOpenSys/QIBM/ProdData/JavaVM/jdk70/64bit/jre` This property is used for output only. See "Support for multiple Java Development Kits (JDKs)" on page 6 for details.
java.library.path	This property is used for locating native method libraries for the application as well as internal JVM native libraries. The default value is obtained from the concatenation of two lists: the IBM i library list and the paths specified for the LIBPATH environment variable. For more information, see "Managing native method libraries" on page 208.
java.net.preferIPv4Stack	• false (no's) - default value • true On dual stack machines, system properties are provided for setting the preferred protocol stack (IPv4 or IPv6) as well as the preferred address family types (inet4 or inet6). IPv6 stack is preferred by default, because on a dual-stack machine IPv6 socket can talk to both IPv4 and IPv6 peers. This setting can be changed with this property. For more information, see the Networking IPv6 User Guide.

Java property	Default value
java.net.preferIPv6Addresses	• true • false (no's) (default value) Even though IPv6 is available on the operating system, the default preference is to prefer an IPv4-mapped address over an IPv6 address. This property controls whether IPv6 (true) or IPv4 (false) addresses are used. For more information, see the Networking IPv6 User Guide.
java.use.policy	true
java.vendor	IBM Corporation
java.vendor.url	http://www.ibm.com
java.vm.name	IBM J9 VM
java.vm.specification.name	Java Virtual Machine Specification
java.vm.specification.vendor	Oracle America, Inc.
java.vm.specification.version	1.0
java.vm.vendor	IBM Corporation
java.vm.version	• Java SE 7: 2.6 • Java SE 71: 2.7 • Java SE 8: 2.8
os.arch	ppc/ppc64
os.name	OS/400®
os.version	V7R3M0 (default value) Obtains the IBM i release level from the Retrieve Product Information application programming interface (API).
os400.certificateContainer	Directs Java secure sockets layer (SSL) support to use the specified certificate container for the Java program that was started and the property that was specified. If you specify the `os400.secureApplication` system property, this system property is ignored. For example, enter `-Dos400.certificateContainer=/home/username/mykeyfile.kdb` or any other keyfile in the integrated file system.
os400.certificateLabel	You can specify this system property in conjunction with the `os400.certificateContainer` system property. This property lets you select which certificate in the specified container you want secure sockets layer (SSL) to use. For example, enter `-Dos400.certificateLabel=myCert`, where `myCert` is the label name that you assign to the certificate through the Digital Certificate Manager (DCM) when you create or import the certificate.
os400.child.stdio.convert	Controls the data conversion for stdin, stdout, and stderr in Java. Data conversion between ASCII data and Extended Binary Coded Decimal Interchange Code (EBCDIC) data occurs by default in the Java virtual machine. Using this property to turn on and turn off these conversions only affects child processes that this process starts by using the `Runtime.exec()` in which the command being run is a command based on Java. This property value becomes the default value for `os400.stdio.convert` in the child processes. See "Values for os400.stdio.convert and os400.child.stdio.convert system properties" on page 19.
os400.display.properties	If this value is set to 'true', then all of the Java Virtual Machine properties are printed to standard out. No other values are recognized.

Java property	Default value
os400.file.create.auth, os400.dir.create.auth	These properties specify authorities assigned to files and directories. Specifying the properties without any values or with unsupported values results in a public authority of *NONE. You can specify `os400.file.create.auth=RWX` or `os400.dir.create.auth=RWX`, where R=read, W=write, and X=execute. Any combination of these authorities is valid.
os400.job.file.encoding	This property is used for output only. It lists the character encoding equivalent to the job CCSID of the IBM i job the JVM is running in.
os400.secureApplication	Associates the Java program that starts when using this system property (`os400.secureApplication`) with the registered secure application name. You can view registered secure application names by using the Digital Certificate Manager (DCM).
os400.security.properties	Allows full control over which `java.security` file you use. When you specify this property, the J2SE does not use any other `java.security` files, including the J2SE specific `java.security` default.
os400.stderr	Allows mapping stderr to a file or socket. See "os400.stdin, os400.stdout, and os400.stderr system property values" on page 19.
os400.stdin	Allows mapping stdin to a file or socket. See "os400.stdin, os400.stdout, and os400.stderr system property values" on page 19.
os400.stdin.allowed	Specifies whether stdin is allowed (1) or not allowed (0). The default value is 1 for interactive jobs and 0 for a batch jobs.
os400.stdio.convert	Allows control of the data conversion for stdin, stdout, and stderr in Java. Data conversion occurs by default in the Java virtual machine to convert ASCII data to or from EBCDIC. You can turn these conversions on or off with this property, which affects the current Java program. See "Values for os400.stdio.convert and os400.child.stdio.convert system properties" on page 19. For Java programs started using the `Runtime.exec()` method, see os400.child.stdio.convert.
os400.stdout	Allows mapping stdout to a file or socket. See default values.
os400.xrun.option	This property can be used in place of the `-Xrun` option on the `java` command to run an agent program during JVM startup.
os400.vm.inputargs	This property is used for output only. It will display the arguments that the JVM received as inputs. This property can be useful for debugging what was specified on JVM startup.
user.timezone	• The JVM selects the value for this property by using the QTIMZON value for the current job. The name in the 'Alternate Name' field of this object is the value used for this property. The value in the 'Alternate Name' field must be at least 3 characters in length, or it will not be used. • If the 'Alternate Name' field in the QTIMZON object is less than 3 characters in length, the JVM will attempt to find a matching GMT value based on the current system offset. Example: A QTIMZON object with an empty Alternate Name field and an offset of -5 would result in setting `user.timezone=GMT-5`. • If a value still has not been found, the JVM defaults user.timezone to Universal Time Coordinate (UTC). For more information, see Time zone IDs that can be specified for the user.timezone property in the WebSphere® Software Information Center.

Related concepts

"Customizing your IBM i server for Java usage" on page 12

After you install Java on your server, you can customize your server.

Values for os400.stdio.convert and os400.child.stdio.convert system properties

The following tables show the system values for the `os400.stdio.convert` and `os400.child.stdio.convert` system properties.

*Table 2. System values for **os400.stdio.convert***

Value	Description
Y (default)	All stdio converts to or from the `file.encoding` value to job CCSID during read or write.
N	No stdio conversion is performed during read or write.

*Table 3. System values for **os400.child.stdio.convert***

Value	Description
N (default)	No stdio conversion is performed during read or write.
Y	All stdio converts to or from the `file.encoding` value to job CCSID during read or write.

os400.stdin, os400.stdout, and os400.stderr system property values

The following table shows the system values for os400.stdin, os400.stdout, and os400.stderr system properties.

Value	Example name	Description	Example
File	SomeFileName	SomeFileName is an absolute path or a path relative to the current directory.	file:/QIBM/UserData/Java400/Output.file
Port	HostName	Port address	port:myhost:2000
Port	TCPAddress	Port address	port:1.1.11.111:2000

Internationalization

You can customize your Java programs for a specific region of the world by creating internationalized Java program. By using time zones, locales, and character encoding, you can ensure that your Java program reflects the correct time, place, and language.

Related concepts

Java classpath

The Java virtual machine uses the Java classpath to find classes during runtime. Java commands and tools also use the classpath to locate classes. The default system classpath, the CLASSPATH environment variable, and the classpath command parameter all determine what directories are searched when looking for a particular class.

Java system properties

Java system properties determine the environment in which you run your Java programs. They are similar to system values or environment variables in IBM i.

IBM i globalization

Java Internationalization

Time zone configuration

When you have Java programs that are sensitive to time zones, you should configure the time zone on your system so that your Java programs use the correct time.

The simplest method for configuring time zone is to set the QTIMZON system value to one of the *TIMZON objects provided by IBM i. To correctly determine the local time, the Java virtual machine (JVM) requires that both the QUTCOFFSET system value and the `user.timezone` Java system property be set properly. Setting the QTIMZON system value does both of these for you. The TIMZON objects contain an alternate long name that specifies the Java `user.timezone` value that will be used, so you should select the QTIMZON value that contains the appropriate alternate name. For example, TIMZON object QN0600CST2 contains the alternate name America/Chicago and provides the correct time support for the United States Central time zone.

Note: The `user.timezone` system property setting provided by the QTIMZON system value can be overridden by specifying the `user.timezone` value explicitly on the command line or in the SystemDefault.properties file. This allows each Java job to have its own unique `user.timezone` value so that multiple time zones can be supported on the same system.

Related concepts

Java character encodings
Java programs can convert data in different formats, enabling your applications to transfer and use information from many kinds of international character sets.

Related reference

Examples: Creating an internationalized Java program
If you need to customize a Java program for a specific region of the world, you can create an internationalized Java program with Java locales.

IBM i system value: QTIMZON

Work with Time Zone Desc (WRKTIMZON) CL command

TimeZone Javadoc reference information

Java character encodings

Java programs can convert data in different formats, enabling your applications to transfer and use information from many kinds of international character sets.

Internally, the Java virtual machine (JVM) always operates with data in Unicode. However, all data transferred into or out of the JVM is in a format matching the file.encoding property. Data read into the JVM is converted from file.encoding to Unicode and data sent out of the JVM is converted from Unicode to file.encoding.

Data files for Java programs are stored in the integrated file system. Files in the integrated file system are tagged with a coded character set identifier (CCSID) that identifies the character encoding of the data contained in the file.

When data is read by a Java program, it is expected to be in the character encoding matching file.encoding. When data is written to a file by a Java program, it is written in a character encoding matching file.encoding. This also applies to Java source code files (.java files) processed by the javac command and to data sent and received through Transmission Control Protocol/Internet Protocol (TCP/IP) sockets using the java.net package.

Data read from or written to System.in, System.out, and System.err are handled differently than data read from or written to other sources when they are assigned to stdin, stdout, and stderr. Since stdin, stdout, and stderr are normally attached to EBCDIC devices on the IBM i server, a conversion is performed by the JVM on the data to convert from the normal character encoding of file.encoding to a CCSID matching the IBM i job CCSID. When System.in, System.out, or System.err are redirected to a file or socket and are not directed to stdin, stdout, or stderr, this additional data conversion is not performed and the data remains in a character encoding matching file.encoding.

When data must be read into or written from a Java program in a character encoding other than file.encoding, the program can use the Java IO classes java.io.InputStreamReader, java.io.FileReader, java.io.OutputStreamReader, and java.io.FileWriter. These Java classes allow specifying a file.encoding value that takes precedence over the default file.encoding property currently in use by the JVM.

Data to or from the DB2® database converts to or from the CCSID of the IBM i database through the JDBC APIs .

Data that is transferred to or from other programs through Java Native Interface does not get converted.

Related concepts

Time zone configuration
When you have Java programs that are sensitive to time zones, you should configure the time zone on your system so that your Java programs use the correct time.

Related reference

Examples: Creating an internationalized Java program
If you need to customize a Java program for a specific region of the world, you can create an internationalized Java program with Java locales.

File.encoding values and IBM i CCSID

This table shows the relation between possible `file.encoding` values and the closest matching IBM i coded character set identifier (CCSID).

For more information regarding `file.encoding` support, see Supported encodings by Oracle.

file.encoding	**CCSID**	**Description**
ASCII	367	American Standard Code for Information Interchange
Big5	950	8-bit ASCII T-Chinese BIG-5
Big5_HKSCS	950	Big5_HKSCS
Big5_Solaris	950	Big5 with seven additional Hanzi ideograph character mappings for the Solaris zh_TW.BIG5 locale
CNS11643	964	Chinese National Character Set for traditional Chinese
Cp037	037	IBM EBCDIC US, Canada, Netherlands
Cp273	273	IBM EBCDIC Germany, Austria
Cp277	277	IBM EBCDIC Denmark, Norway
Cp278	278	IBM EBCDIC Finland, Sweden
Cp280	280	IBM EBCDIC Italy
Cp284	284	IBM EBCDIC Spanish, Latin America
Cp285	285	IBM EBCDIC UK
Cp297	297	IBM EBCDIC France
Cp420	420	IBM EBCDIC Arabic
Cp424	424	IBM EBCDIC Hebrew
Cp437	437	8-bit ASCII US PC
Cp500	500	IBM EBCDIC International
Cp737	737	8-bit ASCII Greek MS-DOS
Cp775	775	8-bit ASCII Baltic MS-DOS
Cp838	838	IBM EBCDIC Thailand

file.encoding	CCSID	Description
Cp850	850	8-bit ASCII Latin-1 Multinational
Cp852	852	8-bit ASCII Latin-2
Cp855	855	8-bit ASCII Cyrillic
Cp856	0	8-bit ASCII Hebrew
Cp857	857	8-bit ASCII Latin-5
Cp860	860	8-bit ASCII Portugal
Cp861	861	8-bit ASCII Iceland
Cp862	862	8-bit ASCII Hebrew
Cp863	863	8-bit ASCII Canada
Cp864	864	8-bit ASCII Arabic
Cp865	865	8-bit ASCII Denmark, Norway
Cp866	866	8-bit ASCII Cyrillic
Cp868	868	8-bit ASCII Urdu
Cp869	869	8-bit ASCII Greek
Cp870	870	IBM EBCDIC Latin-2
Cp871	871	IBM EBCDIC Iceland
Cp874	874	8-bit ASCII Thailand
Cp875	875	IBM EBCDIC Greek
Cp918	918	IBM EBCDIC Urdu
Cp921	921	8-bit ASCII Baltic
Cp922	922	8-bit ASCII Estonia
Cp930	930	IBM EBCDIC Japanese Extended Katakana
Cp933	933	IBM EBCDIC Korean
Cp935	935	IBM EBCDIC Simplified Chinese
Cp937	937	IBM EBCDIC Traditional Chinese
Cp939	939	IBM EBCDIC Japanese Extended Latin
Cp942	942	8-bit ASCII Japanese
Cp942C	942	Variant of Cp942
Cp943	943	Japanese PC data mixed for open env
Cp943C	943	Japanese PC data mixed for open env
Cp948	948	8-bit ASCII IBM Traditional Chinese
Cp949	944	8-bit ASCII Korean KSC5601
Cp949C	949	Variant of Cp949
Cp950	950	8-bit ASCII T-Chinese BIG-5
Cp964	964	EUC Traditional Chinese

file.encoding	CCSID	Description
Cp970	970	EUC Korean
Cp1006	1006	ISO 8-bit Urdu
Cp1025	1025	IBM EBCDIC Cyrillic
Cp1026	1026	IBM EBCDIC Turkey
Cp1046	1046	8-bit ASCII Arabic
Cp1097	1097	IBM EBCDIC Farsi
Cp1098	1098	8-bit ASCII Farsi
Cp1112	1112	IBM EBCDIC Baltic
Cp1122	1122	IBM EBCDIC Estonia
Cp1123	1123	IBM EBCDIC Ukraine
Cp1124	0	ISO 8-bit Ukraine
Cp1140	1140	Variant of Cp037 with Euro character
Cp1141	1141	Variant of Cp273 with Euro character
Cp1142	1142	Variant of Cp277 with Euro character
Cp1143	1143	Variant of Cp278 with Euro character
Cp1144	1144	Variant of Cp280 with Euro character
Cp1145	1145	Variant of Cp284 with Euro character
Cp1146	1146	Variant of Cp285 with Euro character
Cp1147	1147	Variant of Cp297 with Euro character
Cp1148	1148	Variant of Cp500 with Euro character
Cp1149	1149	Variant of Cp871 with Euro character
Cp1250	1250	MS-Win Latin-2
Cp1251	1251	MS-Win Cyrillic
Cp1252	1252	MS-Win Latin-1
Cp1253	1253	MS-Win Greek
Cp1254	1254	MS-Win Turkish
Cp1255	1255	MS-Win Hebrew
Cp1256	1256	MS-Win Arabic
Cp1257	1257	MS-Win Baltic
Cp1258	1251	MS-Win Russian
Cp1381	1381	8-bit ASCII S-Chinese GB
Cp1383	1383	EUC Simplified Chinese
Cp33722	33722	EUC Japanese
EUC_CN	1383	EUC for Simplified Chinese

file.encoding	CCSID	Description
EUC_JP	5050	EUC for Japanese
EUC_JP_LINUX	0	JISX 0201, 0208 , EUC encoding Japanese
EUC_KR	970	EUC for Korean
EUC_TW	964	EUC for Traditional Chinese
GB2312	1381	8-bit ASCII S-Chinese GB
GB18030	1392	Simplified Chinese, PRC standard
GBK	1386	New simplified Chinese 8-bit ASCII 9
ISCII91	806	ISCII91 encoding of Indic scripts
ISO2022CN	965	ISO 2022 CN, Chinese (conversion to Unicode only)
ISO2022_CN_CNS	965	CNS11643 in ISO 2022 CN form, Traditional Chinese (conversion from Unicode only)
ISO2022_CN_GB	1383	GB2312 in ISO 2022 CN form, Simplified Chinese (conversion from Unicode only)
ISO2022CN_CNS	965	7-bit ASCII for Traditional Chinese
ISO2022CN_GB	1383	7-bit ASCII for Simplified Chinese
ISO2022JP	5054	7-bit ASCII for Japanese
ISO2022KR	25546	7-bit ASCII for Korean
ISO8859_1	819	ISO 8859-1 Latin Alphabet No. 1
ISO8859_2	912	ISO 8859-2 ISO Latin-2
ISO8859_3	0	ISO 8859-3 ISO Latin-3
ISO8859_4	914	ISO 8859-4 ISO Latin-4
ISO8859_5	915	ISO 8859-5 ISO Latin-5
ISO8859_6	1089	ISO 8859-6 ISO Latin-6 (Arabic)
ISO8859_7	813	ISO 8859-7 ISO Latin-7 (Greek/Latin)
ISO8859_8	916	ISO 8859-8 ISO Latin-8 (Hebrew)
ISO8859_9	920	ISO 8859-9 ISO Latin-9 (ECMA-128, Turkey)
ISO8859_13	0	Latin Alphabet No. 7
ISO8859_15	923	ISO8859_15
ISO8859_15_FDIS	923	ISO 8859-15, Latin alphabet No. 9
ISO-8859-15	923	ISO 8859-15, Latin Alphabet No. 9
JIS0201	897	Japanese industry standard X0201
JIS0208	5052	Japanese industry standard X0208
JIS0212	0	Japanese industry standard X0212
JISAutoDetect	0	Detects and converts from Shift-JIS, EUC-JP, ISO 2022 JP (conversion to Unicode only)

file.encoding	CCSID	Description
Johab	0	Korean composition Hangul encoding (full)
K018_R	878	Cyrillic
KSC5601	949	8-bit ASCII Korean
MacArabic	1256	Macintosh Arabic
MacCentralEurope	1282	Macintosh Latin-2
MacCroatian	1284	Macintosh Croatian
MacCyrillic	1283	Macintosh Cyrillic
MacDingbat	0	Macintosh Dingbat
MacGreek	1280	Macintosh Greek
MacHebrew	1255	Macintosh Hebrew
MacIceland	1286	Macintosh Iceland
MacRoman	0	Macintosh Roman
MacRomania	1285	Macintosh Romania
MacSymbol	0	Macintosh Symbol
MacThai	0	Macintosh Thai
MacTurkish	1281	Macintosh Turkish
MacUkraine	1283	Macintosh Ukraine
MS874	874	MS-Win Thailand
MS932	943	Windows Japanese
MS936	936	Windows Simplified Chinese
MS949	949	Windows Korean
MS950	950	Windows Traditional Chinese
MS950_HKSCS	NA	Windows Traditional Chinese with Hong Kong S.A.R. of China extensions
SJIS	932	8-bit ASCII Japanese
TIS620	874	Thai industry standard 620
US-ASCII	367	American Standard Code for Information Interchange
UTF8	1208	UTF-8
UTF-16	1200	Sixteen-bit UCS Transformation Format, byte order identified by an optional byte-order mark
UTF-16BE	1200	Sixteen-bit Unicode Transformation Format, big-endian byte order
UTF-16LE	1200	Sixteen-bit Unicode Transformation Format, little-endian byte order
UTF-8	1208	Eight-bit UCS Transformation Format
Unicode	13488	UNICODE, UCS-2
UnicodeBig	13488	Same as Unicode

file.encoding	CCSID	Description
UnicodeBigUnmarked		Unicode with no byte-order mark
UnicodeLittle		Unicode with little-endian byte order
UnicodeLittleUnmarked		UnicodeLittle with no byte-order mark

For default values, see Default file.encoding values.

Default file.encoding values

This table shows how the `file.encoding` value is set based on the PASE for i coded character set identifier (CCSID) when the Java virtual machine starts.

Note: The PASE for i CCSID is set based on the job language ID and country ID. For more information about how PASE for i determines what CCSID to use, see IBM PASE for i Locales.

PASE for i CCSID	Default `file.encoding`	Description
813	ISO8859_7	Latin-7 (Greek/Latin)
819	ISO8859_1	Latin-1
874	TIS620	Thai
912	ISO8859_2	Latin-2 (Czech/Czech Republic, Croatian/Croatia, Hungarian/Hungary, Polish/Poland)
915	ISO8859_5	Cyrillic 8-bit (Bulgaria)
916	ISO8859_8	Hebrew (Israel)
920	ISO8859_9	Latin-5 (Turkey extended)
921	Cp921	Baltic 8-bit (Lithuanian/Lithuania, Latvian/Latvia)
922	Cp922	Estonia ISO-8
923	ISO8859_15	Latin-9
1046	Cp1046	Windows Arabic
1089	ISO8859_6	Arabic
1208	UTF-8	Eight-bit UCS Transformation Format
1252	Cp1252	Windows Latin-1

Examples: Creating an internationalized Java program

If you need to customize a Java program for a specific region of the world, you can create an internationalized Java program with Java locales.

Creating an internationalized Java program involves several tasks:

1. Isolate the locale-sensitive code and data. For example, strings, dates, and numbers in your program.
2. Set or get the locale using the Locale class.
3. Format dates and numbers to specify a locale if you do not want to use the default locale.
4. Create resource bundles to handle strings and other locale-sensitive data.

Review the following examples, which offer ways to help you complete the tasks required to create an internationalized Java program:

- "Example: Internationalization of dates using the java.util.DateFormat class" on page 373
- "Example: Internationalization of numeric display using the java.util.NumberFormat class" on page 373
- "Example: Internationalization of locale-specific data using the java.util.ResourceBundle class" on page 374

Related concepts

Time zone configuration
When you have Java programs that are sensitive to time zones, you should configure the time zone on your system so that your Java programs use the correct time.

Java character encodings
Java programs can convert data in different formats, enabling your applications to transfer and use information from many kinds of international character sets.

Release-to-release compatibility

This topic describes considerations when you are moving Java applications from an earlier release to the most current release.

You should take into account the following compatibility issues when running Java applications in the current release:

- IBM Technology for Java only supports JVMTI interfaces from PASE for i. As a result, JVMTI agents will need to be ported to PASE for i.
- When using PASE for i native methods, the architecture of the native code needs to match the architecture of the JVM. That is, object binaries need to be compiled as 32-bit binaries for a 32-bit JVM, or as 64-bit binaries for a 64-bit JVM. This also applies to agents, such as user-provided JVMTI agents.
- The Java system property, `java.version`, is not recognized as an input property for the IBM Technology for Java JVM. In prior releases, the Classic JVM was available which honored the `java.version` Java system property as input for determining which JDK to use. Beginning in IBM i 7.1, the IBM Technology for Java is the only available JVM and it requires the environment variable JAVA_HOME to be specified to determine which JDK to use.
- In the Classic JVM, the Java method `System.getenv()` would return the value of the appropriate ILE environment variable. In IBM Technology for Java, this will instead return the PASE for i environment variable. This may cause issues where the user sets an environment variable in an ILE native method and expects a later call to `System.getenv()` to retrieve it. In general, the user needs to be aware that ILE and PASE for i have their own disjoint sets of environment variables.
- Support for direct processing stopped in IBM i 6.1. In IBM i 7.1, the Java program commands will continue to be supported, but only when using them to target a previous release. See the Release-to-release compatibility section for IBM i 6.1 for additional information. From i 7.2, all java program commands are no longer supported.

Related concepts

"What's new for IBM i 7.3" on page 1
Read about new or significantly changed information for the IBM Developer Kit for Java topic collection.

Database access from Java programs

Java programs can access database files in several ways.

Accessing your IBM i database with the Java JDBC driver

The Java JDBC driver, also known as the "native" driver, provides programmatic access to IBM i database files. Using the Java Database Connectivity (JDBC) API, applications written in the Java language can access JDBC database functions with embedded Structured Query Language (SQL), run SQL statements, retrieve results, and propagate changes back to the database. The JDBC API can also be used to interact with multiple data sources in a distributed, heterogeneous environment.

The SQL99 Command Language Interface (CLI), on which the JDBC API is based, is the basis for ODBC. JDBC provides a natural and easy-to-use mapping from the Java programming language to the abstractions and concepts defined in the SQL standard.

Oracle JDBC documentation
Native JDBC Driver FAQs
JDBC 4.0 API Specification

Getting started with JDBC

The Java Database Connectivity (JDBC) driver shipped with Java on IBM i is called the IBM Developer Kit for Java JDBC driver. This driver is also commonly known as the native JDBC driver.

To select which JDBC driver suits your needs, consider the following suggestions:

- Programs running directly on a server where the database resides should use the native JDBC driver for performance. This includes most servlet and JavaServer Pages (JSP) solutions, and applications written to run locally on a system.
- Programs that must connect to a remote IBM i server, use IBM Toolbox for Java JDBC classes. The IBM Toolbox for Java JDBC driver is a robust implementation of JDBC and is provided as part of IBM Toolbox for Java. Being pure Java, the IBM Toolbox for Java JDBC driver is trivial to set up for clients and requires little server setup.
- Programs that run on a IBM i server and need to connect to a remote, non-IBM i database use the native JDBC driver and set up a Distributed Relational Database Architecture™ (DRDA) connection to that remote server.

Types of JDBC drivers

This topic defines the Java Database Connectivity (JDBC) driver types. Driver types are used to categorize the technology used to connect to the database. A JDBC driver vendor uses these types to describe how their product operates. Some JDBC driver types are better suited for some applications than others.

Type 1

Type 1 drivers are "bridge" drivers. They use another technology such as Open Database Connectivity (ODBC) to communicate with a database. This is an advantage because ODBC drivers exist for many Relational Database Management System (RDBMS) platforms. The Java Native Interface (JNI) is used to call ODBC functions from the JDBC driver.

A Type 1 driver needs to have the bridge driver installed and configured before JDBC can be used with it. This can be a serious drawback for a production application. Type 1 drivers cannot be used in an applet since applets cannot load native code.

Type 2

Type 2 drivers use a native API to communicate with a database system. Java native methods are used to invoke the API functions that perform database operations. Type 2 drivers are generally faster than Type 1 drivers.

Type 2 drivers need native binary code installed and configured to work. A Type 2 driver also uses the JNI. You cannot use a Type 2 driver in an applet since applets cannot load native code. A Type 2 JDBC driver may require some Database Management System (DBMS) networking software to be installed.

The Developer Kit for Java JDBC driver is a Type 2 JDBC driver.

Type 3

These drivers use a networking protocol and middleware to communicate with a server. The server then translates the protocol to DBMS function calls specific to DBMS.

Type 3 JDBC drivers are the most flexible JDBC solution because they do not require any native binary code on the client. A Type 3 driver does not need any client installation.

Type 4

A Type 4 driver uses Java to implement a DBMS vendor networking protocol. Since the protocols are usually proprietary, DBMS vendors are generally the only companies providing a Type 4 JDBC driver.

Type 4 drivers are all Java drivers. This means that there is no client installation or configuration. However, a Type 4 driver may not be suitable for some applications if the underlying protocol does not handle issues such as security and network connectivity well.

The IBM Toolbox for Java JDBC driver is a Type 4 JDBC driver, indicating that the API is a pure Java networking protocol driver.

Related concepts

JDBC requirements
This topic indicates the requirements you need to access Core JDBC and the Java Transaction API (JTA).

Related tasks

JDBC tutorial
The following is a tutorial on writing a Java Database Connectivity (JDBC) program and running it on IBM i with the native JDBC driver. It is designed to show you the basic steps required for your program to run JDBC.

Setting up JNDI for the Java examples
DataSources work hand-in-hand with the Java Naming and Directory Interface (JNDI). JNDI is a Java abstraction layer for directory services just as Java Database Connectivity (JDBC) is an abstraction layer for databases.

JDBC requirements

This topic indicates the requirements you need to access Core JDBC and the Java Transaction API (JTA).

Before you write and deploy your JDBC applications, you may need to include specific JAR files in your classpath.

Core JDBC

For core Java Database Connectivity (JDBC) access to the local database, there are no requirements. All support is built in, preinstalled, and configured.

JDBC compliance

The native JDBC driver is compliant with all relevant JDBC specifications. The compliance level of the JDBC driver is not dependent on the IBM i release, but on the JDK release you use. The native JDBC driver's compliance level for the various JDKs is listed as follows:

Version	Jar File	JDBC Driver's Compliance Level
J2SE 7	db2_classes16.jar	JDBC 4.1
J2SE 8	db2_classes18.jar	JDBC 4.2

Related concepts

Types of JDBC drivers

This topic defines the Java Database Connectivity (JDBC) driver types. Driver types are used to categorize the technology used to connect to the database. A JDBC driver vendor uses these types to describe how their product operates. Some JDBC driver types are better suited for some applications than others.

Related tasks

JDBC tutorial
The following is a tutorial on writing a Java Database Connectivity (JDBC) program and running it on IBM i with the native JDBC driver. It is designed to show you the basic steps required for your program to run JDBC.

Setting up JNDI for the Java examples
DataSources work hand-in-hand with the Java Naming and Directory Interface (JNDI). JNDI is a Java abstraction layer for directory services just as Java Database Connectivity (JDBC) is an abstraction layer for databases.

JDBC tutorial

The following is a tutorial on writing a Java Database Connectivity (JDBC) program and running it on IBM i with the native JDBC driver. It is designed to show you the basic steps required for your program to run JDBC.

The example creates a table and populates it with some data. The program processes a query to get that data out of the database and to display it on the screen.

Run the example program

To run the example program, perform the following steps:

1. Copy the program to your workstation.
   a) Copy the example and paste it into a file on your workstation.
   b) Save the file with the same name as the public class provided and with the .java extension. In this case, you must name the file BasicJDBC.java on your local workstation.
2. Transfer the file from your workstation to your server. From a command prompt, enter the following commands:

```
ftp <server name>
<Enter your user ID>
<Enter your password>
cd /home/cujo
put BasicJDBC.java
quit
```

   For these commands to work, you must have a directory in which to put the file. In the example, /home/cujo is the location, but you can use any location you want.

   Note: It is possible that the FTP commands mentioned previously may be different for you based on how your server is set up, but they should be similar. It does not matter how you transfer the file to your server as long as you transfer it into the integrated file system.
3. Make sure you set your classpath to the directory where you put the file in so that your Java commands find the file when you run them. From a CL command line, you can use WRKENVVAR to see what environment variables are set for your user profile.
   - If you see an environment variable named CLASSPATH, you must ensure that the location where you put the .java file in is in the string of directories listed there or add it if the location has not been specified.
   - If there is no CLASSPATH environment variable, you must add one. This can be accomplished with the following command:

```
ADDENVVAR ENVVAR(CLASSPATH)
VALUE('/home/cujo:/QIBM/ProdData/Java400/jdk15/lib/tools.jar')
```

   Note: To compile Java code from the CL command, you must include the tools.jar file. This JAR file includes the javac command.

4. Compile the Java file into a class file. Enter the following command from the CL command line:

```
JAVA CLASS(com.sun.tools.javac.Main) PARM(My_Program_Name.java)
java BasicJDBC
```

You can also compile the Java file from QSH:

```
cd /home/cujo
javac BasicJDBC.java
```

QSH automatically ensures that the tools.jar file can be found. As a result, you do not have to add it to your classpath. The current directory is also in the classpath. By issuing the change directory (cd) command, the BasicJDBC.java file is also found.

Note: You can also compile the file on your workstation and use FTP to send the class file to your server in binary mode. This is an example of Java's ability to run on any platform.

Run the program by using the following command from either the CL command line or from QSH:

```
java BasicJDBC
```

The output is as follows:

```
----------------------
| 1 | Frank Johnson  |
|                    |
| 2 | Neil Schwartz  |
|                    |
| 3 | Ben Rodman     |
|                    |
| 4 | Dan Gloore     |
----------------------
There were 4 rows returned.
Output is complete.
Java program completed.
```

Related concepts

Types of JDBC drivers
This topic defines the Java Database Connectivity (JDBC) driver types. Driver types are used to categorize the technology used to connect to the database. A JDBC driver vendor uses these types to describe how their product operates. Some JDBC driver types are better suited for some applications than others.

JDBC requirements
This topic indicates the requirements you need to access Core JDBC and the Java Transaction API (JTA).

Related tasks

Setting up JNDI for the Java examples
DataSources work hand-in-hand with the Java Naming and Directory Interface (JNDI). JNDI is a Java abstraction layer for directory services just as Java Database Connectivity (JDBC) is an abstraction layer for databases.

IBM Toolbox for Java JDBC driver Web site

Oracle JDBC documentation

Example: JDBC
This is an example of how to use the BasicJDBC program. This program uses the native JDBC driver for the IBM Developer Kit for Java to build a simple table and process a query that displays the data in that table.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
//////////////////////////////////////////////////////////////////////////////////
//
// BasicJDBC example.  This program uses the native JDBC driver for the
// Developer Kit for Java to build a simple table and process a query
// that displays the data in that table.
```

```
//
// Command syntax:
//    BasicJDBC
//
//////////////////////////////////////////////////////////////////////////////////
//
// This source is an example of the native JDBC driver.
// IBM grants you a nonexclusive license to use this as an example
// from which you can generate similar function tailored to
// your own specific needs.
//
// This sample code is provided by IBM for illustrative purposes
// only. These examples have not been thoroughly tested under all
// conditions. IBM, therefore, cannot guarantee or imply
// reliability, serviceability, or function of these programs.
//
// All programs contained herein are provided to you "AS IS"
// without any warranties of any kind.  The implied warranties of
// merchantability and fitness for a particular purpose are
// expressly disclaimed.
//
// IBM Developer Kit for Java
// (C) Copyright IBM Corp. 2001
// All rights reserved.
// US Government Users Restricted Rights -
// Use, duplication, or disclosure restricted
// by GSA ADP Schedule Contract with IBM Corp.
//
//////////////////////////////////////////////////////////////////////////////////

// Include any Java classes that are to be used. In this application,
// many classes from the java.sql package are used and the
// java.util.Properties class is also used as part of obtaining
// a connection to the database.
import java.sql.*;
import java.util.Properties;

// Create a public class to encapsulate the program.
public class BasicJDBC {

    // The connection is a private variable of the object.
    private Connection connection = null;

    // Any class that is to be an 'entry point' for running
    // a program must have a main method. The main method
    // is where processing begins when the program is called.
    public static void main(java.lang.String[] args) {

        // Create an object of type BasicJDBC. This
        // is fundamental to object-oriented programming. Once
        // an object is created, call various methods on
        // that object to accomplish work.
        // In this case, calling the constructor for the object
        // creates a database connection that the other
        // methods use to do work against the database.
        BasicJDBC test = new BasicJDBC();

        // Call the rebuildTable method. This method ensures that
        // the table used in this program exists and looks
        // correct. The return value is a boolean for
        // whether or not rebuilding the table completed
        // successfully. If it did no, display a message
        // and exit the program.
        if (!test.rebuildTable()) {
            System.out.println("Failure occurred while setting up " +
                               " for running the test.");
            System.out.println("Test will not continue.");
            System.exit(0);
        }

        // The run query method is called next. This method
        // processes an SQL select statement against the table that
        // was created in the rebuildTable method. The output of
        // that query is output to standard out for you to view.
        test.runQuery();

        // Finally, the cleanup method is called. This method
        // ensures that the database connection that the object has
        // been hanging on to is closed.
        test.cleanup();
    }
```

```
    /**
    This is the constructor for the basic JDBC test. It creates a database
    connection that is stored in an instance variable to be used in later
    method calls.
    **/
    public BasicJDBC() {

        // One way to create a database connection is to pass a URL
        // and a java Properties object to the DriverManager. The following
        // code constructs a Properties object that has your user ID and
        // password. These pieces of information are used for connecting
        // to the database.
        Properties properties = new Properties ();
        properties.put("user", "cujo");
        properties.put("password", "newtiger");

        // Use a try/catch block to catch all exceptions that can come out of the
        // following code.
        try {
            // The DriverManager must be aware that there is a JDBC driver available
            // to handle a user connection request. The following line causes the
            // native JDBC driver to be loaded and registered with the DriverManager.
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");

            // Create the database Connection object that this program uses in all
            // the other method calls that are made. The following code specifies
            // that a connection is to be established to the local database and that
            // that connection should conform to the properties that were set up
            // previously (that is, it should use the user ID and password specified).
            connection = DriverManager.getConnection("jdbc:db2:*local", properties);

        } catch (Exception e) {
            // If any of the lines in the try/catch block fail, control transfers to
            // the following line of code. A robust application tries to handle the
            // problem or provide more details to you. In this program, the error
            // message from the exception is displayed and the application allows
            // the program to return.
            System.out.println("Caught exception: " + e.getMessage());
        }
    }


    /**
    Ensures that the qgpl.basicjdbc table looks you want it to at the start of
    the test.

    @returns boolean    Returns true if the table was rebuild successfully;
                        returns false if any failure occurred.
    **/
    public boolean rebuildTable() {
        // Wrap all the functionality in a try/catch block so an attempt is
        // made to handle any errors that may happen within this method.
        try {

            // Statement objects are used to process SQL statements against the
            // database. The Connection object is used to create a Statement
            // object.
            Statement s = connection.createStatement();

            try {
                // Build the test table from scratch. Process an update statement
                // that attempts to delete the table if it currently exists.
                s.executeUpdate("drop table qgpl.basicjdbc");
            } catch (SQLException e) {
                // Do not perform anything if an exception occurred. Assume
                // that the problem is that the table that was dropped does not
                // exist and that it can be created next.
            }

            // Use the statement object to create our table.
            s.executeUpdate("create table qgpl.basicjdbc(id int, name char(15))");

            // Use the statement object to populate our table with some data.
            s.executeUpdate("insert into qgpl.basicjdbc values(1, 'Frank Johnson')");
            s.executeUpdate("insert into qgpl.basicjdbc values(2, 'Neil Schwartz')");
            s.executeUpdate("insert into qgpl.basicjdbc values(3, 'Ben Rodman')");
            s.executeUpdate("insert into qgpl.basicjdbc values(4, 'Dan Gloore')");

            // Close the SQL statement to tell the database that it is no longer
            // needed.
            s.close();
```

```
            // If the entire method processed successfully, return true. At this point,
            // the table has been created or refreshed correctly.
            return true;

        } catch (SQLException sqle) {
            // If any of our SQL statements failed (other than the drop of the table
            // that was handled in the inner try/catch block), the error message is
            // displayed and false is returned to the caller, indicating that the table
            // may not be complete.
            System.out.println("Error in rebuildTable: " + sqle.getMessage());
            return false;
        }
    }



    /**
    Runs a query against the demonstration table and the results are displayed to
    standard out.
    **/
    public void runQuery() {
        // Wrap all the functionality in a try/catch block so an attempts is
        // made to handle any errors that might happen within this
        // method.
        try {
            // Create a Statement object.
            Statement s = connection.createStatement();

            // Use the statement object to run an SQL query. Queries return
            // ResultSet objects that are used to look at the data the query
            // provides.
            ResultSet rs = s.executeQuery("select * from qgpl.basicjdbc");

            // Display the top of our 'table' and initialize the counter for the
            // number of rows returned.
            System.out.println("--------------------");
            int i = 0;

            // The ResultSet next method is used to process the rows of a
            // ResultSet. The next method must be called once before the
            // first data is available for viewing. As long as next returns
            // true, there is another row of data that can be used.
            while (rs.next()) {

                // Obtain both columns in the table for each row and write a row to
                // our on-screen table with the data. Then, increment the count
                // of rows that have been processed.
                System.out.println("| " + rs.getInt(1) + " | " + rs.getString(2) + "|");
                i++;
            }

            // Place a border at the bottom on the table and display the number of rows
            // as output.
            System.out.println("--------------------");
            System.out.println("There were " + i + " rows returned.");
            System.out.println("Output is complete.");

        } catch (SQLException e) {
            // Display more information about any SQL exceptions that are
            // generated as output.
            System.out.println("SQLException exception: ");
            System.out.println("Message:....." + e.getMessage());
            System.out.println("SQLState:...." + e.getSQLState());
            System.out.println("Vendor Code:." + e.getErrorCode());
            e.printStackTrace();
        }
    }


    /**
    The following method ensures that any JDBC resources that are still
    allocated are freed.
    **/
    public void cleanup() {
        try {
            if (connection != null)
                connection.close();
        } catch (Exception e) {
            System.out.println("Caught exception: ");
            e.printStackTrace();
        }
```

```
    }
}
```

Setting up JNDI for the Java examples

DataSources work hand-in-hand with the Java Naming and Directory Interface (JNDI). JNDI is a Java abstraction layer for directory services just as Java Database Connectivity (JDBC) is an abstraction layer for databases.

JNDI is used most often with the Lightweight Directory Access Protocol (LDAP), but it may also be used with the CORBA Object Services (COS), the Java Remote Method Invocation (RMI) registry, or the underlying file system. This varied use is accomplished by means of the various directory service providers that turn common JNDI requests into specific directory service requests.

Note: Keep in mind that using RMI can be a complex undertaking. Before you choose RMI as a solution, be sure that you understand the ramifications of doing so. A good place to begin assessing RMI is Java Remote Method Invocation (RMI) Home.

The DataSource samples were designed using the JNDI file system service provider. If you want to run the examples provided, there must be a JNDI service provider in place.

Follow these directions to set up the environment for the file system service provider:

1. Download the file system JNDI support from Oracle JNDI site.
2. Transfer (using FTP or another mechanism) fscontext.jar and providerutil.jar to your system and put them in /QIBM/UserData/Java400/ext. This is the extensions directory and the JAR files that you place here are found automatically when you run your application (that is, you do not need them in your classpath).

Once you have support for a service provider for JNDI, you must set up the context information for your applications. This can be accomplished by putting the required information in a SystemDefault.properties file. There are several places on the system where you can specify default properties, but the best way is to create a text file called SystemDefault.properties in your home directory (that is, at /home/).

To create a file, use the following lines or add them to your existing file:

```
# Needed env settings for JNDI.
java.naming.factory.initial=com.sun.jndi.fscontext.RefFSContextFactory
java.naming.provider.url=file:/DataSources/jdbc
```

These lines specify that the file system service provider handles JNDI requests and that /DataSources/jdbc is the root for tasks that use JNDI. You can change this location, but the directory that you specify must exist. The location that you specify is where the example DataSources are bound and deployed.

Related concepts

Types of JDBC drivers
This topic defines the Java Database Connectivity (JDBC) driver types. Driver types are used to categorize the technology used to connect to the database. A JDBC driver vendor uses these types to describe how their product operates. Some JDBC driver types are better suited for some applications than others.

JDBC requirements
This topic indicates the requirements you need to access Core JDBC and the Java Transaction API (JTA).

Related tasks

JDBC tutorial
The following is a tutorial on writing a Java Database Connectivity (JDBC) program and running it on IBM i with the native JDBC driver. It is designed to show you the basic steps required for your program to run JDBC.

Connections

The Connection object represents a connection to a data source in Java Database Connectivity (JDBC). It is through Connection objects that Statement objects are created for processing SQL statements against the database. An application program can have multiple connections at one time. These Connection objects can all connect to the same database or connect to different databases.

Obtaining a connection in JDBC can be accomplished in two ways:

- Through the DriverManager class.
- By using DataSources.

Using DataSources to obtain a connection is preferred because it enhances application portability and maintainability. It also allows an application to transparently use connection and statement pooling, and distributed transactions.

Related concepts

Create various types of Statement objects for interacting with the database
A Statement object is used for processing a static SQL statement and obtaining the results produced by it. Only one ResultSet for each Statement object can be open at a time. All statement methods that process an SQL statement implicitly close a statement's current ResultSet if an open one exists.

Control transactions against the database
A transaction is a logical unit of work. To complete a logical unit of work, several actions may need to be taken against a database.

Retrieve metadata about the database
The DatabaseMetaData interface is implemented by the IBM Developer Kit for Java JDBC driver to provide information about its underlying data sources. It is used primarily by application servers and tools to determine how to interact with a given data source. Applications may also use DatabaseMetaData methods to obtain information about a data source, but this is less typical.

Java DriverManager class

DriverManager is a static class in the Java 2 Plaform, Standard Edition (J2SE) and Java SE Development Kit (JDK). DriverManager manages the set of Java Database Connectivity (JDBC) drivers that are available for an application to use.

Applications can use multiple JDBC drivers concurrently if necessary. Each application specifies a JDBC driver by using a Uniform Resource Locator (URL). By passing a URL for a specific JDBC driver to the DriverManager, the application informs the DriverManager about which type of JDBC connection should be returned to the application.

Prior to JDBC 4.0, the DriverManager must be made aware of the available JDBC drivers so it can hand out connections. By making a call to the Class.forName method, it loads a class into the running Java virtual machine (JVM) based on its string name that is passed into the method. The following is an example of the class.forName method being used to load the native JDBC driver:

Example: Load the native JDBC driver

```
// Load the native JDBC driver into the DriverManager to make it
// available for getConnection requests.

Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
```

JDBC drivers are designed to tell the DriverManager about themselves automatically when their driver implementation class loads.

In JDBC 4.0, JDBC driver are automatically loaded and the Class.forName call is not necessary.

Once the native JDBC driver is available for the DriverManager with which to work, the following line of code requests a Connection object using the native JDBC URL:

Example: Request a Connection object

```
// Get a connection that uses the native JDBC driver.

Connection c = DriverManager.getConnection("jdbc:db2:*local");
```

The simplest form of JDBC URL is a list of three values that are separated by colons. The first value in the list represents the protocol which is always `jdbc` for JDBC URLs. The second value is the subprotocol and db2 or `db2iSeries` is used to specifiy the native JDBC driver. The third value is the system name to establish the connection to a specific system. There are two special values that can be used to connect to the local database. They are *LOCAL and localhost (both are case insensitive). A specific system name can also be provided as follows:

```
Connection c =
  DriverManager.getConnection("jdbc:db2:rchasmop");
```

This creates a connection to the rchasmop system. If the system to which you are trying to connect is a remote system (for example, through the Distributed Relational Database Architecture), the system name from the relational database directory must be used.

Note: When not specified, the user ID and password currently used to sign in are also used to establish the connection to the database.

Note: The IBM DB2 JDBC Universal driver also uses the db2 subprotocol. To assure that the native JDBC driver will handle the URL, applications need to use the jdbc:db2iSeries:xxxx URL instead of the jdbc:db2:xxxx URL. If the application does not want the native driver to accept URLS with the db2 subprotocol, then the application should load the class com.ibm.db2.jdbc.app.DB2iSeriesDriver, instead of com.ibm.db2.jdbc.app.DB2Driver. When this class is loaded, the native driver no longer handles URLs containing the db2 subprotocol.

Properties

The DriverManager.getConnection method takes a single string URL indicated previously and is only one of the methods on DriverManager to obtain a Connection object. There is also another version of the DriverManager.getConnection method that takes a user ID and password. The following is an example of this version:

Example: DriverManager.getConnection method taking a user ID and password

```
// Get a connection that uses the native JDBC driver.

Connection c = DriverManager.getConnection("jdbc:db2:*local", "cujo", "newtiger");
```

The line of code attempts to connect to the local database as user cujo with password newtiger no matter who is running the application. There is also a version of the DriverManager.getConnection method that takes a java.util.Properties object to allow further customization. The following is an example:

Example: DriverManager.getConnection method taking a java.util.Properties object

```
// Get a connection that uses the native JDBC driver.

Properties prop = new java.util.Properties();
prop.put("user", "cujo");
prop.put("password","newtiger");
Connection c = DriverManager.getConnection("jdbc:db2:*local", prop);
```

The code is functionally equivalent to the version previously mentioned that passed the user ID and password as parameters.

Refer to Connection properties for a complete list of connection properties for the native JDBC driver.

URL properties

Another way to specify properties is to place them in a list on the URL object itself. Each property in the list is separated by a semi-colon and the list must be of the form `property name=property value`. This is just a shortcut and does not significantly change the way processing is performed as the following example shows:

Example: Specify URL properties

```
// Get a connection that uses the native JDBC driver.

Connection c =
DriverManager.getConnection("jdbc:db2:*local;user=cujo;password=newtiger");
```

The code is again functionally equivalent to the examples mentioned previously.

If a property value is specified in both a properties object and on the URL object, the URL version takes precedence over the version in the properties object. The following is an example:

Example: URL properties

```
// Get a connection that uses the native JDBC driver.
Properties prop = new java.util.Properties();
prop.put("user", "someone");
prop.put("password","something");
Connection c =
DriverManager.getConnection("jdbc:db2:*local;user=cujo;password=newtiger",
prop);
```

Related concepts

Using DataSources with UDBDataSource
DataSource interfaces allow additional flexibility in using Java Database Connectivity (JDBC) drivers.

Related reference

JDBC driver connection properties
This table contains valid JDBC driver connection properties, their values, and their descriptions.

DataSource properties
For each JDBC driver connection property, there is a corresponding data source method. This table contains the valid data source properties.

JDBC driver connection properties

This table contains valid JDBC driver connection properties, their values, and their descriptions.

Property	Values	Meaning
access	all, read call, read only	This value can be used to restrict the type of operations that can be done with a specific connection. The default value is all and basically means that the connection has full access to the JDBC API. The read call value allows the connection to do only queries and call stored procedures. An attempt to update the database through an SQL statement is stopped. The read only value can be used to restrict a connection to only queries. Stored procedure calls and update statements are stopped.
auto commit	true, false	This value is used to set the auto commit setting of the connection. The default value is true unless the transaction isolation property has been set to a value other than none. In that case, the default value is false.
batch style	2.0, 2.1	The JDBC 2.1 specification defines a second method for how exceptions in a batch update can be handled. The driver can comply with either of these. The default is to work as defined in the JDBC 2.0 specification.

Property	Values	Meaning
block size	0, 8, 16, 32, 64, 128, 256, 512	This is the number of rows that are fetched at a time for a result set. For typical forward-only processing of a result set, a block of this size is obtained. Then the database is not accessed because each row is processed by your application. The database requests another block of data only when the end of the block is reached. This value is only used if the blocking enabled property is set to true. Setting the block size property to 0 has the same effect as setting the blocking enabled property to false. The default is to use blocking with a block size of 32. This is a fairly arbitrary decision and the default could change in the future. Blocking is not used on scrollable result sets.
blocking enabled	true, false	This value is used to determine if the connection uses blocking on result set row retrieval. Blocking can significantly improve the performance of processing result sets. By default, this property is set to true.
commit hold	false, true	This value specifies if "commit hold" is used when the `connection.commit()` method is called. When "commit hold" is used, cursors and other database resources are not closed or freed when commit is called. The default value is false.
concurrent access resolution	1, 2, 3	This property specifies whether "currently committed" access is used on the connection. The following values are possible: **1** "currently committed" will be used **2** "wait for outcome" will be used **3** "skip locks" will be used The default value is 2.

Property	Values	Meaning
cursor hold	true, false	This value specifies if result sets remain open when a transaction is committed. A value of true means that an application is able to access its open result sets after commit is called. A value of false means that commit closes any open cursors under the connection. By default, this property is set to true. This value property serves as a default value for all result sets made for the connection. With cursor holdability support added in JDBC 3.0, this default is simply replaced if an application specifies a different holdability later. If you are migrating to JDBC 3.0 from an earlier version, be aware that cursor holdability support was not added until JDBC 3.0. In earlier versions, the default value of "true" was sent at connect time, but it was not yet recognized by the JVM. Therefore, the cursor hold property will not impact database functionality until JDBC 3.0.
cursor sensitivity	asensitive, sensitive	Specifies the cursor sensitivity used by ResultSet.TYPE_SCROLL_SENSITIVE cursors. By default, the native JDBC driver creates asensitive cursors for ResultSet.TYPE_SCROLL_SENSITIVE cursors.
data truncation	true, false	This value specifies if truncation of character data should cause warnings and exceptions to be generated (true) or if the data should just be silently truncated (false). If the default is true, data truncation of character fields are honored.
date format	julian, mdy, dmy, ymd, usa, iso, eur, jis	This property allows you to change how dates are formatted.
date separator	/(slash), -(dash), .(period), ,(comma), b	This property allows you to change what the date separator is. This is only valid in combination with some of the dateFormat values (according to system rules).
decfloat rounding mode	round half even, round half up, round down, round ceiling, round floor, round half down, round up, round half even	This property specifies the rounding mode to be used by decimal float operations. The default value is round half even.
decimal separator	.(period), ,(comma)	This property allows you to change what the decimal separator is.
direct map	true, false	This property specifies if database direct map optimizations will be used when retrieving result sets from the database. The default value is true.

Property	Values	Meaning
do escape processing	true, false	This property sets a flag for if statements under the connection must do escape processing. Using escape processing is a way to code your SQL statements so that they are generic and similar for all platforms, but then the database reads the escape clauses and substitutes the proper system specific version for the user. This is good, except that it forces extra work on the system. In the case where you know you are only using SQL statements that already contain valid IBM i SQL syntax, it is recommended that this value be set to false to increase performance. The default value for this property is true, as it must be compliant with the JDBC specification (that is, escape processing is active by default). This value is added due to a shortcoming of the JDBC specification. You can only set escape processing to off in the Statement class. That works well if you are dealing with simple statements. You create your statement, turn off escape processing, and start processing statements. However, in the case of prepared statements and callable statements, this scheme does not work. You supply the SQL statement at the time that the prepared or callable statement is constructed and it does not change after that. So the statement is prepared up front and changing the escape processing after that is meaningless. Having this connection property allows a way to get around the extra overhead.
errors	basic, full	This property allows the full system second-level error text to be returned in SQLException object messages. The default is basic which returns only the standard message text.
extended metadata	true, false	The property specifies if the driver should request extended metadata from the database. Setting this property to true increases the accuracy of the information returned from the following ResultSetMetaData methods: • getColumnLabel(int) • getSchemaName(int) • getTableName(int) • isReadOnly(int) • isSearchable(int) • isWriteable(int) Setting this property to true may slow performance because it requires retrieving more information from the database.

Property	Values	Meaning
ignore warnings	A command separated list of SQL states that should be ignored.	By default, the native JDBC driver will internally create a java.sql.SQLWarning object for each warning returned by the database. This property specifies a list of SQL states for which the native JDBC driver should not create warning objects. For example, a warning with the SQLSTATE 0100C is created every time a result set is returned from a stored procedure. This warning can be safely ignored to improve the performance of applications that call stored procedures.
libraries	A space-separated list of libraries. (A list of libraries can also be separated by colons or commas.)	This property allows a list of libraries to be placed into the server job's library list or a specific default library to be set. The naming property affects how this property works. In the default case, where naming is set to sql, JDBC works like ODBC. The library list has no effect on how the connection processes. There is a default library for all unqualified tables. By default, that library has the same name as the user profile that is connected. If the libraries property is specified, the first library in the list becomes the default library. If a default library is specified on the connection URL (as in jdbc:db2:*local/mylibrary), that overrides any value in this property. In the case where naming is set system, each of the libraries specified for this property is added to the user portion of the library list and the library list is searched to resolve unqualified table references.
lob block size	4096,65536,262144,1048576,4194304,16777216	This property specifies the number of bytes read from the InputStream when processing a block of data. The default value of this connection property is 1048576 bytes. Lower values will typically cause slower performance when processing lob data.
lob threshold	Any value under 500000	This property tells the driver to place the actual values into the result set storage instead of locators for lob columns if the lob column is smaller than the threshold size. This property acts against the column size, not the lob data size itself. For example, if the lob column is defined to hold up to 1 MB for each lob, but all the column values are under 500 KB, locators are still used. Note that the size limit is set as it is to allow blocks of data to be fetched without danger of not always growing data blocks larger than the 16 MB maximum allocation size. With large result sets, it is still easy to exceed this limit, which causes fetches to fail. Care must be taken in how the block size property and this property interact with the size of a data block. The default is 0. Locators are always used for lob data.
maximum precision	31, 63	This value specifies the maximum precision used for decimal and numeric types. The default value is 31.

Property	Values	Meaning
maximum scale	0-63	This value specifies the maximum scale (number of decimal positions to the right of the decimal point) that is returned used by decimal and numeric types. The value can range from 0 to the maximum precision. The default value is 31.
minimum divide scale	0-9	This value specifies the minimum divide scale (number of decimal positions to the right of the decimal point) that is returned for both intermediary and result data types. The value can range from 0 to 9, not to exceed the maximum scale. If 0 is specified, minimum divide scale is not used. The default value is 0.
naming	sql, system	This property allows you to use either the traditional IBM i naming syntax or the standard SQL naming syntax. System naming means that you use a /(slash) character to separate collection and table values, and SQL naming means that you use a .(period) character to separate the values. The setting of this value has ramifications for what the default library is also. See the libraries property above for further information about this. The default is to use SQL naming.
password	anything	This property allows for a password to be specified for the connection. This property does not work correctly without also specifying the user property. These properties allow for connections to be made to the database as a user other than the one that is running the IBM i job. Specifying the user and password properties have the same effect as using the connection method with the signature `getConnection(String url, String userId, String password)`.
prefetch	true, false	This property specifies if the driver fetches the first data for a result set immediately after processing or wait until the data is requested. If the default is true, data is prefetched. For applications using the Native JDBC driver, this is rarely an issue. The property exists primarily for internal use with Java stored procedures and user-defined functions where it is important that the database engine does not fetch any data from result sets on your behalf before you request it.
qaqqinilib	library name	This property specifies the library that contains the qaqqini file to use. A qaqqini file contains all of the attributes that can potentially impact the performance of the Db2® for i database engine.

Property	Values	Meaning
query optimize goal	1, 2	This property specifies the goal the server should use with optimization of queries. This setting corresponds to the server's QAQQINI option called OPTIMIZATION_GOAL. The following values are possible: **1** Optimize query for first block of data (*FIRSTIO) **2** Optimize query for entire result set (*ALLIO) The default value is 2.
reuse objects	true, false	This property specifies if the driver attempts to reuse some types of objects after you close them. This is a performance enhancement. The default is true.

servermode subsystem	*SAME, subsystem name	This property specifies the subsystem in which the associated QSQSRVR jobs will run. The default behavior is to have the jobs run in the QSYSWRK subsystem. If the value *SAME is used, then the QSQSRVR jobs will run in the same subsystem as the job using the native JDBC driver.
		In order for a QSQSRVR job to run in a different subsystem, a QSQSRVR prestart job entry for that subsystem must exist. The following commands can be used to create a QSQSRVR prestart job entry.
		`ENDSBS sbs`
		`ADDPJE SBSD(library/sbsd) PGM(QSYS/QSQSRVR) STRJOBS(*YES) INLJOBS(x) THRESHOLD(y) ADLJOBS(z) MAXUSE(*NOMAX)`
		`STRSBS sbs`
		Where *sbs* is the subsystem, *library* is the library in which the subsystem description *sbsd* is located in, and *x*, *y*, and *z* are numeric values for the corresponding parameters on the Add Prestart Job Entry (ADDPJE) command.
		If a prestart job entry does not exist for QSQSRVR within the subsystem, the QSQSRVR job will utilize a Batch Immediate job (BCI) instead of a Prestart job (PJ). This Batch Immediate job typically runs in the same subsystem as the job using the native JDBC driver.

time format	hms, usa, iso, eur, jis	This property allows you to change how time values are formatted.
time separator	:(colon), .(period), ,(comma), b	This property allows you to change what the time separator is. This is only valid in combination with some of the timeFormat values (according to system rules).
trace	true, false	This property allows for turning on tracing of the connection. It can be used as a simple debugging aide. The default value is false, which does not use tracing.

Property	Values	Meaning
transaction isolation	none, read committed, read uncommitted, repeatable read, serializable	This property allows you to set the transaction isolation level for the connection. There is no difference between setting this property to a specific level and specifying a level on the setTransactionIsolation method in the Connection interface. The default value for this property is none, as JDBC defaults to auto-commit mode.
translate binary	true, false	This property can be used to force the JDBC driver to treat binary and varbinary data values as if they were char and varchar data values. When binary data is to be treated the same as character data, the CCSID of the job will be used as the CCSID of the data. The default for this property is false, where binary data is not treated the same as character data.
translate hex	binary, character	This value is used to select the data type used by hex constants in SQL expression. The binary setting indicates that hex constants will use the BINARY data type. The character setting indicates that hex contants will use the CHARACTER FOR BIT DATA data type. The default setting is character.
use block insert	true, false	This property allows the native JDBC driver to go into a block insert mode for inserting blocks of data into the database. This is an optimized version of the batch update. This optimized mode can only be used in applications that ensure that they do not break certain system constraints or data insert failures and potentially corrupt data. Applications that turn on this property only connect to the local system when attempting to perform batched updates. They do use DRDA to establish remote connections because blocked insert cannot be managed over DRDA. Applications must also ensure that PreparedStatements with an SQL insert statement and a values clause make all the insert values parameters. No constants are permitted in the values list. This is a requirement of the blocked insert engine of the system. The default is false.
user	anything	This property allows for a user ID to be specified for the connection. This property does not work correctly without also specifying the password property. These properties allow for connections to be made to the database as a user other than the one that is running the IBM i job. Specifying the user and password properties has the same effect as using the connection method with the signature getConnection(String url, String userId, String password).

Related concepts

Java DriverManager class
DriverManager is a static class in the Java 2 Plaform, Standard Edition (J2SE) and Java SE Development Kit (JDK). DriverManager manages the set of Java Database Connectivity (JDBC) drivers that are available for an application to use.

Using DataSources with UDBDataSource
DataSource interfaces allow additional flexibility in using Java Database Connectivity (JDBC) drivers.

Related reference

DataSource properties
For each JDBC driver connection property, there is a corresponding data source method. This table contains the valid data source properties.

Using DataSources with UDBDataSource

DataSource interfaces allow additional flexibility in using Java Database Connectivity (JDBC) drivers.

The use of DataSources can be split into two phases:

- **Deployment**

  Deployment is a setup phase that occurs before a JDBC application actually runs. Deployment usually involves setting up a DataSource to have specific properties and then binding it into a directory service through the use of the Java Naming and Directory Interface (JNDI). The directory service is most commonly the Lightweight Directory Access Protocol (LDAP), but could be a number of others such as Common Object Request Broker Architecture (CORBA) Object Services, Java Remote Method Invocation (RMI), or the underlying file system.

- **Use**

  By decoupling the deployment from the runtime use of the DataSource, the DataSource setup can be reused by many applications. By changing some aspect of the deployment, all the applications that use that DataSource automatically pick up the changes.

Note: Keep in mind that using RMI can be a complex undertaking. Before you choose RMI as a solution, be sure that you understand the ramifications of doing so.

An advantage of DataSources is that they allow JDBC drivers to do work on behalf of the application without having an impact on the application development process directly. For more information, see the following:

- "Using DataSource support for object pooling" on page 117
- "DataSource-based statement pooling" on page 121
- "JDBC distributed transactions" on page 70

UDBDataSourceBind

The "Example: Creating a UDBDataSource and binding it with JNDI" on page 47 program is an example of creating a UDBDataSource and getting it bound with JNDI. This program accomplishes all the basic tasks requested. Namely, it instantiates a UDBDataSource object, sets properties on this object, retrieves a JNDI context, and binds the object to a name within the JNDI context.

The deployment time code is vendor-specific. The application must import the specific DataSource implementation that it wants to work with. In the import list, the package-qualified UDBDataSource class is imported. The most unfamiliar part of this application is the work done with JNDI (for example, the retrieval of the Context object and the call to bind). For additional information, see JNDI by Oracle.

Once this program has been run and has successfully completed, there is a new entry in a JNDI directory service called SimpleDS. This entry is at the location specified by the JNDI context. The DataSource implementation is now deployed. An application program can make use of this DataSource to retrieve database connections and JDBC-related work.

UDBDataSourceUse

The “Example: Obtaining an initial context before binding UDBDataSource” on page 48 program is an example of a JDBC application that uses the previously deployed application.

The JDBC application obtains an initial context as it did before binding the UDBDataSource in the previous example. The lookup method is then used on that context to return an object of type DataSource for the application to use.

Note: The runtime application is only interested in the methods of the DataSource interface, so there is no need for it to be aware of the implementation class. This makes applications portable.

Suppose that UDBDataSourceUse is a complex application that runs a large operation within your organization. You have a dozen or more similar large applications within your organization. You have to change the name of one of the systems in your network. By running a deployment tool and changing a single UDBDataSource property, you would be able to get this new behavior in all your applications without changing the code for them. One of the benefits of DataSources is that they allow you to consolidate system setup information. Another major benefit is that they allow drivers to implement functionality invisible to the application such as connection pooling, statement pooling and support for distributed transactions.

After analyzing UDBDataSourceBind and UDBDataSourceUse closely, you may have wondered how the DataSource object knew what to do. There is no code to specify a system, a user ID, or a password in either of these programs. The UDBDataSource class has defaults values for all properties; by default, it connects to the local IBM i with the user profile and password of the running application. If you wanted to ensure that the connection was made with the user profile cujo instead, you could have accomplished this in two ways:

- Set the user ID and password as DataSource properties.
- Use the DataSource getConnection method that takes a user ID and password at runtime.

There are a number of properties that can be specified for the UDBDataSource as there are properties that can be specified for connections created with the DriverManager. Refer to “DataSource properties” on page 49 for a list of supported properties for the native JDBC driver.

While these lists are similar, it is not certain to be similar in future releases. You are encouraged to start coding to the DataSource interface.

Note: The native JDBC driver also has two other DataSource implementations: DB2DataSource and DB2StdDataSource. These implementations have been deprecated and direct use of them is not recommended. These implementations may be removed in a future release.

Related concepts

Java DriverManager class
DriverManager is a static class in the Java 2 Plaform, Standard Edition (J2SE) and Java SE Development Kit (JDK). DriverManager manages the set of Java Database Connectivity (JDBC) drivers that are available for an application to use.

Related reference

JDBC driver connection properties
This table contains valid JDBC driver connection properties, their values, and their descriptions.

DataSource properties
For each JDBC driver connection property, there is a corresponding data source method. This table contains the valid data source properties.

Example: Creating a UDBDataSource and binding it with JNDI
This is an example of how to create a UDBDataSource and bind it with JNDI.

Note: By using the code examples, you agree to the terms of the “Code license and disclaimer information” on page 489.

```
// Import the required packages. At deployment time,
// the JDBC driver-specific class that implements
```

```
// DataSource must be imported.
import java.sql.*;
import javax.naming.*;
import com.ibm.db2.jdbc.app.UDBDataSource;

public class UDBDataSourceBind
{
    public static void main(java.lang.String[] args)
    throws Exception
    {
        // Create a new UDBDataSource object and give it
        // a description.
        UDBDataSource ds = new UDBDataSource();
        ds.setDescription("A simple UDBDataSource");

        // Retrieve a JNDI context. The context serves
        // as the root for where objects are bound or
        // found in JNDI.
        Context ctx = new InitialContext();

        // Bind the newly created UDBDataSource object
        // to the JNDI directory service, giving it a name
        // that can be used to look up this object again
        // at a later time.
        ctx.rebind("SimpleDS", ds);
    }
}
```

Example: Obtaining an initial context before binding UDBDataSource
The following example obtains an initial context before binding the UDBDataSource. The lookup method is then used on that context to return an object of type DataSource for the application to use.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
// Import the required packages. There is no
// driver-specific code needed in runtime
// applications.
import java.sql.*;
import javax.sql.*;
import javax.naming.*;

public class UDBDataSourceUse
{
    public static void main(java.lang.String[] args)
    throws Exception
    {
        // Retrieve a JNDI context. The context serves
        // as the root for where objects are bound or
        // found in JNDI.
        Context ctx = new InitialContext();

        // Retrieve the bound UDBDataSource object using the
        // name with which it was previously bound. At runtime,
        // only the DataSource interface is used, so there
        // is no need to convert the object to the UDBDataSource
        // implementation class. (There is no need to know what
        // the implementation class is. The logical JNDI name is
        // only required).
        DataSource ds = (DataSource) ctx.lookup("SimpleDS");


        // Once the DataSource is obtained, it can be used to establish
        // a connection. This Connection object is the same type
        // of object that is returned if the DriverManager approach
        // to establishing connection is used. Thus, so everything from
        // this point forward is exactly like any other JDBC
        // application.
        Connection connection = ds.getConnection();

        // The connection can be used to create Statement objects and
        // update the database or process queries as follows.
        Statement statement = connection.createStatement();
        ResultSet rs = statement.executeQuery("select * from qsys2.sysprocs");
        while (rs.next()) {
            System.out.println(rs.getString(1) + "." + rs.getString(2));
        }
```

```
        // The connection is closed before the application ends.
        connection.close();
    }
}
```

DataSource properties

For each JDBC driver connection property, there is a corresponding data source method. This table contains the valid data source properties.

For some properties, you can refer to the corresponding driver connection property for more information.

Set method (data type)	Values	Description
setAccess(String)	"all", "read call", "read only"	Refer to the access connection property.
setAutoCommit(boolean)	"true", "false"	Refer to the auto commit connection property.
setBatchStyle(String)	"2.0", "2.1"	Refer to the batch style connection property.
setBlockSize(int)	"0", "8", "16", "32", "64", "128", "256", "512"	Refer to the block size connection property.
setCommitHold(boolean)	"true", "false"	Refer to the commit hold connection property.
setConcurrentAccessResolution(int)	1,2,3	Refer to the concurrent access resolution property
setCursorHold(boolean)	"true", "false"	Refer to the cursor hold connection property.
setCursorSensitivity(String)	"sensitive", "asensitive"	Refer to the cursor sensitivity connection property.
setDataTruncation(boolean)	"true", "false"	Refer to the data truncation connection property.
setDatabaseName(String)	Any name	This property specifies the database to which the DataSource attempts to connect. The default is *LOCAL. The database name must either exist in the relational database directory on the system that runs the application or be the special value *LOCAL or localhost to specify the local system.
setDataSourceName(String)	Any name	This property allows the passing of a ConnectionPoolDataSource Java Naming and Directory Interface (JNDI) name to support connection pooling.
setDateFormat(String)	"julian", "mdy", "dmy", "ymd", "usa", "iso", "eur", "jis"	Refer to the date format connection property.
setDateSeparator(String)	"/", "-", ".", ",", "b"	Refer to the date separator connection property.
setDecimalSeparator(String)	".", ","	Refer to the decimal separator connection property.

Set method (data type)	Values	Description
setDecfloatRoundingMode(String)	"round half even", "round half up", "round down", "round ceiling", "round floor", "round half down", "round up", "round half even"	Refer to the decfloat rounding mode connection property.
setDescription(String)	Any name	This property allows the setting of this DataSource object's text description.
setDirectMap(boolean)	"true", "false"	Refer to the direct map connection property.
setDoEscapeProcessing(boolean)	"true", "false"	Refer to the do escape processing connection property.
setFullErrors(boolean)	"true", "false"	Refer to the errors connection property.
setIgnoreWarnings(String)	Comma separated list of SQLSTATEs.	Refer to the ignore warnings connection property.
setLibraries(String)	A space-separated list of libraries	Refer to the libraries connection property.
setLobThreshold(int)	Any value under 500000	Refer to the lob threshold connection property.
setLoginTimeout(int)	Any value	This property is currently ignored and is planned for future use.
setMaximumPrecision(int)	31, 63	Refer to the maximum precision connection property.
setMaximumScale(int)	0-63	Refer to the maximum scale connection property.
setMinimumDivideScale(int)	0-9	Refer to the minimum divide scale connection property.
setNetworkProtocol(int)	Any value	This property is currently ignored and is planned for future use.
setPassword(String)	Any string	Refer to the password connection property.
setPortNumber(int)	Any value	This property is currently ignored and is planned for future use.
setPrefetch(boolean)	"true", "false"	Refer to the prefetch connection property.
setQaqqinilib(String)	library name	Refer to the qaqqinilib connection property.
setQueryOptimizeGoal(String)	1, 2	Refer to the query optimize goal connection property.
setReuseObjects(boolean)	"true", "false"	Refer to the reuse objects connection property.
setServermodeSubsystem(String)	"*SAME", subsystem name	Refer to the servermode subsystem property

Set method (data type)	Values	Description
setServerName(String)	Any name	This property is currently ignored and is planned for future use.
setSystemNaming(boolean)	"true", "false"	Refer to the naming connection property.
setTimeFormat(String)	"hms", "usa", "iso", "eur", "jis"	Refer to the time format connection property.
setTimeSeparator(String)	":", ".", ",", "b"	Refer to the time separator connection property.
setTransactionIsolationLevel(String)	"none", "read committed", "read uncommitted", "repeatable read", "serializable"	Refer to the transaction isolation connection property.
setTranslateBinary(Boolean)	"true", "false"	Refer to the translate binary connection property.
setUseBlockInsert(boolean)	"true", "false"	Refer to the use block insert connection property.
setUser(String)	anything	Refer to the user connection property.

Related concepts

Java DriverManager class
DriverManager is a static class in the Java 2 Plaform, Standard Edition (J2SE) and Java SE Development Kit (JDK). DriverManager manages the set of Java Database Connectivity (JDBC) drivers that are available for an application to use.

Using DataSources with UDBDataSource
DataSource interfaces allow additional flexibility in using Java Database Connectivity (JDBC) drivers.

Related reference

JDBC driver connection properties
This table contains valid JDBC driver connection properties, their values, and their descriptions.

JVM properties for JDBC

Some settings used by the native JDBC driver cannot be set using a connection property. These settings must be set for the JVM in which the native JDBC driver is running. These settings are used for all connections created by the native JDBC driver.

The native driver recognizes the following JVM properties:

Property	Values	Meaning
jdbc.db2.cli.trace	true,false	Setting this value to true will cause the native JDBC driver to trace the CLI calls made by the driver.

Property	Values	Meaning
jdbc.db2.job.sort.sequence	default value = *HEX	Setting this property to true causes the native JDBC driver to use the Job Sort Sequence of the user that starts the job instead of using the default value of *HEX. Setting it to anything else or leaving it unset will cause JDBC to continue to use the default of *HEX. Take careful note of what this means. When JDBC connections pass in different user profiles on connection requests, the sort sequence of the user profile that starts the server is used for all of the connections. This is an environment attribute that is set at startup time, not a dynamic connection attribute.
jdbc.db2.statementDumpThreshold	integer value	Setting this value specifies the threshold at which the JDBC driver will dump information about the currently allocated statements. This information is usefully when debugging issues where statements are not being correctly closed. The output is placed in the directory /tmp/jdbcHandleDumps-XXXX where XXXX is the job name. In that directory there will be a file for each connection dumped. The information will be first dumped when the threshold is reached. The threshold is then changed to two times the previous threshold. The information will be dumped again if the new threshold is reached.
jdbc.db2.trace	1 or error = Trace error information 2 or info = Trace information and error information 3 or verbose = Trace verbose, information, and error information 4 or all or true = Trace all possible information	This property turns on tracing for the JDBC driver. It should be used when reporting a problem.

Property	Values	Meaning
jdbc.db2.trace.config	stdout = Trace information is sent to stdout (default value) usrtrc = Trace information is sent to a user trace. The CL command Dump User Trace Buffer (DMPUSRTRC) can be used to obtain the trace information. file://<pathtofile> = Trace information is send to a file. If the file name contains "%j", the "%j" will be replaced by the job name. An example of <pathtofile> is /tmp/jdbc.%j.trace.txt.	This property is used to specify where the output of the trace should go.
jdbc.db2.traceMonitor	true,false	Setting this value to true enables dynamic tracing for the Native JDBC driver. The trace settings can then be **modified** dynamically by running the JDBC trace administration program (com.ibm.db2.jdbc.T). This program is run with the following parameters. java com.ibm.db2.jdbc.app.T <JOBNUMBER-USER-NAME> <traceCommands>... <traceCommands> one or more of: jdbc.db2.trace=0 | 1 | 2 | 3 jdbc.db2.cli.trace=true | false jdbc.db2.trace.config=stdout | usrtrc | file://<pathtofile> Where an example of <pathtofile> is /home/jdbc.%j.trace.txt

DatabaseMetaData interface

The DatabaseMetaData interface is implemented by the IBM Developer Kit for Java JDBC driver to provide information about its underlying data sources. It is used primarily by application servers and tools to determine how to interact with a given data source. Applications may also use DatabaseMetaData methods to obtain information about a data source, but this is less typical.

The DatabaseMetaData interface includes over 150 methods that can be categorized according to the types of information they provide. These are described below. The DatabaseMetaData interface also contains over 40 fields that are constants used as return values for various DatabaseMetaData methods.

See "Changes in JDBC 3.0" and "Changes in JDBC 4.0" below for information about changes made to methods in the DatabaseMetaData interface.

Creating a DatabaseMetaData object

A DatabaseMetaData object is created with the Connection method getMetaData. Once the object is created, it can be used to dynamically find information about the underlying data source. The following example creates a DatabaseMetaData object and uses it to determine the maximum number of characters allowed for a table name:

Example: Create a DatabaseMetaData object

```
// con is a Connection object
DatabaseMetaData dbmd = con.getMetadata();
int maxLen = dbmd.getMaxTableNameLength();
```

Retrieving general information

Some DatabaseMetaData methods are used to dynamically find general information about a data source as well as to obtain details about its implementation. Some of these methods include the following:

- getURL
- getUserName
- getDatabaseProductVersion, getDriverMajorVersion, and getDriverMinorVersion
- getSchemaTerm, getCatalogTerm, and getProcedureTerm
- nullsAreSortedHigh, and nullsAreSortedLow
- usesLocalFiles, and usesLocalFilePerTable
- getSQLKeywords

Determining feature support

A large group of DatabaseMetaData methods can be used to determine whether a given feature or set of features is supported by the driver or underlying data source. Beyond this, there are methods that describe what level of support is provided. Some of the methods that describe support for individual features include the following:

- supportsAlterTableWithDropColumn
- supportsBatchUpdates
- supportsTableCorrelationNames
- supportsPositionedDelete
- supportsFullOuterJoins
- supportsStoredProcedures
- supportsMixedCaseQuotedIdentifiers

Methods to describe a level of feature support include the following:

- supportsANSI92EntryLevelSQL
- supportsCoreSQLGrammar

Data source limits

Another group of methods provide the limits imposed by a given data source. Some of the methods in this category include the following:

- getMaxRowSize
- getMaxStatementLength
- getMaxTablesInSelect
- getMaxConnections
- getMaxCharLiteralLength
- getMaxColumnsInTable

Methods in this group return the limit value as an integer. A return value of zero means that there is either no limit or the limit is unknown.

SQL objects and their attributes

A number of DatabaseMetaData methods provide information about the SQL objects that populate a given data source. These methods can determine the attributes of SQL objects. These methods also return ResultSet objects in which each row describes a particular object. For example, the getUDTs method returns a ResultSet object in which there is a row for each user-defined table (UDT) that has been defined in the data source. Examples of this category include the following:

- getSchemas and getCatalogs
- getTables
- getPrimaryKeys
- getProcedures and getProcedureColumns
- getUDTs

Transaction support

A small group of methods provide information about the transaction semantics supported by the data source. Examples of this category include the following:

- supportsMultipleTransactions
- getDefaultTransactionIsolation

See “Example: Returning a list of tables using the DatabaseMetaData interface” on page 58 for an example of how to use the DatabaseMetaData interface.

Changes in JDBC 3.0

There are changes to the return values for some of the methods in JDBC 3.0. The following methods have been updated in JDBC 3.0 to add fields to the ResultSets that they return.

- getTables
- getColumns
- getUDTs
- getSchemas

Note: If an application is being developed using Java Development Kit (JDK) 1.4, you may recognize that there are a certain number of columns being returned when testing. You write your application and expect to access all of these columns. However, if the application is being designed to also run on previous releases of the JDK, the application receives an SQLException when it tries to access these fields that do not exist in earlier JDK releases. “Example: Using metadata ResultSets that have more than one column” on page 59 is an example of how an application can be written to work with several JDK releases.

Changes in JDBC 4.0

In V6R1, the command language interface (CLI) is changing the implementation of the MetaData APIs to also call the SYSIBM stored procedures. Because of this, the JDBC MetaData methods will use the SYSIBM procedures directly on V6R1, regardless of JDK level. You will notice the following differences due to this change:

- The native JDBC driver previously permitted the user of `localhost` as the catalog name for most of the methods. In JDBC 4.0, the native JDBC driver will not return any information if `localhost` is specified.
- The native JDBC driver always returned an empty result set when the nullable parameter of getBestRowIdentifier was set to false. This will be corrected to return the proper result.
- The values returned by getColumns for the BUFFER_LENGTH, SQL_DATA_TYPE, and SQL_DATETIME_SUB columns may be different. These values should not be used in a JDBC application because the JDBC specification defines these columns as "unused."

- The native JDBC driver previously recognized the table and schema parameters of getCrossReference, getExportedKeys, getImportedKeys, and getPrimaryKeys as a "pattern." Now, the table and schema parameters must match the name as stored in the database.
- Views used for implementing system defined views were previously described by getTables() as SYSTEM TABLES. To be consistent with the DB2 family, these views are now described as VIEWS.
- Column names returned by getProcedures are different. These column names are not defined by the JDBC 4.0 specification. Also, the remarks column for getProcedures used to return "" if no information was available. It now returns null.

Table 4. Column names returned by getProcedures in JDBC 4.0

Column #	Previous name	Name under JDBC 4.0
4	RESERVED1	NUM_INPUT_PARAMS
5	RESERVED2	NUM_OUTPUT_PARAMS
6	RESERVED3	NUM_RESULT_SETS

- Some values returned by getProcedureColumns for various data types have changed, as shown below:

Table 5. Values returned by getProcedureColumns in JDBC 4.0

Data type	Column	Previous value	Value in JDBC 4.0
ALL	Remarks	""	null
INTEGER	Length	Null	4
SMALLINT	Length	Null	2
BIGINT	dataType	19 (incorrect)	-5
BIGINT	Length	Null	8
DECIMAL	Length	Null	precision + scale
NUMERIC	Length	Null	precision + scale
DOUBLE	TypeName	DOUBLE PRECISION	DOUBLE
DOUBLE	Length	Null	8
FLOAT	TypeName	DOUBLE PRECISION	DOUBLE
FLOAT	Length	Null	8
REAL	Length	Null	4
DATE	Precision	null	10
DATE	Length	10	6
TIME	Precision	null	8
TIME	Length	8	6
TIME	Scale	null	0
TIMESTAMP	Precision	null	26
TIMESTAMP	Length	26	16
TIMESTAMP	Scale	null	6
CHAR	typeName	CHARACTER	CHAR
CHAR	Precision	null	same as length

Table 5. Values returned by getProcedureColumns in JDBC 4.0 (continued)			
Data type	**Column**	**Previous value**	**Value in JDBC 4.0**
VARCHAR	typeName	CHARACTER VARYING	VARCHAR
VARCHAR	Precision	null	same as length
CLOB	dataType	null (incorrect)	2005
CLOB	typeName	CHARACTER LARGE OBJECT	CLOB
CLOB	Precision	null	same as length
CHAR FOR BIT DATA	dataType	1 (CHAR)	-2 (BINARY)
CHAR FOR BIT DATA	typeName	CHARACTER	CHAR () FOR BIT DATA
CHAR FOR BIT DATA	Precision	null	same as length
BLOB	dataType	null (incorrect)	2004
BLOB	typeName	BINARY LARGE OBJECT	BLOB
BLOB	Precision	null	same as length
DATALINK	dataType	null (incorrect)	70
DATALINK	Precision	null	same as length
VARCHAR FOR BIT DATA	dataType	12 (VARCHAR)	-3 (VARBINARY)
VARCHAR FOR BIT DATA	typeName	CHARACTER VARYING	VARCHAR () FOR BIT DATA
VARCHAR FOR BIT DATA	Precision	null	same as length

Restriction on READ ONLY stored procedures

The native JDBC supports the `access = read only` property. This property is enforced at the JDBC level. Because of this, the MetaData procedures should still work if this property is set. However, it is possible to use the native JDBC driver from a database stored procedure that is defined as READ ONLY. In this case, the MetaData procedures will not work.

New method: getClientInfoProperties()

The getClientInfoProperties method retrieves a list of the client info properties that the driver supports. Each client info property is stored in a SQL special register. The native JDBC driver will return a result set with the following information:

Table 6. Information returned by the getClientInfoProperties method			
Name	**Maximum length**	**Default value**	**Description**
ApplicationName	255	blank	The name of the application currently utilizing the connection

Table 6. Information returned by the getClientInfoProperties method (continued)			
Name	**Maximum length**	**Default value**	**Description**
ClientUser	255	blank	The name of the user that the application using the connection is performing work for. This may not be the same as the user name that was used in establishing the connection
ClientHostname	255	blank	The hostname of the computer that the application using the connection is running on
ClientAccounting	255	blank	Accounting information.

The SQL special registers corresponding to the client info properties are as follows:

Table 7. SQL special registers	
Name	**SQL special register**
ApplicationName	CURRENT CLIENT_APPLNAME
ClientUser	CURRENT CLIENT_USERID
ClientHostname	CURRENT CLIENT_WRKSTNNAME
ClientAccounting	CURRENT CLIENT_ACCTNG

The clientInfoProperties may be set using the Connection object's setClientInfo method.

Related concepts

"ResultSets" on page 100

The ResultSet interface provides access to the results generated by running queries. Conceptually, data of a ResultSet can be thought of as a table with a specific number of columns and a specific number of rows. By default, the table rows are retrieved in sequence. Within a row, column values can be accessed in any order.

Example: Returning a list of tables using the DatabaseMetaData interface

This example shows how to return a list of tables.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
// Connect to the server.
Connection c = DriverManager.getConnection("jdbc:db2:mySystem");

// Get the database meta data from the connection.
DatabaseMetaData dbMeta = c.getMetaData();

// Get a list of tables matching this criteria.
String catalog = "myCatalog";
String schema = "mySchema";
String table  = "myTable%"; // % indicates search pattern
String types[] = {"TABLE", "VIEW", "SYSTEM TABLE"}:
ResultSet rs = dbMeta.getTables(catalog, schema, table, types);

// ... iterate through the ResultSet to get the values.
```

```
// Close the connection.
c.close():
```

Related reference

Example: Using metadata ResultSets that have more than one column
This is an example of how to use metadata ResultSets that have more than one column.

Example: Using metadata ResultSets that have more than one column

This is an example of how to use metadata ResultSets that have more than one column.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
//////////////////////////////////////////////////////////////////////////////////
//
// SafeGetUDTs example. This program demonstrates one way to deal with
// metadata ResultSets that have more columns in JDK 1.4 than they
// had in previous releases.
//
// Command syntax:
//    java SafeGetUDTs
//
//////////////////////////////////////////////////////////////////////////////////
//
// This source is an example of the IBM Developer for Java JDBC driver.
// IBM grants you a nonexclusive license to use this as an example
// from which you can generate similar function tailored to
// your own specific needs.
//
// This sample code is provided by IBM for illustrative purposes
// only. These examples have not been thoroughly tested under all
// conditions. IBM, therefore, cannot guarantee or imply
// reliability, serviceability, or function of these programs.
//
// All programs contained herein are provided to you "AS IS"
// without any warranties of any kind.  The implied warranties of
// merchantability and fitness for a particular purpose are
// expressly disclaimed.
//
// IBM Developer Kit for Java
// (C) Copyright IBM Corp. 2001
// All rights reserved.
// US Government Users Restricted Rights -
// Use, duplication, or disclosure restricted
// by GSA ADP Schedule Contract with IBM Corp.
//
//////////////////////////////////////////////////////////////////////////////////

import java.sql.*;

public class SafeGetUDTs {

    public static int jdbcLevel;

    // Note: Static block runs before main begins.
    // Therefore, there is access to jdbcLevel in
    // main.
    {
        try {
            Class.forName("java.sql.Blob");

            try {
                Class.forName("java.sql.ParameterMetaData");
                // Found a JDBC 3.0 interface.  Must support JDBC 3.0.
                jdbcLevel = 3;
            } catch (ClassNotFoundException ez) {
                // Could not find the JDBC 3.0 ParameterMetaData class.
                // Must be running under a JVM with only JDBC 2.0
                // support.
                jdbcLevel = 2;
            }

        } catch (ClassNotFoundException ex) {
            // Could not find the JDBC 2.0 Blob class.  Must be
            // running under a JVM with only JDBC 1.0 support.
            jdbcLevel = 1;
        }
    }
```

```
    // Program entry point.
    public static void main(java.lang.String[] args)
    {
        Connection c = null;

        try {
            // Get the driver registered.
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");

            c = DriverManager.getConnection("jdbc:db2:*local");
            DatabaseMetaData dmd = c.getMetaData();

            if (jdbcLevel == 1) {
                System.out.println("No support is provided for getUDTs.  Just return.");
                System.exit(1);
            }

            ResultSet rs = dmd.getUDTs(null, "CUJOSQL", "SSN%", null);
            while (rs.next()) {

                // Fetch all the columns that have been available since the
                // JDBC 2.0 release.
                System.out.println("TYPE_CAT is " + rs.getString("TYPE_CAT"));
                System.out.println("TYPE_SCHEM is " + rs.getString("TYPE_SCHEM"));
                System.out.println("TYPE_NAME is " + rs.getString("TYPE_NAME"));
                System.out.println("CLASS_NAME is " + rs.getString("CLASS_NAME"));
                System.out.println("DATA_TYPE is " + rs.getString("DATA_TYPE"));
                System.out.println("REMARKS is " + rs.getString("REMARKS"));

                // Fetch all the columns that were added in JDBC 3.0.
                if (jdbcLevel > 2) {
                    System.out.println("BASE_TYPE is " + rs.getString("BASE_TYPE"));
                }
            }
        } catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
        } finally {
            if (c != null) {
                try {
                    c.close();
                } catch (SQLException e) {
                    // Ignoring shutdown exception.
                }
            }
        }
    }
}
```

Related reference

Example: Returning a list of tables using the DatabaseMetaData interface
This example shows how to return a list of tables.

Java exceptions

The Java language uses exceptions to provide error-handling capabilities for its programs. An exception is an event that occurs when you run your program that disrupts the normal flow of instructions.

The Java runtime system and many classes from Java packages throw exceptions in some circumstances by using the throw statement. You can use the same mechanism to throw exceptions in your Java programs.

Java SQLException class

The SQLException class and its subtypes provide information about errors and warnings that occur while a data source is being accessed.

Unlike most of JDBC, which is defined by interfaces, the exception support is provided in classes. The base class for exceptions that occur while running JDBC applications is SQLException. Every method of the JDBC API is declared as being able to throw SQLExceptions. SQLException is an extension of java.lang.Exception and provides additional information related to failures that happen in a database context. Specifically, the following information is available from an SQLException:

- Text description

- SQLState
- Error code
- A reference to any other exceptions that also occurred

ExceptionExample is a program that properly handles catching an (expected in this case) SQLException and dumping all the information that it provides.

As noted, SQLException objects are thrown when errors occur. This is correct, but is not the complete picture. In practice, the native JDBC driver rarely throws actual SQLExceptions. It throws instances of its own SQLException subclasses. This allows you to determine more information about what has actually failed as is shown below.

DB2Exception.java

DB2Exception objects are not thrown directly either. This base class is used to hold functionality that is common to all JDBC exceptions. There are two subclasses of this class that are be the standard exceptions that JDBC throws. These subclasses are DB2DBException.java and DB2JDBCException.java. DB2DBExceptions are exceptions that are reported to you that have come directly from the database. DB2JDBCExceptions are thrown when the JDBC driver finds problems on its own. Splitting the exception class hierarchy in this manner allows you to handle the two types of exceptions differently.

DB2DBException.java

As stated, DB2DBExceptions are exceptions that come directly from the database. These are encountered when the JDBC driver make a call to the CLI and gets back an SQLERROR return code. The CLI function SQLError is called to get the message text, SQLState, and vendor code in these cases. The replacement text for the SQLMessage is also retrieved and returned to you. The DatabaseException class causes an error that the database recognizes and reports to the JDBC driver to build the exception object for.

DB2JDBCException.java

DB2JDBCExceptions are generated for error conditions that come from the JDBC driver itself. The functionality of this exception class is fundamentally different; the JDBC driver itself handles message language translation of exception and other issues that the operating system and database handle for exceptions originating within the database. Wherever possible, the JDBC driver adheres to the SQLStates of the database. The vendor code for exceptions that the JDBC driver throws is always -99999. DB2DBExceptions that are recognized and returned by the CLI layer often also have the -99999 error code. The JDBCException class causes an error that the JDBC driver recognizes and builds the exception for itself. When run during development of the release, the following output was created. Notice that the top of the stack contains DB2JDBCException. This is an indication that the error is being reported from the JDBC driver prior to ever making the request to the database.

Related concepts

SQLWarning
Methods in some interfaces generate an SQLWarning object if the methods cause a database access warning.

DataTruncation and silent truncation
DataTruncation is a subclass of SQLWarning. While SQLWarnings are not thrown, DataTruncation objects are sometimes thrown and attached like other SQLWarning objects. Silent truncation occurs when the

size of a column exceeds the size specified by the setMaxFieldSize statement method, but no warning or exception is reported.

Example: SQLException

This is an example of catching an SQLException and dumping all the information that it provides.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
import java.sql.*;

public class ExceptionExample {

    public static Connection connection = null;

    public static void main(java.lang.String[] args) {

        try {
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
            connection = DriverManager.getConnection("jdbc:db2:*local");

            Statement s = connection.createStatement();
            int count = s.executeUpdate("insert into cujofake.cujofake values(1, 2,3)");

            System.out.println("Did not expect that table to exist.");


        } catch (SQLException e) {
            System.out.println("SQLException exception: ");
            System.out.println("Message:....." + e.getMessage());
            System.out.println("SQLState:...." + e.getSQLState());
            System.out.println("Vendor Code:." + e.getErrorCode());
            System.out.println("----------------------------------------------------");
            e.printStackTrace();
        } catch (Exception ex) {
            System.out.println("An exception other than an SQLException was thrown: ");
            ex.printStackTrace();
        } finally {
            try {
                if (connection != null) {
                    connection.close();
                }
            } catch (SQLException e) {
                System.out.println("Exception caught attempting to shutdown...");
            }
        }
    }
}
```

SQLWarning

Methods in some interfaces generate an SQLWarning object if the methods cause a database access warning.

Methods in the following interfaces can generate an SQLWarning:

- Connection
- Statement and its subtypes, PreparedStatement and CallableStatement
- ResultSet

When a method generates an SQLWarning object, the caller is not informed that a data access warning has occurred. The getWarnings method must be called on the appropriate object to retrieve the SQLWarning object. However, the DataTruncation subclass of SQLWarning may be thrown in some circumstances. It should be noted that the native JDBC driver opts to ignore some database-generated warnings for increased efficiency. For example, a warning is generated by the system when you attempt to retrieve data beyond the end of a ResultSet through the ResultSet.next method. In this case, the next method is defined to return false instead of true, informing you of the error. It is unnecessary to create an object to restate this, so the warning is simply ignored.

If multiple data access warnings occur, they are chained to the first one and can be retrieved by calling the SQLWarning.getNextWarning method. If there are no more warnings in the chain, getNextWarning returns `null`.

Subsequent SQLWarning objects continue to be added to the chain until the next statement is processed or, in the case of a ResultSet object, when the cursor is repositioned. As a result, all SQLWarning objects in the chain are removed.

Using Connection, Statement, and ResultSet objects can cause SQLWarnings to be generated. SQLWarnings are informational messages indicating that while a particular operation has completed successfully, there might be other information of which you should be aware. SQLWarnings are an extension of the SQLException class, but they are not thrown. They are instead attached to the object that causes their generation. When an SQLWarning is generated, nothing happens to inform the application that the warning has been generated. Your application must actively request warning information.

Like SQLExceptions, SQLWarnings can be chained to one another. You can call the clearWarnings method on a Connection, Statement, or ResultSet object to clear the warnings for that object.

Note: Calling the clearWarnings method does not clear all warnings. It only clears the warnings that are associated with a particular object.

The JDBC driver clears SQLWarning objects at specific times if you do not clear them manually. SQLWarning objects are cleared when the following actions are taken:

- For the Connection interface, warnings are cleared on the creation of a new Statement, PreparedStatement, or CallableStatement object.
- For the Statement interface, warnings are cleared when the next statement is processed (or when the statement is processed again for PreparedStatements and CallableStatements).
- For the ResultSet interface, warnings are cleared when the cursor is repositioned.

Related concepts

Java SQLException class
The SQLException class and its subtypes provide information about errors and warnings that occur while a data source is being accessed.

DataTruncation and silent truncation
DataTruncation is a subclass of SQLWarning. While SQLWarnings are not thrown, DataTruncation objects are sometimes thrown and attached like other SQLWarning objects. Silent truncation occurs when the size of a column exceeds the size specified by the setMaxFieldSize statement method, but no warning or exception is reported.

DataTruncation and silent truncation

DataTruncation is a subclass of SQLWarning. While SQLWarnings are not thrown, DataTruncation objects are sometimes thrown and attached like other SQLWarning objects. Silent truncation occurs when the size of a column exceeds the size specified by the setMaxFieldSize statement method, but no warning or exception is reported.

DataTruncation objects provide additional information beyond what is returned by an SQLWarning. The available information includes the following:

- The number of bytes of data that have been transferred.
- The column or parameter index that was truncated.
- Whether the index is for a parameter or a ResultSet column.
- Whether the truncation happened when reading from the database or writing to it.
- The amount of data that was actually transferred.

In some instances, the information can be deciphered, but situations arise that are not completely intuitive. For example, if the PreparedStatement's setFloat method is used to insert a value into a column that holds integer values, a DataTruncation may result because the float may be larger than the largest value that the column can hold. In these situations, the byte counts for truncation do not make sense, but it is important for the driver to provide the truncation information.

Report set() and update() methods

There is a subtle difference between JDBC drivers. Some drivers such as the native and IBM Toolbox for Java JDBC drivers catch and report data truncation issues at the time of the parameter setting. This is done either on the PreparedStatement set method or the ResultSet update method. Other drivers report the problem at the time of processing the statement and is accomplished by the execute, executeQuery, or updateRow methods.

Reporting the problem at the time that you provide incorrect data, instead of at the time that processing cannot continue any further, offers these advantages:

- The failure can be addressed in your application when you have a problem instead of addressing the problem at processing time.
- By checking when setting the parameters, the JDBC driver can ensure that the values that are handed to the database at statement processing time are valid. This allows the database to optimize its work and processing can be completed faster.

ResultSet.update() methods throwing DataTruncation exceptions

In some past releases, ResultSet.update() methods posted warnings when truncation conditions existed. This case occurs when the data value is going to be inserted into the database. The specification dictates that JDBC drivers throw exceptions in these cases. As a result, the JDBC driver works in this manner.

There are no significant difference between handling a ResultSet update function that receives a data truncation error and handling a prepared statement parameter set for an update or insert statement that receives an error. In both cases, the problem is identical; you provided data that does not fit where you wanted it.

NUMERIC and DECIMAL truncate to the right side of a decimal point silently. This is how both JDBC for UDB NT works and how interactive SQL on the IBM i platform works.

Note: No value is rounded when a data truncation occurs. Any fractional portion of a parameter that does not fit in a NUMERIC or DECIMAL column is simply lost without warning.

The following are examples, assuming that the value in the values clause is actually a parameter being set on a prepared statement:

```
create table cujosql.test (col1 numeric(4,2))
a) insert into cujosql.test values(22.22)  // works - inserts 22.22
b) insert into cujosql.test values(22.223) // works - inserts 22.22
c) insert into cujosql.test values(22.227) // works - inserts 22.22
d) insert into cujosql.test values(322.22) // fails - Conversion error on assignment to column
COL1.
```

Difference between a data truncation warning and a data truncation exception

The specification states that data truncation on a value to be written to the database throws an exception. If data truncation is not performed on the value being written to the database, a warning is generated. This means that the point at which a data truncation situation is identified, you must also be aware of the statement type that the data truncation is processing. Given this as a requirement, the following lists the behavior of several SQL statement types:

- In a SELECT statement, query parameters never damage database content. Therefore, data truncation situations are always handled by posting warnings.
- In VALUES INTO and SET statements, the input values are only used to generate output values. As a result, warnings are issued.
- In a CALL statement, the JDBC driver cannot determine what a stored procedure does with a parameter. Exceptions are always thrown when a stored procedure parameter truncates.
- All other statement types throw exceptions rather than post warnings.

Data truncation property for Connection and DataSource

There has been a data truncation property available for many releases. The default for that property is true, meaning that data truncation issues are checked and warnings are posted or exceptions are thrown. The property is provided for convenience and performance in cases where you are not concerned that a value does not fit into the database column. You want the driver to put as much of the value as it can into the column.

Data truncation property only affects character and binary-based data types

A couple releases ago, the data truncation property determined whether data truncation exceptions could be thrown. The data truncation property was put in place to have JDBC applications not worry about a value getting truncated when the truncation was not important to them. There are few cases where you would want either the value 00 or 10 stored in the database when applications attempted to insert 100 into a DECIMAL(2,0). Therefore, the JDBC driver's data truncation property was changed to only honor situations where the parameter is for character-based types such as CHAR, VARCHAR, CHAR FOR BIT DATA, and VARCHAR FOR BIT DATA.

Data truncation property is only applied to parameters

The data truncation property is a setting of the JDBC driver and not of the database. As a result, it has no effect on statement literals. For example, the following statements that are processed to insert a value into a CHAR(8) column in the database still fail with the data truncation flag set to false (assume that connection is a java.sql.Connection object allocated elsewhere).

```
Statement stmt = connection.createStatement();
Stmt.executeUpdate("create table cujosql.test (col1 char(8))");
Stmt.executeUpdate("insert into cujosql.test values('dettinger')");
// Fails as the value does not fit into database column.
```

Native JDBC driver throws exceptions for insignificant data truncation

The native JDBC driver does not look at the data that you provide for parameters. Doing so only slows down processing. However, there can be situations where it does not matter to you that a value truncates, but you have not set the data truncation connection property to false.

For example, 'dettinger ', a char(10) that is passed, throws an exception even though everything important about the value fits. This does happen to be how JDBC for UDB NT works; however, it is not the behavior you would get if you passed the value as a literal in an SQL statement. In this case, the database engine would throw out the additional spaces quietly.

The problems with the JDBC driver not throwing an exception are the following:

- Performance overhead is extensive on every set method, whether needed or not. For the majority of cases where there would be no benefit, there is considerable performance overhead on a function as common as setString().
- Your workaround is trivial, for example, calling the trim function on the string value passed in.
- There are issues with the database column to take into account. A space in CCSID 37 is not at all a space in CCSID 65535, or 13488.

Silent truncation

The setMaxFieldSize statement method allows a maximum field size to be specified for any column. If data truncates because its size has exceeded the maximum field size value, no warning or exception is reported. This method, like the data truncation property previously mentioned, only affects character-based types such as CHAR, VARCHAR, CHAR FOR BIT DATA, and VARCHAR FOR BIT DATA.

Related concepts

Java SQLException class

The SQLException class and its subtypes provide information about errors and warnings that occur while a data source is being accessed.

SQLWarning
Methods in some interfaces generate an SQLWarning object if the methods cause a database access warning.

JDBC transactions

A transaction is a logical unit of work. To complete a logical unit of work, several actions may need to be taken against a database.

Transactional support allows applications to ensure the following:

- All the steps to complete a logical unit of work are followed.
- When one of the steps to the unit of work files fails, all the work done as part of that logical unit of work can be undone and the database can return to its previous state before the transaction began.

Transactions are used to provide data integrity, correct application semantics, and a consistent view of data during concurrent access. All Java Database Connectivity (JDBC) compliant drivers must support transactions.

Note: This section only discusses local transactions and the standard JDBC concept of transactions. Java and the native JDBC driver support the Java Transaction API (JTA), distributed transactions, and the two-phase commit protocol (2PC).

All transactional work is handled at the Connection object level. When the work for a transaction completes, it can be finalized by calling the commit method. If the application aborts the transaction, the rollback method is called.

All Statement objects under a connection are a part of the transaction. This means is that if an application creates three Statement objects and uses each object to make changes to the database, when a commit or rollback call happens, the work for all three statements either becomes permanent or is discarded.

The commit and rollback SQL statements are used to finalize transactions when working purely with SQL. These SQL statements cannot be dynamically prepared and you should not attempt to use them in your JDBC applications to complete transactions.

JDBC auto-commit mode

By default, JDBC uses an operation mode called auto-commit. This means that every update to the database is immediately made permanent.

Any situation where a logical unit of work requires more than one update to the database cannot be done safely in auto-commit mode. If something happens to the application or the system after one update is made and before any other updates are made, the first change cannot be undone when running in auto-commit mode.

Because changes are instantly made permanent in auto-commit mode, there is no need for the application to call the commit method or the rollback method. This makes applications easier to write.

Auto-commit mode can be enabled and disabled dynamically during a connection's existence. Auto-commit is enabled in the following way, assuming that data source already exists:

```
Connection connection = dataSource.getConnection();

Connection.setAutoCommit(false);        // Disables auto-commit.
```

If the auto-commit setting is changed in the middle of a transaction, any pending work is automatically committed. An SQLException is generated if auto-commit is enabled for a connection that is part of a distributed transaction.

Related concepts

Transaction isolation levels

Transaction isolation levels specify what data is visible to statements within a transaction. These levels directly impact the level of concurrent access by defining what interaction is possible between transactions against the same target data source.

Savepoints
Savepoints allow the setting of "staging points" in a transaction. Savepoints are checkpoints that the application can roll back to without throwing away the entire transaction.

Transaction isolation levels

Transaction isolation levels specify what data is visible to statements within a transaction. These levels directly impact the level of concurrent access by defining what interaction is possible between transactions against the same target data source.

Database anomalies

Database anomalies are generated results that seem incorrect when looked at from the scope of a single transaction, but are correct when looked at from the scope of all transactions. The different types of database anomalies are described as follows:

- **Dirty** reads occur when:
  1. Transaction A inserts a row into a table.
  2. Transaction B reads the new row.
  3. Transaction A rolls back.

  Transaction B may have done work to the system based on the row inserted by transaction A, but that row never became a permanent part of the database.
- **Nonrepeatable** reads occur when:
  1. Transaction A reads a row.
  2. Transaction B changes the row.
  3. Transaction A reads the same row a second time and gets the new results.
- **Phantom** reads occur when:
  1. Transaction A reads all rows that satisfy a WHERE clause on an SQL query.
  2. Transaction B inserts an additional row that satisfies the WHERE clause.
  3. Transaction A re-evaluates the WHERE condition and picks up the additional row.

Note: Db2 for i does not always expose the application to the allowable database anomalies at the prescribed levels due to its locking strategies.

JDBC transaction isolation levels

There are five levels of transaction isolation in the IBM Developer Kit for Java JDBC API. Listed from least to most restrictive, they are as follows:

JDBC_TRANSACTION_NONE
This is a special constant indicating that the JDBC driver does not support transactions.

JDBC_TRANSACTION_READ_UNCOMMITTED
This level allows transactions to see uncommitted changes to the data. All database anomalies are possible at this level.

JDBC_TRANSACTION_READ_COMMITTED
This level means that any changes made inside a transaction are not visible outside it until the transaction is committed. This prevents dirty reads from being possible.

JDBC_TRANSACTION_REPEATABLE_READ
This level means that rows that are read retain locks so that another transaction cannot change them when the transaction is not completed. This disallows dirty reads and nonrepeatable reads. Phantom read are still possible.

JDBC_TRANSACTION_SERIALIZABLE
Tables are locked for the transaction so that WHERE conditions cannot be changed by other transactions that add values to or remove values from a table. This prevents all types of database anomalies.

The setTransactionIsolation method can be used to change the transaction isolation level for a connection.

Considerations

A common misinterpretation is that the JDBC specification defines the five transactional levels previously mentioned. It is commonly thought that the TRANSACTION_NONE value represents the concept of running without commitment control. The JDBC specification does not define TRANSACTION_NONE in the same manner. TRANSACTION_NONE is defined in the JDBC specification as a level where the driver does not support transactions and is not a JDBC-compliant driver. The NONE level is never reported when the getTransactionIsolation method is called.

The issue is marginally complicated by the fact that a JDBC driver's default transaction isolation level is defined by the implementation. The default level of transaction isolation for the native JDBC driver default transaction isolation level is NONE. This allows the driver to work with files that do not have journals and you are not required to make any specifications such as files in the QGPL library.

The native JDBC driver allows you to pass JDBC_TRANSACTION_NONE to the setTransactionIsolation method or specify none as a connection property. However, the getTransactionIsolation method always reports JDBC_TRANSACTION_READ_UNCOMMITTED when the value is none. It is your application's responsibility to keep track of what level you are running if it is a requirement in your application.

In past releases, the JDBC driver would handle your specifying true for auto-commit by changing the transaction isolation level to none because the system did not have a concept of a true auto-commit mode. This was a close approximation of the functionality, but did not provide the correct results for all scenarios. This is not done anymore; the database decouples the concept of auto-commit from the concept of a transaction isolation level. Therefore, it is completely valid to run at the JDBC_TRANSACTION_SERIALIZABLE level with auto-commit being enabled. The only scenario that is not valid is to run at the JDBC_TRANSACTION_NONE level and not be in auto-commit mode. Your application cannot take control over commit boundaries when the system is not running with a transaction isolation level.

Transaction isolation levels between the JDBC specification and the IBM i platform

The IBM i platform has common names for its transaction isolation levels that do not match those names provided by the JDBC specification. The following table matches the names used by the IBM i platform, but are not equivalent to those used by the JDBC specification:

JDBC level*	IBM i level
JDBC_TRANSACTION_NONE	*NONE or *NC
JDBC_TRANSACTION_READ_UNCOMMITTED	*CHG or *UR
JDBC_TRANSACTION_READ_COMMITTED	*CS
JDBC_TRANSACTION_REPEATABLE_READ	*ALL or *RS
JDBC_TRANSACTION_SERIALIZABLE	*RR

* In this table, the JDBC_TRANSACTION_NONE value is lined up with the IBM i levels *NONE and *NC for clarity. This is not a direct specification-to-IBM i level match.

Related concepts
JDBC auto-commit mode

By default, JDBC uses an operation mode called auto-commit. This means that every update to the database is immediately made permanent.

Savepoints
Savepoints allow the setting of "staging points" in a transaction. Savepoints are checkpoints that the application can roll back to without throwing away the entire transaction.

Savepoints

Savepoints allow the setting of "staging points" in a transaction. Savepoints are checkpoints that the application can roll back to without throwing away the entire transaction.

Savepoints are new in JDBC 3.0, meaning that the application must run on Java Development Kit (JDK) 1.4 or a subsequent release to use them. Moreover, savepoints are new to the Developer Kit for Java, meaning that savepoints are not supported if JDK 1.4 or a subsequent release is not used with previous releases of the Developer Kit for Java.

Note: The system provides SQL statements for working with savepoints. It is advised that JDBC applications do not use these statements directly in an application. Doing so may work, but the JDBC driver loses its ability to track the your savepoints when this is done. At a minimum, mixing the two models (that is, using your own savepoint SQL statements and using the JDBC API) should be avoided.

Setting and rolling back to savepoints

Savepoints can be set throughout the work of a transaction. The application can then roll back to any of these savepoints if something goes wrong and continue processing from that point. In the following example, the application inserts the value FIRST into a database table. After that, a savepoint is set and another value, SECOND, is inserted into the database. A rollback to the savepoint is issued and undoes the work of inserting SECOND, but leaves FIRST as part of the pending transaction. Finally, the value THIRD is inserted and the transaction is committed. The database table contains the values FIRST and THIRD.

Example: Set and roll back to savepoints

```
Statement s = Connection.createStatement();
s.executeUpdate("insert into table1 values ('FIRST')");
Savepoint pt1 = connection.setSavepoint("FIRST SAVEPOINT");
s.executeUpdate("insert into table1 values ('SECOND')";);
connection.rollback(pt1);        // Undoes most recent insert.
s.executeUpdate("insert into table1 values ('THIRD')");
connection.commit();
```

Although it is unlikely to cause problems to set savepoints while in auto-commit mode, they cannot be rolled back as their lives end at the end of a transaction.

Releasing a savepoint

Savepoints can be released by the application with the releaseSavepoint method on the Connection object. Once a savepoint has been released, attempting to roll back to it results in an exception. When a transaction commits or rolls back, all savepoints automatically release. When a savepoint is rolled back, other savepoints that follow it are also released.

Related concepts

JDBC auto-commit mode
By default, JDBC uses an operation mode called auto-commit. This means that every update to the database is immediately made permanent.

Transaction isolation levels

Transaction isolation levels specify what data is visible to statements within a transaction. These levels directly impact the level of concurrent access by defining what interaction is possible between transactions against the same target data source.

JDBC distributed transactions

Typically, transactions in Java Database Connectivity (JDBC) are local. This means that a single connection performs all the work of the transaction and that the connection can only work on one transaction at a time.

When all the work for that transaction has been completed or has failed, commit or rollback is called to make the work permanent, and a new transaction can begin. There is, however, also advanced support for transactions available in Java that provides functionality beyond local transactions. This support is fully specified by the Java Transaction API.

The Java Transaction API (JTA) has support for complex transactions. It also provides support for decoupling transactions from Connection objects. As JDBC is modeled after the Object Database Connectivity (ODBC) and the X/Open Call Level Interface (CLI) specifications, JTA is modeled after the X/Open Extended Architecture (XA) specification. JTA and JDBC work together to decouple transactions from Connection objects. By decoupling transactions from Connection objects, this allows you to have a single connection work on multiple transactions concurrently. Conversely, it allows you to have multiple Connections work on a single transaction.

Note: If you are planning to work with JTA, refer to the Get started with JDBC topic for more information about required Java Archive (JAR) files in your extensions classpath. You want both the JDBC 2.0 optional package and the JTA JAR files (these files are found automatically by the JDK if you are running JDK 1.4 or a subsequent version). These are not found by default.

Transactions with JTA

When JTA and JDBC are used together, there are a series of steps between them to accomplish transactional work. Support for XA is provided through the XADataSource class. This class contains support for setting up connection pooling exactly the same way as its ConnectionPoolDataSource superclass.

With an XADataSource instance, you can retrieve an XAConnection object. The XAConnection object serves as a container for both the JDBC Connection object and an XAResource object. The XAResource object is designed to handle XA transactional support. XAResource handles transactions through objects called transaction IDs (XIDs).

The XID is an interface that you must implement. It represents a Java mapping of the XID structure of the X/Open transaction identifier. This object contains three parts:

- A global transaction's format ID
- A global transaction ID
- A branch qualifier

See the JTA specification for complete details on this interface.

Use UDBXADataSource support for pooling and distributed transactions

The Java Transaction API support provides direct support for connection pooling. UDBXADataSource is an extension of a ConnectionPoolDataSource, allowing application access to pooled XAConnection objects. Since UDBXADataSource is a ConnectionPoolDataSource, the configuration and use of the UDBXADataSource is the same as that described in the Use DataSource support for object pooling topic.

XADataSource properties

In addition to the properties provided by the ConnectionPoolDataSource, the XADataSource interface provides the following properties:

Set method (data type)	Values	Description
setLockTimeout (int)	0 or any positive value	Any positive value is a valid lock timeout (in seconds) at the transaction level. A lock timeout of 0 means that there is no lock timeout value enforced at the transaction level, although there may be one enforced at other levels (the job or the table). The default value is 0.
setTransactionTimeout (int)	0 or any positive value	Any positive value is a valid transaction timeout (in seconds). A transaction timeout of 0 means that there is no transaction timeout value enforced. If the transaction is active for longer than the timeout value, it is marked rollback only, and subsequent attempts to perform work under it causes an exception to occur. The default value is 0.

ResultSets and transactions

Besides demarcating the start and end of a transaction as shown in the previous example, transactions can be suspended for a time and resumed later. This provides a number of scenarios for ResultSet resources that are created during a transaction.

Simple transaction end

When you end a transaction, all open ResultSets that were created under that transaction automatically close. It is recommended that you explicitly close your ResultSets when you are finished using them to ensure maximum parallel processing. However, an exception results if any ResultSets that were opened during a transaction are accessed after the XAResource.end call is made.

Suspend and resume

While a transaction is suspended, access to a ResultSet created while the transaction was active is not allowed and results in an exception. However, once the transaction is resumed, the ResultSet is available again and remains in the same state it was in before the transaction was suspended.

Effecting suspended ResultSets

While a transaction is suspended, the ResultSet cannot be accessed. However, Statement objects can be reprocessed under another transaction to perform work. Because JDBC Statement objects can have only one ResultSet at a time (excluding the JDBC 3.0 support for multiple concurrent ResultSets from a stored procedure call), the ResultSet for the suspended transaction must be closed to fulfill the request of the new transaction. This is exactly what happens.

Note: Although JDBC 3.0 allows a Statement to have multiple ResultSets open simultaneously for a stored procedure call, they are treated as a single unit and all of them close if the Statement is reprocessed under a new transaction. It is not possible to have ResultSets from two transactions active simultaneously for a single statement.

Multiplexing

The JTA API is designed to decouple transactions from JDBC connections. This API allows you to have either multiple connections work on a single transaction or a single connection work on multiple transactions concurrently. This is called **multiplexing** and many complex tasks can be performed that cannot be accomplished with JDBC alone.

For further information on using JTA, see the JTA specification. The JDBC 3.0 specification also contains information on how these two technologies work together to support distributed transactions.

Two-phase commit and transaction logging

The JTA APIs externalize the responsibilities of the distributed two-phase commit protocol completely to the application. As the examples have shown, when using JTA and JDBC to access a database under a JTA transaction, the application uses the XAResource.prepare() and XAResource.commit() methods or just the XAResource.commit() method to commit the changes.

In addition, when accessing multiple distinct databases using a single transaction, it is the application's responsibility to ensure that the two-phase commit protocol and any associated logging required for transaction atomicity across those databases are performed. Typically, the two-phase commit processing across multiple databases (that is, XAResources) and its logging are performed under the control of an application server or transaction monitor so that the application itself does not actually concern itself with these issues.

For example, the application may call some commit() method or return from its processing with no errors. The underlying application server or transaction monitor would then begin processing for each database (XAResource) that participated in the single distributed transaction.

The application server would use extensive logging during the two-phase commit processing. It would call the XAResource.prepare() method in turn for each participant database (XAResource), followed by a call to the XAResource.commit() method for each participant database (XAResource).

If a failure occurs during this processing, the application server's transaction monitor logs allow the application server itself to subsequently use the JTA APIs to recover the distributed transaction. This recovery, under the control of the application server or transaction monitor, allows the application server to get the transaction to a known state at each participant database (XAResource). This ensures a well-known state of the entire distributed transaction across all participant databases.

Related concepts

"Getting started with JDBC" on page 28
The Java Database Connectivity (JDBC) driver shipped with Java on IBM i is called the IBM Developer Kit for Java JDBC driver. This driver is also commonly known as the native JDBC driver.

"Using DataSource support for object pooling" on page 117
You can use DataSources to have multiple applications share a common configuration for accessing a database. This is accomplished by having each application reference the same DataSource name.

Related reference

"ConnectionPoolDataSource properties" on page 119
You can configure the ConnectionPoolDataSource interface by using the set of properties that it provides.

Related information

Java Transaction API 1.0.1 Specification

Example: Using JTA to handle a transaction

This is an example of how to use the Java Transaction API (JTA) to handle a transaction in an application.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
import java.sql.*;
import javax.sql.*;
import java.util.*;
import javax.transaction.*;
```

```
import javax.transaction.xa.*;
import com.ibm.db2.jdbc.app.*;

public class JTACommit {

    public static void main(java.lang.String[] args) {
        JTACommit test = new JTACommit();

        test.setup();
        test.run();
    }


    /**
     * Handle the previous cleanup run so that this test can recommence.
     */
    public void setup() {

        Connection c = null;
        Statement s = null;
        try {
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
            c = DriverManager.getConnection("jdbc:db2:*local");
            s = c.createStatement();

            try {
                s.executeUpdate("DROP TABLE CUJOSQL.JTATABLE");
            } catch (SQLException e) {
                // Ignore... does not exist
            }

            s.executeUpdate("CREATE TABLE CUJOSQL.JTATABLE (COL1 CHAR (50))");
            s.close();
        } finally {
            if (c != null) {
                c.close();
            }
        }
    }


    /**
     * This test uses JTA support to handle transactions.
     */
    public void run() {
        Connection c = null;

        try {
            Context ctx = new InitialContext();

            // Assume the data source is backed by a UDBXADataSource.
            UDBXADataSource ds = (UDBXADataSource) ctx.lookup("XADataSource");

            // From the DataSource, obtain an XAConnection object that
            // contains an XAResource and a Connection object.
            XAConnection  xaConn = ds.getXAConnection();
            XAResource    xaRes  = xaConn.getXAResource();
            Connection    c      = xaConn.getConnection();

            // For XA transactions, a transaction identifier is required.
            // An implementation of the XID interface is not included with the
            // JDBC driver. See Transactions with JTA for a description of
            // this interface to build a class for it.
            Xid xid = new XidImpl();

            // The connection from the XAResource can be used as any other
            // JDBC connection.
            Statement stmt = c.createStatement();

            // The XA resource must be notified before starting any
            // transactional work.
            xaRes.start(xid, XAResource.TMNOFLAGS);

            // Standard JDBC work is performed.
            int count =
              stmt.executeUpdate("INSERT INTO CUJOSQL.JTATABLE VALUES('JTA is pretty fun.')");

            // When the transaction work has completed, the XA resource must
            // again be notified.
            xaRes.end(xid, XAResource.TMSUCCESS);

            // The transaction represented by the transaction ID is prepared
```

```
            // to be committed.
            int rc = xaRes.prepare(xid);

            // The transaction is committed through the XAResource.
            // The JDBC Connection object is not used to commit
            // the transaction when using JTA.
            xaRes.commit(xid, false);


        } catch (Exception e) {
            System.out.println("Something has gone wrong.");
            e.printStackTrace();
        } finally {
            try {
                if (c != null)
                    c.close();
            } catch (SQLException e) {
                System.out.println("Note:  Cleaup exception.");
                e.printStackTrace();
            }
        }
    }
}
```

Related reference

Example: Multiple connections that work on a transaction
This is an example of how to use multiple connections working on a single transaction.

Example: Using a connection with multiple transactions
This is an example of how to use a single connection with multiple transactions.

Example: Suspended ResultSets
This is an example of the how a Statement object is reprocessed under another transaction to perform work.

Example: Ending a transaction
This is an example of ending a transaction in your application.

Example: Suspending and resuming a transaction
This is an example of a transaction that is suspended and then is resumed.

Example: Multiple connections that work on a transaction

This is an example of how to use multiple connections working on a single transaction.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
import java.sql.*;
import javax.sql.*;
import java.util.*;
import javax.transaction.*;
import javax.transaction.xa.*;
import com.ibm.db2.jdbc.app.*;
public class JTAMultiConn {
   public static void main(java.lang.String[] args) {
      JTAMultiConn test = new JTAMultiConn();
      test.setup();
      test.run();
   }
/**
* Handle the previous cleanup run so that this test can recommence.
*/
   public void setup() {
      Connection c = null;
      Statement s = null;
      try {
         Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
         c = DriverManager.getConnection("jdbc:db2:*local");
         s = c.createStatement();
         try {
            s.executeUpdate("DROP TABLE CUJOSQL.JTATABLE");
         }
         catch (SQLException e) {
         // Ignore... does not exist
         }
```

```
         s.executeUpdate("CREATE TABLE CUJOSQL.JTATABLE (COL1 CHAR
                        (50))");
               s.close();
      }
      finally {
         if (c != null) {
            c.close();
         }
      }
   }
/**
* This test uses JTA support to handle transactions.
*/
   public void run() {
      Connection c1 = null;
      Connection c2 = null;
      Connection c3 = null;
      try {
         Context ctx = new InitialContext();
         // Assume the data source is backed by a UDBXADataSource.
         UDBXADataSource ds = (UDBXADataSource)
                                 ctx.lookup("XADataSource");
         // From the DataSource, obtain some XAConnection objects that
         // contain an XAResource and a Connection object.
         XAConnection xaConn1 = ds.getXAConnection();
         XAConnection xaConn2 = ds.getXAConnection();
         XAConnection xaConn3 = ds.getXAConnection();
         XAResource xaRes1 = xaConn1.getXAResource();
         XAResource xaRes2 = xaConn2.getXAResource();
         XAResource xaRes3 = xaConn3.getXAResource();
         c1 = xaConn1.getConnection();
         c2 = xaConn2.getConnection();
         c3 = xaConn3.getConnection();
         Statement stmt1 = c1.createStatement();
         Statement stmt2 = c2.createStatement();
         Statement stmt3 = c3.createStatement();
         // For XA transactions, a transaction identifier is required.
         // Support for creating XIDs is again left to the application
         // program.
         Xid xid = JDXATest.xidFactory();
         // Perform some transactional work under each of the three
         // connections that have been created.
         xaRes1.start(xid, XAResource.TMNOFLAGS);
         int count1 = stmt1.executeUpdate("INSERT INTO " + tableName + "VALUES('Value 1-A')");
         xaRes1.end(xid, XAResource.TMNOFLAGS);

         xaRes2.start(xid, XAResource.TMJOIN);
         int count2 = stmt2.executeUpdate("INSERT INTO " + tableName + "VALUES('Value 1-B')");
         xaRes2.end(xid, XAResource.TMNOFLAGS);

         xaRes3.start(xid, XAResource.TMJOIN);
         int count3 = stmt3.executeUpdate("INSERT INTO " + tableName + "VALUES('Value 1-C')");
         xaRes3.end(xid, XAResource.TMSUCCESS);
         // When completed, commit the transaction as a single unit.
         // A prepare() and commit() or 1 phase commit() is required for
         // each separate database (XAResource) that participated in the
         // transaction. Since the resources accessed (xaRes1, xaRes2, and xaRes3)
         // all refer to the same database, only one prepare or commit is required.
         int rc = xaRes.prepare(xid);
         xaRes.commit(xid, false);
      }
      catch (Exception e) {
         System.out.println("Something has gone wrong.");
         e.printStackTrace();
      }
      finally {
         try {
            if (c1 != null) {
               c1.close();
            }
         }
         catch (SQLException e) {
            System.out.println("Note: Cleaup exception " +
                                  e.getMessage());
         }
         try {
            if (c2 != null) {
               c2.close();
            }
         }
         catch (SQLException e) {
            System.out.println("Note: Cleaup exception " +
```

```
                             e.getMessage());
          }
          try {
             if (c3 != null) {
                c3.close();
             }
          }
          catch (SQLException e) {
             System.out.println("Note: Cleaup exception " +
                                e.getMessage());
          }
       }
    }
}
```

Related reference

Example: Using JTA to handle a transaction
This is an example of how to use the Java Transaction API (JTA) to handle a transaction in an application.

Example: Using a connection with multiple transactions
This is an example of how to use a single connection with multiple transactions.

Example: Suspended ResultSets
This is an example of the how a Statement object is reprocessed under another transaction to perform work.

Example: Ending a transaction
This is an example of ending a transaction in your application.

Example: Suspending and resuming a transaction
This is an example of a transaction that is suspended and then is resumed.

Example: Using a connection with multiple transactions

This is an example of how to use a single connection with multiple transactions.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
import java.sql.*;
import javax.sql.*;
import java.util.*;
import javax.transaction.*;
import javax.transaction.xa.*;
import com.ibm.db2.jdbc.app.*;

public class JTAMultiTx {

    public static void main(java.lang.String[] args) {
        JTAMultiTx test = new JTAMultiTx();

        test.setup();
        test.run();
    }


    /**
     * Handle the previous cleanup run so that this test can recommence.
     */
    public void setup() {

        Connection c = null;
        Statement s = null;
        try {
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
            c = DriverManager.getConnection("jdbc:db2:*local");
            s = c.createStatement();

            try {
                s.executeUpdate("DROP TABLE CUJOSQL.JTATABLE");
            } catch (SQLException e) {
                // Ignore... does not exist
            }

            s.executeUpdate("CREATE TABLE CUJOSQL.JTATABLE (COL1 CHAR (50))");
```

```
            s.close();
        } finally {
            if (c != null) {
                c.close();
            }
        }
    }


    /**
     * This test uses JTA support to handle transactions.
     */
    public void run() {
        Connection c = null;

        try {
            Context ctx = new InitialContext();

            // Assume the data source is backed by a UDBXADataSource.
            UDBXADataSource ds = (UDBXADataSource) ctx.lookup("XADataSource");

            // From the DataSource, obtain an XAConnection object that
            // contains an XAResource and a Connection object.
            XAConnection  xaConn = ds.getXAConnection();
            XAResource    xaRes  = xaConn.getXAResource();
            Connection    c      = xaConn.getConnection();
            Statement     stmt   = c.createStatement();

            // For XA transactions, a transaction identifier is required.
            // This is not meant to imply that all the XIDs are the same.
            // Each XID must be unique to distinguish the various transactions
            // that occur.
            // Support for creating XIDs is again left to the application
            // program.
            Xid xid1 = JDXATest.xidFactory();
            Xid xid2 = JDXATest.xidFactory();
            Xid xid3 = JDXATest.xidFactory();

            // Do work under three transactions for this connection.
            xaRes.start(xid1, XAResource.TMNOFLAGS);
            int count1 = stmt.executeUpdate("INSERT INTO CUJOSQL.JTATABLE VALUES('Value 1-A')");
            xaRes.end(xid1, XAResource.TMNOFLAGS);

            xaRes.start(xid2, XAResource.TMNOFLAGS);
            int count2 = stmt.executeUpdate("INSERT INTO CUJOSQL.JTATABLE VALUES('Value 1-B')");
            xaRes.end(xid2, XAResource.TMNOFLAGS);

            xaRes.start(xid3, XAResource.TMNOFLAGS);
            int count3 = stmt.executeUpdate("INSERT INTO CUJOSQL.JTATABLE VALUES('Value 1-C')");
            xaRes.end(xid3, XAResource.TMNOFLAGS);

            // Prepare all the transactions
            int rc1 = xaRes.prepare(xid1);
            int rc2 = xaRes.prepare(xid2);
            int rc3 = xaRes.prepare(xid3);

            // Two of the transactions commit and one rolls back.
            // The attempt to insert the second value into the table is
            // not committed.
            xaRes.commit(xid1, false);
            xaRes.rollback(xid2);
            xaRes.commit(xid3, false);


        } catch (Exception e) {
            System.out.println("Something has gone wrong.");
            e.printStackTrace();
        } finally {
            try {
                if (c != null)
                    c.close();
            } catch (SQLException e) {
                System.out.println("Note:  Cleaup exception.");
                e.printStackTrace();
            }
        }
    }
}
```

Related reference

Example: Using JTA to handle a transaction
This is an example of how to use the Java Transaction API (JTA) to handle a transaction in an application.

Example: Multiple connections that work on a transaction
This is an example of how to use multiple connections working on a single transaction.

Example: Suspended ResultSets
This is an example of the how a Statement object is reprocessed under another transaction to perform work.

Example: Ending a transaction
This is an example of ending a transaction in your application.

Example: Suspending and resuming a transaction
This is an example of a transaction that is suspended and then is resumed.

Example: Suspended ResultSets

This is an example of the how a Statement object is reprocessed under another transaction to perform work.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
import java.sql.*;
import javax.sql.*;
import java.util.*;
import javax.transaction.*;
import javax.transaction.xa.*;
import com.ibm.db2.jdbc.app.*;

public class JTATxEffect {

    public static void main(java.lang.String[] args) {
        JTATxEffect test = new JTATxEffect();

        test.setup();
        test.run();
    }


    /**
     * Handle the previous cleanup run so that this test can recommence.
     */
    public void setup() {

        Connection c = null;
        Statement s = null;
        try {
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
            c = DriverManager.getConnection("jdbc:db2:*local");
            s = c.createStatement();

            try {
                s.executeUpdate("DROP TABLE CUJOSQL.JTATABLE");
            } catch (SQLException e) {
                // Ignore... does not exist
            }

            s.executeUpdate("CREATE TABLE CUJOSQL.JTATABLE (COL1 CHAR (50))");
            s.executeUpdate("INSERT INTO CUJOSQL.JTATABLE VALUES('Fun with JTA')");
            s.executeUpdate("INSERT INTO CUJOSQL.JTATABLE VALUES('JTA is fun.)");

            s.close();
        } finally {
            if (c != null) {
                c.close();
            }
        }
    }


    /**
     * This test uses JTA support to handle transactions.
     */
```

```
    public void run() {
        Connection c = null;

        try {
            Context ctx = new InitialContext();

            // Assume the data source is backed by a UDBXADataSource.
            UDBXADataSource ds = (UDBXADataSource) ctx.lookup("XADataSource");

            // From the DataSource, obtain an XAConnection object that
            // contains an XAResource and a Connection object.
            XAConnection  xaConn = ds.getXAConnection();
            XAResource    xaRes  = xaConn.getXAResource();
            Connection    c      = xaConn.getConnection();

            // For XA transactions, a transaction identifier is required.
            // An implementation of the XID interface is not included with
            // the JDBC driver. See Transactions with JTA
            // for a description of this interface to build a
            // class for it.
            Xid xid = new XidImpl();

            // The connection from the XAResource can be used as any other
            // JDBC connection.
            Statement stmt = c.createStatement();

            // The XA resource must be notified before starting any
            // transactional work.
            xaRes.start(xid, XAResource.TMNOFLAGS);

            // Create a ResultSet during JDBC processing and fetch a row.
            ResultSet rs = stmt.executeUpdate("SELECT * FROM CUJOSQL.JTATABLE");
            rs.next();

            // The end method is called with the suspend option. The
            // ResultSets associated with the current transaction are 'on hold'.
            // They are neither gone nor accessible in this state.
            xaRes.end(xid, XAResource.TMSUSPEND);


            // In the meantime, other work can be done outside the transaction.
            // The ResultSets under the transaction can be closed if the
            // Statement object used to create them is reused.
            ResultSet nonXARS = stmt.executeQuery("SELECT * FROM CUJOSQL.JTATABLE");
            while (nonXARS.next()) {
                // Process here...
            }


            // Attempt to go back to the suspended transaction. The suspended
            // transaction's ResultSet has disappeared because the statement
            // has been processed again.
            xaRes.start(newXid, XAResource.TMRESUME);
            try {
                rs.next();
            } catch (SQLException ex) {
                System.out.println("This exception is expected. " +
                                   "The ResultSet closed due to another process.");
            }


            // When the transaction had completed, end it
            // and commit any work under it.
            xaRes.end(xid, XAResource.TMNOFLAGS);
            int rc = xaRes.prepare(xid);
            xaRes.commit(xid, false);


        } catch (Exception e) {
            System.out.println("Something has gone wrong.");
            e.printStackTrace();
        } finally {
            try {
                if (c != null)
                    c.close();
            } catch (SQLException e) {
                System.out.println("Note:  Cleaup exception.");
                e.printStackTrace();
            }
        }
```

```
    }
}
```

Related reference

Example: Using JTA to handle a transaction
This is an example of how to use the Java Transaction API (JTA) to handle a transaction in an application.

Example: Multiple connections that work on a transaction
This is an example of how to use multiple connections working on a single transaction.

Example: Using a connection with multiple transactions
This is an example of how to use a single connection with multiple transactions.

Example: Ending a transaction
This is an example of ending a transaction in your application.

Example: Suspending and resuming a transaction
This is an example of a transaction that is suspended and then is resumed.

Example: Ending a transaction

This is an example of ending a transaction in your application.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
import java.sql.*;
import javax.sql.*;
import java.util.*;
import javax.transaction.*;
import javax.transaction.xa.*;
import com.ibm.db2.jdbc.app.*;

public class JTATxEnd {

    public static void main(java.lang.String[] args) {
        JTATxEnd test = new JTATxEnd();

        test.setup();
        test.run();
    }


    /**
     * Handle the previous cleanup run so that this test can recommence.
     */
    public void setup() {

        Connection c = null;
        Statement s = null;
        try {
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
            c = DriverManager.getConnection("jdbc:db2:*local");
            s = c.createStatement();

            try {
                s.executeUpdate("DROP TABLE CUJOSQL.JTATABLE");
            } catch (SQLException e) {
                // Ignore... does not exist
            }

            s.executeUpdate("CREATE TABLE CUJOSQL.JTATABLE (COL1 CHAR (50))");
            s.executeUpdate("INSERT INTO CUJOSQL.JTATABLE VALUES('Fun with JTA')");
            s.executeUpdate("INSERT INTO CUJOSQL.JTATABLE VALUES('JTA is fun.)");

            s.close();
        } finally {
            if (c != null) {
                c.close();
            }
        }
    }


    /**
     * This test use JTA support to handle transactions.
     */
```

```
    public void run() {
        Connection c = null;

        try {
            Context ctx = new InitialContext();

            // Assume the data source is backed by a UDBXADataSource.
            UDBXADataSource ds = (UDBXADataSource) ctx.lookup("XADataSource");

            // From the DataSource, obtain an XAConnection object that
            // contains an XAResource and a Connection object.
            XAConnection  xaConn = ds.getXAConnection();
            XAResource    xaRes  = xaConn.getXAResource();
            Connection    c      = xaConn.getConnection();

            // For XA transactions, transaction identifier is required.
            // An implementation of the XID interface is not included
            // with the JDBC driver. See Transactions with JTA for a
            // description of this interface to build a class for it.
            Xid xid = new XidImpl();

            // The connection from the XAResource can be used as any other
            // JDBC connection.
            Statement stmt = c.createStatement();

            // The XA resource must be notified before starting any
            // transactional work.
            xaRes.start(xid, XAResource.TMNOFLAGS);

            // Create a ResultSet during JDBC processing and fetch a row.
            ResultSet rs = stmt.executeUpdate("SELECT * FROM CUJOSQL.JTATABLE");
            rs.next();

            // When the end method is called, all ResultSet cursors close.
            // Accessing the ResultSet after this point results in an
            // exception being thrown.
            xaRes.end(xid, XAResource.TMNOFLAGS);

            try {
                String value = rs.getString(1);
                System.out.println("Something failed if you receive this message.");
            } catch (SQLException e) {
                System.out.println("The expected exception was thrown.");
            }

            // Commit the transaction to ensure that all locks are
            // released.
            int rc = xaRes.prepare(xid);
            xaRes.commit(xid, false);


        } catch (Exception e) {
            System.out.println("Something has gone wrong.");
            e.printStackTrace();
        } finally {
            try {
                if (c != null)
                    c.close();
            } catch (SQLException e) {
                System.out.println("Note:  Cleaup exception.");
                e.printStackTrace();
            }
        }
    }
}
```

Related reference

Example: Using JTA to handle a transaction
This is an example of how to use the Java Transaction API (JTA) to handle a transaction in an application.

Example: Multiple connections that work on a transaction
This is an example of how to use multiple connections working on a single transaction.

Example: Using a connection with multiple transactions
This is an example of how to use a single connection with multiple transactions.

Example: Suspended ResultSets

This is an example of the how a Statement object is reprocessed under another transaction to perform work.

Example: Suspending and resuming a transaction
This is an example of a transaction that is suspended and then is resumed.

“JDBC distributed transactions” on page 70
Typically, transactions in Java Database Connectivity (JDBC) are local. This means that a single connection performs all the work of the transaction and that the connection can only work on one transaction at a time.

Example: Suspending and resuming a transaction

This is an example of a transaction that is suspended and then is resumed.

Note: By using the code examples, you agree to the terms of the “Code license and disclaimer information” on page 489.

```
import java.sql.*;
import javax.sql.*;
import java.util.*;
import javax.transaction.*;
import javax.transaction.xa.*;
import com.ibm.db2.jdbc.app.*;
import javax.naming.InitialContext;
import javax.naming.Context;

public class JTATxSuspend {

    public static void main(java.lang.String[] args) {
        JTATxSuspend test = new JTATxSuspend();

        test.setup();
        test.run();
    }


    /**
     * Handle the previous cleanup run so that this test can recommence.
     */
    public void setup() {

        Connection c = null;
        Statement s = null;
        try {
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
            c = DriverManager.getConnection("jdbc:db2:*local");
            s = c.createStatement();

            try {
                s.executeUpdate("DROP TABLE CUJOSQL.JTATABLE");
            } catch (SQLException e) {
                // Ignore... doesn't exist
            }

            s.executeUpdate("CREATE TABLE CUJOSQL.JTATABLE (COL1 CHAR (50))");
            s.executeUpdate("INSERT INTO CUJOSQL.JTATABLE VALUES('Fun with JTA')");
            s.executeUpdate("INSERT INTO CUJOSQL.JTATABLE VALUES('JTA is fun.)");

            s.close();
        } finally {
            if (c != null) {
                c.close();
            }
        }
    }


    /**
     * This test uses JTA support to handle transactions.
     */
    public void run() {
        Connection c = null;

        try {
            Context ctx = new InitialContext();

            // Assume the data source is backed by a UDBXADataSource.
```

```
            UDBXADataSource ds = (UDBXADataSource) ctx.lookup("XADataSource");

            // From the DataSource, obtain an XAConnection object that
            // contains an XAResource and a Connection object.
            XAConnection  xaConn = ds.getXAConnection();
            XAResource    xaRes  = xaConn.getXAResource();
            c      = xaConn.getConnection();

            // For XA transactions, a transaction identifier is required.
            // An implementation of the XID interface is not included with
            // the JDBC driver. See topic "Transactions with JTA" for a
            // description of this interface to build a class for it.
            Xid xid = new XidImpl();

            // The connection from the XAResource can be used as any other
            // JDBC connection.
            Statement stmt = c.createStatement();

            // The XA resource must be notified before starting any
            // transactional work.
            xaRes.start(xid, XAResource.TMNOFLAGS);

            // Create a ResultSet during JDBC processing and fetch a row.
            ResultSet rs = stmt.executeQuery("SELECT * FROM CUJOSQL.JTATABLE");
            rs.next();

            // The end method is called with the suspend option. The
            // ResultSets associated with the current transaction are 'on hold'.
            // They are neither gone nor accessible in this state.
            xaRes.end(xid, XAResource.TMSUSPEND);


            // Other work can be performed with the transaction.
            // As an example, you can create a statement and process a query.
            // This work and any other transactional work that the transaction may
            // perform is separate from the work done previously under the XID.
            Statement nonXAStmt = c.createStatement();
            ResultSet nonXARS = nonXAStmt.executeQuery("SELECT * FROM CUJOSQL.JTATABLE");
            while (nonXARS.next()) {
                // Process here...
            }
            nonXARS.close();
            nonXAStmt.close();


            // If an attempt is made to use any suspended transactions
            // resources, an exception results.
            try {
                rs.getString(1);
                System.out.println("Value of the first row is " + rs.getString(1));
            } catch (SQLException e) {
                System.out.println("This was an expected exception - " +
                                   "suspended ResultSet was used.");
            }


            // Resume the suspended transaction and complete the work on it.
            // The ResultSet is exactly as it was before the suspension.
            xaRes.start(newXid, XAResource.TMRESUME);
            rs.next();
            System.out.println("Value of the second row is " + rs.getString(1));


            // When the transaction has completed, end it
            // and commit any work under it.
            xaRes.end(xid, XAResource.TMNOFLAGS);
            int rc = xaRes.prepare(xid);
            xaRes.commit(xid, false);


        } catch (Exception e) {
            System.out.println("Something has gone wrong.");
            e.printStackTrace();
        } finally {
            try {
                if (c != null)
                    c.close();
            } catch (SQLException e) {
                System.out.println("Note:  Cleaup exception.");
                e.printStackTrace();
            }
        }
```

```
    }
}
```

Related reference

Example: Using JTA to handle a transaction
This is an example of how to use the Java Transaction API (JTA) to handle a transaction in an application.

Example: Multiple connections that work on a transaction
This is an example of how to use multiple connections working on a single transaction.

Example: Using a connection with multiple transactions
This is an example of how to use a single connection with multiple transactions.

Example: Suspended ResultSets
This is an example of the how a Statement object is reprocessed under another transaction to perform work.

Example: Ending a transaction
This is an example of ending a transaction in your application.

Statement types

The Statement interface and its PreparedStatement and CallableStatement subclasses are used to process structured query language (SQL) commands against the database. SQL statements cause the generation of ResultSet objects.

Subclasses of the Statement interface are created with a number of methods on the Connection interface. A single Connection object can have many Statement objects created under it simultaneously. In past releases, it was possible to give exact numbers of Statement objects that could be created. It is impossible to do so in this release because different types of Statement objects take different numbers of "handles" within the database engine. Therefore, the types of Statement objects you are using influence the number of statements that can be active under a connection at a single time.

An application calls the Statement.close method to indicate that the application has finished processing a statement. All Statement objects are closed when the connection that created them is closed. However, you should not fully rely on this behavior to close Statement objects. For example, if your application changes so that a connection pool is used instead of explicitly closing the connections, the application "leaks" statement handles because the connections never close. Closing Statement objects as soon as they are no longer required allows external database resources that the statement is using to be released immediately.

The native JDBC driver attempts to detect statement leaks and handles them on you behalf. However, relying on that support results in poorer performance.

Due to the inheritance hierarchy that CallableStatement extends PreparedStatement which extends Statement, features of each interface are available in the class that extend the interface. For example, features of the Statement class are also supported in the PreparedStatement and CallableStatement classes. The main exception is the executeQuery, executeUpdate, and execute methods on the Statement class. These methods take in an SQL statement to dynamically process and cause exceptions if you attempt to use them with PreparedStatement or CallableStatement objects.

Statement objects

A Statement object is used for processing a static SQL statement and obtaining the results produced by it. Only one ResultSet for each Statement object can be open at a time. All statement methods that process an SQL statement implicitly close a statement's current ResultSet if an open one exists.

Create statements

Statement objects are created from Connection objects with the createStatement method. For example, assuming a Connection object named conn already exists, the following line of code creates a Statement object for passing SQL statements to the database:

```
Statement stmt = conn.createStatement();
```

Specify ResultSet characteristics

The characteristics of ResultSets are associated with the statement that eventually creates them. The Connection.createStatement method allows you to specify these ResultSet characteristics. The following are some examples of valid calls to the createStatement method:

Example: The createStatement method

```
// The following is new in JDBC 2.0

Statement stmt2 = conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_UPDATEABLE);

// The following is new in JDBC 3.0

Statement stmt3 = conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY, ResultSet.HOLD_CURSOR_OVER_COMMIT);
```

For more information about these characteristics, see ResultSets.

Process statements

Processing SQL statements with a Statement object is accomplished with the executeQuery(), executeUpdate(), and execute() methods.

Return results from SQL queries

If an SQL query statement returning a ResultSet object is to be processed, the executeQuery() method should be used. You can refer to the example program that uses a Statement object's executeQuery method to obtain a ResultSet.

Note: If an SQL statement processed with executeQuery does not return a ResultSet, an SQLException is thrown.

Return update counts for SQL Statements

If the SQL is known to be a Data Definition Language (DDL) statement or a Data Manipulation Language (DML) statement returning an update count, the executeUpdate() method should be used. The StatementExample program uses a Statement object's executeUpdate method.

Process SQL statements where the expected return is unknown

If the SQL statement type is not known, the execute method should be used. Once this method has been processed, the JDBC driver can tell the application what types of results the SQL statement has generated through API calls. The execute method returns true if the result is at least one ResultSet and false if the return value is an update count. Given this information, applications can use the statement method's getUpdateCount or getResultSet to retrieve the return value from processing the SQL statement. The StatementExecute program uses the execute method on a Statement object. This program expects a parameter to be passed that is an SQL statement. Without looking at the text of the SQL that you provide, the program processes the statement and determines information about what was processed.

Note: Calling the getUpdateCount method when the result is a ResultSet returns -1. Calling the getResultSet method when the result is an update count returns null.

The cancel method

The methods of the native JDBC driver are synchronized to prevent two threads running against the same object from corrupting the object. An exception is the cancel method. The cancel method can be used by one thread to stop a long running SQL statement on another thread for the same object. The native JDBC driver cannot force the thread to stop doing work; it can only request that it stop whatever task it was doing. For this reason, it still takes time for a cancelled statement to stop. The cancel method can be used to halt runaway SQL queries on the system.

Related concepts

PreparedStatements
PreparedStatements extend the Statement interface and provide support for adding parameters to SQL statements.

CallableStatements
The JDBC CallableStatement interface extends PreparedStatement and provides support for output and input/output parameters. The CallableStatement interface also has support for input parameters that is provided by the PreparedStatement interface.

Example: Using the Statement object's executeUpdate method
This is an example of how to use the Statement object's executeUpdate method.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
import java.sql.*;
import java.util.Properties;

public class StatementExample {

    public static void main(java.lang.String[] args)
    {

        // Suggestion:  Load these from a properties object.
        String DRIVER = "com.ibm.db2.jdbc.app.DB2Driver";
        String URL    = "jdbc:db2://*local";

        // Register the native JDBC driver. If the driver cannot be
        // registered, the test cannot continue.
        try {
            Class.forName(DRIVER);
        } catch (Exception e) {
            System.out.println("Driver failed to register.");
            System.out.println(e.getMessage());
            System.exit(1);
        }

        Connection c = null;
        Statement s = null;

        try {
            // Create the connection properties.
            Properties properties = new Properties ();
            properties.put ("user", "userid");
            properties.put ("password", "password");

            // Connect to the local database.
            c = DriverManager.getConnection(URL, properties);

            // Create a Statement object.
            s = c.createStatement();
            // Delete the test table if it exists. Note: This
            // example assumes that the collection MYLIBRARY
            // exists on the system.
            try {
                s.executeUpdate("DROP TABLE MYLIBRARY.MYTABLE");
            } catch (SQLException e) {
                // Just continue... the table probably does not exist.
            }

            // Run an SQL statement that creates a table in the database.
            s.executeUpdate("CREATE TABLE MYLIBRARY.MYTABLE (NAME VARCHAR(20), ID INTEGER)");

            // Run some SQL statements that insert records into the table.
            s.executeUpdate("INSERT INTO MYLIBRARY.MYTABLE (NAME, ID) VALUES ('RICH', 123)");
            s.executeUpdate("INSERT INTO MYLIBRARY.MYTABLE (NAME, ID) VALUES ('FRED', 456)");
            s.executeUpdate("INSERT INTO MYLIBRARY.MYTABLE (NAME, ID) VALUES ('MARK', 789)");

            // Run an SQL query on the table.
            ResultSet rs = s.executeQuery("SELECT * FROM MYLIBRARY.MYTABLE");
```

```
            // Display all the data in the table.
            while (rs.next()) {
                System.out.println("Employee " + rs.getString(1) + " has ID " + rs.getInt(2));
            }

        } catch (SQLException sqle) {
            System.out.println("Database processing has failed.");
            System.out.println("Reason: " + sqle.getMessage());
        } finally {
            // Close database resources
            try {
                if (s != null) {
                    s.close();
                }
            } catch (SQLException e) {
                System.out.println("Cleanup failed to close Statement.");
            }
             }

            try {
                if (c != null) {
                    c.close();
                }
            } catch (SQLException e) {
                System.out.println("Cleanup failed to close Connection.");
            }
        }
    }
}
```

PreparedStatements

PreparedStatements extend the Statement interface and provide support for adding parameters to SQL statements.

SQL statements that are passed to the database go through a two-step process in returning results to you. They are first prepared and then are processed. With Statement objects, these two phases appear to be one phase to your applications. PreparedStatements allow these two steps to be broken apart. The preparation step occurs when the object is created and the processing step occurs when the executeQuery, executeUpdate, or execute method are called on the PreparedStatement object.

Being able to split the SQL processing into separate phases are meaningless without the addition of parameter markers. Parameter markers are placed in an application so that it can tell the database that it does not have a specific value at preparation time, but that it provides one before processing time. Parameter markers are represented in SQL statements by question marks.

Parameter markers make it possible to make general SQL statements that are used for specific requests. For example, take the following SQL query statement:

```
SELECT * FROM EMPLOYEE_TABLE WHERE LASTNAME = 'DETTINGER'
```

This is a specific SQL statement that returns only one value; that is, information about an employee named Dettinger. By adding a parameter marker, the statement can become more flexible:

```
SELECT * FROM EMPLOYEE_TABLE WHERE LASTNAME = ?
```

By simply setting the parameter marker to a value, information can be obtained about any employee in the table.

PreparedStatements provide significant performance improvements over Statements because the previous Statement example can go through the preparation phase only once and then be processed repeatedly with different values for the parameter.

Note: Using PreparedStatements is a requirement to support the native JDBC driver's statement pooling.

For more information about using prepared statements, including creating prepared statements, specifying result set characteristics, working with auto-generated keys, and setting parameter markers, see the following pages:

Related concepts

Statement objects
A Statement object is used for processing a static SQL statement and obtaining the results produced by it. Only one ResultSet for each Statement object can be open at a time. All statement methods that process an SQL statement implicitly close a statement's current ResultSet if an open one exists.

CallableStatements
The JDBC CallableStatement interface extends PreparedStatement and provides support for output and input/output parameters. The CallableStatement interface also has support for input parameters that is provided by the PreparedStatement interface.

Creating and using PreparedStatements
The prepareStatement method is used to create new PreparedStatement objects. Unlike the createStatement method, the SQL statement must be supplied when the PreparedStatement object is created. At that time, the SQL statement is precompiled for use.

For example, assuming a Connection object named conn already exists, the following example creates a PreparedStatement object and prepares the SQL statement for processing within the database.

```
PreparedStatement ps = conn.prepareStatement("SELECT * FROM EMPLOYEE_TABLE
                                              WHERE LASTNAME = ?");
```

Specifing ResultSet characteristics and auto-generated key support

As with the createStatement method, the prepareStatement method is overloaded to provide support for specifying ResultSet characteristics. The prepareStatement method also has variations for working with auto-generated keys. The following are some examples of valid calls to the prepareStatement method:

Example: The prepareStatement method

Note: Read the Code example disclaimer for important legal information.

```
// New in JDBC 2.0

PreparedStatement ps2 = conn.prepareStatement("SELECT * FROM
    EMPLOYEE_TABLE WHERE LASTNAME = ?",

ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_UPDATEABLE);

// New in JDBC 3.0

PreparedStatement ps3 = conn.prepareStatement("SELECT * FROM
    EMPLOYEE_TABLE WHERE LASTNAME = ?",
    ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATEABLE,
    ResultSet.HOLD_CURSOR_OVER_COMMIT);

PreparedStatement ps4 = conn.prepareStatement("SELECT * FROM
    EMPLOYEE_TABLE WHERE LASTNAME = ?", Statement.RETURN_GENERATED_KEYS);
```

Handling parameters

Before a PreparedStatement object can be processed, each of the parameter markers must be set to some value. The PreparedStatement object provides a number of methods for setting parameters. All methods are of the form set<Type>, where <Type> is a Java data type. Some examples of these methods include setInt, setLong, setString, setTimestamp, setNull, and setBlob. Nearly all of these methods take two parameters:

- The first parameter is the index of the parameter within the statement. Parameter markers are numbered, starting with 1.
- The second parameter is the value to set the parameter to. There are a couple set<Type> methods that have additional parameters such as the length parameter on setBinaryStream.

Consult the Javadoc for the java.sql package for more information. Given the prepared SQL statement in the previous examples for the ps object, the following code illustrates how the parameter value is specified before processing:

```
ps.setString(1,'Dettinger');
```

If an attempt is made to process a PreparedStatement with parameter markers that have not been set, an SQLException is thrown.

Note: Once set, parameter markers hold the same value between processes unless the following situations occur:

- The value is changed by another call to a set method.
- The value is removed when the clearParameters method is called.

The clearParameters method flags all parameters as being unset. After the call to clearParameters has been made, all the parameters must have the set method called again before the next process.

ParameterMetaData support

A new ParameterMetaData interface allows you to retrieve information about a parameter. This support is the compliment to ResultSetMetaData and is similar. Information such as the precision, scale, data type, data type name, and whether the parameter allows the null value are all provided.

Related concepts

Processing PreparedStatements
Processing SQL statements with a PreparedStatement object is accomplished with the executeQuery, executeUpdate, and execute methods like Statement objects are processed. Unlike Statement versions, no parameters are passed on these methods because the SQL statement was already provided when the object was created. Because PreparedStatement extends Statement, applications can attempt to call versions of executeQuery, executeUpdate, and execute methods that take a SQL statement. Doing so results in an SQLException being thrown.

Related reference

Example: Using PreparedStatement to obtain a ResultSet
This is an example of using a PreparedStatement object's executeQuery method to obtain a ResultSet.

Example: ParameterMetaData
This is an example of using the ParameterMetaData interface to retrieve information about parameters.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
//////////////////////////////////////////////////////////////////////////////////
//
// ParameterMetaData example. This program demonstrates
// the new support of JDBC 3.0 for learning information
// about parameters to a PreparedStatement.
//
// Command syntax:
//    java PMD
//
//////////////////////////////////////////////////////////////////////////////////
//
// This source is an example of the IBM Developer for Java JDBC driver.
// IBM grants you a nonexclusive license to use this as an example
// from which you can generate similar function tailored to
// your own specific needs.
//
// This sample code is provided by IBM for illustrative purposes
// only. These examples have not been thoroughly tested under all
// conditions. IBM, therefore, cannot guarantee or imply
// reliability, serviceability, or function of these programs.
//
// All programs contained herein are provided to you "AS IS"
// without any warranties of any kind.  The implied warranties of
// merchantability and fitness for a particular purpose are
// expressly disclaimed.
```

```
//
// IBM Developer Kit for Java
// (C) Copyright IBM Corp. 2001
// All rights reserved.
// US Government Users Restricted Rights -
// Use, duplication, or disclosure restricted
// by GSA ADP Schedule Contract with IBM Corp.
//
//////////////////////////////////////////////////////////////////////////////////

import java.sql.*;

public class PMD {

    // Program entry point.
    public static void main(java.lang.String[] args)
    throws Exception
    {
        // Obtain setup.
        Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
        Connection c = DriverManager.getConnection("jdbc:db2:*local");
        PreparedStatement ps = c.prepareStatement("INSERT INTO CUJOSQL.MYTABLE
VALUES(?, ?, ?)");
        ParameterMetaData pmd = ps.getParameterMetaData();

        for (int i = 1; i < pmd.getParameterCount(); i++) {
            System.out.println("Parameter number " + i);
            System.out.println("  Class name is " + pmd.getParameterClassName(i));
            // Note: Mode relates to input, output or inout
            System.out.println("  Mode is " + pmd.getParameterClassName(i));
            System.out.println("  Type is " + pmd.getParameterType(i));
            System.out.println("  Type name is " + pmd.getParameterTypeName(i));
            System.out.println("  Precision is " + pmd.getPrecision(i));
            System.out.println("  Scale is " + pmd.getScale(i));
            System.out.println("  Nullable? is " + pmd.isNullable(i));
            System.out.println("  Signed? is " + pmd.isSigned(i));
        }
    }
}
```

Processing PreparedStatements
Processing SQL statements with a PreparedStatement object is accomplished with the executeQuery, executeUpdate, and execute methods like Statement objects are processed. Unlike Statement versions, no parameters are passed on these methods because the SQL statement was already provided when the object was created. Because PreparedStatement extends Statement, applications can attempt to call versions of executeQuery, executeUpdate, and execute methods that take a SQL statement. Doing so results in an SQLException being thrown.

Returning results from SQL queries

If an SQL query statement that returns a ResultSet object is to be processed, the executeQuery method should be used. The PreparedStatementExample program uses a PreparedStatement object's executeQuery method to obtain a ResultSet.

Note: If an SQL statement processed with the executeQuery method does not return a ResultSet, an SQLException is thrown.

Returning update counts for SQL statements

If the SQL is known to be a Data Definition Language (DDL) statement or a Data Manipulation Language (DML) statement that returns an update count, the executeUpdate method should be used. The PreparedStatementExample sample program uses a PreparedStatement object's executeUpdate method.

Processing SQL statements where the expected return is unknown

If the SQL statement type is not known, the execute method should be used. Once this method has been processed, the JDBC driver can tell the application what results types the SQL statement generated through API calls. The execute method returns true if the result is at least one ResultSet and false if the return value is an update count. Given this information, applications can use the getUpdateCount or getResultSet statement methods to retrieve the return value from processing the SQL statement.

Note: Calling the getUpdateCount method when the result is a ResultSet returns -1. Calling the getResultSet method when the result is an update count returns null.

Related concepts

Creating and using PreparedStatements
The prepareStatement method is used to create new PreparedStatement objects. Unlike the createStatement method, the SQL statement must be supplied when the PreparedStatement object is created. At that time, the SQL statement is precompiled for use.

Related reference

Example: Using PreparedStatement to obtain a ResultSet
This is an example of using a PreparedStatement object's executeQuery method to obtain a ResultSet.

Example: Using PreparedStatement to obtain a ResultSet
This is an example of using a PreparedStatement object's executeQuery method to obtain a ResultSet.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
import java.sql.*;
import java.util.Properties;

public class PreparedStatementExample {

    public static void main(java.lang.String[] args)
    {
        // Load the following from a properties object.
        String DRIVER = "com.ibm.db2.jdbc.app.DB2Driver";
        String URL    = "jdbc:db2://*local";

        // Register the native JDBC driver. If the driver cannot
        // be registered, the test cannot continue.
        try {
            Class.forName(DRIVER);
        } catch (Exception e) {
            System.out.println("Driver failed to register.");
            System.out.println(e.getMessage());
            System.exit(1);
        }

        Connection c = null;
        Statement s = null;

        //    This program creates a table that is
        //    used by prepared statements later.
        try {
            // Create the connection properties.
            Properties properties = new Properties ();
            properties.put ("user", "userid");
            properties.put ("password", "password");

            // Connect to the local database.
            c = DriverManager.getConnection(URL, properties);

            // Create a Statement object.
            s = c.createStatement();
            // Delete the test table if it exists.  Note that
            // this example assumes throughout that the collection
            // MYLIBRARY exists on the system.
            try {
                s.executeUpdate("DROP TABLE MYLIBRARY.MYTABLE");
            } catch (SQLException e) {
                // Just continue... the table probably did not exist.
            }

            // Run an SQL statement that creates a table in the database.
            s.executeUpdate("CREATE TABLE MYLIBRARY.MYTABLE (NAME VARCHAR(20), ID INTEGER)");

        } catch (SQLException sqle) {
            System.out.println("Database processing has failed.");
            System.out.println("Reason: " + sqle.getMessage());
        } finally {
```

```
            // Close database resources
            try {
                if (s != null) {
                    s.close();
                }
            } catch (SQLException e) {
                System.out.println("Cleanup failed to close Statement.");
            }
        }


        //    This program then uses a prepared statement to insert many
        //    rows into the database.
        PreparedStatement ps = null;
        String[] nameArray = {"Rich", "Fred", "Mark", "Scott", "Jason",
            "John", "Jessica", "Blair", "Erica", "Barb"};
        try {
            // Create a PreparedStatement object that is used to insert data into the
            // table.
            ps = c.prepareStatement("INSERT INTO MYLIBRARY.MYTABLE (NAME, ID) VALUES (?, ?)");

            for (int i = 0; i < nameArray.length; i++) {
                ps.setString(1, nameArray[i]);      // Set the Name from our array.
                ps.setInt(2, i+1);                  // Set the ID.
                ps.executeUpdate();
            }

        } catch (SQLException sqle) {
            System.out.println("Database processing has failed.");
            System.out.println("Reason: " + sqle.getMessage());
        } finally {
            // Close database resources
            try {
                if (ps != null) {
                    ps.close();
                }
            } catch (SQLException e) {
                System.out.println("Cleanup failed to close Statement.");
            }
        }


        //    Use a prepared statement to query the database
        //    table that has been created and return data from it. In
        //    this example, the parameter used is arbitrarily set to
        //    5, meaning return all rows where the ID field is less than
        //    or equal to 5.
        try {
            ps = c.prepareStatement("SELECT * FROM MYLIBRARY.MYTABLE " +
                                    "WHERE ID <= ?");

            ps.setInt(1, 5);

            // Run an SQL query on the table.
            ResultSet rs = ps.executeQuery();
            // Display all the data in the table.
            while (rs.next()) {
                System.out.println("Employee " + rs.getString(1) + " has ID " + rs.getInt(2));
            }

        } catch (SQLException sqle) {
            System.out.println("Database processing has failed.");
            System.out.println("Reason: " + sqle.getMessage());
        } finally {
            // Close database resources
            try {
                if (ps != null) {
                    ps.close();
                }
            } catch (SQLException e) {
                System.out.println("Cleanup failed to close Statement.");
            }

            try {
                if (c != null) {
                    c.close();
                }
            } catch (SQLException e) {
                System.out.println("Cleanup failed to close Connection.");
            }

        }
```

```
    }
}
```

Related concepts

Creating and using PreparedStatements
The prepareStatement method is used to create new PreparedStatement objects. Unlike the createStatement method, the SQL statement must be supplied when the PreparedStatement object is created. At that time, the SQL statement is precompiled for use.

Processing PreparedStatements
Processing SQL statements with a PreparedStatement object is accomplished with the executeQuery, executeUpdate, and execute methods like Statement objects are processed. Unlike Statement versions, no parameters are passed on these methods because the SQL statement was already provided when the object was created. Because PreparedStatement extends Statement, applications can attempt to call versions of executeQuery, executeUpdate, and execute methods that take a SQL statement. Doing so results in an SQLException being thrown.

CallableStatements

The JDBC CallableStatement interface extends PreparedStatement and provides support for output and input/output parameters. The CallableStatement interface also has support for input parameters that is provided by the PreparedStatement interface.

The CallableStatement interface allows the use of SQL statements to call stored procedures. Stored procedures are programs that have a database interface. These programs possess the following:

- They can have input and output parameters, or parameters that are both input and output.
- They can have a return value.
- They have the ability to return multiple ResultSets.

Conceptually in JDBC, a stored procedure call is a single call to the database, but the program associated with the stored procedure may process hundreds of database requests. The stored procedure program may also perform a number of other programmatic tasks not typically done with SQL statements.

Because CallableStatements follow the PreparedStatement model of decoupling the preparation and processing phases, they have the potential for optimized reuse (see "PreparedStatements" on page 87 for details). Since SQL statements of a stored procedure are bound into a program, they are processed as static SQL and further performance benefits can be gained that way. Encapsulating a lot of database work in a single, reusable database call is an example of using stored procedures optimally. Only this call goes over the network to the other system, but the request can accomplish a lot of work on the remote system.

Creating CallableStatements

The prepareCall method is used to create new CallableStatement objects. As with the prepareStatement method, the SQL statement must be supplied at the time that the CallableStatement object is created. At that time, the SQL statement is precompiled. For example, assuming a Connection object named conn already exists, the following creates a CallableStatement object and completes the preparation phase of getting the SQL statement ready for processing within the database:

```
PreparedStatement ps = conn.prepareStatement("? = CALL ADDEMPLOYEE(?, ?, ?");
```

The ADDEMPLOYEE stored procedure takes input parameters for a new employee name, his social security number, and his manager's user ID. From this information, multiple company database tables may be updated with information about the employee such as his start date, division, department, and so on. Further, a stored procedure is a program that may generate standard user IDs and e-mail addresses for that employee. The stored procedure may also send an e-mail to the hiring manager with initial usernames and passwords; the hiring manager can then provide the information to the employee.

The ADDEMPLOYEE stored procedure is set up to have a return value. The return code may be a success or failure code that the calling program can use when a failure occurs. The return value may also be defined as the new employee's company ID number. Finally, the stored procedure program could have processed queries internally and have left the ResultSets from those queries open and available for the calling program. Querying all the new employee's information and making it available to the caller through a returned ResultSet is reasonable.

How to accomplish each of these types of tasks is covered in the following sections.

Specifying ResultSet characteristics and auto-generated key support

As with createStatement and prepareStatement, there are multiple versions of prepareCall that provide support for specifying ResultSet characteristics. Unlike prepareStatement, the prepareCall method does not provide variations for working with auto-generated keys from CallableStatements (JDBC 3.0 does not support this concept.) The following are some examples of valid calls to the prepareCall method:

Example: The prepareCall method

```
// The following is new in JDBC 2.0

CallableStatement cs2 = conn.prepareCall("? = CALL ADDEMPLOYEE(?, ?, ?)",
    ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATEABLE);

// New in JDBC 3.0

CallableStatement cs3 = conn.prepareCall("? = CALL ADDEMPLOYEE(?, ?, ?)",
    ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATEABLE,
    ResultSet.HOLD_CURSOR_OVER_COMMIT);
```

Handling parameters

As stated, CallableStatement objects may take three types of parameters:

- **IN**

  IN parameters are handled in the same manner as PreparedStatements. The various set methods of the inherited PreparedStatement class are used to set the parameters.
- **OUT**

  OUT parameters are handled with the registerOutParameter method. The most common form of registerOutParameter takes an index parameter as the first parameter and an SQL type as the second parameter. This tells the JDBC driver what to expect for data from the parameter when the statement is processed. There are two other variations on the registerOutParameter method that can be found in the java.sql package Javadoc.
- **INOUT**

  INOUT parameters require that the work for both IN parameters and OUT parameters be done. For each INOUT parameter, you must call a set method and the registerOutParameter method before the statement can be processed. Failing to set or register any parameter results in an SQLException being thrown when the statement is processed.

Refer to "Example: Creating a procedure with input and output parameters" on page 98 for more information.

As with PreparedStatements, CallableStatement parameter values remain the same between processes unless you call a set method again. The clearParameters method does not affect parameters that are registered for output. After calling clearParameters, all IN parameters must be set to a value again, but all OUT parameters do not have to be registered again.

Note: The concept of parameters must not be confused with the index of a parameter marker. A stored procedure call expects a certain number of parameters that are passed to it. A particular SQL statement has ? characters (parameter markers) in it to represent values that are supplied at runtime. Consider the following example to see the difference between the two concepts:

```
CallableStatement cs = con.prepareCall("CALL PROC(?, "SECOND", ?)");

cs.setString(1, "First");     //Parameter marker 1, Stored procedure parm 1

cs.setString(2, "Third");     //Parameter marker 2, Stored procedure parm 3
```

Accessing stored procedure parameters by name

Parameters to stored procedures have names associated with them as the following stored procedure declaration shows:

Example: Stored procedure parameters

```
CREATE
PROCEDURE MYLIBRARY.APROC
    (IN PARM1 INTEGER)
LANGUAGE SQL SPECIFIC MYLIBRARY.APROC
BODY: BEGIN
    <Perform a task here...>
END BODY
```

There is a single integer parameter with the name PARM1. In JDBC 3.0, there is support for specifying stored procedure parameters by name as well as by index. The code to set up a CallableStatement for this procedure is as follows:

```
CallableStatement cs = con.prepareCall("CALL APROC(?)");

cs.setString("PARM1", 6);     //Sets input parameter at index 1 (PARM1) to 6.
```

Related concepts

Statement objects
A Statement object is used for processing a static SQL statement and obtaining the results produced by it. Only one ResultSet for each Statement object can be open at a time. All statement methods that process an SQL statement implicitly close a statement's current ResultSet if an open one exists.

PreparedStatements
PreparedStatements extend the Statement interface and provide support for adding parameters to SQL statements.

Processing CallableStatements
Processing SQL stored procedure calls with a JDBC CallableStatement object is accomplished with the same methods that are used with a PreparedStatement object.

Return results for stored procedures

If an SQL query statement is processed within a stored procedure, the query results can be made available to the program calling the stored procedure. Multiple queries can also be called within the stored procedure and the calling program can process all the ResultSets that are available.

See "Example: Creating a procedure with multiple ResultSets" on page 96 for more information.

Note: If a stored procedure is processed with executeQuery and it does not return a ResultSet, an SQLException is thrown.

Access ResultSets concurrently

Return results for stored procedures deals with ResultSets and stored procedures and provides an example that works with all Java Development Kit (JDK) releases. In the example, the ResultSets are processed in order from the first ResultSet that the stored procedure opened to the last ResultSet opened. One ResultSet is closed before the next is used.

In JDK 1.4 and subsequent versions, there is support for working with ResultSets from stored procedures concurrently.

Note: This feature was added to the underlying system support through the Command Line Interface (CLI) in V5R2. As a result, JDK 1.4 or a subsequent version of the JDK running on a system before V5R2 does not have this support available to it.

Return update counts for stored procedures

Returning update counts for stored procedures is a feature discussed in the JDBC specification, but it is not currently supported on the IBM i platform. There is no way to return multiple update counts from a stored procedure call. If an update count is needed from a processed SQL statement within a stored procedure, there are two ways of returning the value:

- Return the value as an output parameter.
- Pass back the value as the return value from the parameter. This is a special case of an output parameter. See Process stored procedures that have a return for more information.

Process stored procedures where the expected return is unknown

If the results from a stored procedure call are not known, the execute method should be used. Once this method has been processed, the JDBC driver can tell the application what types of results the stored procedure generated through API calls. The execute method returns true if the result is one or more ResultSets. Updating counts do not come from stored procedure calls.

Process stored procedures that have a return value

The IBM i platform supports stored procedures that have a return value similar to a function's return value. The return value from a stored procedure is labeled like other parameters marks and is labeled such that it is assigned by the stored procedure call. An example of this is as follows:

```
? = CALL MYPROC(?, ?, ?)
```

The return value from a stored procedure call is always an integer type and must be registered like any other output parameter.

See "Example: Creating a procedure with return values" on page 99 for more information.

Related reference

Example: Creating a procedure with multiple ResultSets
This example shows how to access a database and then create a procedure with multiple ResultSets using JDBC.

Example: Creating a procedure with input and output parameters
This example shows how to access a database using JDBC and then create a procedure with input and output parameters.

Example: Creating a procedure with return values
This example shows how to access a database using JDBC and then create a procedure with return values.

Example: Creating a procedure with multiple ResultSets
This example shows how to access a database and then create a procedure with multiple ResultSets using JDBC.

Note: Read the Code example disclaimer for important legal information.

```
import java.sql.*;
import java.util.Properties;

public class CallableStatementExample1 {

    public static void main(java.lang.String[] args) {

        // Register the Native JDBC driver.  If we cannot
        // register the driver, the test cannot continue.
        try {
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");

            // Create the connection properties
            Properties properties = new Properties ();
            properties.put ("user", "userid");
            properties.put ("password", "password");
```

```
            // Connect to the local server database
            Connection c = DriverManager.getConnection("jdbc:db2://*local", properties);

            Statement s = c.createStatement();

            // Create a procedure with multiple ResultSets.
            String sql = "CREATE PROCEDURE MYLIBRARY.SQLSPEX1 " +
                         "RESULT SET 2 LANGUAGE SQL READS SQL DATA SPECIFIC MYLIBRARY.SQLSPEX1
" +
                         "EX1: BEGIN " +
                         "   DECLARE C1 CURSOR FOR SELECT * FROM QSYS2.SYSPROCS " +
                         "               WHERE SPECIFIC_SCHEMA = 'MYLIBRARY';  " +
                         "   DECLARE C2 CURSOR FOR SELECT * FROM QSYS2.SYSPARMS  " +
                         "               WHERE SPECIFIC_SCHEMA = 'MYLIBRARY';  " +
                         "   OPEN C1; " +
                         "   OPEN C2; " +
                         "   SET RESULT SETS CURSOR C1, CURSOR C2; " +
                         "END EX1 ";

            try {
                s.executeUpdate(sql);
                } catch (SQLException e) {
                // NOTE:  We are ignoring the error here.  We are making
                //        the assumption that the only reason this fails
                //        is because the procedure already exists. Other
                //        reasons that it could fail are because the C compiler
                //        is not found to compile the procedure or because
                //        collection MYLIBRARY does not exist on the system.
            }
            s.close();

            // Now use JDBC to run the procedure and get the results back. In
            // this case we are going to get information about 'MYLIBRARY's stored
            // procedures (which is also where we created this procedure, thereby
            // ensuring that there is something to get.
            CallableStatement cs = c.prepareCall("CALL MYLIBRARY.SQLSPEX1");

            ResultSet rs = cs.executeQuery();

            // We now have the first ResultSet object that the stored procedure
            // left open.  Use it.
            int i = 1;
            while (rs.next()) {
                System.out.println("MYLIBRARY stored procedure
                                   " + i + " is " + rs.getString(1) + "." +
                                   rs.getString(2));
                i++;
            }
            System.out.println("");

            // Now get the next ResultSet object from the system - the previous
            // one is automatically closed.
            if (!cs.getMoreResults()) {
                System.out.println("Something went wrong.  There should have
                                    been another ResultSet, exiting.");
                System.exit(0);
            }
            rs = cs.getResultSet();

             // We now have the second ResultSet object that the stored procedure
            // left open.  Use that one.
            i = 1;
            while (rs.next()) {
                System.out.println("MYLIBRARY procedure " + rs.getString(1)
                                    + "." + rs.getString(2) +
                                   " parameter:  " + rs.getInt(3) + " direction:
                                   " + rs.getString(4) +
                                   " data type: " + rs.getString(5));
                i++;
            }

            if (i == 1) {
                System.out.println("None of the stored procedures have any parameters.");
            }

            if (cs.getMoreResults()) {
                System.out.println("Something went wrong,
                                    there should not be another ResultSet.");
                System.exit(0);
            }
```

```
            cs.close();  // close the CallableStatement object
            c.close();   // close the Connection object.

        } catch (Exception e) {
            System.out.println("Something failed..");
            System.out.println("Reason: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

Related concepts

Processing CallableStatements
Processing SQL stored procedure calls with a JDBC CallableStatement object is accomplished with the same methods that are used with a PreparedStatement object.

Related reference

Example: Creating a procedure with input and output parameters
This example shows how to access a database using JDBC and then create a procedure with input and output parameters.

Example: Creating a procedure with return values
This example shows how to access a database using JDBC and then create a procedure with return values.

Example: Creating a procedure with input and output parameters
This example shows how to access a database using JDBC and then create a procedure with input and output parameters.

Note: Read the Code example disclaimer for important legal information.

```
import java.sql.*;
import java.util.Properties;

public class CallableStatementExample2 {

    public static void main(java.lang.String[] args) {

        // Register the Native JDBC driver.  If we cannot
        // register the driver, the test cannot continue.
        try {
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");

            // Create the connection properties
            Properties properties = new Properties ();
            properties.put ("user", "userid");
            properties.put ("password", "password");

            // Connect to the local server database
            Connection c = DriverManager.getConnection("jdbc:db2://*local", properties);

            Statement s = c.createStatement();

            // Create a procedure with in, out, and in/out parameters.
            String sql = "CREATE PROCEDURE MYLIBRARY.SQLSPEX2 " +
                         "(IN P1 INTEGER, OUT P2 INTEGER, INOUT P3 INTEGER) " +
                         "LANGUAGE SQL SPECIFIC MYLIBRARY.SQLSPEX2 " +
                         "EX2: BEGIN " +
                         "   SET P2 = P1 + 1; " +
                         "   SET P3 = P3 + 1; " +
                         "END EX2 ";

            try {
                s.executeUpdate(sql);
            } catch (SQLException e) {
                // NOTE:  We are ignoring the error here.  We are making
                //        the assumption that the only reason this fails
                //        is because the procedure already exists.  Other
                //        reasons that it could fail are because the C compiler
                //        is not found to compile the procedure or because
                //        collection MYLIBRARY does not exist on the system.
            }
            s.close();

            // Prepare a callable statement used to run the procedure.
```

```
            CallableStatement cs = c.prepareCall("CALL MYLIBRARY.SQLSPEX2(?, ?, ?)");

            // All input parameters must be set and all output parameters must
            // be registered.  Notice that this means we have two calls to make
            // for an input output parameter.
            cs.setInt(1, 5);
            cs.setInt(3, 10);
            cs.registerOutParameter(2, Types.INTEGER);
            cs.registerOutParameter(3, Types.INTEGER);

            // Run the procedure
            cs.executeUpdate();

            // Verify the output parameters have the desired values.
            System.out.println("The value of P2 should be P1 (5) + 1 = 6.  -->  " +
cs.getInt(2));
            System.out.println("The value of P3 should be P3 (10) + 1 = 11.  -->  " +
cs.getInt(3));

            cs.close();  // close the CallableStatement object
            c.close();   // close the Connection object.

        } catch (Exception e) {
            System.out.println("Something failed..");
            System.out.println("Reason: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

Related concepts

Processing CallableStatements
Processing SQL stored procedure calls with a JDBC CallableStatement object is accomplished with the same methods that are used with a PreparedStatement object.

Related reference

Example: Creating a procedure with multiple ResultSets
This example shows how to access a database and then create a procedure with multiple ResultSets using JDBC.

Example: Creating a procedure with return values
This example shows how to access a database using JDBC and then create a procedure with return values.

Example: Creating a procedure with return values
This example shows how to access a database using JDBC and then create a procedure with return values.

Note: Read the Code example disclaimer for important legal information.

```
import java.sql.*;
import java.util.Properties;

public class CallableStatementExample3 {

    public static void main(java.lang.String[] args) {

        // Register the native JDBC driver. If the driver cannot
        // be registered, the test cannot continue.
        try {
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");

            // Create the connection properties
            Properties properties = new Properties ();
            properties.put ("user", "userid");
            properties.put ("password", "password");

            // Connect to the local server database
            Connection c = DriverManager.getConnection("jdbc:db2://*local", properties);

            Statement s = c.createStatement();

            // Create a procedure with a return value.
            String sql = "CREATE PROCEDURE MYLIBRARY.SQLSPEX3 " +
                         " LANGUAGE SQL SPECIFIC MYLIBRARY.SQLSPEX3 " +
```

```
                         " EX3: BEGIN " +
                         "      RETURN 1976; " +
                         " END EX3 ";

            try {
                s.executeUpdate(sql);
            } catch (SQLException e) {
                // NOTE:  The error is ignored here. The assumptions is
                //        made that the only reason this fails is
                //        because the procedure already exists. Other
                //        reasons that it could fail are because the C compiler
                //        is not found to compile the procedure or because
                //        collection MYLIBRARY does not exist on the system.
            }
            s.close();

            // Prepare a callable statement used to run the procedure.
            CallableStatement cs = c.prepareCall("? = CALL MYLIBRARY.SQLSPEX3");

            // You still need to register the output parameter.
            cs.registerOutParameter(1, Types.INTEGER);

            // Run the procedure.
            cs.executeUpdate();

            // Show that the correct value is returned.
            System.out.println("The return value
                                should always be 1976 for this example:
                                -->  " + cs.getInt(1));

            cs.close();  // close the CallableStatement object
            c.close();   // close the Connection object.

        } catch (Exception e) {
            System.out.println("Something failed..");
            System.out.println("Reason: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

Related concepts

Processing CallableStatements
Processing SQL stored procedure calls with a JDBC CallableStatement object is accomplished with the same methods that are used with a PreparedStatement object.

Related reference

Example: Creating a procedure with multiple ResultSets
This example shows how to access a database and then create a procedure with multiple ResultSets using JDBC.

Example: Creating a procedure with input and output parameters
This example shows how to access a database using JDBC and then create a procedure with input and output parameters.

ResultSets

The ResultSet interface provides access to the results generated by running queries. Conceptually, data of a ResultSet can be thought of as a table with a specific number of columns and a specific number of rows. By default, the table rows are retrieved in sequence. Within a row, column values can be accessed in any order.

ResultSet characteristics

This topic discusses ResultSet characteristics such ResultSet types, concurrency, ability to close the ResultSet by committing the connection object, and specification of ResultSet characteristics.

By default, all created ResultSets have a type of forward only, a concurrency of read only, and cursors are held over commit boundaries. An exception to this is that WebSphere currently changes the cursor holdability default so that cursors are implicitly closed when committed. These characteristics are configurable through methods that are accessible on Statement, PreparedStatement, and CallableStatement objects.

ResultSet types

The ResultSet type specifies the following about the ResultSet:

- Whether the ResultSet is scrollable.
- The types of Java Database Connectivity (JDBC) ResultSets that are defined by constants on the ResultSet interface.

Definitions of these ResultSet types are as follows:

TYPE_FORWARD_ONLY
A cursor that can only be used to process from the beginning of a ResultSet to the end of it. This is the default type.

TYPE_SCROLL_INSENSITIVE
A cursor that can be used to scroll through a ResultSet. This type of cursor is insensitive to changes made to the database while it is open. It contains rows that satisfy the query when the query was processed or when data is fetched.

TYPE_SCROLL_SENSITIVE
A cursor that can be used to scroll in various ways through a ResultSet. This type of cursor is sensitive to changes made to the database while it is open. Changes to the database have a direct impact on the ResultSet data.

JDBC 1.0 ResultSets are always forward only. Scrollable cursors were added in JDBC 2.0.

Note: The blocking enabled and block size connection properties affect the degree of sensitivity of a TYPE_SCROLL_SENSITIVE cursor. Blocking enhances performance by caching data in the JDBC driver layer itself.

Concurrency

Concurrency determines whether the ResultSet can be updated. The types are again defined by constants in the ResultSet interface. The available concurrency settings are as follows:

CONCUR_READ_ONLY
A ResultSet that can only be used for reading data out of the database. This is the default setting.

CONCUR_UPDATEABLE
A ResultSet that allows you to make changes to it. These changes can be placed into the underlying database.

JDBC 1.0 ResultSets are always forward only. Updateable ResultSets were added in JDBC 2.0.

Note: According to the JDBC specification, the JDBC driver is allowed to change the ResultSet type of the ResultSet concurrency setting if the values cannot be used together. In such cases, the JDBC driver places a warning on the Connection object.

There is one situation where the application specifies a TYPE_SCROLL_INSENSITIVE, CONCUR_UPDATEABLE ResultSet. Insensitivity is implemented in the database engine by making a copy of the data. You are then not allowed to make updates through that copy to the underlying database. If you specify this combination, the driver changes the sensitivity to TYPE_SCROLL_SENSITIVE and create the warning indicating that your request has been changed.

Holdability

The holdability characteristic determines whether calling commit on the Connection object closes the ResultSet. The JDBC API for working with the holdability characteristic is new in version 3.0. However, the native JDBC driver has provided a connection property for several releases that allows you to specify that default for all ResultSets created under the connection. The API support overrides any setting for the connection property. Values for the holdability characteristic are defined by ResultSet constants and are as follows:

HOLD_CURSOR_OVER_COMMIT
: All open cursors remain open when the commit clause is called. This is the native JDBC default value.

CLOSE_CURSORS_ON_COMMIT
: All open cursors are closed when commit clause is called.

Note: Calling rollback on a connection always closes all open cursors. This is a little known fact, but a common way for databases to handle cursors.

According to the JDBC specification, the default for cursor holdability is implementation-defined. Some platforms choose to use CLOSE_CURSORS_ON_COMMIT as the default. This does not usually become an issue for most applications, but you must be aware of what the driver you are working with does if you are working with cursors across commit boundaries. The IBM Toolbox for Java JDBC driver also uses the HOLD_CURSORS_ON_COMMIT default, but the JDBC driver for UDB for Windows NT has a default of CLOSE_CURSORS_ON_COMMIT.

Specifying ResultSet characteristics

A ResultSet's characteristics do not change once the ResultSet object has been created. Therefore, the characteristics have be specified before creating the object. You can specify these characteristics through overloaded variations of the createStatement, prepareStatement, and prepareCall methods.

Note: There are ResultSet methods to obtain the ResultSet type and the concurrency of the ResultSet, but there is no method to obtain the holdability of the ResultSet.

Related concepts

Cursor movement
The IBM i Java Database Connectivity (JDBC) drivers support scrollable ResultSets. With a scrollable ResultSet, you can process rows of data in any order using a number of cursor-positioning methods.

Retrieving ResultSet data
The ResultSet object provides several methods for obtaining column data for a row. All are of the form get<*Type*>, where <*Type*> is a Java data type. Some examples of these methods include getInt, getLong, getString, getTimestamp, and getBlob. Nearly all of these methods take a single parameter that is either the column index within the ResultSet or the column name.

Creating ResultSets
To create a ResultSet object, you can use executeQuery methods, or other methods. This topic describes options for creating ResultSets.

"Statement objects" on page 84
A Statement object is used for processing a static SQL statement and obtaining the results produced by it. Only one ResultSet for each Statement object can be open at a time. All statement methods that process an SQL statement implicitly close a statement's current ResultSet if an open one exists.

"CallableStatements" on page 93
The JDBC CallableStatement interface extends PreparedStatement and provides support for output and input/output parameters. The CallableStatement interface also has support for input parameters that is provided by the PreparedStatement interface.

"PreparedStatements" on page 87
PreparedStatements extend the Statement interface and provide support for adding parameters to SQL statements.

"Cursor movement" on page 106
The IBM i Java Database Connectivity (JDBC) drivers support scrollable ResultSets. With a scrollable ResultSet, you can process rows of data in any order using a number of cursor-positioning methods.

Related tasks

Changing ResultSets
With the IBM i JDBC drivers, you can change ResultSets by performing several tasks.

"Changing ResultSets" on page 110

With the IBM i JDBC drivers, you can change ResultSets by performing several tasks.

Related reference

Example: ResultSet interface
This is an example of how to use the ResultSet interface.

“JDBC driver connection properties” on page 38
This table contains valid JDBC driver connection properties, their values, and their descriptions.

“DataSource properties” on page 49
For each JDBC driver connection property, there is a corresponding data source method. This table contains the valid data source properties.

Example: Sensitive and insensitive ResultSets
The following example shows the difference between sensitive and insensitive ResultSets when rows are inserted into a table.

Note: By using the code examples, you agree to the terms of the “Code license and disclaimer information” on page 489.

```
import java.sql.*;

public class Sensitive {

    public Connection connection = null;

    public static void main(java.lang.String[] args) {
        Sensitive test = new Sensitive();

        test.setup();
        test.run("sensitive");
        test.cleanup();

        test.setup();
        test.run("insensitive");
        test.cleanup();
    }


    public void setup() {

        try {
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
            connection = DriverManager.getConnection("jdbc:db2:*local");

            Statement s = connection.createStatement();
            try {
                s.executeUpdate("drop table cujosql.sensitive");
            } catch (SQLException e) {
                // Ignored.
            }

            s.executeUpdate("create table cujosql.sensitive(col1 int)");
            s.executeUpdate("insert into cujosql.sensitive values(1)");
            s.executeUpdate("insert into cujosql.sensitive values(2)");
            s.executeUpdate("insert into cujosql.sensitive values(3)");
            s.executeUpdate("insert into cujosql.sensitive values(4)");
            s.executeUpdate("insert into cujosql.sensitive values(5)");
            s.close();

        } catch (Exception e) {
            System.out.println("Caught exception: " + e.getMessage());
            if (e instanceof SQLException) {
                SQLException another = ((SQLException) e).getNextException();
                System.out.println("Another:  " + another.getMessage());
            }
        }
    }


    public void run(String sensitivity) {
        try {
            Statement s = null;
            if (sensitivity.equalsIgnoreCase("insensitive")) {
                System.out.println("creating a TYPE_SCROLL_INSENSITIVE cursor");
                s = connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
```

```
                    ResultSet.CONCUR_READ_ONLY);
            } else {
                System.out.println("creating a TYPE_SCROLL_SENSITIVE cursor");
                s = connection.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                    ResultSet.CONCUR_READ_ONLY);
            }

            ResultSet rs = s.executeQuery("select * From cujosql.sensitive");

            // Fetch the five values that are there.
            rs.next();
            System.out.println("value is " + rs.getInt(1));
            rs.next();
            System.out.println("value is " + rs.getInt(1));
            rs.next();
            System.out.println("value is " + rs.getInt(1));
            rs.next();
            System.out.println("value is " + rs.getInt(1));
            rs.next();
            System.out.println("value is " + rs.getInt(1));
            System.out.println("fetched the five rows...");

            // Note:  If you fetch the last row, the ResultSet looks
            //        closed and subsequent new rows that are added
            //        are not be recognized.

            // Allow another statement to insert a new value.
            Statement s2 = connection.createStatement();
            s2.executeUpdate("insert into cujosql.sensitive values(6)");
            s2.close();

            // Whether a row is recognized is based on the sensitivity setting.
            if (rs.next()) {
                System.out.println("There is a row now: " + rs.getInt(1));
            } else {
                System.out.println("No more rows.");
            }


        } catch (SQLException e) {
            System.out.println("SQLException exception: ");
            System.out.println("Message:....." + e.getMessage());
            System.out.println("SQLState:...." + e.getSQLState());
            System.out.println("Vendor Code:." + e.getErrorCode());
            System.out.println("-------------------------------------");
            e.printStackTrace();
        }
        catch (Exception ex) {
            System.out.println("An exception other than an SQLException was thrown: ");
            ex.printStackTrace();
        }
    }



    public void cleanup() {
        try {
            connection.close();
        } catch (Exception e) {
            System.out.println("Caught exception: ");
            e.printStackTrace();
        }
    }
}
```

Example: ResultSet sensitivity

The following example shows how a change can affect a where clause of an SQL statement based on the sensitivity of the ResultSet.

Some of the formatting in this example may be incorrect in order to fit this example on a printed page.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
import java.sql.*;

public class Sensitive2 {
```

```
    public Connection connection = null;

    public static void main(java.lang.String[] args) {
        Sensitive2 test = new Sensitive2();

        test.setup();
        test.run("sensitive");
        test.cleanup();

        test.setup();
        test.run("insensitive");
        test.cleanup();
    }


    public void setup() {

        try {
            System.out.println("Native JDBC used");
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
            connection = DriverManager.getConnection("jdbc:db2:*local");

            Statement s = connection.createStatement();
            try {
                s.executeUpdate("drop table cujosql.sensitive");
            } catch (SQLException e) {
                // Ignored.
            }

            s.executeUpdate("create table cujosql.sensitive(col1 int)");
            s.executeUpdate("insert into cujosql.sensitive values(1)");
            s.executeUpdate("insert into cujosql.sensitive values(2)");
            s.executeUpdate("insert into cujosql.sensitive values(3)");
            s.executeUpdate("insert into cujosql.sensitive values(4)");
            s.executeUpdate("insert into cujosql.sensitive values(5)");


            try {
                s.executeUpdate("drop table cujosql.sensitive2");
            } catch (SQLException e) {
                // Ignored.
            }

            s.executeUpdate("create table cujosql.sensitive2(col2 int)");
            s.executeUpdate("insert into cujosql.sensitive2 values(1)");
            s.executeUpdate("insert into cujosql.sensitive2 values(2)");
            s.executeUpdate("insert into cujosql.sensitive2 values(3)");
            s.executeUpdate("insert into cujosql.sensitive2 values(4)");
            s.executeUpdate("insert into cujosql.sensitive2 values(5)");

            s.close();

        } catch (Exception e) {
            System.out.println("Caught exception: " + e.getMessage());
            if (e instanceof SQLException) {
                SQLException another = ((SQLException) e).getNextException();
                System.out.println("Another:  " + another.getMessage());
            }
        }
    }


    public void run(String sensitivity) {
        try {

            Statement s = null;
            if (sensitivity.equalsIgnoreCase("insensitive")) {
                System.out.println("creating a TYPE_SCROLL_INSENSITIVE cursor");
                s = connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                    ResultSet.CONCUR_READ_ONLY);
            } else {
                System.out.println("creating a TYPE_SCROLL_SENSITIVE cursor");
                s = connection.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                    ResultSet.CONCUR_READ_ONLY);
            }

            ResultSet rs = s.executeQuery("select col1, col2 From cujosql.sensitive,
                                           cujosql.sensitive2 where col1 = col2");

            rs.next();
            System.out.println("value is " + rs.getInt(1));
            rs.next();
```

```
            System.out.println("value is " + rs.getInt(1));
            rs.next();
            System.out.println("value is " + rs.getInt(1));
            rs.next();
            System.out.println("value is " + rs.getInt(1));

            System.out.println("fetched the four rows...");

            // Another statement creates a value that does not fit the where clause.
            Statement s2 =
              connection.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
              ResultSet.CONCUR_UPDATEABLE);
            ResultSet rs2 = s2.executeQuery("select *
            from cujosql.sensitive where col1 = 5 FOR UPDATE");
            rs2.next();
            rs2.updateInt(1, -1);
            rs2.updateRow();
            s2.close();

            if (rs.next()) {
                System.out.println("There is still a row: " + rs.getInt(1));
            } else {
                System.out.println("No more rows.");
            }

        } catch (SQLException e) {
            System.out.println("SQLException exception: ");
            System.out.println("Message:....." + e.getMessage());
            System.out.println("SQLState:...." + e.getSQLState());
            System.out.println("Vendor Code:." + e.getErrorCode());
            System.out.println("----------------------------");
            e.printStackTrace();
        }
        catch (Exception ex) {
            System.out.println("An exception other
            than an SQLException was thrown: ");
            ex.printStackTrace();
        }
    }



    public void cleanup() {
        try {
            connection.close();
        } catch (Exception e) {
            System.out.println("Caught exception: ");
            e.printStackTrace();
        }
    }
}
```

Cursor movement

The IBM i Java Database Connectivity (JDBC) drivers support scrollable ResultSets. With a scrollable ResultSet, you can process rows of data in any order using a number of cursor-positioning methods.

The ResultSet.next method is used to move through a ResultSet one row at a time. With Java Database Connectivity (JDBC) 2.0, the IBM i JDBC drivers support scrollable ResultSets. Scrollable ResultSets allow processing the rows of data in any order by using the previous, absolute, relative, first, and last methods.

By default, JDBC ResultSets are always forward only, meaning that the only valid cursor-positioning method to call is next(). You have to explicitly request a scrollable ResultSet. See ResultSet types for more information.

With a scrollable ResultSet, you can use the following cursor-positioning methods:

Method	**Description**
Next	This method moves the cursor forward one row in the ResultSet. The method returns true if the cursor is positioned on a valid row and false otherwise.
Previous	The method moves the cursor backward one row in the ResultSet. The method returns true if the cursor is positioned on a valid row and false otherwise.

Method	Description
First	The method moves the cursor to the first row in the ResultSet. The method returns true if the cursor is positioned on the first row and false if the ResultSet is empty.
Last	The method moves the cursor to the last row in the ResultSet. The method returns true if the cursor is positioned on the last row and false if the ResultSet is empty.
BeforeFirst	The method moves the cursor immediately before the first row in the ResultSet. For an empty ResultSet, this method has no effect. There is no return value from this method.
AfterLast	The method moves the cursor immediately after the last row in the ResultSet. For an empty ResultSet, this method has no effect. There is no return value from this method.
Relative (int rows)	The method moves the cursor relative to its current position. • If rows is 0, this method has no effect. • If rows is positive, the cursor is moved forward that many rows. If there are fewer rows between the current position and the end of the ResultSet than specified by the input parameters, this method operates like afterLast. • If rows is negative, the cursor is moved backward that many rows. If there are fewer rows between the current position and the end of the ResultSet than specified by the input parameter, this method operates like beforeFirst. The method returns true if the cursor in positioned on a valid row and false otherwise.
Absolute (int row)	The method moves the cursor to the row specified by row value. If row value is positive, the cursor is positioned that many rows from the beginning of the ResultSet. The first row is numbered 1, the second is 2, and so on. If there are fewer rows in the ResultSet than specified by the row value, this method operates the same way as afterLast. If row value is negative, the cursor is positioned that many rows from the end of the ResultSet. The last row is numbered -1, the second to last is -2, and so on. If there are fewer rows in the ResultSet than specified by the row value, this method operates the same way beforeFirst. If row value is 0, this method operates the same way as beforeFirst. The method returns true if the cursor is positioned on a valid row and false otherwise.

Related concepts

ResultSet characteristics
This topic discusses ResultSet characteristics such ResultSet types, concurrency, ability to close the ResultSet by committing the connection object, and specification of ResultSet characteristics.

Retrieving ResultSet data
The ResultSet object provides several methods for obtaining column data for a row. All are of the form get<*Type*>, where <*Type*> is a Java data type. Some examples of these methods include getInt, getLong, getString, getTimestamp, and getBlob. Nearly all of these methods take a single parameter that is either the column index within the ResultSet or the column name.

Creating ResultSets

To create a ResultSet object, you can use executeQuery methods, or other methods. This topic describes options for creating ResultSets.

Related tasks

Changing ResultSets
With the IBM i JDBC drivers, you can change ResultSets by performing several tasks.

Related reference

Example: ResultSet interface
This is an example of how to use the ResultSet interface.

Retrieving ResultSet data

The ResultSet object provides several methods for obtaining column data for a row. All are of the form get<*Type*>, where <*Type*> is a Java data type. Some examples of these methods include getInt, getLong, getString, getTimestamp, and getBlob. Nearly all of these methods take a single parameter that is either the column index within the ResultSet or the column name.

ResultSet columns are numbered, starting with 1. If the column name is used and there is more than one column in the ResultSet with the same name, the first one is returned. There are some get<Type> methods that have additional parameters, such as the optional Calendar object, which can be passed to getTime, getDate, and getTimestamp. Refer to the Javadoc for the java.sql package for full details.

For get methods that return objects, the return value is null when the column in the ResultSet is null. For primitive types, null cannot be returned. In these cases, the value is 0 or false. If an application must distinguish between null, and 0 or false, the wasNull method can be used immediately after the call. This method can then determine whether the value was an actual 0 or false value, or if that value was returned because the ResultSet value was indeed null.

ResultSetMetaData support

When the getMetaData method is called on a ResultSet object, the method returns a ResultSetMetaData object describing the columns of that ResultSet object. When the SQL statement being processed is unknown until runtime, the ResultSetMetaData can be used to determine what get methods should be used to retrieve the data. The following code example uses ResultSetMetaData to determine each column type in the result set:

```
ResultSet rs = stmt.executeQuery(sqlString);
ResultSetMetaData rsmd = rs.getMetaData();
int colType [] = new int[rsmd.getColumnCount()];
for (int idx = 0, int col = 1; idx < colType.length; idx++, col++)
colType[idx] = rsmd.getColumnType(col);
```

Related concepts

ResultSet characteristics
This topic discusses ResultSet characteristics such ResultSet types, concurrency, ability to close the ResultSet by committing the connection object, and specification of ResultSet characteristics.

Cursor movement
The IBM i Java Database Connectivity (JDBC) drivers support scrollable ResultSets. With a scrollable ResultSet, you can process rows of data in any order using a number of cursor-positioning methods.

Creating ResultSets
To create a ResultSet object, you can use executeQuery methods, or other methods. This topic describes options for creating ResultSets.

Related tasks

Changing ResultSets
With the IBM i JDBC drivers, you can change ResultSets by performing several tasks.

Related reference

Example: ResultSet interface

This is an example of how to use the ResultSet interface.

Example: ResultSetMetaData interface

This program demonstrates using a ResultSetMetaData and a ResultSet to display all the metadata about a ResultSet created querying a table. The user passes the value for the table and the library.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
import java.sql.*;

/**
ResultSetMetaDataExample.java

This program demonstrates using a ResultSetMetaData and
a ResultSet to display all the metadata about a ResultSet
created querying a table.  The user passes the value for the
table and library.
**/
public class ResultSetMetaDataExample {

    public static void main(java.lang.String[] args)
    {
        if (args.length != 2) {
            System.out.println("Usage:  java ResultSetMetaDataExample <library> <table>");
            System.out.println("where <library> is the library that contains <table>");
            System.exit(0);
        }

        Connection con = null;
        Statement s = null;
        ResultSet rs = null;
        ResultSetMetaData rsmd = null;

        try {
            // Get a database connection and prepare a statement.
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
            con = DriverManager.getConnection("jdbc:db2:*local");

            s = con.createStatement();

            rs = s.executeQuery("SELECT * FROM " + args[0] + "." + args[1]);
            rsmd = rs.getMetaData();

            int colCount = rsmd.getColumnCount();
            int rowCount = 0;
        for (int i = 1; i <= colCount; i++) {
                System.out.println("Information about column " + i);
                System.out.println("   Name.........: " + rsmd.getColumnName(i));
                System.out.println("   Data Type.....: " + rsmd.getColumnType(i) +
                                   " ( " + rsmd.getColumnTypeName(i) + " )");
                System.out.println("   Precision.....: " + rsmd.getPrecision(i));
                System.out.println("   Scale.........: " + rsmd.getScale(i));
                System.out.print  ("   Allows Nulls..: ");
                if (rsmd.isNullable(i)==0)
                    System.out.println("false");
                else
                    System.out.println("true");
            }

        } catch (Exception e) {
            // Handle any errors.
            System.out.println("Oops... we have an error... ");
            e.printStackTrace();
        } finally {
            // Ensure we always clean up.  If the connection gets closed, the
            // statement under it closes as well.
            if (con != null) {
                try {
                    con.close();
                } catch (SQLException e) {
                    System.out.println("Critical error - cannot close connection object");
                }
            }
        }
    }
}
```

Changing ResultSets

With the IBM i JDBC drivers, you can change ResultSets by performing several tasks.

The default setting for ResultSets is read only. However, with Java Database Connectivity (JDBC) 2.0, the IBM i JDBC drivers provide complete support for updateable ResultSets.

You can refer to "ResultSet characteristics" on page 100 on how to update ResultSets.

Update rows

Rows may be updated in a database table through the ResultSet interface. The steps involved in this process are the following:

1. Change the values for a specific row using various update<*Type*> methods, where <*Type*> is a Java data type. These update<*Type*> methods correspond to the get<*Type*> methods available for retrieving values.
2. Apply the rows to the underlying database.

The database itself is not updated until the second step. Updating columns in a ResultSet without calling the updateRow method does not make any changes to the database.

Planned updates to a row can be thrown away with the cancelUpdates method. Once the updateRow method is called, changes to the database are final and cannot be undone.

Note: The rowUpdated method always returns false as the database does not have a way to point out which rows have been updated. Correspondingly, the updatesAreDetected method returns false.

Delete rows

Rows may be deleted in a database table through the ResultSet interface. The deleteRow method is provided and deletes the current row.

Insert rows

Rows may be inserted into a database table through the ResultSet interface. This process makes use of an "insert row" which applications specifically move the cursor to and build the values they want to insert into the database. The steps involved in this process are as follows:

1. Position the cursor on the insert row.
2. Set each of the values for the columns in the new row.
3. Insert the row into the database and optionally move the cursor back to the current row within the ResultSet.

Note: New rows are not inserted into the table where the cursor is positioned. They are typically added to the end of the table data space. A relational database is not position-dependent by default. For example, you should not expect to move the cursor to the third row and insert something that shows up before the forth row when subsequent users fetch the data.

Support for positioned updates

Besides the method for updating the database through a ResultSet, SQL statements can be used to issue positioned updates. This support relies on using named cursors. JDBC provides the setCursorName method from Statement and the getCursorName method from ResultSet to provide access to these values.

Two DatabaseMetaData methods, supportsPositionedUpdated and supportsPositionedDelete, both return true as this feature is supported with the native JDBC driver.

Related concepts

ResultSet characteristics
This topic discusses ResultSet characteristics such ResultSet types, concurrency, ability to close the ResultSet by committing the connection object, and specification of ResultSet characteristics.

Cursor movement

The IBM i Java Database Connectivity (JDBC) drivers support scrollable ResultSets. With a scrollable ResultSet, you can process rows of data in any order using a number of cursor-positioning methods.

Retrieving ResultSet data
The ResultSet object provides several methods for obtaining column data for a row. All are of the form get<*Type*>, where <*Type*> is a Java data type. Some examples of these methods include getInt, getLong, getString, getTimestamp, and getBlob. Nearly all of these methods take a single parameter that is either the column index within the ResultSet or the column name.

Creating ResultSets
To create a ResultSet object, you can use executeQuery methods, or other methods. This topic describes options for creating ResultSets.

Related reference

Example: ResultSet interface
This is an example of how to use the ResultSet interface.

Example: Removing values from a table through another statement's cursor
This Java example shows how to remove values from a table through another statement's cursor.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
import java.sql.*;

public class UsingPositionedDelete {
    public Connection connection = null;
    public static void main(java.lang.String[] args) {
        UsingPositionedDelete test = new UsingPositionedDelete();

        test.setup();
        test.displayTable();

        test.run();
        test.displayTable();

        test.cleanup();
    }



/**
Handle all the required setup work.
**/
    public void setup() {
        try {
            // Register the JDBC driver.
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");

            connection = DriverManager.getConnection("jdbc:db2:*local");

            Statement s = connection.createStatement();
            try {
                s.executeUpdate("DROP TABLE CUJOSQL.WHERECUREX");
            } catch (SQLException e) {
                // Ignore problems here.
            }

            s.executeUpdate("CREATE TABLE CUJOSQL.WHERECUREX ( " +
                            "COL_IND INT, COL_VALUE CHAR(20)) ");

            for (int i = 1; i <= 10; i++) {
                s.executeUpdate("INSERT INTO CUJOSQL.WHERECUREX VALUES(" + i + ", 'FIRST')");
            }

            s.close();

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


/**
```

```
In this section, all the code to perform the testing should
be added. If only one connection to the database is needed,
the global variable 'connection' can be used.
**/
    public void run() {
        try {
            Statement stmt1 = connection.createStatement();

            // Update each value using next().
            stmt1.setCursorName("CUJO");
            ResultSet rs = stmt1.executeQuery ("SELECT * FROM CUJOSQL.WHERECUREX " +
                                               "FOR UPDATE OF COL_VALUE");

            System.out.println("Cursor name is " + rs.getCursorName());

            PreparedStatement stmt2 = connection.prepareStatement
                                      ("DELETE FROM " + " CUJOSQL.WHERECUREX WHERE CURRENT OF "
+
                                      rs.getCursorName ());


            // Loop through the ResultSet and update every other entry.
            while (rs.next ()) {
                if (rs.next())
                    stmt2.execute ();
            }


            // Clean up the resources after they have been used.
            rs.close ();
            stmt2.close ();


        } catch (Exception e) {
            System.out.println("Caught exception: ");
            e.printStackTrace();
        }
    }



/**
In this section, put all clean-up work for testing.
**/
    public void cleanup() {
        try {
            // Close the global connection opened in setup().
            connection.close();

        } catch (Exception e) {
            System.out.println("Caught exception: ");
            e.printStackTrace();
        }
    }



/**
Display the contents of the table.
**/
    public void displayTable()
    {
        try {
            Statement s = connection.createStatement();
            ResultSet rs = s.executeQuery ("SELECT * FROM CUJOSQL.WHERECUREX");

            while (rs.next ()) {
                System.out.println("Index " + rs.getInt(1) + " value " + rs.getString(2));
            }

            rs.close ();
            s.close();
            System.out.println("-----------------------------------------");
        } catch (Exception e) {
            System.out.println("Caught exception: ");
            e.printStackTrace();
        }
    }
}
```

Example: Changing values with a statement through another statement's cursor
This Java example shows how to change values with a statement through another statement's cursor.

Note: By using the code examples, you agree to the terms of the “Code license and disclaimer information” on page 489.

```
import java.sql.*;

public class UsingPositionedUpdate {
    public Connection connection = null;
    public static void main(java.lang.String[] args) {

        UsingPositionedUpdate test = new UsingPositionedUpdate();

        test.setup();
        test.displayTable();

        test.run();
        test.displayTable();

        test.cleanup();
    }



/**
Handle all the required setup work.
**/
    public void setup() {
        try {
            // Register the JDBC driver.
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");

            connection = DriverManager.getConnection("jdbc:db2:*local");

            Statement s = connection.createStatement();
            try {
                s.executeUpdate("DROP TABLE CUJOSQL.WHERECUREX");
            } catch (SQLException e) {
                // Ignore problems here.
            }

            s.executeUpdate("CREATE TABLE CUJOSQL.WHERECUREX ( " +
                            "COL_IND INT, COL_VALUE CHAR(20)) ");

            for (int i = 1; i <= 10; i++) {
                s.executeUpdate("INSERT INTO CUJOSQL.WHERECUREX VALUES(" + i + ", 'FIRST')");
            }

            s.close();

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


/**
In this section, all the code to perform the testing should
be added. If only one connection to the database is required,
the global variable 'connection' can be used.
**/
    public void run() {
        try {
            Statement stmt1 = connection.createStatement();

            // Update each value using next().
            stmt1.setCursorName("CUJO");
            ResultSet rs = stmt1.executeQuery ("SELECT * FROM CUJOSQL.WHERECUREX " +
                                               "FOR UPDATE OF COL_VALUE");

            System.out.println("Cursor name is " + rs.getCursorName());

            PreparedStatement stmt2 = connection.prepareStatement ("UPDATE "
                                                                   + " CUJOSQL.WHERECUREX
                                                                       SET COL_VALUE =
'CHANGED'
                                                                       WHERE CURRENT OF "
                                                                   + rs.getCursorName ());
```

```
            // Loop through the ResultSet and update every other entry.
            while (rs.next ()) {
                if (rs.next())
                    stmt2.execute ();
            }


            // Clean up the resources after they have been used.
            rs.close ();
            stmt2.close ();


        } catch (Exception e) {
            System.out.println("Caught exception: ");
            e.printStackTrace();
        }
    }



/**
In this section, put all clean-up work for testing.
**/
    public void cleanup() {
        try {
            // Close the global connection opened in setup().
            connection.close();

        } catch (Exception e) {
            System.out.println("Caught exception: ");
            e.printStackTrace();
        }
    }



/**
Display the contents of the table.
**/
    public void displayTable()
    {
        try {
            Statement s = connection.createStatement();
            ResultSet rs = s.executeQuery ("SELECT * FROM CUJOSQL.WHERECUREX");

            while (rs.next ()) {
                System.out.println("Index " + rs.getInt(1) + " value " + rs.getString(2));
            }

            rs.close ();
            s.close();
            System.out.println("-----------------------------------------");
        } catch (Exception e) {
            System.out.println("Caught exception: ");
            e.printStackTrace();
        }
    }
}
```

Creating ResultSets

To create a ResultSet object, you can use executeQuery methods, or other methods. This topic describes options for creating ResultSets.

These methods are from the Statement, PreparedStatement, or CallableStatement interfaces. There are, however, other available methods. For example, DatabaseMetaData methods such as getColumns, getTables, getUDTs, getPrimaryKeys, and so on, return ResultSets. It is also possible to have a single SQL statement return multiple ResultSets for processing. You can also use the getResultSet method to retrieve a ResultSet object after calling the execute method provided by the Statement, PreparedStatement, or CallableStatement interfaces.

See "Example: Creating a procedure with multiple ResultSets" on page 96 for more information.

Close ResultSets

While a ResultSet object is automatically closed when the Statement object with which it is associated closes, it is recommended that you close ResultSet objects when you are finished using them. By doing so, you immediately free internal database resources that can increase application throughput.

It is also important to close ResultSets generated by DatabaseMetaData calls. Because you do not directly have access to the Statement object that was used to create these ResultSets, you do not call close on the Statement object directly. These objects are linked together in such a way that the JDBC driver closes the internal Statement object when you close the external ResultSet object. When these objects are not closed manually, the system continues to work; however, it uses more resources than is necessary.

Note: The holdability characteristic of ResultSets can also close ResultSets automatically on you behalf. Calling close multiple times on a ResultSet object is allowed.

Related concepts

ResultSet characteristics
This topic discusses ResultSet characteristics such ResultSet types, concurrency, ability to close the ResultSet by committing the connection object, and specification of ResultSet characteristics.

Cursor movement
The IBM i Java Database Connectivity (JDBC) drivers support scrollable ResultSets. With a scrollable ResultSet, you can process rows of data in any order using a number of cursor-positioning methods.

Retrieving ResultSet data
The ResultSet object provides several methods for obtaining column data for a row. All are of the form get<*Type*>, where <*Type*> is a Java data type. Some examples of these methods include getInt, getLong, getString, getTimestamp, and getBlob. Nearly all of these methods take a single parameter that is either the column index within the ResultSet or the column name.

Related tasks

Changing ResultSets
With the IBM i JDBC drivers, you can change ResultSets by performing several tasks.

Related reference

Example: ResultSet interface
This is an example of how to use the ResultSet interface.

“Statement objects” on page 84
A Statement object is used for processing a static SQL statement and obtaining the results produced by it. Only one ResultSet for each Statement object can be open at a time. All statement methods that process an SQL statement implicitly close a statement's current ResultSet if an open one exists.

“PreparedStatements” on page 87
PreparedStatements extend the Statement interface and provide support for adding parameters to SQL statements.

“CallableStatements” on page 93
The JDBC CallableStatement interface extends PreparedStatement and provides support for output and input/output parameters. The CallableStatement interface also has support for input parameters that is provided by the PreparedStatement interface.

“DatabaseMetaData interface” on page 53
The DatabaseMetaData interface is implemented by the IBM Developer Kit for Java JDBC driver to provide information about its underlying data sources. It is used primarily by application servers and tools to determine how to interact with a given data source. Applications may also use DatabaseMetaData methods to obtain information about a data source, but this is less typical.

Example: ResultSet interface

This is an example of how to use the ResultSet interface.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
import java.sql.*;

/**
ResultSetExample.java

This program demonstrates using a ResultSetMetaData and
a ResultSet to display all the data in a table even though
the program that gets the data does not know what the table
is going to look like (the user passes in the values for the
table and library).
**/
public class ResultSetExample {

    public static void main(java.lang.String[] args)
    {
        if (args.length != 2) {
            System.out.println("Usage:  java ResultSetExample <library> <table>");
            System.out.println(" where <library> is the library that contains <table>");
            System.exit(0);
        }

        Connection con = null;
        Statement s = null;
        ResultSet rs = null;
        ResultSetMetaData rsmd = null;

        try {
            // Get a database connection and prepare a statement.
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
            con = DriverManager.getConnection("jdbc:db2:*local");

            s = con.createStatement();

            rs = s.executeQuery("SELECT * FROM " + args[0] + "." + args[1]);
            rsmd = rs.getMetaData();

            int colCount = rsmd.getColumnCount();
            int rowCount = 0;
            while (rs.next()) {
                rowCount++;
                System.out.println("Data for row " + rowCount);
                for (int i = 1; i <= colCount; i++)
                    System.out.println("   Row " + i + ": " + rs.getString(i));
            }

        } catch (Exception e) {
            // Handle any errors.
            System.out.println("Oops... we have an error... ");
            e.printStackTrace();
        } finally {
            // Ensure we always clean up.  If the connection gets closed, the
            // statement under it closes as well.
            if (con != null) {
                try {
                    con.close();
                } catch (SQLException e) {
                    System.out.println("Critical error - cannot close connection object");
                }
            }
        }
    }
}
```

Related concepts

ResultSet characteristics

This topic discusses ResultSet characteristics such ResultSet types, concurrency, ability to close the ResultSet by committing the connection object, and specification of ResultSet characteristics.

Cursor movement

The IBM i Java Database Connectivity (JDBC) drivers support scrollable ResultSets. With a scrollable ResultSet, you can process rows of data in any order using a number of cursor-positioning methods.

Retrieving ResultSet data
The ResultSet object provides several methods for obtaining column data for a row. All are of the form get<*Type*>, where <*Type*> is a Java data type. Some examples of these methods include getInt, getLong, getString, getTimestamp, and getBlob. Nearly all of these methods take a single parameter that is either the column index within the ResultSet or the column name.

Creating ResultSets
To create a ResultSet object, you can use executeQuery methods, or other methods. This topic describes options for creating ResultSets.

Related tasks

Changing ResultSets
With the IBM i JDBC drivers, you can change ResultSets by performing several tasks.

JDBC object pooling

Object pooling is an important consideration for Java Database Connectivity (JDBC) and performance. Since many objects used in JDBC are expensive to create such as Connection, Statement, and ResultSet objects, significant performance benefits can be achieved by reusing these objects instead of creating every time you need them.

Many applications already handle object pooling on your behalf. For example, WebSphere has extensive support for pooling JDBC objects and allows you to control how the pool is managed. Because of this, you can get the functionality you want without being concerned about your own pooling mechanisms. However, when the support is not provided, you must find a solution for all but trivial applications.

Using DataSource support for object pooling

You can use DataSources to have multiple applications share a common configuration for accessing a database. This is accomplished by having each application reference the same DataSource name.

By using DataSources, many applications can be changed from a central location. For example, if you change the name of a default library used by all your applications and you have used a single DataSource to obtain connections for all of them, you can update the name of the collection in that DataSource. All of your applications then start using the new default library.

When using DataSources to obtain connections for an application, you can use the native JDBC driver's built-in support for connection pooling. This support is provided as an implementation of the ConnectionPoolDataSource interface.

Pooling is accomplished by handing out "logical" Connection objects instead of physical Connection objects. A **logical Connection object** is a connection object that is returned by a pooled Connection object. Each logical connection object acts as a temporary handle to the physical connection represented by the pooled connection object. To the application, when the Connection object is returned, there is no noticeable difference between the two. The subtle difference comes when you call the close method on the Connection object. This call invalidates the logical connection and returns the physical connection to the pool where another application is able to use the physical connection. This technique lets many logical connection objects reuse a single physical connection.

Set up connection pooling

Connection pooling is accomplished by creating a DataSource object that references a ConnectionPoolDataSource object. ConnectionPoolDataSource objects have properties that can be set for handling various aspects of pool maintenance.

Refer to the example on how to set up connection pooling with UDBDataSource and UDBConnectionPoolDataSource more details. You can also see the Java Naming and Directory Interface (JNDI) for details about the role JNDI plays in this example.

From the example, the link that binds the two DataSource objects together is the dataSourceName. The link tells the DataSource object to defer establishing connections to the ConnectionPoolDataSource object that manages pooling automatically.

Pooling and non-pooling applications

There is no difference between an application that uses Connection pooling and one that does not. Therefore, pooling support can be added after the application code is complete, without making any changes to the application code.

The following is output from running the previous program locally during development.

Start timing the non-pooling DataSource version... Time spent: 6410

Start timing the pooling version... Time spent: 282

Java program completed.

By default, a UDBConnectionPoolDataSource pools a single connection. If an application needs a connection several times and only needs one connection at a time, using UDBConnectionPoolDataSource is a perfect solution. If you need many simultaneous connections, you must configure your ConnectionPoolDataSource"ConnectionPoolDataSource properties" on page 119 to match your needs and resources.

Related concepts

DataSource-based statement pooling
The maxStatements property, available on the UDBConnectionPoolDataSource interface, allows for statement pooling within the connection pool. Statement pooling only has an effect on PreparedStatements and CallableStatements. Statement objects are not pooled.

Building your own connection pooling
You can develop your own connection and statement pooling without requiring support for DataSources or relying on another product.

"Java Naming and Directory Interface" on page 488
The Java Naming and Directory Interface (JNDI) is part of the JavaSoft platform application programming interface (API). With JNDI, you can connect seamlessly to multiple naming and directory services. You can build powerful and portable directory-enabled Java applications by using this interface.

Related reference

ConnectionPoolDataSource properties
You can configure the ConnectionPoolDataSource interface by using the set of properties that it provides.

"Example: Setting up connection pooling with UDBDataSource and UDBConnectionPoolDataSource" on page 118
This is an example of how to use connection pooling with UDBDataSource and UDBConnectionPoolDataSource.

"ConnectionPoolDataSource properties" on page 119
You can configure the ConnectionPoolDataSource interface by using the set of properties that it provides.

Example: Setting up connection pooling with UDBDataSource and UDBConnectionPoolDataSource
This is an example of how to use connection pooling with UDBDataSource and UDBConnectionPoolDataSource.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
import java.sql.*;
import javax.naming.*;
import com.ibm.db2.jdbc.app.UDBDataSource;
import com.ibm.db2.jdbc.app.UDBConnectionPoolDataSource;

public class ConnectionPoolingSetup
{
    public static void main(java.lang.String[] args)
```

```
    throws Exception
    {
        // Create a ConnectionPoolDataSource implementation
        UDBConnectionPoolDataSource cpds = new UDBConnectionPoolDataSource();
        cpds.setDescription("Connection Pooling DataSource object");

        // Establish a JNDI context and bind the connection pool data source
        Context ctx = new InitialContext();
        ctx.rebind("ConnectionSupport", cpds);

        // Create a standard data source that references it.
        UDBDataSource ds = new UDBDataSource();
        ds.setDescription("DataSource supporting pooling");
        ds.setDataSourceName("ConnectionSupport");
        ctx.rebind("PoolingDataSource", ds);
  }
}
```

Example: Testing the performance of connection pooling
This is an example of how to test the performance of the pooling example against the performance of the non-pooling example.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
import java.sql.*;
import javax.naming.*;
import java.util.*;
import javax.sql.*;

public class ConnectionPoolingTest
{
    public static void main(java.lang.String[] args)
    throws Exception
    {
        Context ctx = new InitialContext();
        // Do the work without a pool:
        DataSource ds = (DataSource) ctx.lookup("BaseDataSource");
        System.out.println("\nStart timing the non-pooling DataSource version...");

        long startTime = System.currentTimeMillis();
        for (int i = 0; i < 100; i++) {
            Connection c1 = ds.getConnection();
            c1.close();
        }
        long endTime = System.currentTimeMillis();
        System.out.println("Time spent: " + (endTime - startTime));

        // Do the work with pooling:
        ds = (DataSource) ctx.lookup("PoolingDataSource");
        System.out.println("\nStart timing the pooling version...");

        startTime = System.currentTimeMillis();
        for (int i = 0; i < 100; i++) {
            Connection c1 = ds.getConnection();
            c1.close();
        }
        endTime = System.currentTimeMillis();
        System.out.println("Time spent: " + (endTime - startTime));
    }
}
```

ConnectionPoolDataSource properties

You can configure the ConnectionPoolDataSource interface by using the set of properties that it provides.

Descriptions of these properties are provided in the following table.

Property	Description
initialPoolSize	When the pool is first instantiated, this property determines how many connections are placed into the pool. If this value is specified outside the range of minPoolSize and maxPoolSize, either minPoolSize or maxPoolSize is used as the number of initial connections to create.
maxPoolSize	As the pool is used, more connections may be requested than the pool has in it. This property specifies the maximum number of connections allowed to be created in the pool. Applications do not "block" and wait for a connection to be returned to the pool when the pool is at its maximum size and all connections are in use. Instead, the JDBC driver constructs a new connection based on the DataSource properties and returns the connection. If a maxPoolSize of 0 is specified, the pool is allowed to grow unbounded as long as the system has resources available to hand out.
minPoolSize	Spikes in using the pool can cause it to increase the number of connections in it. If the activity level diminishes to the point where some Connections are never pulled out of the pool, the resources are being taken up for no particular reason. In such cases, the JDBC driver has the ability to release some of the connections that it has accumulated. This property allows you to tell the JDBC to release connections, ensuring that it always has a certain number of connections available to use. If a minPoolSize of 0 is specified, it is possible for the pool to free all of its connections and for the application to actually pay for the connection time for each connection request.
maxIdleTime	Connections keep track of how long they have been sitting around without being used. This property specifies how long an application allows connections to be unused before they are released (that is, there are more connections than are needed). This property is a time in seconds and does not specify when the actual close occurs. It specifies when enough time has passed that the connection should be released.
propertyCycle	This property represents the number of seconds that are allowed to pass between the enforcement of these rules.

Note: Setting either the maxIdleTime or the propertyCycle time to 0 means that the JDBC driver does not check for connections to be removed from the pool on its own. The rules specified for initial, min, and max size are still enforced.

When maxIdleTime and propertyCycle are not 0, a management thread is used to watch over the pool. The thread wakes up every propertyCycle second and checks all the connections in the pool to see which ones have been there without being used for more than maxIdleTime seconds. Connections fitting this criterion are removed from the pool until the minPoolSize is reached.

Related concepts

Using DataSource support for object pooling
You can use DataSources to have multiple applications share a common configuration for accessing a database. This is accomplished by having each application reference the same DataSource name.

DataSource-based statement pooling
The maxStatements property, available on the UDBConnectionPoolDataSource interface, allows for statement pooling within the connection pool. Statement pooling only has an effect on PreparedStatements and CallableStatements. Statement objects are not pooled.

Building your own connection pooling
You can develop your own connection and statement pooling without requiring support for DataSources or relying on another product.

DataSource-based statement pooling

The maxStatements property, available on the UDBConnectionPoolDataSource interface, allows for statement pooling within the connection pool. Statement pooling only has an effect on PreparedStatements and CallableStatements. Statement objects are not pooled.

The implementation of statement pooling is similar to that of connection pooling. When the application calls Connection.prepareStatement("select * from tablex"), the pooling module checks if the Statement object has already been prepared under the connection. If it has, a logical PreparedStatement object is handed to you instead of the physical object. When you call close, the Connection object is returned to the pool, the logical Connection object is thrown away, and the Statement object can be reused.

The maxStatements property allows the DataSource to specify how many statements can be pooled under a connection. A value of 0 indicates that statement pooling should not be used. When the statement pool is full, a least recently used algorithm is applied to determine which statement is to be thrown out.

The example below tests one DataSource that uses connection pooling only and the other DataSource that uses statement and connection pooling.

The following example is output from running this program locally during development.

Deploying statement pooling data source Start timing the connection pooling only version... Time spent: 26312

Starting timing the statement pooling version... Time spent: 2292 Java program completed

Related concepts

Using DataSource support for object pooling
You can use DataSources to have multiple applications share a common configuration for accessing a database. This is accomplished by having each application reference the same DataSource name.

Building your own connection pooling
You can develop your own connection and statement pooling without requiring support for DataSources or relying on another product.

Related reference

ConnectionPoolDataSource properties

You can configure the ConnectionPoolDataSource interface by using the set of properties that it provides.

Example: Testing the performance of two DataSources
This is an example of testing one DataSource that uses connection pooling only and another DataSource that uses statement and connection pooling.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
import java.sql.*;
import javax.naming.*;
import java.util.*;
import javax.sql.*;
import com.ibm.db2.jdbc.app.UDBDataSource;
import com.ibm.db2.jdbc.app.UDBConnectionPoolDataSource;

public class StatementPoolingTest
{

    public static void main(java.lang.String[] args)
    throws Exception
    {
        Context ctx = new InitialContext();

        System.out.println("deploying statement pooling data source");
        deployStatementPoolDataSource();

        // Do the work with connection pooling only.
        DataSource ds = (DataSource) ctx.lookup("PoolingDataSource");
        System.out.println("\nStart timing the connection pooling only version...");

        long startTime = System.currentTimeMillis();
        for (int i = 0; i < 100; i++) {
            Connection c1 = ds.getConnection();
            PreparedStatement ps = c1.prepareStatement("select * from qsys2.sysprocs");
            ResultSet rs = ps.executeQuery();
            c1.close();
        }
        long endTime = System.currentTimeMillis();
        System.out.println("Time spent: " + (endTime - startTime));


        // Do the work with statement pooling added.
        ds = (DataSource) ctx.lookup("StatementPoolingDataSource");
        System.out.println("\nStart timing the statement pooling version...");

        startTime = System.currentTimeMillis();
        for (int i = 0; i < 100; i++) {
            Connection c1 = ds.getConnection();
            PreparedStatement ps = c1.prepareStatement("select * from qsys2.sysprocs");
            ResultSet rs = ps.executeQuery();
            c1.close();
        }
        endTime = System.currentTimeMillis();
        System.out.println("Time spent: " + (endTime - startTime));
    }



    private static void deployStatementPoolDataSource()
    throws Exception
    {
        // Create a ConnectionPoolDataSource implementation
        UDBConnectionPoolDataSource cpds = new UDBConnectionPoolDataSource();
        cpds.setDescription("Connection Pooling DataSource object with Statement pooling");
        cpds.setMaxStatements(10);

        // Establish a JNDI context and bind the connection pool data source
        Context ctx = new InitialContext();
        ctx.rebind("StatementSupport", cpds);


        // Create a standard datasource that references it.
        UDBDataSource ds = new UDBDataSource();
        ds.setDescription("DataSource supporting statement pooling");
        ds.setDataSourceName("StatementSupport");
        ctx.rebind("StatementPoolingDataSource", ds);
```

```
    }
}
```

Building your own connection pooling

You can develop your own connection and statement pooling without requiring support for DataSources or relying on another product.

Without connection pooling, there is too much database work for every request. That is, you get a connection, get a statement, process the statement, close the statement, and close the connection. Instead of discarding everything after each request, there is a way to reuse portions of this process. **Connection pooling** is replacing the create connection code with code to obtain a connection from the pool, and then replacing the close connection code with code to return the connection to the pool for use.

The connection pool's constructor creates the connections and places them in the pool. The pool class has take and put methods for locating a connection to use and for returning the connection to the pool when done working with the connection. These methods are synchronized because the pool object is a shared resource, but you do not want multiple threads to simultaneously try to manipulate the pooled resources.

Building your own statement pooling

When using connection pooling, time is wasted when creating and closing a statement when each statement is processed. This is an example of wasting an object that can be reused.

To reuse an object, you can use the prepared statement class. In most applications, the same SQL statements are reused with minor changes. For example, one iteration through an application might generate the following query:

```
SELECT * from employee where salary > 100000
```

The next iteration might generate the following query:

```
SELECT * from employee where salary > 50000
```

This is the same query, but it uses a different parameter. Both queries can be accomplished with the following query:

```
SELECT * from employee where salary > ?
```

You can then set the parameter marker (denoted by the question mark) to 100000 when processing the first query and 50000 when processing the second query. This enhances performance for three reasons beyond what the connection pool can offer:

- Fewer objects are created. A PreparedStatement object is created and reused instead of creating a Statement object for every request. Therefore, you run fewer constructors.
- The database work to set up the SQL statement (called the **prepare**) can be reused. Preparing SQL statements is reasonably expensive as it involves determining what the SQL statement text says and how the system should accomplish the task requested.
- When removing the additional object creations, there is a benefit that is not often considered. There is no need to destroy what was not created. This model is easier on the Java garbage collector and also benefits performance over time with many users.

Considerations

Performance improves through replication. If an item is not reused, then it is wasting resources to pool it.

Most applications contain critical sections of code. Typically, an application uses 80 to 90 percent of its processing time on only 10 to 20 percent of the code. If there are 10,000 SQL statements potentially used in an application, not all of them are pooled. The objective is to identify and pool the SQL statements that are used in the application's critical sections of code.

Creating objects in a Java implementation can carry a heavy cost. The pooling solution can be used with advantage. Objects used in the process are created at the beginning, before other users attempt to use the system. These objects are reused as often as required. Performance is excellent and it is possible to fine-tune the application over time to facilitate its use for greater numbers of users. As a result, more objects are pooled. Moreover, it permits more efficient multithreading of the application's database access to gain greater throughput.

Java (using JDBC) is based on dynamic SQL and tends to be slow. Pooling can minimize this problem. By preparing the statements at startup, access to the database can be rendered static. There is little difference in performance between dynamic and static SQL after the statement is prepared.

The performance of database access in Java can be efficient and can be accomplished without sacrificing object-oriented design or code maintainability. Writing code to build statement and connection pooling is not difficult. Furthermore, the code can be changed and enhanced to support multiple applications and application types (Web-based, client/server) and so on.

Related concepts

Using DataSource support for object pooling
You can use DataSources to have multiple applications share a common configuration for accessing a database. This is accomplished by having each application reference the same DataSource name.

DataSource-based statement pooling
The maxStatements property, available on the UDBConnectionPoolDataSource interface, allows for statement pooling within the connection pool. Statement pooling only has an effect on PreparedStatements and CallableStatements. Statement objects are not pooled.

Related reference

ConnectionPoolDataSource properties
You can configure the ConnectionPoolDataSource interface by using the set of properties that it provides.

Batch updates

Batch update support allows any updates to the database to be passed as a single transaction between the user program and the database. This procedure can significantly improve performance when many updates must be performed at once.

For example, if a large company requires its newly hired employees to start work on a Monday, this requirement makes it necessary to process many updates (in this case, insertions) to the employee database at one time. Creating a batch of updates and submitting them to the database as one unit can save you processing time.

There are two types of batch updates:

- Batch updates that use Statement objects.
- Batch updates that use PreparedStatement objects.

Statement batch update

To perform a Statement batch update, you must turn off auto-commit. In Java Database Connectivity (JDBC), auto-commit is on by default. Auto-commit means any updates to the database are committed after each SQL statement is processed. If you want to treat a group of statements being handed to the database as one functional group, you do not want the database committing each statement individually. If you do not turn off auto-commit and a statement in the middle of the batch fails, you cannot roll back the entire batch and try it again because half of the statements have been made final. Further, the additional work of committing each statement in a batch creates a lot of overhead.

See “JDBC transactions” on page 66 for more details.

After turning off auto-commit, you can create a standard Statement object. Instead of processing statements with methods such as executeUpdate, you add them to the batch with the addBatch method. Once you have added all the statements you want to the batch, you can process all of them with the executeBatch method. You can empty the batch at anytime with the clearBatch method.

The following example shows how you can use these methods:

Example: Statement batch update

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
connection.setAutoCommit(false);
Statement statement = connection.createStatement();
statement.addBatch("INSERT INTO TABLEX VALUES(1, 'Cujo')");
statement.addBatch("INSERT INTO TABLEX VALUES(2, 'Fred')");
statement.addBatch("INSERT INTO TABLEX VALUES(3, 'Mark')");
int [] counts = statement.executeBatch();
connection.commit();
```

In this example, an array of integers is returned from the executeBatch method. This array has one integer value for each statement that is processed in the batch. If values are being inserted into the database, the value for each statement is 1 (that is, assuming successful processing). However, some of the statements may be update statements that affect multiple rows. If you put any statements in the batch other than INSERT, UPDATE, or DELETE, an exception occurs.

Related concepts

JDBC BatchUpdateException
An important consideration of batch updates is what action to take when a call to the executeBatch method fails. In this case, a new type of exception, called BatchUpdateException, is thrown. The BatchUpdateException is a subclass of SQLException and it allows you to call all the same methods you have always called to receive the message, the SQLState, and vendor code.

Blocked inserts with JDBC
You can use a blocked insert operation to insert several rows into a database table at a time.

Related reference

PreparedStatement batch update
A preparedStatement batch is similar to the Statement batch; however, a preparedStatement batch always works off the same prepared statement, and you only change the parameters to that statement.

PreparedStatement batch update

A preparedStatement batch is similar to the Statement batch; however, a preparedStatement batch always works off the same prepared statement, and you only change the parameters to that statement.

The following is an example that uses a preparedStatement batch.

Note: Read the Code example disclaimer for important legal information.

```
connection.setAutoCommit(false);
PreparedStatement statement =
  connection.prepareStatement("INSERT INTO TABLEX VALUES(?, ?)");
statement.setInt(1, 1);
statement.setString(2, "Cujo");
statement.addBatch();
statement.setInt(1, 2);
statement.setString(2, "Fred");
statement.addBatch();
statement.setInt(1, 3);
statement.setString(2, "Mark");
statement.addBatch();
int [] counts = statement.executeBatch();
connection.commit();
```

Related concepts

JDBC BatchUpdateException
An important consideration of batch updates is what action to take when a call to the executeBatch method fails. In this case, a new type of exception, called BatchUpdateException, is thrown. The BatchUpdateException is a subclass of SQLException and it allows you to call all the same methods you have always called to receive the message, the SQLState, and vendor code.

Blocked inserts with JDBC

You can use a blocked insert operation to insert several rows into a database table at a time.

Related reference

Statement batch update
To perform a Statement batch update, you must turn off auto-commit. In Java Database Connectivity (JDBC), auto-commit is on by default. Auto-commit means any updates to the database are committed after each SQL statement is processed. If you want to treat a group of statements being handed to the database as one functional group, you do not want the database committing each statement individually. If you do not turn off auto-commit and a statement in the middle of the batch fails, you cannot roll back the entire batch and try it again because half of the statements have been made final. Further, the additional work of committing each statement in a batch creates a lot of overhead.

JDBC BatchUpdateException

An important consideration of batch updates is what action to take when a call to the executeBatch method fails. In this case, a new type of exception, called BatchUpdateException, is thrown. The BatchUpdateException is a subclass of SQLException and it allows you to call all the same methods you have always called to receive the message, the SQLState, and vendor code.

BatchUpdateException also provides the getUpdateCounts method that returns an integer array. The integer array contains update counts from all the statements in the batch that were processed up to the point where the failure occurred. The array length tells you which statement in the batch failed. For example, if the array returned in the exception has a length of three, the fourth statement in the batch failed. Therefore, from the single BatchUpdateException object that is returned, you can determine the update counts for all the statements that were successful, which statement failed, and all the information about the failure.

The standard performance of processing batched updates is equivalent to the performance of processing each statement independently. You can refer to Blocked insert support for more information about optimized support for batch updates. You should still use the new model when coding and take advantage of future performance optimizations.

Note: In the JDBC 2.1 specification, a different option is provided for how exception conditions for batch updates are handled. JDBC 2.1 introduces a model where the processing batch continues after a batch entry fails. A special update count is placed in the array of update count integers that is returned for each entry that fails. This allows large batches to continue processing even though one of their entries fails. See the JDBC 2.1 or JDBC 3.0 specification for details on these two modes of operation. By default, the native JDBC driver uses the JDBC 2.0 definition. The driver provides a Connection property that is used when using DriverManager to establish connections. The driver also provides a DataSource property that is used when using DataSources to establish connections. These properties allow applications to choose how they want batch operations to handle failures.

Related concepts

Blocked inserts with JDBC
You can use a blocked insert operation to insert several rows into a database table at a time.

Related reference

Statement batch update
To perform a Statement batch update, you must turn off auto-commit. In Java Database Connectivity (JDBC), auto-commit is on by default. Auto-commit means any updates to the database are committed after each SQL statement is processed. If you want to treat a group of statements being handed to the database as one functional group, you do not want the database committing each statement individually. If you do not turn off auto-commit and a statement in the middle of the batch fails, you cannot roll back the entire batch and try it again because half of the statements have been made final. Further, the additional work of committing each statement in a batch creates a lot of overhead.

PreparedStatement batch update

A preparedStatement batch is similar to the Statement batch; however, a preparedStatement batch always works off the same prepared statement, and you only change the parameters to that statement.

Blocked inserts with JDBC

You can use a blocked insert operation to insert several rows into a database table at a time.

A blocked insert is a special type of operation on IBM i that provides a highly optimized way to insert several rows into a database table at a time. Blocked inserts can be thought of as a subset of batched updates. Batched updates can be any form of an update request, but blocked inserts are specific. However, blocked insert types of batched updates are common; the native JDBC driver has been changed to take advantage of this feature.

Because of system restrictions when using blocked insert support, the default setting for the native JDBC driver is to have blocked insert disabled. It can be enabled through a Connection property or a DataSource property. Most of the restrictions when using a blocked insert can be checked and handled on your behalf, but a few restrictions cannot; thus, this is the reason for turning off blocked insert support by default. The list of restrictions is as follows:

- The SQL statement used must be an INSERT statement with a VALUES clause, meaning that it is not an INSERT statement with SUBSELECT. The JDBC driver recognizes this restriction and takes the appropriate course of action.
- A PreparedStatement must be used, meaning that there is no optimized support for Statement objects. The JDBC driver recognizes this restriction and takes the appropriate course of action.
- The SQL statement must specify parameter markers for all the columns in the table. This means that you cannot either use constant values for a column or allow the database to insert default values for any of the columns. The JDBC driver does not have a mechanism to handle testing for specific parameter markers in your SQL statement. If you set the property to perform optimized blocked insertions and you do not avoid defaults or constants in your SQL statements, the values that end up in the database table are not correct.
- The connection must be to the local system. This means that a connection using DRDA to access a remote system cannot be used because DRDA does not support a blocked insert operation. The JDBC driver does not have a mechanism to handle testing for a connection to a local system. If you set the property to perform an optimized blocked insertion and you attempt to connect to a remote system, the processing of the batch update fails.

This code example shows how to enable support for blocked insert processing. The only difference between this code and a version that does not use blocked insert support is `use block insert=true` that is added to the Connection URL.

Example: Blocked insert processing

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
// Create a database connection
Connection c = DriverManager.getConnection("jdbc:db2:*local;use block insert=true");
BigDecimal bd = new BigDecimal("123456");

// Create a PreparedStatement to insert into a table with 4 columns
PreparedStatement ps =
  c.prepareStatement("insert into cujosql.xxx values(?, ?, ?, ?)");

// Start timing...
for (int i = 1; i <= 10000; i++) {
  ps.setInt(1, i);                       // Set all the parameters for a row
  ps.setBigDecimal(2, bd);
  ps.setBigDecimal(3, bd);
  ps.setBigDecimal(4, bd);
  ps.addBatch();                         //Add the parameters to the batch
}

// Process the batch
int[] counts = ps.executeBatch();

// End timing...
```

In similar test cases, a blocked insert is several times faster than performing the same operations when a blocked insert is not used. For example, the test performed on the previous code was nine time faster using blocked inserts. Cases that only use primitive types instead of objects can be up to sixteen times faster. In applications where there is a significant amount of work going on, change your expectations appropriately.

Related concepts

JDBC BatchUpdateException
An important consideration of batch updates is what action to take when a call to the executeBatch method fails. In this case, a new type of exception, called BatchUpdateException, is thrown. The BatchUpdateException is a subclass of SQLException and it allows you to call all the same methods you have always called to receive the message, the SQLState, and vendor code.

Related reference

Statement batch update
To perform a Statement batch update, you must turn off auto-commit. In Java Database Connectivity (JDBC), auto-commit is on by default. Auto-commit means any updates to the database are committed after each SQL statement is processed. If you want to treat a group of statements being handed to the database as one functional group, you do not want the database committing each statement individually. If you do not turn off auto-commit and a statement in the middle of the batch fails, you cannot roll back the entire batch and try it again because half of the statements have been made final. Further, the additional work of committing each statement in a batch creates a lot of overhead.

PreparedStatement batch update
A preparedStatement batch is similar to the Statement batch; however, a preparedStatement batch always works off the same prepared statement, and you only change the parameters to that statement.

Advanced data types

Advanced SQL3 data types give you a tremendous amount of flexibility. They are ideal for storing serialized Java objects, Extensible Markup Language (XML) documents, and multimedia data such as songs, product pictures, employee photographs, and movie clips. Java Database Connectivity (JDBC) 2.0 and higher provide support for working with these data types that are a part of the SQL99 standard.

Distinct types

The distinct type is a user-defined type that is based on a standard database type. For example, you can define a Social Security Number type, SSN, that is a CHAR(9) internally. The following SQL statement creates such a DISTINCT type.

```
CREATE DISTINCT TYPE CUJOSQL.SSN AS CHAR(9)
```

A distinct type always maps to a built-in data type. For more information about how and when to use distinct types in the context of SQL, consult the SQL reference manuals.

To use distinct types in JDBC, you access them the same way that you access an underlying type. The getUDTs method is a new method that allows you to query what distinct types are available on the system. The Example: Distinct types program shows the following:

- The creation of a distinct type.
- The creation of a table that uses it.
- The use of a PreparedStatement to set a distinct type parameter.
- The use of a ResultSet to return a distinct type.
- The use of the metadata Application Programming Interface (API) call to getUDTs to learn about a distinct type.

For more information, see the Example: Distinct types subtopic that shows various commons tasks you can perform by using distinct types.

Large Objects

There are three types of Large Objects (LOBs):

- Binary Large Objects (BLOBs)
- Character Large Objects (CLOBs)
- Double Byte Character Large Objects (DBCLOBs)

DBCLOBs are similar to CLOBs except for their internal storage representation of the character data. Because Java and JDBC externalize all character data as Unicode, there is only support in JDBC for CLOBs. DBCLOBs work interchangeable with the CLOB support from a JDBC perspective.

Binary Large Objects

In many ways, a Binary Large Object (BLOB) column is similar to a CHAR FOR BIT DATA column that can be made large. You can store anything in these columns that can be represented as a stream of nontranslated bytes. Often, BLOB columns are used to store serialized Java objects, pictures, songs, and other binary data.

You can use BLOBs the same way you can use other standard database types. You can pass them to stored procedures, use them in prepared statements, and update them in result sets. The PreparedStatement class has a setBlob method for passing BLOBs to the database, and the ResultSet class adds a getBlob class for retrieving them from the database. A BLOB is represented in a Java program by a BLOB object that is a JDBC interface.

Character Large Objects

Character Large Objects (CLOBs) are the character data complement to BLOBs. Instead of storing data in the database without translation, the data is stored in the database as text and is processed the same way as a CHAR column. As with BLOBs, JDBC 2.0 provides functions for dealing directly with CLOBs. The PreparedStatement interface contains a setClob method and the ResultSet interface contains a getClob method.

Although BLOB and CLOB columns work like CHAR FOR BIT DATA and CHAR columns, this is conceptually how they work from an external user's perspective. Internally, they are different; because of the potentially enormous size of Large Object (LOB) columns, you typically work indirectly with data. For example, when a block of rows is fetched from the database, you do not move a block of LOBs to the ResultSet. You move pointers called LOB locators (that is, four-byte integers) into the ResultSet instead. However, it is not necessary to know about locators when working with LOBs in JDBC.

Datalinks

Datalinks are encapsulated values that contain a logical reference from the database to a file stored outside the database. Datalinks are represented and used from a JDBC perspective in two different ways, depending on whether you are using JDBC 2.0 or earlier, or you are using JDBC 3.0 or later.

Unsupported SQL3 data types

There are other SQL3 data types that have been defined and for which the JDBC API provides support. These are ARRAY, REF, and STRUCT. Presently, IBM i does not support these types. Therefore, the JDBC driver does not provide any form of support for them.

Related reference

"Example: Distinct types" on page 139

This is an example of how to use distinct types.

Writing code that uses BLOBs

There are a number of tasks that can be accomplished with database Binary Large Object (BLOB) columns through the Java Database Connectivity (JDBC) Application Programming Interface (API). The following topics briefly discuss these tasks and include examples on how to accomplish them.

Read BLOBs from the database and insert BLOBs into the database

With the JDBC API, there are ways to get BLOBs out of the database and ways to put BLOBs into the database. However, there is no standardized way to create a Blob object. This is not a problem if your database is already full of BLOBs, but it poses a problem if you want to work with BLOBs from scratch through JDBC. Instead of defining a constructor for the Blob and Clob interfaces of the JDBC API, support is provided for placing BLOBs into the database and getting them out of the database directly as other types. For example, the setBinaryStream method can work with a database column of type Blob. The Example: Blobs topic shows some of the common ways that a BLOB can be put into the database or retrieved from the database.

Work with the Blob object API

BLOBs are defined in JDBC as an interface of which the various drivers provide implementations. This interface has a series of methods that can be used to interact with the Blob object. The Example: Use Blobs shows some of the common tasks that can be performed using this API. Consult the JDBC Javadoc for a complete list of available methods on the Blob object.

Use JDBC 3.0 support to update BLOBs

In JDBC 3.0, there is support for making changes to LOB objects. These changes can be stored into BLOB columns in the database. The Example: Update Blobs topic shows some of the tasks that can be performed with BLOB support in JDBC 3.0.

Related concepts

Writing code that uses CLOBs
There are a number of tasks that can be performed with database CLOB and DBCLOB columns through the Java Database Connectivity (JDBC) Application Programming Interface (API). The following topics briefly discuss these tasks and include examples on how to accomplish them.

Writing code that uses Datalinks
How you work with Datalinks is dependent on what release you are working with. In JDBC 3.0, there is support to work directly with Datalink columns using the getURL and putURL methods.

Related reference

Example: Distinct types
This is an example of how to use distinct types.

"Example: BLOB" on page 130
This is an example of how a BLOB can be put into the database or retrieved from the database.

"Example: Updating BLOBs" on page 132
This is an example of how to update BLOBs in your Java applications.

Example: BLOB

This is an example of how a BLOB can be put into the database or retrieved from the database.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
/////////////////////////////////////////
// PutGetBlobs is an example application
// that shows how to work with the JDBC
// API to obtain and put BLOBs to and from
// database columns.
//
```

```
// The results of running this program
// are that there are two BLOB values
// in a new table. Both are identical
// and contain 500k of random byte
// data.
/////////////////////////////////////////
import java.sql.*;
import java.util.Random;

public class PutGetBlobs {
   public static void main(String[] args)
   throws SQLException
   {
       // Register the native JDBC driver.
       try {
          Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
      } catch (Exception e) {
          System.exit(1);  // Setup error.
      }

      // Establish a Connection and Statement with which to work.
      Connection c = DriverManager.getConnection("jdbc:db2:*local");
      Statement s = c.createStatement();

      // Clean up any previous run of this application.
      try {
          s.executeUpdate("DROP TABLE CUJOSQL.BLOBTABLE");
      } catch (SQLException e) {
          // Ignore it - assume the table did not exist.
      }

      // Create a table with a BLOB column. The default BLOB column
      // size is 1 MB.
      s.executeUpdate("CREATE TABLE CUJOSQL.BLOBTABLE (COL1 BLOB)");

      // Create a PreparedStatement object that allows you to put
      // a new Blob object into the database.
      PreparedStatement ps = c.prepareStatement("INSERT INTO CUJOSQL.BLOBTABLE VALUES(?)");

      // Create a big BLOB value...
      Random random = new Random ();
      byte [] inByteArray = new byte[500000];
      random.nextBytes (inByteArray);

      // Set the PreparedStatement parameter. Note: This is not
      // portable to all JDBC drivers. JDBC drivers do not have
      // support when using setBytes for BLOB columns. This is used to
      // allow you to generate new BLOBs. It also allows JDBC 1.0
      // drivers to work with columns containing BLOB data.
      ps.setBytes(1, inByteArray);

      // Process the statement, inserting the BLOB into the database.
      ps.executeUpdate();

      // Process a query and obtain the BLOB that was just inserted out
      // of the database as a Blob object.
      ResultSet rs = s.executeQuery("SELECT * FROM CUJOSQL.BLOBTABLE");
      rs.next();
      Blob blob = rs.getBlob(1);


      // Put that Blob back into the database through
      // the PreparedStatement.
      ps.setBlob(1, blob);
      ps.execute();

      c.close(); // Connection close also closes stmt and rs.
   }
}
```

Related reference

Example: Updating BLOBs
This is an example of how to update BLOBs in your Java applications.

Example: Using BLOBs

This is an example of how to use BLOBs in your Java applications.

Example: Updating BLOBs
This is an example of how to update BLOBs in your Java applications.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
/////////////////////////////////////////
// UpdateBlobs is an example application
// that shows some of the APIs providing
// support for changing Blob objects
// and reflecting those changes to the
// database.
//
// This program must be run after
// the PutGetBlobs program has completed.
/////////////////////////////////////////
import java.sql.*;

public class UpdateBlobs {
   public static void main(String[] args)
   throws SQLException
   {
       // Register the native JDBC driver.
       try {
          Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
      } catch (Exception e) {
          System.exit(1);  // Setup error.
      }

      Connection c = DriverManager.getConnection("jdbc:db2:*local");
      Statement s = c.createStatement();

      ResultSet rs = s.executeQuery("SELECT * FROM CUJOSQL.BLOBTABLE");

      rs.next();
      Blob blob1 = rs.getBlob(1);
      rs.next();
      Blob blob2 = rs.getBlob(1);


      // Truncate a BLOB.
      blob1.truncate((long) 150000);
      System.out.println("Blob1's new length is " + blob1.length());

      // Update part of the BLOB with a new byte array.
      // The following code obtains the bytes that are at
      // positions 4000-4500 and set them to positions 500-1000.

      // Obtain part of the BLOB as a byte array.
      byte[] bytes = blob1.getBytes(4000L, 4500);

      int bytesWritten = blob2.setBytes(500L, bytes);

      System.out.println("Bytes written is " + bytesWritten);

      // The bytes are now found at position 500 in blob2
      long startInBlob2 = blob2.position(bytes, 1);

      System.out.println("pattern found starting at position " + startInBlob2);

      c.close(); // Connection close also closes stmt and rs.
   }
}
```

Related reference

Example: BLOB
This is an example of how a BLOB can be put into the database or retrieved from the database.

Example: Using BLOBs

This is an example of how to use BLOBs in your Java applications.

Example: Using BLOBs
This is an example of how to use BLOBs in your Java applications.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
/////////////////////////////////////////
// UseBlobs is an example application
// that shows some of the APIs associated
// with Blob objects.
//
// This program must be run after
// the PutGetBlobs program has completed.
/////////////////////////////////////////
import java.sql.*;

public class UseBlobs {
   public static void main(String[] args)
   throws SQLException
   {
       // Register the native JDBC driver.
       try {
          Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
      } catch (Exception e) {
          System.exit(1);  // Setup error.
      }

      Connection c = DriverManager.getConnection("jdbc:db2:*local");
      Statement s = c.createStatement();

      ResultSet rs = s.executeQuery("SELECT * FROM CUJOSQL.BLOBTABLE");

      rs.next();
      Blob blob1 = rs.getBlob(1);
      rs.next();
      Blob blob2 = rs.getBlob(1);


      // Determine the length of a LOB.
      long end = blob1.length();
      System.out.println("Blob1 length is " + blob1.length());

      // When working with LOBs, all indexing that is related to them
      // is 1-based, and is not 0-based like strings and arrays.
      long startingPoint = 450;
      long endingPoint = 500;

      // Obtain part of the BLOB as a byte array.
      byte[] outByteArray = blob1.getBytes(startingPoint, (int)endingPoint);

      // Find where a sub-BLOB or byte array is first found within a
      // BLOB. The setup for this program placed two identical copies of
      // a random BLOB into the database. Thus, the start position of the
      // byte array extracted from blob1 can be found in the starting
      // position in blob2. The exception would be if there were 50
      // identical random bytes in the LOBs previously.
      long startInBlob2 = blob2.position(outByteArray, 1);

      System.out.println("pattern found starting at position " + startInBlob2);

      c.close(); // Connection close closes stmt and rs too.
   }
}
```

Related reference

Example: BLOB
This is an example of how a BLOB can be put into the database or retrieved from the database.

Example: Updating BLOBs

This is an example of how to update BLOBs in your Java applications.

Writing code that uses CLOBs

There are a number of tasks that can be performed with database CLOB and DBCLOB columns through the Java Database Connectivity (JDBC) Application Programming Interface (API). The following topics briefly discuss these tasks and include examples on how to accomplish them.

Read CLOBs from the database and insert CLOBs into the database

With the JDBC API, there are ways to get CLOBs out of the database and ways to put CLOBs into the database. However, there is no standardized way to create a Clob object. This is not a problem if your database is already full of CLOBs, but it poses a problem if you want to work with CLOBs from scratch through JDBC. Instead of defining a constructor for the Blob and Clob interfaces of the JDBC API, support is provided for placing CLOBs into the database and getting them out of the database directly as other types. For example, the setCharacterStream method can work with a database column of type Clob. The Example: CLOB topic shows some of the common ways that a CLOB can be put into the database or retrieved from the database.

Work with the Clob object API

CLOBs are defined in JDBC as an interface of which the various drivers provide implementations. This interface has a series of methods that can be used to interact with the Clob object. The Example: Use Clobs topic shows some of the common tasks that can be performed using this API. Consult the JDBC Javadoc for a complete list of available methods on the Clob object.

Use JDBC 3.0 support to update CLOBs

In JDBC 3.0, there is support for making changes to LOB objects. These changes can be stored into CLOB columns in the database. The Example: Update Clobs topic shows some of the tasks that can be performed with CLOB support in JDBC 3.0.

Related concepts

Writing code that uses BLOBs
There are a number of tasks that can be accomplished with database Binary Large Object (BLOB) columns through the Java Database Connectivity (JDBC) Application Programming Interface (API). The following topics briefly discuss these tasks and include examples on how to accomplish them.

Writing code that uses Datalinks
How you work with Datalinks is dependent on what release you are working with. In JDBC 3.0, there is support to work directly with Datalink columns using the getURL and putURL methods.

Related reference

Example: Distinct types
This is an example of how to use distinct types.

"Example: CLOB" on page 134
This is an example of how a CLOB can be put into the database or retrieved from the database.

"Example: Using CLOBs" on page 137
This is an example of how to use CLOBs in your Java applications.

"Example: Updating CLOBs" on page 136
This is an example of how to update CLOBs in your Java applications.

Example: CLOB

This is an example of how a CLOB can be put into the database or retrieved from the database.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
/////////////////////////////////////////
// PutGetClobs is an example application
```

```
// that shows how to work with the JDBC
// API to obtain and put CLOBs to and from
// database columns.
//
// The results of running this program
// are that there are two CLOB values
// in a new table. Both are identical
// and contain about 500k of repeating
// text data.
//////////////////////////////////////////
import java.sql.*;

public class PutGetClobs {
   public static void main(String[] args)
   throws SQLException
   {
       // Register the native JDBC driver.
       try {
          Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
      } catch (Exception e) {
          System.exit(1);  // Setup error.
      }

      // Establish a Connection and Statement with which to work.
      Connection c = DriverManager.getConnection("jdbc:db2:*local");
      Statement s = c.createStatement();

      // Clean up any previous run of this application.
      try {
          s.executeUpdate("DROP TABLE CUJOSQL.CLOBTABLE");
      } catch (SQLException e) {
          // Ignore it - assume the table did not exist.
      }

      // Create a table with a CLOB column. The default CLOB column
      // size is 1 MB.
      s.executeUpdate("CREATE TABLE CUJOSQL.CLOBTABLE (COL1 CLOB)");

      // Create a PreparedStatement object that allow you to put
      // a new Clob object into the database.
      PreparedStatement ps = c.prepareStatement("INSERT INTO CUJOSQL.CLOBTABLE VALUES(?)");

      // Create a big CLOB value...
      StringBuffer buffer = new StringBuffer(500000);
      while (buffer.length() < 500000) {
          buffer.append("All work and no play makes Cujo a dull boy.");
      }
      String clobValue = buffer.toString();

      // Set the PreparedStatement parameter. This is not
      // portable to all JDBC drivers. JDBC drivers do not have
      // to support setBytes for CLOB columns. This is done to
      // allow you to generate new CLOBs. It also
      // allows JDBC 1.0 drivers a way to work with columns containing
      // Clob data.
      ps.setString(1, clobValue);

      // Process the statement, inserting the clob into the database.
      ps.executeUpdate();

      // Process a query and get the CLOB that was just inserted out of the
      // database as a Clob object.
      ResultSet rs = s.executeQuery("SELECT * FROM CUJOSQL.CLOBTABLE");
      rs.next();
      Clob clob = rs.getClob(1);


      // Put that Clob back into the database through
      // the PreparedStatement.
      ps.setClob(1, clob);
      ps.execute();

      c.close(); // Connection close also closes stmt and rs.
   }
}
```

Related reference

Example: Updating CLOBs

This is an example of how to update CLOBs in your Java applications.

Example: Using CLOBs
This is an example of how to use CLOBs in your Java applications.

Example: Updating CLOBs
This is an example of how to update CLOBs in your Java applications.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
/////////////////////////////////////////
// UpdateClobs is an example application
// that shows some of the APIs providing
// support for changing Clob objects
// and reflecting those changes to the
// database.
//
// This program must be run after
// the PutGetClobs program has completed.
/////////////////////////////////////////
import java.sql.*;

public class UpdateClobs {
   public static void main(String[] args)
   throws SQLException
   {
       // Register the native JDBC driver.
       try {
          Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
      } catch (Exception e) {
          System.exit(1);  // Setup error.
      }

      Connection c = DriverManager.getConnection("jdbc:db2:*local");
      Statement s = c.createStatement();

      ResultSet rs = s.executeQuery("SELECT * FROM CUJOSQL.CLOBTABLE");

      rs.next();
      Clob clob1 = rs.getClob(1);
      rs.next();
      Clob clob2 = rs.getClob(1);


      // Truncate a CLOB.
      clob1.truncate((long) 150000);
      System.out.println("Clob1's new length is " + clob1.length());

      // Update a portion of the CLOB with a new String value.
      String value = "Some new data for once";
      int charsWritten = clob2.setString(500L, value);

      System.out.println("Characters written is " + charsWritten);

      // The bytes can be found at position 500 in clob2
      long startInClob2 = clob2.position(value, 1);

      System.out.println("pattern found starting at position " + startInClob2);

      c.close(); // Connection close also closes stmt and rs.
   }
}
```

Related reference

Example: CLOB
This is an example of how a CLOB can be put into the database or retrieved from the database.

Example: Using CLOBs

This is an example of how to use CLOBs in your Java applications.

Example: Using CLOBs
This is an example of how to use CLOBs in your Java applications.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
/////////////////////////////////////////
// UpdateClobs is an example application
// that shows some of the APIs providing
// support for changing Clob objects
// and reflecting those changes to the
// database.
//
// This program must be run after
// the PutGetClobs program has completed.
/////////////////////////////////////////
import java.sql.*;

public class UseClobs {
   public static void main(String[] args)
   throws SQLException
   {
       // Register the native JDBC driver.
       try {
          Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
      } catch (Exception e) {
          System.exit(1);  // Setup error.
      }

      Connection c = DriverManager.getConnection("jdbc:db2:*local");
      Statement s = c.createStatement();

      ResultSet rs = s.executeQuery("SELECT * FROM CUJOSQL.CLOBTABLE");

      rs.next();
      Clob clob1 = rs.getClob(1);
      rs.next();
      Clob clob2 = rs.getClob(1);


      // Determine the length of a LOB.
      long end = clob1.length();
      System.out.println("Clob1 length is " + clob1.length());

      // When working with LOBs, all indexing that is related to them
      // is 1-based, and not 0-based like strings and arrays.
      long startingPoint = 450;
      long endingPoint = 50;

      // Obtain part of the CLOB as a byte array.
      String outString = clob1.getSubString(startingPoint, (int)endingPoint);
      System.out.println("Clob substring is " + outString);

      // Find where a sub-CLOB or string is first found within a
      // CLOB. The setup for this program placed two identical copies of
      // a repeating CLOB into the database. Thus, the start position of the
      // string extracted from clob1 can be found in the starting
      // position in clob2 if the search begins close to the position where
      // the string starts.
      long startInClob2 = clob2.position(outString, 440);

      System.out.println("pattern found starting at position " + startInClob2);

      c.close(); // Connection close also closes stmt and rs.
   }
}
```

Related reference

Example: CLOB
This is an example of how a CLOB can be put into the database or retrieved from the database.

Example: Updating CLOBs

This is an example of how to update CLOBs in your Java applications.

Writing code that uses Datalinks

How you work with Datalinks is dependent on what release you are working with. In JDBC 3.0, there is support to work directly with Datalink columns using the getURL and putURL methods.

With previous JDBC versions, you had to work with Datalink columns as if they were String columns. Presently, the database does not support automatic conversions between Datalink and character data types. As a result, you need perform some type casting in your SQL statements.

Related concepts

Writing code that uses BLOBs
There are a number of tasks that can be accomplished with database Binary Large Object (BLOB) columns through the Java Database Connectivity (JDBC) Application Programming Interface (API). The following topics briefly discuss these tasks and include examples on how to accomplish them.

Writing code that uses CLOBs
There are a number of tasks that can be performed with database CLOB and DBCLOB columns through the Java Database Connectivity (JDBC) Application Programming Interface (API). The following topics briefly discuss these tasks and include examples on how to accomplish them.

Related reference

Example: Distinct types
This is an example of how to use distinct types.

Example: Datalink
This example application shows how to use the JDBC API to handle datalink database columns.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
/////////////////////////////////////////
// PutGetDatalinks is an example application
// that shows how to use the JDBC
// API to handle datalink database columns.
/////////////////////////////////////////
import java.sql.*;
import java.net.URL;
import java.net.MalformedURLException;

public class PutGetDatalinks {
    public static void main(String[] args)
    throws SQLException
    {
        // Register the native JDBC driver.
        try {
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
        } catch (Exception e) {
            System.exit(1);  // Setup error.
        }

        // Establish a Connection and Statement with which to work.
        Connection c = DriverManager.getConnection("jdbc:db2:*local");
        Statement s = c.createStatement();

        // Clean up any previous run of this application.
        try {
            s.executeUpdate("DROP TABLE CUJOSQL.DLTABLE");
        } catch (SQLException e) {
            // Ignore it - assume the table did not exist.
        }

        // Create a table with a datalink column.
        s.executeUpdate("CREATE TABLE CUJOSQL.DLTABLE (COL1 DATALINK)");

        // Create a PreparedStatement object that allows you to add
        // a new datalink into the database. Since conversing
        // to a datalink cannot be accomplished directly in the database, you
        // can code the SQL statement to perform the explicit conversion.
        PreparedStatement ps = c.prepareStatement("INSERT INTO CUJOSQL.DLTABLE
                                                   VALUES(DLVALUE( CAST(? AS VARCHAR(100))))");
```

```
        // Set the datalink. This URL points you to a topic about
        // the new features of JDBC 3.0.
        ps.setString (1, "http://www.ibm.com/developerworks/java/library/j-jdbcnew/index.html");

        // Process the statement, inserting the CLOB into the database.
        ps.executeUpdate();

        // Process a query and obtain the CLOB that was just inserted out of the
        // database as a Clob object.
        ResultSet rs = s.executeQuery("SELECT * FROM CUJOSQL.DLTABLE");
        rs.next();
        String datalink = rs.getString(1);


        // Put that datalink value into the database through
        // the PreparedStatement. Note: This function requires JDBC 3.0
        // support.
        /*
        try {
            URL url = new URL(datalink);
            ps.setURL(1, url);
            ps.execute();
        } catch (MalformedURLException mue) {
            // Handle this issue here.
        }

        rs = s.executeQuery("SELECT * FROM CUJOSQL.DLTABLE");
        rs.next();
        URL url = rs.getURL(1);
        System.out.println("URL value is " + url);
        */

        c.close(); // Connection close also closes stmt and rs.
    }
}
```

Example: Distinct types

This is an example of how to use distinct types.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
/////////////////////////////////////////
// This example program shows examples of
// various common tasks that can be done
// with distinct types.
/////////////////////////////////////////
import java.sql.*;

public class Distinct {
    public static void main(String[] args)
    throws SQLException
    {
        // Register the native JDBC driver.
        try {
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
        } catch (Exception e) {
            System.exit(1);  // Setup error.
        }

        Connection c = DriverManager.getConnection("jdbc:db2:*local");
        Statement s = c.createStatement();

        // Clean up any old runs.
        try {
            s.executeUpdate("DROP TABLE CUJOSQL.SERIALNOS");
        } catch (SQLException e) {
            // Ignore it and assume the table did not exist.
        }

        try {
            s.executeUpdate("DROP DISTINCT TYPE CUJOSQL.SSN");
        } catch (SQLException e) {
            // Ignore it and assume the table did not exist.
        }

        // Create the type, create the table, and insert a value.
        s.executeUpdate("CREATE DISTINCT TYPE CUJOSQL.SSN AS CHAR(9)");
        s.executeUpdate("CREATE TABLE CUJOSQL.SERIALNOS (COL1 CUJOSQL.SSN)");
```

```
        PreparedStatement ps = c.prepareStatement("INSERT INTO CUJOSQL.SERIALNOS VALUES(?)");
        ps.setString(1, "399924563");
        ps.executeUpdate();
        ps.close();

        // You can obtain details about the types available with new metadata in
        // JDBC 2.0
        DatabaseMetaData dmd = c.getMetaData();

        int types[] = new int[1];
        types[0] = java.sql.Types.DISTINCT;

        ResultSet rs = dmd.getUDTs(null, "CUJOSQL", "SSN", types);
        rs.next();
        System.out.println("Type name " + rs.getString(3) +
                           " has type " + rs.getString(4));


        // Access the data you have inserted.
        rs = s.executeQuery("SELECT COL1 FROM CUJOSQL.SERIALNOS");
        rs.next();
        System.out.println("The SSN is " + rs.getString(1));

        c.close(); // Connection close also closes stmt and rs.
    }
}
```

Related concepts

Writing code that uses BLOBs
There are a number of tasks that can be accomplished with database Binary Large Object (BLOB) columns through the Java Database Connectivity (JDBC) Application Programming Interface (API). The following topics briefly discuss these tasks and include examples on how to accomplish them.

Writing code that uses CLOBs
There are a number of tasks that can be performed with database CLOB and DBCLOB columns through the Java Database Connectivity (JDBC) Application Programming Interface (API). The following topics briefly discuss these tasks and include examples on how to accomplish them.

Writing code that uses Datalinks
How you work with Datalinks is dependent on what release you are working with. In JDBC 3.0, there is support to work directly with Datalink columns using the getURL and putURL methods.

JDBC RowSets

RowSets were originally added to the Java Database Connectivity (JDBC) 2.0 Optional Package. Unlike some of the better-known interfaces of the JDBC specification, the RowSet specification is designed to be more of a framework than an actual implementation. The RowSet interfaces define a set of core functionality that all RowSets have. RowSet implementation providers have considerable freedom to define the functionality that is needed to fit their needs in a specific problem space.

RowSet characteristics

You can request certain properties to be satisfied by the rowsets. Common properties include the set of interfaces to be supported by the resulting rowset.

RowSets are ResultSets

The RowSet interface extends the ResultSet interface which means that RowSets have the ability to perform all the functions that ResultSets can do. For example, RowSets can be scrollable and updateable.

RowSets can be disconnected from the database

There are two categories of RowSets:

Connected
: While connected RowSets are populated with data, they always have internal connections to the underlying database open and serve as wrappers around a ResultSet implementation.

Disconnected
Disconnected RowSets are not required to maintain connections to their data source at all times. Disconnected RowSets can be detached from the database, be used in a variety of ways, and then be reconnected to the database to mirror any changes made to them.

RowSets are JavaBeans components

RowSets have support for event handling based on the JavaBeans event-handling model. They also have properties that can be set. These properties can be used by the RowSet to perform the following:

- Establish a connection to the database.
- Process an SQL statement.
- Determine features of the data that the RowSet represents and handle other internal features of the RowSet object.

RowSets are serializable

RowSets can be serialized and deserialized to allow them to flow over a network connection, be written out to a flat file (that is, a text document without any word processing or other structure characters), and so on.

Related concepts

DB2CachedRowSet
The DB2CachedRowSet object is a disconnected RowSet, meaning that it can be used without being connected to the database. Its implementation adheres closely to the description of a CachedRowSet. The DB2CachedRowSet is a container for rows of data from a ResultSet. The DB2CachedRowSet holds all its own data so it does not need to maintain a connection to the database other than explicitly while reading or writing data to the database.

DB2JdbcRowSet
The DB2JdbcRowSet is a connected RowSet, meaning that it can only be used with the support of an underlying Connection object, PreparedStatement object, or ResultSet object. Its implementation adheres closely to the description of a JdbcRowSet.

DB2CachedRowSet

The DB2CachedRowSet object is a disconnected RowSet, meaning that it can be used without being connected to the database. Its implementation adheres closely to the description of a CachedRowSet. The DB2CachedRowSet is a container for rows of data from a ResultSet. The DB2CachedRowSet holds all its own data so it does not need to maintain a connection to the database other than explicitly while reading or writing data to the database.

Related concepts

RowSet characteristics
You can request certain properties to be satisfied by the rowsets. Common properties include the set of interfaces to be supported by the resulting rowset.

DB2JdbcRowSet
The DB2JdbcRowSet is a connected RowSet, meaning that it can only be used with the support of an underlying Connection object, PreparedStatement object, or ResultSet object. Its implementation adheres closely to the description of a JdbcRowSet.

Using DB2CachedRowSet

Because the DB2CachedRowSet object can be disconnected and serialized, it is useful in environments where it is not always practical to run a full JDBC driver (for example, on Personal Digital Assistants (PDAs) and Java-enabled cell phones).

Since the DB2CachedRowSet object is contained in memory and its data is always known, it can serve as a highly optimized form of a scrollable ResultSet for applications. Whereas DB2ResultSets that are scrollable typically pay a performance penalty because their random movements interfere with the JDBC driver's ability to cache rows of data, RowSets do not have this issue.

Two methods are provided on DB2CachedRowSet that create new RowSets:

- The createCopy method creates a new RowSet that is identical to the copied one.
- The createShared method creates a new RowSet that shares the same underlying data as the original.

You can use the createCopy method to hand out common ResultSets to clients. If the table data is not changing, creating a copy of a RowSet and passing it to each client is more efficient than running a query against the database each time.

You can use the createShared method to improve your database's performance by allowing several people to use the same data. For example, assume that you have a Web site that shows the top twenty best-selling products on your home page when a customer connects. You want the information on your main page to be updated regularly, but running the query to get the most frequently purchased items every time a customer visits your main page is not practical. Using the createShared method, you can effectively create "cursors" for each customer without having to either process the query again or store an enormous amount of information in memory. When appropriate, the query to find the most frequently purchased products can be run again. The new data can populate the RowSet that is used to create the shared cursors and the servlets can use them.

DB2CachedRowSets provide a delayed processing feature. This feature allows multiple query requests to be grouped together and processed against the database as a single request. See the "Creating and populating a DB2CachedRowSet" on page 143 topic to eliminate some of the computational stress that the database would otherwise be under.

Because the RowSet must keep careful track of any changes that happen to it so that they are reflected back to the database, there is support for functions that undo changes or allow you to see all changes have been made. For example, there is a showDeleted method that can be used to tell the RowSet to let you fetch deleted rows. There are also cancelRowInsert and cancelRowDelete methods to undo row insertions and deletions, respectfully, after they have been made.

The DB2CachedRowSet object offers better interoperability with other Java APIs because of its event handling support and its toCollection methods that allow a RowSet or a portion of it to be converted into a Java collection.

The event handling support of DB2CachedRowSet can be used in graphical user interface (GUI) applications to control displays, for logging information about changes to the RowSet as they are made, or to find information about changes to sources other than RowSets. See "DB2JdbcRowSet events" on page 160 for details.

For information on the event model and event handling, see "DB2JdbcRowSet" on page 158 as this support works identically for both types of RowSets.

Related concepts

DB2CachedRowSet features
In addition to working like a ResultSet, the DB2CachedRowSet class has some additional functionality that makes it more flexible to use. Methods are provided for turning either the entire Java Database Connectivity (JDBC) RowSet or just a portion of it into a Java collection. Moreover, because of their disconnected nature, DB2CachedRowSets do not have a strict one-to-one relationship with ResultSets.

Related reference

Creating and populating a DB2CachedRowSet
There are several ways to place data into a DB2CachedRowSet: the populate method, DB2CachedRowSet properties with DataSources, DB2CachedRowSet properties and JDBC URLs, the setConnection(Connection) method , the execute(Connection) method, and the execute(int) method.

Accessing DB2CachedRowSet data and cursor manipulation
This topic provides information about accessing DB2CachedRowSet data and various cursor manipulation functions.

Changing DB2CachedRowSet data and reflecting changes back to the data source

This topic provides information about making changes to rows in a DB2CachedRowSet and then updating the underlying database.

Creating and populating a DB2CachedRowSet
There are several ways to place data into a DB2CachedRowSet: the populate method, DB2CachedRowSet properties with DataSources, DB2CachedRowSet properties and JDBC URLs, the setConnection(Connection) method , the execute(Connection) method, and the execute(int) method.

Using the populate method

DB2CachedRowSets have a populate method that can be used to put data into the RowSet from a DB2ResultSet object. The following is an example of this approach.

Example: Use the populate method

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
// Establish a connection to the database.
Connection conn = DriverManager.getConnection("jdbc:db2:*local");

// Create a statement and use it to perform a query.
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("select col1 from cujosql.test_table");

// Create and populate a DB2CachedRowSet from it.
DB2CachedRowSet crs = new DB2CachedRowSet();
crs.populate(rs);

// Note: Disconnect the ResultSet, Statement,
// and Connection used to create the RowSet.
rs.close();
stmt.close();
conn.close();

// Loop through the data in the RowSet.
while (crs.next()) {
  System.out.println("v1 is " + crs.getString(1));
}

crs.close();
```

Using DB2CachedRowSet properties and DataSources

DB2CachedRowSets have properties that allow the DB2CachedRowSets to accept an SQL query and a DataSource name. They then use the SQL query and DataSource name to create data for themselves. The following is an example of this approach. The reference to the DataSource named BaseDataSource is assumed to be a valid DataSource that has been previously set up.

Example: Use DB2CachedRowSet properties and DataSources

Note: Read the Code example disclaimer for important legal information.

```
// Create a new DB2CachedRowSet
DB2CachedRowSet crs = new DB2CachedRowSet();

// Set the properties that are needed for
// the RowSet to use a DataSource to populate itself.
crs.setDataSourceName("BaseDataSource");
crs.setCommand("select col1 from cujosql.test_table");

// Call the RowSet execute method. This causes
// the RowSet to use the DataSource and SQL query
// specified to populate itself with data. Once
// the RowSet is populated, it disconnects from the database.
crs.execute();

// Loop through the data in the RowSet.
while (crs.next()) {
  System.out.println("v1 is " + crs.getString(1));
}
```

```
// Eventually, close the RowSet.
crs.close();
```

Using DB2CachedRowSet properties and JDBC URLs

DB2CachedRowSets have properties that allow the DB2CachedRowSets to accept an SQL query and a JDBC URL. They then use the query and JDBC URL to create data for themselves. The following is an example of this approach.

Example: Use DB2CachedRowSet properties and JDBC URLs

Note: Read the Code example disclaimer for important legal information.

```
// Create a new DB2CachedRowSet
DB2CachedRowSet crs = new DB2CachedRowSet();

// Set the properties that are needed for
// the RowSet to use a JDBC URL to populate itself.
crs.setUrl("jdbc:db2:*local");
crs.setCommand("select col1 from cujosql.test_table");

// Call the RowSet execute method. This causes
// the RowSet to use the DataSource and SQL query
// specified to populate itself with data. Once
// the RowSet is populated, it disconnects from the database.
crs.execute();

// Loop through the data in the RowSet.
while (crs.next()) {
  System.out.println("v1 is " + crs.getString(1));
}

// Eventually, close the RowSet.
crs.close();
```

Using the setConnection(Connection) method to use an existing database connection

To promote the reuse of JDBC Connection objects, the DB2CachedRowSet provides a mechanism for passing an established Connection object to the DB2CachedRowSet that is used to populate the RowSet. If a user-supplied Connection object is passed in, the DB2CachedRowSet does not disconnect it after populating itself.

Example: Use setConnection(Connection) method to use an existing database connection

Note: Read the Code example disclaimer for important legal information.

```
// Establish a JDBC connection to the database.
Connection conn = DriverManager.getConnection("jdbc:db2:*local");

// Create a new DB2CachedRowSet
DB2CachedRowSet crs = new DB2CachedRowSet();

// Set the properties that are needed for the
// RowSet to use an already connected connection
// to populate itself.
crs.setConnection(conn);
crs.setCommand("select col1 from cujosql.test_table");

// Call the RowSet execute method. This causes
// the RowSet to use the connection that it was provided
// with previously. Once the RowSet is populated, it does not
// close the user-supplied connection.
crs.execute();

// Loop through the data in the RowSet.
while (crs.next()) {
  System.out.println("v1 is " + crs.getString(1));
}

// Eventually, close the RowSet.
crs.close();
```

Using the execute(Connection) method to use an existing database connection

To promote the reuse of JDBC Connection objects, the DB2CachedRowSet provides a mechanism for passing an established Connection object to the DB2CachedRowSet when the execute method is called. If a user-supplied Connection object is passed in, the DB2CachedRowSet does not disconnect it after populating itself.

Example: Use execute(Connection) method to use an existing database connection

Note: Read the Code example disclaimer for important legal information.

```
// Establish a JDBC connection to the database.
Connection conn = DriverManager.getConnection("jdbc:db2:*local");

// Create a new DB2CachedRowSet
DB2CachedRowSet crs = new DB2CachedRowSet();

// Set the SQL statement that is to be used to
// populate the RowSet.
crs.setCommand("select col1 from cujosql.test_table");

// Call the RowSet execute method, passing in the connection
// that should be used. Once the Rowset is populated, it does not
// close the user-supplied connection.
crs.execute(conn);

// Loop through the data in the RowSet.
while (crs.next()) {
  System.out.println("v1 is " + crs.getString(1));
}

// Eventually, close the RowSet.
crs.close();
```

Using the execute(int) method to group database requests

To reduce the database's workload, the DB2CachedRowSet provides a mechanism for grouping SQL statements for several threads into one processing request for the database.

Example: Use execute(int) method to group database requests

Note: Read the Code example disclaimer for important legal information.

```
// Create a new DB2CachedRowSet
DB2CachedRowSet crs = new DB2CachedRowSet();

// Set the properties that are needed for
// the RowSet to use a DataSource to populate itself.
crs.setDataSourceName("BaseDataSource");
crs.setCommand("select col1 from cujosql.test_table");

// Call the RowSet execute method. This causes
// the RowSet to use the DataSource and SQL query
// specified to populate itself with data. Once
// the RowSet is populated, it disconnects from the database.
// This version of the execute method accepts the number of seconds
// that it is willing to wait for its results. By
// allowing a delay, the RowSet can group the requests
// of several users and only process the request against
// the underlying database once.
crs.execute(5);

// Loop through the data in the RowSet.
while (crs.next()) {
  System.out.println("v1 is " + crs.getString(1));
}

// Eventually, close the RowSet.
crs.close();
```

Related concepts

Using DB2CachedRowSet

Because the DB2CachedRowSet object can be disconnected and serialized, it is useful in environments where it is not always practical to run a full JDBC driver (for example, on Personal Digital Assistants (PDAs) and Java-enabled cell phones).

DB2CachedRowSet features
In addition to working like a ResultSet, the DB2CachedRowSet class has some additional functionality that makes it more flexible to use. Methods are provided for turning either the entire Java Database Connectivity (JDBC) RowSet or just a portion of it into a Java collection. Moreover, because of their disconnected nature, DB2CachedRowSets do not have a strict one-to-one relationship with ResultSets.

Related reference

Accessing DB2CachedRowSet data and cursor manipulation
This topic provides information about accessing DB2CachedRowSet data and various cursor manipulation functions.

Changing DB2CachedRowSet data and reflecting changes back to the data source
This topic provides information about making changes to rows in a DB2CachedRowSet and then updating the underlying database.

Accessing DB2CachedRowSet data and cursor manipulation
This topic provides information about accessing DB2CachedRowSet data and various cursor manipulation functions.

RowSets depend on ResultSet methods. For many operations, such as DB2CachedRowSet data access and cursor movement, there is no difference at the application level between using a ResultSet and using a RowSet.

Accessing DB2CachedRowSet data

RowSets and ResultSets access data in the same manner. In the following example, the program creates a table and populates it with various data types using JDBC. Once the table is ready, a DB2CachedRowSet is created and populated with the information from the table. The example also uses various get methods of the RowSet class.

Example: Access DB2CachedRowSet data

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
import java.sql.*;
import javax.sql.*;
import com.ibm.db2.jdbc.app.*;
import java.io.*;
import java.math.*;

public class TestProgram
{
  public static void main(String args[])
  {
    // Register the driver.
    try {
      Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
    }
    catch (ClassNotFoundException ex) {
      System.out.println("ClassNotFoundException: " +
                 ex.getMessage());
      // No need to go any further.
      System.exit(1);
    }

    try {
      Connection conn = DriverManager.getConnection("jdbc:db2:*local");

      Statement stmt = conn.createStatement();

      // Clean up previous runs
      try {
        stmt.execute("drop table cujosql.test_table");
      }
      catch (SQLException ex) {
```

```
        System.out.println("Caught drop table: " + ex.getMessage());
      }

      // Create test table
      stmt.execute("Create table cujosql.test_table (col1 smallint, col2 int, " +
              "col3 bigint, col4 real, col5 float, col6 double, col7 numeric, " +
              "col8 decimal, col9 char(10), col10 varchar(10), col11 date, " +
              "col12 time, col13 timestamp)");
      System.out.println("Table created.");

      // Insert some test rows
      stmt.execute("insert into cujosql.test_table values (1, 1, 1, 1.5, 1.5, 1.5, 1.5, 1.5,
'one', 'one',
                   {d '2001-01-01'}, {t '01:01:01'}, {ts '1998-05-26 11:41:12.123456'})");

      stmt.execute("insert into cujosql.test_table values (null, null, null, null, null, null,
null, null,
                   null, null, null, null, null)");
      System.out.println("Rows inserted");

      ResultSet rs = stmt.executeQuery("select * from cujosql.test_table");
      System.out.println("Query executed");

      // Create a new rowset and populate it...
      DB2CachedRowSet crs = new DB2CachedRowSet();
      crs.populate(rs);
      System.out.println("RowSet populated.");

      conn.close();
      System.out.println("RowSet is detached...");

      System.out.println("Test with getObject");
      int count = 0;
      while (crs.next()) {
        System.out.println("Row " + (++count));
        for (int i = 1; i <= 13; i++) {
          System.out.println("  Col " + i + " value " + crs.getObject(i));
        }
      }

      System.out.println("Test with getXXX... ");
      crs.first();
      System.out.println("Row 1");
      System.out.println("  Col 1 value " + crs.getShort(1));
      System.out.println("  Col 2 value " + crs.getInt(2));
      System.out.println("  Col 3 value " + crs.getLong(3));
      System.out.println("  Col 4 value " + crs.getFloat(4));
      System.out.println("  Col 5 value " + crs.getDouble(5));
      System.out.println("  Col 6 value " + crs.getDouble(6));
      System.out.println("  Col 7 value " + crs.getBigDecimal(7));
      System.out.println("  Col 8 value " + crs.getBigDecimal(8));
      System.out.println("  Col 9 value " + crs.getString(9));
      System.out.println("  Col 10 value " + crs.getString(10));
      System.out.println("  Col 11 value " + crs.getDate(11));
      System.out.println("  Col 12 value " + crs.getTime(12));
      System.out.println("  Col 13 value " + crs.getTimestamp(13));
      crs.next();
      System.out.println("Row 2");
      System.out.println("  Col 1 value " + crs.getShort(1));
      System.out.println("  Col 2 value " + crs.getInt(2));
      System.out.println("  Col 3 value " + crs.getLong(3));
      System.out.println("  Col 4 value " + crs.getFloat(4));
      System.out.println("  Col 5 value " + crs.getDouble(5));
      System.out.println("  Col 6 value " + crs.getDouble(6));
      System.out.println("  Col 7 value " + crs.getBigDecimal(7));
      System.out.println("  Col 8 value " + crs.getBigDecimal(8));
      System.out.println("  Col 9 value " + crs.getString(9));
      System.out.println("  Col 10 value " + crs.getString(10));
      System.out.println("  Col 11 value " + crs.getDate(11));
      System.out.println("  Col 12 value " + crs.getTime(12));
      System.out.println("  Col 13 value " + crs.getTimestamp(13));

      crs.close();
    }
    catch (Exception ex) {
      System.out.println("SQLException: " + ex.getMessage());
                 ex.printStackTrace();
    }
  }
}
```

Cursor manipulation

RowSets are scrollable and act exactly like a scrollable ResultSet. In the following example, the program creates a table and populates it with data using JDBC. Once the table is ready, a DB2CachedRowSet object is created and is populated with the information from the table. The example also uses various cursor manipulation functions.

Example: Cursor manipulation

```
import java.sql.*;
import javax.sql.*;
import com.ibm.db2.jdbc.app.DB2CachedRowSet;

public class RowSetSample1
{
  public static void main(String args[])
  {
    // Register the driver.
    try {
      Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
    }
    catch (ClassNotFoundException ex) {
      System.out.println("ClassNotFoundException: " +
                 ex.getMessage());
      // No need to go any further.
      System.exit(1);
  }

  try {
    Connection conn = DriverManager.getConnection("jdbc:db2:*local");

    Statement stmt = conn.createStatement();

    // Clean up previous runs
    try {
      stmt.execute("drop table cujosql.test_table");
    }
    catch (SQLException ex) {
      System.out.println("Caught drop table: " + ex.getMessage());
    }

    // Create a test table
    stmt.execute("Create table cujosql.test_table (col1 smallint)");
    System.out.println("Table created.");

    // Insert some test rows
    for (int i = 0; i < 10; i++) {
      stmt.execute("insert into cujosql.test_table values (" + i + ")");
    }
    System.out.println("Rows inserted");

    ResultSet rs = stmt.executeQuery("select col1 from cujosql.test_table");
    System.out.println("Query executed");

    // Create a new rowset and populate it...
    DB2CachedRowSet crs = new DB2CachedRowSet();
    crs.populate(rs);
    System.out.println("RowSet populated.");

    conn.close();
    System.out.println("RowSet is detached...");

    System.out.println("Use next()");
    while (crs.next()) {
      System.out.println("v1 is " + crs.getShort(1));
    }

    System.out.println("Use previous()");
    while (crs.previous()) {
      System.out.println("value is " + crs.getShort(1));
    }

    System.out.println("Use relative()");
    crs.next();
    crs.relative(9);
    System.out.println("value is " + crs.getShort(1));

    crs.relative(-9);
    System.out.println("value is " + crs.getShort(1));
```

```
    System.out.println("Use absolute()");
    crs.absolute(10);
    System.out.println("value is " + crs.getShort(1));
    crs.absolute(1);
    System.out.println("value is " + crs.getShort(1));
    crs.absolute(-10);
    System.out.println("value is " + crs.getShort(1));
    crs.absolute(-1);
    System.out.println("value is " + crs.getShort(1));

    System.out.println("Test beforeFirst()");
    crs.beforeFirst();
    System.out.println("isBeforeFirst is " + crs.isBeforeFirst());
    crs.next();
    System.out.println("move one... isFirst is " + crs.isFirst());

    System.out.println("Test afterLast()");
    crs.afterLast();
    System.out.println("isAfterLast is " + crs.isAfterLast());
    crs.previous();
    System.out.println("move one... isLast is " + crs.isLast());

    System.out.println("Test getRow()");
    crs.absolute(7);
    System.out.println("row should be (7) and is " + crs.getRow() +
                       " value should be (6) and is " + crs.getShort(1));

    crs.close();
  }
  catch (SQLException ex) {
    System.out.println("SQLException: " + ex.getMessage());
  }
 }
}
```

Related concepts

Using DB2CachedRowSet
Because the DB2CachedRowSet object can be disconnected and serialized, it is useful in environments where it is not always practical to run a full JDBC driver (for example, on Personal Digital Assistants (PDAs) and Java-enabled cell phones).

DB2CachedRowSet features
In addition to working like a ResultSet, the DB2CachedRowSet class has some additional functionality that makes it more flexible to use. Methods are provided for turning either the entire Java Database Connectivity (JDBC) RowSet or just a portion of it into a Java collection. Moreover, because of their disconnected nature, DB2CachedRowSets do not have a strict one-to-one relationship with ResultSets.

Related reference

Creating and populating a DB2CachedRowSet
There are several ways to place data into a DB2CachedRowSet: the populate method, DB2CachedRowSet properties with DataSources, DB2CachedRowSet properties and JDBC URLs, the setConnection(Connection) method , the execute(Connection) method, and the execute(int) method.

Changing DB2CachedRowSet data and reflecting changes back to the data source
This topic provides information about making changes to rows in a DB2CachedRowSet and then updating the underlying database.

Changing DB2CachedRowSet data and reflecting changes back to the data source
This topic provides information about making changes to rows in a DB2CachedRowSet and then updating the underlying database.

The DB2CachedRowSet uses the same methods as the standard ResultSet interface for making changes to the data in the RowSet object. There is no difference at the application level between changing the data of a RowSet and changing the data of a ResultSet. The DB2CachedRowSet provides the acceptChanges method that is used to reflect changes to the RowSet back to the database where the data came from.

Delete, insert, and update rows in a DB2CachedRowSet

DB2CachedRowSets can be updated. In the following example, the program creates a table and populates it with data using JDBC. Once the table is ready, a DB2CachedRowSet is created and is populated with the information from the table. The example also uses various methods that can be used to update the RowSet and shows how the use of the showDeleted property that allows the application to fetch rows even after they have been deleted. Further, the cancelRowInsert and cancelRowDelete methods are used in the example to allow row insertion or deletion to be undone.

Example: Delete, insert, and update rows in a DB2CachedRowSet

Note: Read the Code example disclaimer for important legal information.

```
import java.sql.*;
import javax.sql.*;
import com.ibm.db2.jdbc.app.DB2CachedRowSet;

public class RowSetSample2
{
  public static void main(String args[])
  {
    // Register the driver.
    try {
      Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
    }
    catch (ClassNotFoundException ex) {
      System.out.println("ClassNotFoundException: " +
                ex.getMessage());

      // No need to go any further.
      System.exit(1);
    }

    try {
      Connection conn = DriverManager.getConnection("jdbc:db2:*local");

      Statement stmt = conn.createStatement();

      // Clean up previous runs
      try {
        stmt.execute("drop table cujosql.test_table");
      }

      catch (SQLException ex) {
        System.out.println("Caught drop table: " + ex.getMessage());
      }

      // Create test table
      stmt.execute("Create table cujosql.test_table (col1 smallint)");
      System.out.println("Table created.");

      // Insert some test rows
      for (int i = 0; i < 10; i++) {
        stmt.execute("insert into cujosql.test_table values (" + i + ")");
      }
      System.out.println("Rows inserted");

      ResultSet rs = stmt.executeQuery("select col1 from cujosql.test_table");
      System.out.println("Query executed");

      // Create a new rowset and populate it...
      DB2CachedRowSet crs = new DB2CachedRowSet();
      crs.populate(rs);
      System.out.println("RowSet populated.");

      conn.close();
      System.out.println("RowSet is detached...");

      System.out.println("Delete the first three rows");
      crs.next();
      crs.deleteRow();
      crs.next();
      crs.deleteRow();
      crs.next();
      crs.deleteRow();

      crs.beforeFirst();
      System.out.println("Insert the value -10 into the RowSet");
```

```
      crs.moveToInsertRow();
      crs.updateShort(1, (short)-10);
      crs.insertRow();
      crs.moveToCurrentRow();

      System.out.println("Update the rows to be the negative of what they now are");
      crs.beforeFirst();
      while (crs.next())
        short value = crs.getShort(1);
        value = (short)-value;
        crs.updateShort(1, value);
        crs.updateRow();
      }

      crs.setShowDeleted(true);

      System.out.println("RowSet is now (value - inserted - updated - deleted)");
      crs.beforeFirst();
      while (crs.next()) {
        System.out.println("value is " + crs.getShort(1) + " " +
                   crs.rowInserted() + " " +
                   crs.rowUpdated() + " " +
                   crs.rowDeleted());
      }

      System.out.println("getShowDeleted is " + crs.getShowDeleted());

      System.out.println("Now undo the inserts and deletes");
      crs.beforeFirst();
      crs.next();
      crs.cancelRowDelete();
      crs.next();
      crs.cancelRowDelete();
      crs.next();
      crs.cancelRowDelete();
      while (!crs.isLast()) {
        crs.next();
      }

      crs.cancelRowInsert();

      crs.setShowDeleted(false);

      System.out.println("RowSet is now (value - inserted - updated - deleted)");
      crs.beforeFirst();
      while (crs.next()) {
        System.out.println("value is " + crs.getShort(1) + " " +
                   crs.rowInserted() + " " +
                   crs.rowUpdated() + " " +
                   crs.rowDeleted());
      }

      System.out.println("finally show that calling cancelRowUpdates works");
      crs.first();
      crs.updateShort(1, (short) 1000);
      crs.cancelRowUpdates();
      crs.updateRow();
      System.out.println("value of row is " + crs.getShort(1));
      System.out.println("getShowDeleted is " + crs.getShowDeleted());

      crs.close();

    }

    catch (SQLException ex) {
      System.out.println("SQLException: " + ex.getMessage());
    }
  }
}
```

Reflect changes to a DB2CachedRowSet back to the underlying database

Once changes have been made to a DB2CachedRowSet, they only exist as long as the RowSet object exists. That is, making changes to a disconnected RowSet has no effect on the database. To reflect the changes of a RowSet in the underlying database, the acceptChanges method is used. This method tells the disconnected RowSet to re-establish a connection to the database and attempt to make the changes that have been made to the RowSet to the underlying database. If the changes cannot be safely made to the database due to conflicts with other database changes after the RowSet was created, an exception is thrown and the transaction is rolled back.

Example: Reflect changes to a DB2CachedRowSet back to the underlying database

Note: Read the Code example disclaimer for important legal information.

```
import java.sql.*;
import javax.sql.*;
import com.ibm.db2.jdbc.app.DB2CachedRowSet;

public class RowSetSample3
{
  public static void main(String args[])
  {
    // Register the driver.
    try {
      Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
    }
    catch (ClassNotFoundException ex) {
      System.out.println("ClassNotFoundException: " +
                ex.getMessage());
      // No need to go any further.
      System.exit(1);
    }

    try {
      Connection conn = DriverManager.getConnection("jdbc:db2:*local");

      Statement stmt = conn.createStatement();

      // Clean up previous runs
      try {
        stmt.execute("drop table cujosql.test_table");
      }
      catch (SQLException ex) {
        System.out.println("Caught drop table: " + ex.getMessage());
      }

      // Create test table
      stmt.execute("Create table cujosql.test_table (col1 smallint)");
      System.out.println("Table created.");

      // Insert some test rows
      for (int i = 0; i < 10; i++) {
        stmt.execute("insert into cujosql.test_table values (" + i + ")");
      }
      System.out.println("Rows inserted");

      ResultSet rs = stmt.executeQuery("select col1 from cujosql.test_table");
      System.out.println("Query executed");

      // Create a new rowset and populate it...
      DB2CachedRowSet crs = new DB2CachedRowSet();
      crs.populate(rs);
      System.out.println("RowSet populated.");

      conn.close();
      System.out.println("RowSet is detached...");

      System.out.println("Delete the first three rows");
      crs.next();
      crs.deleteRow();
      crs.next();
      crs.deleteRow();
      crs.next();
      crs.deleteRow();

      crs.beforeFirst();
      System.out.println("Insert the value -10 into the RowSet");
      crs.moveToInsertRow();
      crs.updateShort(1, (short)-10);
      crs.insertRow();
      crs.moveToCurrentRow();

      System.out.println("Update the rows to be the negative of what they now are");
      crs.beforeFirst();
      while (crs.next()) {
        short value = crs.getShort(1);
        value = (short)-value;
```

```
        crs.updateShort(1, value);
        crs.updateRow();
      }

      System.out.println("Now accept the changes to the database");

      crs.setUrl("jdbc:db2:*local");
      crs.setTableName("cujosql.test_table");

      crs.acceptChanges();
      crs.close();

      System.out.println("And the database table looks like this:");
      conn = DriverManager.getConnection("jdbc:db2:localhost");
      stmt = conn.createStatement();
      rs = stmt.executeQuery("select col1 from cujosql.test_table");
      while (rs.next()) {
        System.out.println("Value from table is " + rs.getShort(1));
      }

      conn.close();

    }
    catch (SQLException ex) {
      System.out.println("SQLException: " + ex.getMessage());
    }
  }
}
```

Related concepts

Using DB2CachedRowSet
Because the DB2CachedRowSet object can be disconnected and serialized, it is useful in environments where it is not always practical to run a full JDBC driver (for example, on Personal Digital Assistants (PDAs) and Java-enabled cell phones).

DB2CachedRowSet features
In addition to working like a ResultSet, the DB2CachedRowSet class has some additional functionality that makes it more flexible to use. Methods are provided for turning either the entire Java Database Connectivity (JDBC) RowSet or just a portion of it into a Java collection. Moreover, because of their disconnected nature, DB2CachedRowSets do not have a strict one-to-one relationship with ResultSets.

Related reference

Creating and populating a DB2CachedRowSet
There are several ways to place data into a DB2CachedRowSet: the populate method, DB2CachedRowSet properties with DataSources, DB2CachedRowSet properties and JDBC URLs, the setConnection(Connection) method , the execute(Connection) method, and the execute(int) method.

Accessing DB2CachedRowSet data and cursor manipulation
This topic provides information about accessing DB2CachedRowSet data and various cursor manipulation functions.

DB2CachedRowSet features

In addition to working like a ResultSet, the DB2CachedRowSet class has some additional functionality that makes it more flexible to use. Methods are provided for turning either the entire Java Database Connectivity (JDBC) RowSet or just a portion of it into a Java collection. Moreover, because of their disconnected nature, DB2CachedRowSets do not have a strict one-to-one relationship with ResultSets.

In addition to working like a ResultSet as several examples have shown, the DB2CachedRowSet class has some additional functionality that makes it more flexible to use. Methods are provided for turning either the entire Java Database Connectivity (JDBC) RowSet or just a portion of it into a Java collection. Moreover, because of their disconnected nature, DB2CachedRowSets do not have a strict one-to-one relationship with ResultSets.

With the methods provided by DB2CachedRowSet, you can perform the following tasks:

Obtaining collections from DB2CachedRowSets

There are three methods that return some form of a collection from a DB2CachedRowSet object. They are the following:

- **toCollection** returns an ArrayList (that is, one entry for each row) of vectors (that is, one entry for each column).
- **toCollection(int columnIndex)** returns a vector containing the value for each row from the given column.
- **getColumn(int columnIndex)** returns an array containing the value for each column for a given column.

The major difference between toCollection(int columnIndex) and getColumn(int columnIndex) is that the getColumn method can return an array of primitive types. Therefore, if columnIndex represents a column that has integer data, an integer array is returned and not an array containing java.lang.Integer objects.

The following example shows how you can use these methods.

Example: Obtain collections from DB2CachedRowSets

Note: Read the Code example disclaimer for important legal information.

```
import java.sql.*;
import javax.sql.*;
import com.ibm.db2.jdbc.app.DB2CachedRowSet;
import java.util.*;

public class RowSetSample4
{
  public static void main(String args[])
  {
    // Register the driver.
    try {
      Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
    }
    catch (ClassNotFoundException ex) {
      System.out.println("ClassNotFoundException: " +
                 ex.getMessage());
      // No need to go any further.
      System.exit(1);
    }

    try {
      Connection conn = DriverManager.getConnection("jdbc:db2:*local");
      Statement stmt = conn.createStatement();

      // Clean up previous runs
      try {
        stmt.execute("drop table cujosql.test_table");
      }
      catch (SQLException ex) {
        System.out.println("Caught drop table: " + ex.getMessage());
      }

      // Create test table
      stmt.execute("Create table cujosql.test_table (col1 smallint, col2 smallint)");
      System.out.println("Table created.");

      // Insert some test rows
      for (int i = 0; i < 10; i++) {
        stmt.execute("insert into cujosql.test_table values (" + i + ", " + (i + 100) + ")");
      }
      System.out.println("Rows inserted");

      ResultSet rs = stmt.executeQuery("select * from cujosql.test_table");
      System.out.println("Query executed");

      // Create a new rowset and populate it...
      DB2CachedRowSet crs = new DB2CachedRowSet();
      crs.populate(rs);
      System.out.println("RowSet populated.");

      conn.close();
      System.out.println("RowSet is detached...");

      System.out.println("Test the toCollection() method");
      Collection collection = crs.toCollection();
```

```
        ArrayList map = (ArrayList) collection;

        System.out.println("size is " + map.size());
        Iterator iter = map.iterator();
        int row = 1;
        while (iter.hasNext()) {
          System.out.print("row [" + (row++) + "]: \t");

          Vector vector = (Vector)iter.next();
          Iterator innerIter = vector.iterator();
          int i = 1;
          while (innerIter.hasNext()) {
            System.out.print(" [" + (i++) + "]=" + innerIter.next() + "; \t");
          }
          System.out.println();
        }
        System.out.println("Test the toCollection(int) method");
        collection = crs.toCollection(2);
        Vector vector = (Vector) collection;

        iter = vector.iterator();

        while (iter.hasNext()) {
          System.out.println("Iter: Value is " + iter.next());
        }

        System.out.println("Test the getColumn(int) method");
        Object values = crs.getColumn(2);
        short[] shorts = (short [])values;

        for (int i =0; i < shorts.length; i++) {
          System.out.println("Array: Value is " + shorts[i]);
        }
      }
      catch (SQLException ex) {
        System.out.println("SQLException: " + ex.getMessage());
      }
    }
}
```

Creating copies of RowSets

The createCopy method creates a copy of the DB2CachedRowSet. All the data associated with the RowSet is replicated along with all control structures, properties, and status flags.

The following example shows how you can use this method.

Example: Create copies of RowSets

Note: Read the Code example disclaimer for important legal information.

```
import java.sql.*;
import javax.sql.*;
import com.ibm.db2.jdbc.app.*;
import java.io.*;

public class RowSetSample5
{
  public static void main(String args[])
  {
    // Register the driver.
    try {
      Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
    }
    catch (ClassNotFoundException ex) {
      System.out.println("ClassNotFoundException: " +
                 ex.getMessage());
      // No need to go any further.
      System.exit(1);
    }

    try {
      Connection conn = DriverManager.getConnection("jdbc:db2:*local");

      Statement stmt = conn.createStatement();

      // Clean up previous runs
      try {
```

```
        stmt.execute("drop table cujosql.test_table");
      }
      catch (SQLException ex) {
        System.out.println("Caught drop table: " + ex.getMessage());
      }

      // Create test table
      stmt.execute("Create table cujosql.test_table (col1 smallint)");
      System.out.println("Table created.");

      // Insert some test rows
      for (int i = 0; i < 10; i++) {
        stmt.execute("insert into cujosql.test_table values (" + i + ")");
      }
      System.out.println("Rows inserted");

      ResultSet rs = stmt.executeQuery("select col1 from cujosql.test_table");
      System.out.println("Query executed");

      // Create a new rowset and populate it...
      DB2CachedRowSet crs = new DB2CachedRowSet();
      crs.populate(rs);
      System.out.println("RowSet populated.");

      conn.close();
      System.out.println("RowSet is detached...");

      System.out.println("Now some new RowSets from one.");
      DB2CachedRowSet crs2 = crs.createCopy();
      DB2CachedRowSet crs3 = crs.createCopy();

      System.out.println("Change the second one to be negated values");
      crs2.beforeFirst();
      while (crs2.next()) {
        short value = crs2.getShort(1);
        value = (short)-value;
        crs2.updateShort(1, value);
        crs2.updateRow();
      }

      crs.beforeFirst();
      crs2.beforeFirst();
      crs3.beforeFirst();
      System.out.println("Now look at all three of them again");

      while (crs.next()) {
        crs2.next();
        crs3.next();
        System.out.println("Values: crs: " + crs.getShort(1) + ", crs2: " + crs2.getShort(1) +
                   ", crs3: " + crs3.getShort(1));
      }
    }
    catch (Exception ex) {
      System.out.println("SQLException: " + ex.getMessage());
      ex.printStackTrace();
    }

  }

}
```

Creating shares for RowSets

The createShared method creates a new RowSet object with high-level status information and allows two RowSet objects to share the same underlying physical data.

The following example shows how you can use this method.

Example: Create shares of RowSets

Note: Read the Code example disclaimer for important legal information.

```
import java.sql.*;
import javax.sql.*;
import com.ibm.db2.jdbc.app.*;
import java.io.*;

public class RowSetSample5
```

```
{
  public static void main(String args[])
  {
    // Register the driver.
    try {
      Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
    }
    catch (ClassNotFoundException ex) {
      System.out.println("ClassNotFoundException: " +
                 ex.getMessage());
      // No need to go any further.
      System.exit(1);
    }

    try {
      Connection conn = DriverManager.getConnection("jdbc:db2:*local");

      Statement stmt = conn.createStatement();

      // Clean up previous runs
      try {
        stmt.execute("drop table cujosql.test_table");
      }
      catch (SQLException ex) {
        System.out.println("Caught drop table: " + ex.getMessage());
      }

      // Create test table
      stmt.execute("Create table cujosql.test_table (col1 smallint)");
      System.out.println("Table created.");

      // Insert some test rows
      for (int i = 0; i < 10; i++) {
        stmt.execute("insert into cujosql.test_table values (" + i + ")");
      }
      System.out.println("Rows inserted");

      ResultSet rs = stmt.executeQuery("select col1 from cujosql.test_table");
      System.out.println("Query executed");

      // Create a new rowset and populate it...
      DB2CachedRowSet crs = new DB2CachedRowSet();
      crs.populate(rs);
      System.out.println("RowSet populated.");

      conn.close();
      System.out.println("RowSet is detached...");

      System.out.println("Test the createShared functionality (create 2 shares)");
      DB2CachedRowSet crs2 = crs.createShared();
      DB2CachedRowSet crs3 = crs.createShared();

      System.out.println("Use the original to update value 5 of the table");
      crs.absolute(5);
      crs.updateShort(1, (short)-5);
      crs.updateRow();

      crs.beforeFirst();
      crs2.afterLast();

      System.out.println("Now move the cursors in opposite directions of the same data.");

      while (crs.next()) {
        crs2.previous();
        crs3.next();
        System.out.println("Values: crs: " + crs.getShort(1) + ", crs2: " + crs2.getShort(1) +
                   ", crs3: " + crs3.getShort(1));
      }
      crs.close();
      crs2.close();
      crs3.close();
    }
    catch (Exception ex) {
      System.out.println("SQLException: " + ex.getMessage());
                 ex.printStackTrace();
    }
  }
}
```

Related concepts

Using DB2CachedRowSet
Because the DB2CachedRowSet object can be disconnected and serialized, it is useful in environments where it is not always practical to run a full JDBC driver (for example, on Personal Digital Assistants (PDAs) and Java-enabled cell phones).

Related reference

Creating and populating a DB2CachedRowSet
There are several ways to place data into a DB2CachedRowSet: the populate method, DB2CachedRowSet properties with DataSources, DB2CachedRowSet properties and JDBC URLs, the setConnection(Connection) method , the execute(Connection) method, and the execute(int) method.

Accessing DB2CachedRowSet data and cursor manipulation
This topic provides information about accessing DB2CachedRowSet data and various cursor manipulation functions.

Changing DB2CachedRowSet data and reflecting changes back to the data source
This topic provides information about making changes to rows in a DB2CachedRowSet and then updating the underlying database.

DB2JdbcRowSet

The DB2JdbcRowSet is a connected RowSet, meaning that it can only be used with the support of an underlying Connection object, PreparedStatement object, or ResultSet object. Its implementation adheres closely to the description of a JdbcRowSet.

Use DB2JdbcRowSet

Because the DB2JdbcRowSet object supports events described in the Java Database Connectivity (JDBC) 3.0 specification for all RowSets, it can serve as an intermediate object between a local database and other objects that must be notified about changes to the database data.

As an example, assume that you are working in an environment where you have a main database and several Personal Digital Assistants (PDAs) that use a wireless protocol to connect to it. A DB2JdbcRowSet object can be used to move to a row and update it by using a master application that is running on the server. The row update causes an event to be generated by the RowSet component. If there is a service running that is responsible for sending out updates to the PDAs, it can register itself as a "listener" of the RowSet. Each time that it receives a RowSet event, it can generate the appropriate update and send it out to the wireless devices.

Refer to Example: DB2JdbcRowSet events for more information.

Create JDBCRowSets

There are several methods provided for creating a DB2JDBCRowSet object. Each is outlined as follows.

Use DB2JdbcRowSet properties and DataSources

DB2JdbcRowSets have properties that accept an SQL query and a DataSource name. The DB2JdbcRowSets are then ready to be used. The following is an example of this approach. The reference to the DataSource named BaseDataSource is assumed to be a valid DataSource that has been previously set up.

Example: Use DB2JdbcRowSet properties and DataSources

Note: Read the Code example disclaimer for important legal information.

```
// Create a new DB2JdbcRowSet
DB2JdbcRowSet jrs = new DB2JdbcRowSet();

// Set the properties that are needed for
// the RowSet to be processed.
jrs.setDataSourceName("BaseDataSource");
jrs.setCommand("select col1 from cujosql.test_table");

// Call the RowSet execute method. This method causes
```

```
    // the RowSet to use the DataSource and SQL query
    // specified to prepare itself for data processing.
    jrs.execute();

    // Loop through the data in the RowSet.
    while (jrs.next()) {
        System.out.println("v1 is " + jrs.getString(1));
    }

    // Eventually, close the RowSet.
    jrs.close();
```

Use DB2JdbcRowSet properties and JDBC URLs

DB2JdbcRowSets have properties that accept an SQL query and a JDBC URL. The DB2JdbcRowSets are then ready to be used. The following is an example of this approach:

Example: Use DB2JdbcRowSet properties and JDBC URLs

Note: Read the Code example disclaimer for important legal information.

```
    // Create a new DB2JdbcRowSet
    DB2JdbcRowSet jrs = new DB2JdbcRowSet();

    // Set the properties that are needed for
    // the RowSet to be processed.
    jrs.setUrl("jdbc:db2:*local");
    jrs.setCommand("select col1 from cujosql.test_table");

    // Call the RowSet execute method. This causes
    // the RowSet to use the URL and SQL query specified
    // previously to prepare itself for data processing.
    jrs.execute();

    // Loop through the data in the RowSet.
    while (jrs.next()) {
        System.out.println("v1 is " + jrs.getString(1));
    }

    // Eventually, close the RowSet.
    jrs.close();
```

Use the setConnection(Connection) method to use an existing database connection

To promote the reuse of JDBC Connection objects, the DB2JdbcRowSet allows you to pass an established connection to the DB2JdbcRowSet. This connection is used by the DB2JdbcRowSet to prepare itself for usage when the execute method is called.

Example: Use the setConnection method

Note: Read the Code example disclaimer for important legal information.

```
    // Establish a JDBC Connection to the database.
    Connection conn = DriverManager.getConnection("jdbc:db2:*local");

    // Create a new DB2JdbcRowSet.
    DB2JdbcRowSet jrs = new DB2JdbcRowSet();

    // Set the properties that are needed for
    // the RowSet to use an established connection.
    jrs.setConnection(conn);
    jrs.setCommand("select col1 from cujosql.test_table");

    // Call the RowSet execute method. This causes
    // the RowSet to use the connection that it was provided
    // previously to prepare itself for data processing.
    jrs.execute();

    // Loop through the data in the RowSet.
    while (jrs.next()) {
        System.out.println("v1 is " + jrs.getString(1));
    }

    // Eventually, close the RowSet.
    jrs.close();
```

Access data and cursor movement

Manipulation of the cursor position and access to the database data through a DB2JdbcRowSet are handled by the underlying ResultSet object. Tasks that can be done with a ResultSet object also apply to the DB2JdbcRowSet object.

Change data and reflecting changes to the underlying database

Support for updating the database through a DB2JdbcRowSet is handled completely by the underlying ResultSet object. Tasks that can be done with a ResultSet object also apply to the DB2JdbcRowSet object.

Related concepts

RowSet characteristics
You can request certain properties to be satisfied by the rowsets. Common properties include the set of interfaces to be supported by the resulting rowset.

DB2CachedRowSet
The DB2CachedRowSet object is a disconnected RowSet, meaning that it can be used without being connected to the database. Its implementation adheres closely to the description of a CachedRowSet. The DB2CachedRowSet is a container for rows of data from a ResultSet. The DB2CachedRowSet holds all its own data so it does not need to maintain a connection to the database other than explicitly while reading or writing data to the database.

DB2JdbcRowSet events
All RowSet implementations support event handling for situations that are of interest to other components. This support allows application components to "talk" to each other when events happen to them. For example, updating a database row through a RowSet can cause a Graphical User Interface (GUI) table shown to you to update itself.

In the following example, the main method does the update to the RowSet and is your core application. The listener is part of your wireless server used by your disconnected clients in the field. It is possible to tie these two aspects of a business together without getting the code for the two processes intermingled. While the event support of RowSets was designed primarily for updating GUIs with database data, it works perfectly for this type of application problem.

Example: DB2JdbcRowSet events

Note: Read the Code example disclaimer for important legal information.

```
import java.sql.*;
import javax.sql.*;
import com.ibm.db2.jdbc.app.DB2JdbcRowSet;

public class RowSetEvents {
    public static void main(String args[])
    {
        // Register the driver.
        try {
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
        } catch (ClassNotFoundException ex) {
            System.out.println("ClassNotFoundException: " +
                               ex.getMessage());
            // No need to go any further.
            System.exit(1);
        }


        try {
            // Obtain the JDBC Connection and Statement needed to set
            // up this example.
            Connection conn = DriverManager.getConnection("jdbc:db2:*local");
            Statement stmt = conn.createStatement();

            // Clean up any previous runs.
            try {
                stmt.execute("drop table cujosql.test_table");
            } catch (SQLException ex) {
                System.out.println("Caught drop table: " + ex.getMessage());
            }
```

```
            // Create the test table
            stmt.execute("Create table cujosql.test_table (col1 smallint)");
            System.out.println("Table created.");

            // Populate the table with data.
            for (int i = 0; i < 10; i++) {
                stmt.execute("insert into cujosql.test_table values (" + i + ")");
            }
            System.out.println("Rows inserted");

            // Remove the setup objects.
            stmt.close();
            conn.close();

            // Create a new rowset and set the properties need to
            // process it.
            DB2JdbcRowSet jrs = new DB2JdbcRowSet();
            jrs.setUrl("jdbc:db2:*local");
            jrs.setCommand("select col1 from cujosql.test_table");
            jrs.setConcurrency(ResultSet.CONCUR_UPDATEABLE);

            // Give the RowSet object a listener. This object handles
            // special processing when certain actions are done on
            // the RowSet.
            jrs.addRowSetListener(new MyListener());

            // Process the RowSet to provide access to the database data.
            jrs.execute();

            // Cause a few cursor change events. These events cause the cursorMoved
            // method in the listener object to get control.
            jrs.next();
            jrs.next();
            jrs.next();

            // Cause a row change event to occur. This event causes the rowChanged method
            // in the listener object to get control.
            jrs.updateShort(1, (short)6);
            jrs.updateRow();

            // Finally, cause a RowSet change event to occur. This causes the
            // rowSetChanged method in the listener object to get control.
            jrs.execute();

            // When completed, close the RowSet.
            jrs.close();
        } catch (SQLException ex) {
            ex.printStackTrace();
        }
    }
}


/**
 * This is an example of a listener. This example prints messages that show
 * how control flow moves through the application and offers some
 * suggestions about what might be done if the application were fully implemented.
 */
class MyListener
implements RowSetListener {
    public void cursorMoved(RowSetEvent rse) {
        System.out.println("Event to do:  Cursor position changed.");
        System.out.println("  For the remote system, do nothing ");
        System.out.println("  when this event happened. The remote view of the data");
        System.out.println("  could be controlled separately from the local view.");
        try {
            DB2JdbcRowSet rs = (DB2JdbcRowSet) rse.getSource();
            System.out.println("row is " + rs.getRow() + ". \n\n");
        } catch (SQLException e) {
            System.out.println("To do:  Properly handle possible problems.");
        }
    }

    public void rowChanged(RowSetEvent rse) {
        System.out.println("Event to do:  Row changed.");
        System.out.println("  Tell the remote system that a row has changed. Then,");
        System.out.println("  pass all the values only for that row to the  ");
        System.out.println("  remote system.");
        try {
            DB2JdbcRowSet rs = (DB2JdbcRowSet) rse.getSource();
            System.out.println("new values are " + rs.getShort(1) + ". \n\n");
        } catch (SQLException e) {
```

```
                System.out.println("To do:  Properly handle possible problems.");
            }
        }

        public void rowSetChanged(RowSetEvent rse) {
            System.out.println("Event to do:  RowSet changed.");
            System.out.println("  If there is a remote RowSet already established, ");
            System.out.println("  tell the remote system that the values it ");
            System.out.println("  has should be thrown out. Then, pass all ");
            System.out.println("  the current values to it.\n\n");
        }
}
```

Performance tips for the native JDBC driver

The native JDBC driver is designed to be a high performance Java interface for working with the database. However, getting the best possible performance requires that you build your applications in a way that takes advantage of the strengths the native JDBC driver has to offer. The following tips are considered good JDBC programming practice. Most are not specific to the native JDBC driver. Therefore, applications written according to these guidelines also perform well if used with JDBC drivers other than the native JDBC driver.

Avoiding SELECT * SQL queries

SELECT * FROM... is a common way to state a query in SQL. Often, however, you do not need to query all the fields. For each column that is to be returned, the JDBC driver must do the additional work of binding and returning the row. Even if your application never uses a particular column, the JDBC driver has to be made aware of it and has to reserve space for its use. If your tables have few columns that are not used, this is not significant overhead. For a large number of unused columns, however, the overhead can be significant. A better solution is to list the columns that your application is interested in individually, like this:

```
SELECT COL1, COL2, COL3 FROM...
```

Using getXXX(int) instead of getXXX(String)

Use the ResultSet getXXX methods that take numeric values instead of the versions that take column names. While the freedom to use your column names instead of numeric constants seems like an advantage, the database itself only knows how to deal with column indexes. Therefore, each getXXX method with a column name you call must be resolved by the JDBC driver before it can be passed to the database. Because getXXX methods are typically called inside loops that could be run millions of times, this little bit of overhead can rapidly accumulate.

Avoiding getObject calls for Java primitive types

When getting values from the database of primitive types (ints, longs, floats, and so on), it is faster to use the get method specific to the primitive type (getInt, getLong, getFloat) than to use getObject. The getObject call does the work of the get for the primitive type, and then creates an object to return to you. This is typically done in loops, potentially creating millions of objects with short lifespans. Using getObject for primitive commands has the added drawback of frequently activating the garbage collector, further degrading performance.

Using PreparedStatement over Statement

If you are writing an SQL statement that is used more than once, it performs better as a PreparedStatement than as a Statement object. Every time you run a statement, you go through a two step process: the statement is prepared, and then the statement is processed. When you use a prepared statement, the statement is prepared only at the time that it is constructed, not each time it is run. Though it is recognized that a PreparedStatement performs faster than a Statement, this advantage is often neglected by programmers. Due to the performance boost that PreparedStatements provide, it is wise to use them in the design of your applications wherever possible (see "Consider using PreparedStatement pooling" on page 164 below).

Avoiding DatabaseMetaData calls

Be aware that some of the DatabaseMetaData calls can be expensive. In particular, the getBestRowIdentifier, getCrossReference, getExportedKeys, and getImportedKeys methods can be costly. Some DataBaseMetaData calls involve complex join conditions over system-level tables. Use them only if you need their information, not just for convenience.

Using the correct commit level for your application

JDBC provides several commit levels which determine how multiple transactions against the system affect each other (see Transactions for more details). The default is to use the lowest commit level. This means that transactions can see some of each other's work through commit boundaries. This introduces the possibility of certain database anomalies. Some programmers increase the commit level so that they do not have to worry about these anomalies occurring. Be aware that higher commit levels involve the database hanging onto more course-grained locks. This limits the amount of concurrency that the system can have, severely slowing the performance of some applications. Often, the anomaly conditions cannot occur because of the design of the application in the first place. Take time to understand what you are trying to accomplish and limit your transaction isolation level to the lowest level you can safely use.

Consider storing data in Unicode

Java requires all character data that it works with (Strings) to be in Unicode. Therefore, any table that does not have Unicode data requires the JDBC driver to translate the data back and forth as it is put into the database and retrieved out of the database. If the table is already in Unicode, the JDBC driver does not need to translate the data and can therefore place the data from the database faster. Take care to understand that data in Unicode may not work with non-Java applications, which do not know how to deal with Unicode. Also keep in mind that non-character data does not perform any faster, as there is never a translation of this data. Another consideration is that data stored in Unicode takes up twice as much space as single byte data does. If you have many character columns that are read many times, however, the performance gained by storing your data in Unicode can be significant.

Using stored procedures

The use of stored procedures is supported in Java. Stored procedures can perform faster by allowing the JDBC driver to run static SQL instead of dynamic SQL. Do not create stored procedures for each individual SQL statement you run in your program. Where possible, however, create a stored procedure that runs a group of SQL statements.

Using BigInt instead of Numeric or Decimal

Instead of using Numeric or Decimal fields that have a scale of 0, use the BigInt data type. BigInt translates directly into the Java primitive type Long whereas Numeric or Decimal data types translate into String or BigDecimal objects. As noted in "Avoiding DatabaseMetaData calls" on page 163, using primitive data types is preferable to using types that require object creation.

Explicitly closing your JDBC resources when done with them

ResultSets, Statements, and Connections should be explicitly closed by the application when they are no longer needed. This allows the resources to be cleaned up in the most efficient way possible and can increase performance. Further, database resources that are not explicitly closed can cause resource leaks and database locks to be held longer than necessary. This can lead to application failures or reduced concurrency in applications.

Using connection pooling

Connection pooling is a strategy by which JDBC Connection objects get reused for multiple users instead of each user request creating its own Connection object. Connection objects are expensive to create. Instead of having each user create a new one, a pool of them should be shared in applications where performance is critical. Many products (such as WebSphere) provide Connection pooling support that can be used with little additional effort on the user's part. If you do not want to use a product with connection pooling support, or prefer to build your own for better control over how the pool works and performs, it is reasonably easy to do so.

Consider using PreparedStatement pooling

Statement pooling works similarly to Connection pooling. Instead of just putting Connections into a pool, put an object that contains the Connection and the PreparedStatements a pool. Then, retrieve that object and access the specific statement you want to use. This can dramatically increase performance.

Using efficient SQL

Because JDBC is built on top of SQL, just about anything that makes for efficient SQL also makes for efficient JDBC. Hence, JDBC benefits from optimized queries, wisely chosen indices, and other aspects of good SQL design.

Accessing databases using DB2 SQLJ support

DB2 Structured Query Language for Java (SQLJ) support is based on the SQLJ ANSI standard. The DB2 SQLJ support is contained in the IBM Developer Kit for Java. DB2 SQLJ support allows you to create, build, and run embedded SQL for Java applications.

The SQLJ support provided by the IBM Developer Kit for Java includes the SQLJ run-time classes, and is available in /QIBM/ProdData/Java400/ext/runtime.zip.

SQLJ setup

Before you can use SQLJ in Java applications on your server, you need to prepare your server to use SQLJ. For more information, see the SQLJ setup topic.

SQLJ tools

The following tools are also included in the SQLJ support provided by the IBM Developer Kit for Java:

- The SQLJ translator, `sqlj`, replaces embedded SQL statements in the SQLJ program with Java source statements and generates a serialized profile that contains information about the SQLJ operations that are found in the SQLJ program.
- The DB2 SQLJ Profile Customizer, `db2profc`, precompiles the SQL statements stored in the generated profile and generates a package in the DB2 database.
- The DB2 SQLJ Profile Printer, `db2profp`, prints the contents of a DB2 customized profile in plain text.
- The SQLJ profile auditor installer, `profdb`, installs and uninstalls debugging class-auditors into an existing set of binary profiles.
- The SQLJ profile conversion tool, `profconv`, converts a serialized profile instance to Java class format.

Note: These tools must be run in the Qshell Interpreter.

DB2 SQLJ restrictions

When you create DB2 applications with SQLJ, you should be aware of the following restrictions:

- DB2 SQLJ support adheres to standard DB2 Universal Database restrictions on issuing SQL statements.
- The DB2 SQLJ profile customizer should only be run on profiles associated with connections to the local database.

- The SQLJ Reference Implementation requires JDK 1.1, or higher. See the Support for multiple Java Development Kits (JDKs) topic for more information on running multiple versions of the Java Development Kit.

Related concepts

"Structured Query Language for Java profiles" on page 165
Profiles are generated by the SQLJ Translator, sqlj, when you translate the SQLJ source file. Profiles are serialized binary files. That is why these files have a .ser extension. These files contain the SQL statements from the associated SQLJ source file.

"Support for multiple Java Development Kits (JDKs)" on page 6
The IBM i platform supports multiple versions of the Java Development Kits (JDKs) and the Java 2 Platform, Standard Edition.

"Embedding SQL statements in your Java application" on page 173
Static SQL statements in SQLJ are in SQLJ clauses. SQLJ clauses begin with `#sql` and end with a semicolon (;) character.

Related tasks

"Setting up your system to use SQLJ" on page 181
Before running a Java program that contains embedded SQLJ statements, ensure that you set up your server to support SQLJ. SQLJ support requires that you modify the CLASSPATH environment variable for your server.

"Compiling and running SQLJ programs" on page 177
If your Java program has embedded SQLJ statements, you need to follow a special procedure to compile and run it.

Structured Query Language for Java profiles

Profiles are generated by the SQLJ Translator, sqlj, when you translate the SQLJ source file. Profiles are serialized binary files. That is why these files have a .ser extension. These files contain the SQL statements from the associated SQLJ source file.

To generate profiles from your SQLJ source code, run the "The structured query language for Java (SQLJ) translator (sqlj)" on page 166 on your .sqlj file.

For more information, see "Compiling and running SQLJ programs" on page 177.

Related concepts

The structured query language for Java (SQLJ) translator (sqlj)
The SQLJ translator, sqlj, generates a serialized profile containing information about the SQL operations found in the SQLJ program. The SQLJ translator uses the /QIBM/ProdData/Java400/ext/translator.zip file.

Precompiling SQL statements in a profile using the DB2 SQLJ profile customizer, db2profc
You can use the DB2 SQLJ Profile Customizer, db2profc, to make your Java application work more efficiently with your database.

Printing the contents of DB2 SQLJ profiles (db2profp and profp)
The DB2 SQLJ Profile Printer, db2profp, prints the contents of a DB2 customized profile in plain text. The Profile Printer, profp, prints the contents of profiles generated by the SQLJ translator in plain text.

SQLJ profile auditor installer (profdb)
The SQLJ profile auditor installer (profdb) installs and uninstalls debugging class-auditors. The debugging class-auditors are installed into an existing set of binary profiles. Once the debugging class-auditors are installed, all RTStatement and RTResultSet calls made during application run time are logged. They can be logged to a file or standard output. The logs can then be inspected to verify the behavior and trace errors of the application. Note that only the calls made to the underlying RTStatement and RTResultSetcall interface at run time are audited.

Embedding SQL statements in your Java application

Static SQL statements in SQLJ are in SQLJ clauses. SQLJ clauses begin with `#sql` and end with a semicolon (;) character.

Related tasks

Converting a serialized profile instance to Java class format using the SQLJ profile conversion tool (profconv)
The SQLJ profile conversion tool (profconv) converts a serialized profile instance to Java class format. The profconv tool is needed because some browsers do not support loading a serialized object from a resource file that is associated with an applet. Run the profconv utility to perform the conversion.

Compiling and running SQLJ programs
If your Java program has embedded SQLJ statements, you need to follow a special procedure to compile and run it.

The structured query language for Java (SQLJ) translator (sqlj)

The SQLJ translator, sqlj, generates a serialized profile containing information about the SQL operations found in the SQLJ program. The SQLJ translator uses the /QIBM/ProdData/Java400/ext/translator.zip file.

For more information about the profile, follow this link: Profile.

Related concepts

Structured Query Language for Java profiles
Profiles are generated by the SQLJ Translator, sqlj, when you translate the SQLJ source file. Profiles are serialized binary files. That is why these files have a .ser extension. These files contain the SQL statements from the associated SQLJ source file.

Precompiling SQL statements in a profile using the DB2 SQLJ profile customizer, db2profc
You can use the DB2 SQLJ Profile Customizer, db2profc, to make your Java application work more efficiently with your database.

Printing the contents of DB2 SQLJ profiles (db2profp and profp)
The DB2 SQLJ Profile Printer, db2profp, prints the contents of a DB2 customized profile in plain text. The Profile Printer, profp, prints the contents of profiles generated by the SQLJ translator in plain text.

SQLJ profile auditor installer (profdb)
The SQLJ profile auditor installer (profdb) installs and uninstalls debugging class-auditors. The debugging class-auditors are installed into an existing set of binary profiles. Once the debugging class-auditors are installed, all RTStatement and RTResultSet calls made during application run time are logged. They can be logged to a file or standard output. The logs can then be inspected to verify the behavior and trace errors of the application. Note that only the calls made to the underlying RTStatement and RTResultSetcall interface at run time are audited.

Embedding SQL statements in your Java application
Static SQL statements in SQLJ are in SQLJ clauses. SQLJ clauses begin with `#sql` and end with a semicolon (;) character.

Related tasks

Converting a serialized profile instance to Java class format using the SQLJ profile conversion tool (profconv)
The SQLJ profile conversion tool (profconv) converts a serialized profile instance to Java class format. The profconv tool is needed because some browsers do not support loading a serialized object from a resource file that is associated with an applet. Run the profconv utility to perform the conversion.

Compiling and running SQLJ programs

If your Java program has embedded SQLJ statements, you need to follow a special procedure to compile and run it.

Precompiling SQL statements in a profile using the DB2 SQLJ profile customizer, db2profc

You can use the DB2 SQLJ Profile Customizer, db2profc, to make your Java application work more efficiently with your database.

The DB2 SQLJ Profile Customizer does the following:

- Precompiles the SQL statements that are stored in a profile and generates a package in the DB2 database.
- Customizes the SQLJ profile by replacing the SQL statements with references to the associated statement in the package that was created.

To precompile the SQL statements in a profile, type in the following at the Qshell command prompt:

```
db2profc MyClass_SJProfile0.ser
```

Where *MyClass_SJProfile0.ser* is the name of the profile you want to precompile.

DB2 SQLJ Profile Customizer usage and syntax

db2profc[*options*] <*SQLJ_profile_name*>

Where *SQLJ_profile_name* is the name of the profile to be printed and *options* is the list of options you want.

The options available for db2profp are the following:

- `-URL=<JDBC_URL>`
- `-user=<username>`
- `-password=<password>`
- `-package=<library_name/package_name>`
- `-commitctrl=<commitment_control>`
- `-datefmt=<date_format>`
- `-datesep=<date_separator>`
- `-timefmt=<time_format>`
- `-timesep=<time_separator>`
- `-decimalpt=<decimal_point>`
- `-stmtCCSID=<CCSID>`
- `-sorttbl=<library_name/sort_sequence_table_name>`
- `-langID=<language_identifier>`

The following are the descriptions of these options:

-URL=<*JDBC_URL*>
Where *JDBC_URL* is the URL of the JDBC connection. The syntax for the URL is:

```
    "jdbc:db2:systemName"
```

For more information, see "Accessing your IBM i database with the Java JDBC driver" on page 27.

-user=<*username*>
Where *username* is your username. The default value is the user ID of the current user that is signed on for local connection.

-password=<*password*>
Where *password* is your password. The default value is the password of the current user that is signed on for local connection.

-package=<*library name/package name*>
Where *library name* is the library where the package is placed, and *package name* is the name of the package to be generated. The default library name is QUSRSYS. The default package name is generated from the name of the profile. The maximum length for the package name is 10 characters. Because the SQLJ profile name is always longer than 10 characters, the default package name that is constructed is different from the profile name. The default package name is constructed by concatenating the first letters of the profile name with the profile key number. If the profile key number is greater than 10 characters long, then the last 10 characters of the profile key number is used for the default package name. For example, the following chart shows some profile names and their default package names:

Profile name	Default package name
App_SJProfile0	App_SJPro0
App_SJProfile01234	App_S01234
App_SJProfile012345678	A012345678
App_SJProfile01234567891	1234567891

-commitctrl=<*commitment_control*>
Where *commitment_control* is the level of commitment control you want. Commitment control can have any one of the following character values:

Value	Definition
C	*CHG. Dirty reads, nonrepeatable reads and phantom reads are possible.
S	*CS. Dirty reads are not possible, but non-repeatable reads and phantom reads are possible.
A	*ALL. Dirty reads and nonrepeatable reads are not possible, but phantom reads are possible.
N	*NONE. Dirty reads, nonrepeatable reads, and phantom reads are not possible. This is the default.

-datefmt=<*date_format*>
Where *date_format* is the type of date formatting you want. Date format can have any one of the following values:

Value	Definition
USA	IBM USA standard (mm.dd.yyyy,hh:mm a.m., hh:mm p.m.)
ISO	International Standards Organization (yyyy-mm-dd, hh.mm.ss) This is the default.
EUR	IBM European Standard (dd.mm.yyyy, hh.mm.ss)
JIS	Japanese Industrial Standard Christian Era (yyyy-mm-dd, hh:mm:ss)
MDY	Month/Day/Year (mm/d/yy)
DMY	Day/Month/Year (dd/mm/yy)
YMD	Year/Month/Day (yy/mm/dd)

Value	Definition
JUL	Julian (yy/ddd)

Date format is used when accessing date result columns. All output date fields are returned in the specified format. For input date strings, the specified value is used to determine whether the date is specified Inc valid format. The default value is ISO.

-datesep=<*date_separator*>

Where *date_separator* is the type of separator you want to use. Date separator is used when accessing date result columns. Date separator can be any of the following values:

Value	Definition
/	A slash is used.
.	A period is used.
,	A comma is used.
-	A dash is used. This is the default.
blank	A space is used.

-timefmt=<*time_format*>

Where *time_format* is the format you want to use to display time fields. Time format is used when accessing time result columns. For input time strings, the specified value is used to determine whether the time is specified in a valid format. Time format can be any one of the following values:

Value	Definition
USA	IBM USA standard (mm.dd.yyyy,hh:mm a.m., hh:mm p.m.)
ISO	International Standards Organization (yyyy-mm-dd, hh.mm.ss) This is the default.
EUR	IBM European Standard (dd.mm.yyyy, hh.mm.ss)
JIS	Japanese Industrial Standard Christian Era (yyyy-mm-dd, hh:mm:ss)
HMS	Hour/Minute/Second (hh:mm:ss)

-timesep=<*time_separator*>

Where *time_separator* is the character you want to use to access your time result columns. Time separator can be any one of the following values:

Value	Definition
:	A colon is used.
.	A period is used. This is the default.
,	A comma is used.
blank	A space is used.

-decimalpt=<*decimal_point*>

Where *decimal_point* is the decimal point you want to use. The decimal point is used for numeric constants in SQL statements. Decimal point can be any one of the following values:

Value	Definition
.	A period is used. This is the default.

Value	Definition
,	A comma is used.

-stmtCCSID=<*CCSID*>
Where *CCSID* is the coded character set identifier for the SQL statements that are prepared into the package. The value of the job during customization time is the default value.

-sorttbl=<*library_name/sort_sequence_table_name*>
Where *library_name/sort_sequence_table_name* is the location and table name of the sort sequence table you want to use. The sort sequence table is used for string comparisons in SQL statements. The library name and sort sequence table name each have limits of 10 characters. The default value is taken from the job during customization time.

-langID=<*language_identifier*>
Where *language identifier* is the language identifier you want to use. The default value for the language identifier is taken from the current job during customization time. The language identifier is used in conjunction with the sort sequence table.

Related concepts

Structured Query Language for Java profiles
Profiles are generated by the SQLJ Translator, sqlj, when you translate the SQLJ source file. Profiles are serialized binary files. That is why these files have a .ser extension. These files contain the SQL statements from the associated SQLJ source file.

The structured query language for Java (SQLJ) translator (sqlj)
The SQLJ translator, sqlj, generates a serialized profile containing information about the SQL operations found in the SQLJ program. The SQLJ translator uses the /QIBM/ProdData/Java400/ext/translator.zip file.

Printing the contents of DB2 SQLJ profiles (db2profp and profp)
The DB2 SQLJ Profile Printer, db2profp, prints the contents of a DB2 customized profile in plain text. The Profile Printer, profp, prints the contents of profiles generated by the SQLJ translator in plain text.

SQLJ profile auditor installer (profdb)
The SQLJ profile auditor installer (profdb) installs and uninstalls debugging class-auditors. The debugging class-auditors are installed into an existing set of binary profiles. Once the debugging class-auditors are installed, all RTStatement and RTResultSet calls made during application run time are logged. They can be logged to a file or standard output. The logs can then be inspected to verify the behavior and trace errors of the application. Note that only the calls made to the underlying RTStatement and RTResultSetcall interface at run time are audited.

Embedding SQL statements in your Java application
Static SQL statements in SQLJ are in SQLJ clauses. SQLJ clauses begin with `#sql` and end with a semicolon (;) character.

Related tasks

Converting a serialized profile instance to Java class format using the SQLJ profile conversion tool (profconv)
The SQLJ profile conversion tool (profconv) converts a serialized profile instance to Java class format. The profconv tool is needed because some browsers do not support loading a serialized object from a resource file that is associated with an applet. Run the profconv utility to perform the conversion.

Compiling and running SQLJ programs
If your Java program has embedded SQLJ statements, you need to follow a special procedure to compile and run it.

Related information

SQL Programming

Printing the contents of DB2 SQLJ profiles (db2profp and profp)

The DB2 SQLJ Profile Printer, db2profp, prints the contents of a DB2 customized profile in plain text. The Profile Printer, profp, prints the contents of profiles generated by the SQLJ translator in plain text.

To print the content of the profiles generated by the SQLJ translator in plain text, use the profp utility as follows:

```
profp MyClass_SJProfile0.ser
```

Where *MyClass_SJProfile0.ser* is the name of the profile you want to print.

To print the content of the DB2 customized version of the profile in plain text, use the db2profp utility as follows:

```
db2profp MyClass_SJProfile0.ser
```

Where *MyClass_SJProfile0.ser* is the name of the profile you want to print.

Note: If you run db2profp on an uncustomized profile, it tells you that the profile has not been customized. If you run profp on a customized profile, it displays the contents of the profile without the customizations.

DB2 SQLJ Profile Printer usage and syntax:

```
db2profp [options] <SQLJ_profile_name>
```

Where *SQLJ_profile_name* is the name of the profile to be printed and *options* is the list of options you want.

The options available for db2profp are the following:

-URL=<*JDBC_URL*>
: Where *JDBC_URL* is the URL you want to connect to. For more information, see "Accessing your IBM i database with the Java JDBC driver" on page 27.

-user=<*username*>
: Where *username* is the user name is your user profile.

-password=<*password*>
: Where *password* is the password of your user profile.

Related concepts

Structured Query Language for Java profiles
Profiles are generated by the SQLJ Translator, sqlj, when you translate the SQLJ source file. Profiles are serialized binary files. That is why these files have a .ser extension. These files contain the SQL statements from the associated SQLJ source file.

The structured query language for Java (SQLJ) translator (sqlj)
The SQLJ translator, sqlj, generates a serialized profile containing information about the SQL operations found in the SQLJ program. The SQLJ translator uses the /QIBM/ProdData/Java400/ext/translator.zip file.

Precompiling SQL statements in a profile using the DB2 SQLJ profile customizer, db2profc
You can use the DB2 SQLJ Profile Customizer, db2profc, to make your Java application work more efficiently with your database.

SQLJ profile auditor installer (profdb)
The SQLJ profile auditor installer (profdb) installs and uninstalls debugging class-auditors. The debugging class-auditors are installed into an existing set of binary profiles. Once the debugging class-auditors are installed, all RTStatement and RTResultSet calls made during application run time are logged. They can be logged to a file or standard output. The logs can then be inspected to verify the behavior and trace errors of the application. Note that only the calls made to the underlying RTStatement and RTResultSetcall interface at run time are audited.

Embedding SQL statements in your Java application

Static SQL statements in SQLJ are in SQLJ clauses. SQLJ clauses begin with `#sql` and end with a semicolon (;) character.

Related tasks

Converting a serialized profile instance to Java class format using the SQLJ profile conversion tool (profconv)
The SQLJ profile conversion tool (profconv) converts a serialized profile instance to Java class format. The profconv tool is needed because some browsers do not support loading a serialized object from a resource file that is associated with an applet. Run the profconv utility to perform the conversion.

Compiling and running SQLJ programs
If your Java program has embedded SQLJ statements, you need to follow a special procedure to compile and run it.

SQLJ profile auditor installer (profdb)

The SQLJ profile auditor installer (profdb) installs and uninstalls debugging class-auditors. The debugging class-auditors are installed into an existing set of binary profiles. Once the debugging class-auditors are installed, all RTStatement and RTResultSet calls made during application run time are logged. They can be logged to a file or standard output. The logs can then be inspected to verify the behavior and trace errors of the application. Note that only the calls made to the underlying RTStatement and RTResultSetcall interface at run time are audited.

To install debugging class-auditors, enter the following at the Qshell command prompt:

```
profdb MyClass_SJProfile0.ser
```

Where *MyClass_SJProfile0.ser* is the name of the profile that was generated by the SQLJ Translator.

To uninstall debugging class-auditors, enter the following at the Qshell command prompt:

```
profdb -Cuninstall MyClass_SJProfile.ser
```

Where *MyClass_SJProfile0.ser* is the name of the profile that was generated by the SQLJ Translator.

Related concepts

Structured Query Language for Java profiles
Profiles are generated by the SQLJ Translator, sqlj, when you translate the SQLJ source file. Profiles are serialized binary files. That is why these files have a .ser extension. These files contain the SQL statements from the associated SQLJ source file.

The structured query language for Java (SQLJ) translator (sqlj)
The SQLJ translator, sqlj, generates a serialized profile containing information about the SQL operations found in the SQLJ program. The SQLJ translator uses the /QIBM/ProdData/Java400/ext/translator.zip file.

Precompiling SQL statements in a profile using the DB2 SQLJ profile customizer, db2profc
You can use the DB2 SQLJ Profile Customizer, db2profc, to make your Java application work more efficiently with your database.

Printing the contents of DB2 SQLJ profiles (db2profp and profp)
The DB2 SQLJ Profile Printer, db2profp, prints the contents of a DB2 customized profile in plain text. The Profile Printer, profp, prints the contents of profiles generated by the SQLJ translator in plain text.

Embedding SQL statements in your Java application
Static SQL statements in SQLJ are in SQLJ clauses. SQLJ clauses begin with `#sql` and end with a semicolon (;) character.

Related tasks

Converting a serialized profile instance to Java class format using the SQLJ profile conversion tool (profconv)
The SQLJ profile conversion tool (profconv) converts a serialized profile instance to Java class format. The profconv tool is needed because some browsers do not support loading a serialized object from a resource file that is associated with an applet. Run the profconv utility to perform the conversion.

Compiling and running SQLJ programs

If your Java program has embedded SQLJ statements, you need to follow a special procedure to compile and run it.

Converting a serialized profile instance to Java class format using the SQLJ profile conversion tool (profconv)

The SQLJ profile conversion tool (profconv) converts a serialized profile instance to Java class format. The profconv tool is needed because some browsers do not support loading a serialized object from a resource file that is associated with an applet. Run the profconv utility to perform the conversion.

- To run the profconv utility, type the following on the Qshell command line:

  ```
  profconv MyApp_SJProfile0.ser
  ```

  where *MyApp_SJProfile0.ser* is the name of profile instance you want to convert.

The `profconv` tool invokes `sqlj -ser2class`. See sqlj for command line options.

Related concepts

Structured Query Language for Java profiles
Profiles are generated by the SQLJ Translator, sqlj, when you translate the SQLJ source file. Profiles are serialized binary files. That is why these files have a .ser extension. These files contain the SQL statements from the associated SQLJ source file.

The structured query language for Java (SQLJ) translator (sqlj)
The SQLJ translator, sqlj, generates a serialized profile containing information about the SQL operations found in the SQLJ program. The SQLJ translator uses the /QIBM/ProdData/Java400/ext/translator.zip file.

Precompiling SQL statements in a profile using the DB2 SQLJ profile customizer, db2profc
You can use the DB2 SQLJ Profile Customizer, db2profc, to make your Java application work more efficiently with your database.

Printing the contents of DB2 SQLJ profiles (db2profp and profp)
The DB2 SQLJ Profile Printer, db2profp, prints the contents of a DB2 customized profile in plain text. The Profile Printer, profp, prints the contents of profiles generated by the SQLJ translator in plain text.

SQLJ profile auditor installer (profdb)
The SQLJ profile auditor installer (profdb) installs and uninstalls debugging class-auditors. The debugging class-auditors are installed into an existing set of binary profiles. Once the debugging class-auditors are installed, all RTStatement and RTResultSet calls made during application run time are logged. They can be logged to a file or standard output. The logs can then be inspected to verify the behavior and trace errors of the application. Note that only the calls made to the underlying RTStatement and RTResultSetcall interface at run time are audited.

Embedding SQL statements in your Java application
Static SQL statements in SQLJ are in SQLJ clauses. SQLJ clauses begin with `#sql` and end with a semicolon (;) character.

Related tasks

Compiling and running SQLJ programs
If your Java program has embedded SQLJ statements, you need to follow a special procedure to compile and run it.

Embedding SQL statements in your Java application

Static SQL statements in SQLJ are in SQLJ clauses. SQLJ clauses begin with `#sql` and end with a semicolon (;) character.

Before you create any SQLJ clauses in your Java application, import the following packages:

- import java.sql.*;
- import sqlj.runtime.*;
- import sqlj.runtime.ref.*;

The simplest SQLJ clauses are clauses that can be processed and consist of the token `#sql` followed by an SQL statement enclosed in braces. For example, the following SQLJ clause may appear wherever a Java statement may legally appear:

```
#sql { DELETE FROM TAB };
```

The previous example deletes all the rows in the table named TAB.

In an SQLJ process clause, the tokens that appear inside the braces are either SQL tokens or host variables. All host variables are distinguished by the colon (:) character. SQL tokens never occur outside the braces of an SQLJ process clause. For example, the following Java method inserts its arguments into an SQL table:

```
public void insertIntoTAB1 (int x, String y, float z) throws SQLException
{
    #sql { INSERT INTO TAB1 VALUES (:x, :y, :z) };
}
```

The method body consists of an SQLJ process clause containing the host variables x, y, and z.

In general, SQL tokens are case insensitive (except for identifiers delimited by double quotation marks), and can be written in upper, lower, or mixed case. *Java* tokens, however, are case sensitive. For clarity in examples, case insensitive SQL tokens are uppercase, and Java tokens are lowercase or mixed case. Throughout this topic, the lowercase `null` is used to represent the Java "null" value, and the uppercase `NULL` is used to represent the SQL "null" value.

The following types of SQL constructs may appear in SQLJ programs:

- Queries For example, SELECT statements and expressions.
- SQL Data Change statements (DML) For example, INSERT, UPDATE, DELETE.
- Data statements For example, FETCH, SELECT..INTO.
- Transaction Control statements For example, COMMIT, ROLLBACK, etc.
- Data Definition Language (DDL, also known as Schema Manipulation Language) statements For example, CREATE, DROP, ALTER.
- Calls to stored procedures For example, CALL MYPROC(:x, :y, :z)
- Invocations of stored functions For example, VALUES(MYFUN(:x))

Related concepts

Structured Query Language for Java profiles
Profiles are generated by the SQLJ Translator, sqlj, when you translate the SQLJ source file. Profiles are serialized binary files. That is why these files have a .ser extension. These files contain the SQL statements from the associated SQLJ source file.

The structured query language for Java (SQLJ) translator (sqlj)
The SQLJ translator, sqlj, generates a serialized profile containing information about the SQL operations found in the SQLJ program. The SQLJ translator uses the /QIBM/ProdData/Java400/ext/translator.zip file.

Precompiling SQL statements in a profile using the DB2 SQLJ profile customizer, db2profc
You can use the DB2 SQLJ Profile Customizer, db2profc, to make your Java application work more efficiently with your database.

Printing the contents of DB2 SQLJ profiles (db2profp and profp)
The DB2 SQLJ Profile Printer, db2profp, prints the contents of a DB2 customized profile in plain text. The Profile Printer, profp, prints the contents of profiles generated by the SQLJ translator in plain text.

SQLJ profile auditor installer (profdb)
The SQLJ profile auditor installer (profdb) installs and uninstalls debugging class-auditors. The debugging class-auditors are installed into an existing set of binary profiles. Once the debugging class-auditors are installed, all RTStatement and RTResultSet calls made during application run time are logged. They can be logged to a file or standard output. The logs can then be inspected to verify the behavior

and trace errors of the application. Note that only the calls made to the underlying RTStatement and RTResultSetcall interface at run time are audited.

Related tasks

Converting a serialized profile instance to Java class format using the SQLJ profile conversion tool (profconv)
The SQLJ profile conversion tool (profconv) converts a serialized profile instance to Java class format. The profconv tool is needed because some browsers do not support loading a serialized object from a resource file that is associated with an applet. Run the profconv utility to perform the conversion.

Compiling and running SQLJ programs
If your Java program has embedded SQLJ statements, you need to follow a special procedure to compile and run it.

Host variables in Structured Query Language for Java

Arguments to embedded SQL statements are passed through host variables. Host variables are variables of the host language, and they can appear in SQL statements.

Host variables have up to three parts:

- A colon (:) prefix.
- A Java host variable that is a Java identifier for a parameter, variable, or field.
- An optional parameter mode identifier.

  This mode identifier can be one of the following:
  IN, OUT, or INOUT.

The evaluation of a Java identifier does not have side effects in a Java program, so it may appear multiple times in the Java code generated to replace an SQLJ clause.

The following query contains the host variable, `:x`. This host variable is the Java variable, field, or parameter x that is visible in the scope containing the query.

```
SELECT COL1, COL2 FROM TABLE1 WHERE :x > COL3
```

Example: Embedding SQL Statements in your Java application

The following example SQLJ application, App.sqlj, uses static SQL to retrieve and update data from the EMPLOYEE table of the DB2 sample database.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
import java.sql.*;
import sqlj.runtime.*;
import sqlj.runtime.ref.*;

#sql iterator App_Cursor1 (String empno, String firstnme) ; // 1
#sql iterator App_Cursor2 (String) ;

class App
{

  /**********************
   **  Register Driver **
   **********************/

 static
 {
   try
   {
     Class.forName("com.ibm.db2.jdbc.app.DB2Driver").newInstance();
   }
   catch (Exception e)
   {
     e.printStackTrace();
   }
 }

  /********************
```

```
  **      Main      **
  ********************/

public static void main(String argv[])
{
  try
  {
    App_Cursor1 cursor1;
    App_Cursor2 cursor2;

    String str1 = null;
    String str2 = null;
    long   count1;

    // URL is jdbc:db2:dbname
    String url = "jdbc:db2:sample";

    DefaultContext ctx = DefaultContext.getDefaultContext();
    if (ctx == null)
    {
      try
      {
        // connect with default id/password
        Connection con = DriverManager.getConnection(url);
        con.setAutoCommit(false);
        ctx = new DefaultContext(con);
      }
      catch (SQLException e)
      {
        System.out.println("Error: could not get a default context");
        System.err.println(e) ;
        System.exit(1);
      }
      DefaultContext.setDefaultContext(ctx);
    }

    // retrieve data from the database
    System.out.println("Retrieve some data from the database.");
    #sql cursor1 = {SELECT empno, firstnme FROM employee}; // 2

    // display the result set
    // cursor1.next() returns false when there are no more rows
    System.out.println("Received results:");
    while (cursor1.next()) // 3
    {
      str1 = cursor1.empno(); // 4
      str2 = cursor1.firstnme();

      System.out.print (" empno= " + str1);
      System.out.print (" firstname= " + str2);
      System.out.println("");
    }
    cursor1.close(); // 9

    // retrieve number of employee from the database
    #sql { SELECT count(*) into :count1 FROM employee }; // 5
    if (1 == count1)
      System.out.println ("There is 1 row in employee table");
    else
      System.out.println ("There are " + count1
                          + " rows in employee table");

    // update the database
    System.out.println("Update the database.");
    #sql { UPDATE employee SET firstnme = 'SHILI' WHERE empno = '000010' };

    // retrieve the updated data from the database
    System.out.println("Retrieve the updated data from the database.");
    str1 = "000010";
    #sql cursor2 = {SELECT firstnme FROM employee WHERE empno = :str1}; // 6

    // display the result set
    // cursor2.next() returns false when there are no more rows
    System.out.println("Received results:");
    while (true)
    {
      #sql { FETCH :cursor2 INTO :str2 }; // 7
      if (cursor2.endFetch()) break; // 8

      System.out.print (" empno= " + str1);
      System.out.print (" firstname= " + str2);
      System.out.println("");
```

```
      }
      cursor2.close(); // 9

      // rollback the update
      System.out.println("Rollback the update.");
      #sql { ROLLBACK work };
      System.out.println("Rollback done.");
    }
    catch( Exception e )
    {
      e.printStackTrace();
    }
  }
}
```

1Declare iterators. This section declares two types of iterators:

- App_Cursor1: Declares column data types and names, and returns the values of the columns according to column name (Named binding to columns).
- App_Cursor2: Declares column data types, and returns the values of the columns by column position (Positional binding to columns).

2Initialize the iterator. The iterator object cursor1 is initialized using the result of a query. The query stores the result in cursor1.

3Advance the iterator to the next row. The cursor1.next() method returns a Boolean `false` if there are no more rows to retrieve.

4Move the data. The named accessor method empno() returns the value of the column named empno on the current row. The named accessor method firstnme() returns the value of the column named firstnme on the current row.

5SELECT data into a host variable. The SELECT statement passes the number of rows in the table into the host variable count1.

6 Initialize the iterator. The iterator object cursor2 is initialized using the result of a query. The query stores the result in cursor2.

7Retrieve the data. The FETCH statement returns the current value of the first column declared in the ByPos cursor from the result table into the host variable str2.

8Check the success of a FETCH.INTO statement. The endFetch() method returns a Boolean true if the iterator is not positioned on a row, that is, if the last attempt to fetch a row failed. The endFetch() method returns false if the last attempt to fetch a row was successful. DB2 attempts to fetch a row when the next() method is called. A FETCH...INTO statement implicitly calls the next() method.

9Close the iterators. The close() method releases any resources held by the iterators. You should explicitly close iterators to ensure that system resources are released in a timely fashion.

Compiling and running SQLJ programs

If your Java program has embedded SQLJ statements, you need to follow a special procedure to compile and run it.

If your Java program has embedded SQLJ statements, you need to follow a special procedure to compile and run it.

1. Set up your server to use SQLJ.
2. Use the SQLJ translator, `sqlj`, on your Java source code with embedded SQL to generate Java source code and associated profiles. There is one profile generated for each connection.

   For example, type in the following command:

   ```
   sqlj MyClass.sqlj
   ```

   where *MyClass.sqlj* is the name of your SQLJ file.

In this example, the SQLJ translator generates a MyClass.java source code file and any associated profiles. The associated profiles are named MyClass_SJProfile0.ser, MyClass_SJProfile1.ser, MyClass_SJProfile2.ser, and so on.

Note: The SQLJ translator automatically compiles the translated Java source code into a class file unless you explicitly turn off the compile option with the `-compile=false` clause.

3. Use the SQLJ Profile Customizer tool, `db2profc`, to install DB2 SQLJ Customizers on generated profiles and create the DB2 packages on the local system.

   For example, type in the command:

   ```
   db2profc MyClass_SJProfile0.ser
   ```

   where *MyClass_SJProfile0.ser* is the name of the profile on which the DB2 SQLJ Customizer is run.

   Note: This step is optional but is recommended to increase runtime performance.

4. Run the Java class file just like any other Java class file.

   For example, type in the command:

   ```
   java MyClass
   ```

   where *MyClass* is the name of your Java class file.

Related concepts

Structured Query Language for Java profiles
Profiles are generated by the SQLJ Translator, sqlj, when you translate the SQLJ source file. Profiles are serialized binary files. That is why these files have a .ser extension. These files contain the SQL statements from the associated SQLJ source file.

The structured query language for Java (SQLJ) translator (sqlj)
The SQLJ translator, sqlj, generates a serialized profile containing information about the SQL operations found in the SQLJ program. The SQLJ translator uses the /QIBM/ProdData/Java400/ext/translator.zip file.

Precompiling SQL statements in a profile using the DB2 SQLJ profile customizer, db2profc
You can use the DB2 SQLJ Profile Customizer, db2profc, to make your Java application work more efficiently with your database.

Printing the contents of DB2 SQLJ profiles (db2profp and profp)
The DB2 SQLJ Profile Printer, db2profp, prints the contents of a DB2 customized profile in plain text. The Profile Printer, profp, prints the contents of profiles generated by the SQLJ translator in plain text.

SQLJ profile auditor installer (profdb)
The SQLJ profile auditor installer (profdb) installs and uninstalls debugging class-auditors. The debugging class-auditors are installed into an existing set of binary profiles. Once the debugging class-auditors are installed, all RTStatement and RTResultSet calls made during application run time are logged. They can be logged to a file or standard output. The logs can then be inspected to verify the behavior and trace errors of the application. Note that only the calls made to the underlying RTStatement and RTResultSetcall interface at run time are audited.

Embedding SQL statements in your Java application
Static SQL statements in SQLJ are in SQLJ clauses. SQLJ clauses begin with `#sql` and end with a semicolon (;) character.

“Embedding SQL statements in your Java application” on page 173
Static SQL statements in SQLJ are in SQLJ clauses. SQLJ clauses begin with `#sql` and end with a semicolon (;) character.

Related tasks

Converting a serialized profile instance to Java class format using the SQLJ profile conversion tool (profconv)

The SQLJ profile conversion tool (profconv) converts a serialized profile instance to Java class format. The profconv tool is needed because some browsers do not support loading a serialized object from a resource file that is associated with an applet. Run the profconv utility to perform the conversion.

Java SQL routines

Your system provides the ability to access Java programs from SQL statements and programs. This can be done using Java stored procedures and Java user-defined functions (UDFs). The IBM i supports both the DB2 and SQLJ conventions for calling Java stored procedures and Java UDFs. Both Java stored procedures and Java UDFs can use Java classes that are stored in JAR files. The IBM i uses stored procedures defined by the *SQLJ Part 1* standard to register JAR files with the database.

Using Java SQL routines

You can access Java programs from SQL statements and programs. This can be done using Java stored procedures and Java user-defined functions (UDFs).

To use Java SQL routines, complete the following tasks:

1. Enable SQLJ

   Because any Java SQL routine may use SQLJ, make SQLJ runtime support always available when running Java 2 Platform, Standard Edition (J2SE). To enable runtime support for SQLJ in J2SE, add a link to the SQLJ runtime.zip file from your extensions directory. For more information, see Setting up your system to use SQLJ.
2. Write the Java methods for the routines

   A Java SQL routine processes a Java method from SQL. This method must be written using either the Db2 for i or SQLJ parameter passing conventions. See Java stored procedures, Java user-defined functions, and Java user-defined table functions for more information about coding a method used by a Java SQL routine.
3. Compile the Java classes

   Java SQL routines written using the Java parameter style may be compiled without any addition setup. However, Java SQL routines using the DB2GENERAL parameter style must extend either the com.ibm.db2.app.UDF class or com.ibm.db2.app.StoredProc class. These classes are contained in the JAR file, /QIBM/ProdData/Java400/ext/db2routines_classes.jar. When using javac to compile these routines, this JAR file must exist in the CLASSPATH. For example, the following command compiles a Java source file containing a routine which uses the DB2GENERAL parameter style:

   ```
   javac -DCLASSPATH=/QIBM/ProdData/Java400/ext/db2routines_classes.jar
   source.java
   ```
4. Make the compiled classes accessible to the Java virtual machine (JVM) used by the database

   The user-defined classes used by the database JVM can either reside in the /QIBM/UserData/OS400/SQLLib/Function directory or in a JAR file registered to the database.

   The /QIBM/UserData/OS400/SQLLib/Function is the IBM i equivalent of /sqllib/function, the directory where Db2 for i stores Java stored procedures and Java UDFs on other platforms. If the class is part of a Java package, it must reside in the appropriate subdirectory. For example, if the runit class is created as part of the foo.bar package, the file runnit.class should be in the integrated file system directory, /QIBM/ProdData/OS400/SQLLib/Function/foo/bar.

   The class file may also be placed in a JAR file that is registered to the database. The JAR file is registered using the SQLJ.INSTALL_JAR stored procedure. This stored procedure is used to assign a JAR ID to a JAR file. This JAR ID is used to identify the JAR file in which the class file resides. See SQLJ procedures that manipulate JAR files for more information on SQLJ.INSTALL_JAR as well as other stored procedures to manipulate JAR files.
5. Register the routine with the database.

   Java SQL routines is registered with the database using the CREATE PROCEDURE and CREATE FUNCTION SQL statements. These statements contain the following elements:

CREATE keywords
The SQL statements to create a Java SQL routine begin with either CREATE PROCEDURE or CREATE STATEMENT.

Name of routine
The SQL statement then identifies the name of the routine that is known to the database. This is the name that is used to access the Java routine from SQL.

Parameters and return value
The SQL statement then identifies the parameters and return values, if applicable, for the Java routine.

LANGUAGE JAVA
The SQL statement uses the keywords LANGUAGE JAVA to indicate that the routine was written in Java.

PARAMETER STYLE KEYWORDS
The SQL statement then identifies the parameter style using the keywords PARAMETER STYLE JAVA or PARAMETER STYLE DB2GENERAL.

External name
The SQL statement then identifies the Java method to be processed as Java SQL routines. The external name has one of two formats:

- If the method is in a class file that is located under the /QIBM/UserData/OS400/SQLLib/Function directory, then the method is identified using the format *classname.methodname*, where *classname* is the fully qualified name of the class and *methodname* is the name of the method.
- If the method is in a JAR file registered to the database, then the method is identified using the format *jarid*:*classname.methodname*, where *jarid* is the JAR ID of the registered JAR file, *classname* is the name of the class, and *methodname* is the name of the method.

System i® Navigator may be used to create a stored procedure or user-defined function that uses the Java parameter style.

6. Use the Java procedure

   A Java stored procedure is called using the SQL CALL statement. A Java UDF is a function that is called as part of another SQL statement.

Related concepts

Java stored procedures
When using Java to write stored procedures, you can use two possible parameter passing styles.

Java user-defined scalar functions
A Java scalar function returns one value from a Java program to the database. For example, a scalar function could be created that returns the sum of two numbers.

SQLJ procedures that manipulate JAR files
Both Java stored procedures and Java UDFs can use Java classes that are stored in Java JAR files.

Parameter passing conventions for Java stored procedures and UDFs
The following table lists how SQL data types are represented in Java stored procedures and UDFs.

"Setting up your system to use SQLJ" on page 181
Before running a Java program that contains embedded SQLJ statements, ensure that you set up your server to support SQLJ. SQLJ support requires that you modify the CLASSPATH environment variable for your server.

"Java stored procedures" on page 181
When using Java to write stored procedures, you can use two possible parameter passing styles.

"Java user-defined scalar functions" on page 185
A Java scalar function returns one value from a Java program to the database. For example, a scalar function could be created that returns the sum of two numbers.

"Java user-defined table functions" on page 190

DB2 provides the ability for a function to return a table. This is useful for exposing information from outside the database to the database in table form. For example, a table can be created that exposes the properties set in the Java virtual machine (JVM) used for Java stored procedures and Java UDFs (both table and scalar).

"SQLJ procedures that manipulate JAR files" on page 192
Both Java stored procedures and Java UDFs can use Java classes that are stored in Java JAR files.

Setting up your system to use SQLJ

Before running a Java program that contains embedded SQLJ statements, ensure that you set up your server to support SQLJ. SQLJ support requires that you modify the CLASSPATH environment variable for your server.

For more information about working with Java classpaths, see the following page:

Java classpath

Using SQLJ and J2SE

To set up SQLJ on a server running any supported version of J2SE, complete the following steps:

1. Add the following files to the CLASSPATH environment variable for your server:
   - /QIBM/ProdData/Os400/Java400/ext/sqlj_classes.jar
   - /QIBM/ProdData/Os400/Java400/ext/translator.zip

   Note: You need to add translator.zip only when you want to run the SQLJ translator (sqlj command). You do not need to add translator.zip if you only want to run compiled Java programs that use SQLJ. for more information, see The SQLJ translator (sqlj)
2. At an IBM i command prompt, use the following command to add a link to runtime.zip from your extensions directory. Type the command on one line, then press **Enter**.

   ```
   ADDLNK OBJ('/QIBM/ProdData/Os400/Java400/ext/runtime.zip')
   NEWLNK('/QIBM/UserData/Java400/ext/runtime.zip')
   ```

   For more information about installing extensions, see the following page:

   Install extensions for the IBM Developer Kit for Java

Java stored procedures

When using Java to write stored procedures, you can use two possible parameter passing styles.

The recommended style is the JAVA parameter style, which matches the parameter style specified in the SQLj: SQL routines standard. The second style, DB2GENERAL, is a parameter style defined by DB2 UDB. The parameter style also determines the conventions that you must use when coding a Java stored procedure.

Additionally, you should also be aware of some restrictions that are placed on Java stored procedures.

Related concepts

Java user-defined scalar functions
A Java scalar function returns one value from a Java program to the database. For example, a scalar function could be created that returns the sum of two numbers.

SQLJ procedures that manipulate JAR files
Both Java stored procedures and Java UDFs can use Java classes that are stored in Java JAR files.

Parameter passing conventions for Java stored procedures and UDFs
The following table lists how SQL data types are represented in Java stored procedures and UDFs.

Related tasks

Using Java SQL routines

You can access Java programs from SQL statements and programs. This can be done using Java stored procedures and Java user-defined functions (UDFs).

JAVA parameter style

When you code a Java stored procedure that uses the JAVA parameter style, you must use these conventions.

- The Java method must be a public void static (not instance) method.
- The parameters of the Java method must be SQL-compatible types.
- A Java method may test for an SQL NULL value when the parameter is a null-capable type (like String).
- Output parameters are returned by using single element arrays.
- The Java method may access the current database using the getConnection method.

Java stored procedures using the JAVA parameter style are public static methods. Within the classes, the stored procedures are identified by their method name and signature. When you call a stored procedure, its signature is generated dynamically, based on the variable types defined by the CREATE PROCEDURE statement.

If a parameter is passed in a Java type that permits the null value, a Java method can compare the parameter to null to determine if an input parameter is an SQL NULL.

The following Java types do not support the null value:

- short
- int
- long
- float
- double

If a null value is passed to a Java type that does not support the null value, an SQL Exception with an error code of -20205 will be returned.

Output parameters are passed as arrays that contain one element. The Java stored procedure can set the first element of the array to set the output parameter.

A connection to the embedding application context is accessed using the following Java Database Connectivity (JDBC) call:

```
connection=DriverManager.getConnection("jdbc:default:connection");
```

This connection then runs SQL statements with JDBC APIs.

The following is a small stored procedure with one input and two outputs. It runs the given SQL query, and returns both the number of rows in the result and the SQLSTATE.

Example: Stored procedure with one input and two outputs

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
package mystuff;

import java.sql.*;
public class sample2 {
  public static void donut(String query, int[] rowCount,
    String[] sqlstate) throws Exception {
  try {
    Connection c=DriverManager.getConnection("jdbc:default:connection");
    Statement s=c.createStatement();
    ResultSet r=s.executeQuery(query);
    int counter=0;
    while(r.next()){
      counter++;
    }
```

```
        r.close(); s.close();
        rowCount[0] = counter;
        }catch(SQLException x){
        sqlstate[0]= x.getSQLState();
       }
      }
     }
```

In the SQLj standard, to return a result set in routines that use the JAVA parameter style, the result set must be set explicitly. When a procedure is created that returns result sets, additional result set parameters are added to the end of the parameter list. For example, the statement

```
CREATE PROCEDURE RETURNTWO()
DYNAMIC RESULT SETS 2
LANGUAGE JAVA
PARAMETER STYLE JAVA
EXTERNAL NAME 'javaClass!returnTwoResultSets'
```

would call a Java method with the signature `public static void returnTwoResultSets(ResultSet[] rs1, ResultSet[] rs2)`.

The output parameters of the result sets must be explicitly set as illustrated in the following example. As in the DB2GENERAL style, the result sets and corresponding statements should not be closed.

Example: Stored procedure that returns two result sets

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
import java.sql.*;
public class javaClass {
  /**
    * Java stored procedure, with JAVA style parameters,
    * that processes two predefined sentences
    * and returns two result sets
    *
    * @param ResultSet[] rs1    first ResultSet
    * @param ResultSet[] rs2    second ResultSet
    */
public static void returnTwoResultSets (ResultSet[] rs1, ResultSet[] rs2) throws Exception
{
  //get caller's connection to the database; inherited from StoredProc
  Connection con = DriverManager.getConnection("jdbc:default:connection");

  //define and process the first select statement
  Statement stmt1 = con.createStatement();
  String sql1 = "select value from table01 where index=1";
  rs1[0] = stmt1.executeQuery(sql1);

  //define and process the second select statement
  Statement stmt2 = con.createStatement();
  Stringsql2 = "select value from table01 where index=2";
  rs2[0] = stmt2.executeQuery(sql2);
  }
}
```

On the server, the additional result set parameters are not examined to determine the ordering of the results sets. The results sets on the server are returned in the order in which they were opened. To ensure compatibility with the SQLj standard, the result should be assigned in the order that they are opened, as previously shown.

DB2GENERAL parameter style

When coding a Java stored procedure that uses the DB2GENERAL parameter style, you must use these conventions.

- The class that defines a Java stored procedure must *extend*, or be a subclass of, the Java com.ibm.db2.app.StoredProc class.
- The Java method must be a public void instance method.
- The parameters of the Java method must be SQL-compatible types.
- A Java method may test for a SQL NULL value using the isNull method.

- The Java method must explicitly set the return parameters using the set method.
- The Java method may access the current database using the getConnection method.

A class that includes a Java stored procedure must extend the class, com.ibm.db2.app.StoredProc. Java stored procedures are public instance methods. Within the classes, the stored procedures are identified by their method name and signature. When you call a stored procedure, its signature is generated dynamically, based on the variable types defined by the CREATE PROCEDURE statement.

The com.ibm.db2.app.StoredProc class provides the isNull method, which permits a Java method to determine if an input parameter is an SQL NULL. The com.ibm.db2.app.StoredProc class also provides set...( ) methods that set output parameters. You must use these methods to set output parameters. If you do not set an output parameter, then the output parameter returns the SQL NULL value.

The com.ibm.db2.app.StoredProc class provides the following routine to fetch a JDBC connection to the embedding application context. A connection to the embedding application context is accessed using the following JDBC call:

```
public Java.sql.Connection getConnection( )
```

This connection then runs SQL statements with JDBC APIs.

The following is a small stored procedure with one input and two outputs. It processes the given SQL query, and returns both the number of rows in the result and the SQLSTATE.

Example: Stored procedure with one input and two outputs

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
package mystuff;

import com.ibm.db2.app.*;
import java.sql.*;
public class sample2 extends StoredProc {
  public void donut(String query, int rowCount,
    String sqlstate) throws Exception {
  try {
    Statement s=getConnection().createStatement();
    ResultSet r=s.executeQuery(query);
    int counter=0;
    while(r.next()){
      counter++;
    }
    r.close(); s.close();
    set(2, counter);
    }catch(SQLException x){
    set(3, x.getSQLState());
    }
  }
}
```

To return a result set in procedures that use the DB2GENERAL parameter style, the result set and the responding statement must be left open at the end of the procedure. The result set that is returned must be closed by the client application. If multiple results sets are returned, they are returned in the order in which they were opened. For example, the following stored procedure returns two results sets.

Example: Stored procedure that returns two results sets

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
public void returnTwoResultSets() throws Exception
{
  // get caller's connection to the database; inherited from StoredProc
  Connection con = getConnection ();
  Statement stmt1 = con.createStatement ();
  String sql1 = "select value from table01 where index=1";
  ResultSet rs1 = stmt1.executeQuery(sql1);
  Statement stmt2 = con.createStatement();
  String sql2 = "select value from table01 where index=2";
```

```
   ResultSet rs2 = stmt2.executeQuery(sql2);
}
```

Restrictions on Java stored procedures

These restrictions apply to Java stored procedures.

- A Java stored procedure should not create additional threads. An additional thread may be created in a job only if the job is multithread capable. Because there is no guarantee that a job that calls an SQL stored procedure is multithread capable, a Java stored procedure should not create additional threads.
- You cannot use adopted authority to access Java class files.
- A Java stored procedure uses the same default version of the JDK as the `java` command. If needed, the version of the JDK used by a Java stored procedure can be changed using a SystemDefault.properties file.
- Since Blob and Clob classes reside in both the java.sql and com.ibm.db2.app packages, the programmer must use the entire name of these classes if both classes are used in the same program. The program must ensure that the Blob and Clob classes from the com.ibm.db2.app are used as the parameters passed to the stored procedure.
- When a Java stored procedure is created, the system generates a program in the library. This program is used to store the procedure definition. The program has a name that is generated by the system. This name can be obtained by examining the job log of the job that created the stored procedure. If the program object is saved and then restored, then the procedure definition is restored. If a Java stored procedure is to be moved from one system to another, you are responsible for moving the program that contains the procedure definition as well as the integrated file system file, which contains the Java class.
- A Java stored procedure cannot set the properties (for example, system naming) of the JDBC connection that is used to connect to the database. The default JDBC connection properties are always used, with the exception that the prefetch property is set to false.

Java user-defined scalar functions

A Java scalar function returns one value from a Java program to the database. For example, a scalar function could be created that returns the sum of two numbers.

Like Java stored procedures, Java scalar functions use one of two parameter styles, Java and DB2GENERAL. When coding a Java user-defined function (UDF), you must be aware of restrictions placed on creating Java scalar functions.

Parameter style Java

The Java parameter style is the style specified by the *SQLJ Part 1: SQL Routines* standard. When coding a Java UDF, use the following conventions.

- The Java method must be a public static method.
- The Java method must return an SQL compatible type. The return value is the result of the method.
- The parameters of the Java method must be SQL compatible types.
- The Java method may test for a SQL NULL for Java types that permit the null value.

For example, given a UDF called sample!test3 that returns INTEGER and takes arguments of type CHAR(5), BLOB(10K), and DATE, DB2 expects the Java implementation of the UDF to have the following signature:

```
import com.ibm.db2.app.*;
public class sample {
   public static int test3(String arg1, Blob arg2, Date arg3) { ... }
}
```

The parameters of a Java method must be SQL compatible types. For example, if a UDF is declared as taking arguments of SQL types t1, t2, and t3, and returning type t4, it is called as a Java method with the expected Java signature:

```
public static T4 name (T1 a, T2 b, T3 c)  { .....}
```

where:

- *name* is the method name
- T1 through T4 are the Java types that correspond to SQL types t1 through t4.
- *a*, *b*, and *c* are arbitrary variable names for the input arguments.

The correlation between SQL types and Java types is found in Parameter passing conventions for stored procedures and UDFs.

SQL NULL values are represented by Java variables that are not initialized. These variables have a Java null value if they are object types. If an SQL NULL is passed to a Java scalar data type, such as int, then an exception condition is raised.

To return a result from a Java UDF when using the JAVA parameter style, simply return the result from the method.

```
{ ....
  return value;
}
```

Like C modules used in UDFs and stored procedures, you cannot use the Java standard I/O streams (System.in, System.out, and System.err) in Java UDFs.

Parameter style DB2GENERAL

Parameter style DB2GENERAL is used by Java UDFs. In this parameter style, the return value is passed as the last parameter of the function and must be set using a *set* method of the com.ibm.db2.app.UDF class.

When coding a Java UDF, the following conventions must be followed:

- The class, which includes the Java UDF, must *extend*, or be a subclass of, the Java com.ibm.db2.app.UDF class.
- For the DB2GENERAL parameter style, the Java method must be a public void instance method.
- The parameters of the Java method must be SQL-compatible types.
- The Java method may test for an SQL NULL value using the isNull method.
- For the DB2GENERAL parameter style, the Java method must explicitly set the return parameter using the set() method.

A class that includes a Java UDF must extend the Java class, com.ibm.db2.app.UDF. A Java UDF that uses the DB2GENERAL parameter style must be a void instance method of the Java class. For example, for a UDF called sample!test3 that returns INTEGER and takes arguments of type CHAR(5), BLOB(10K), and DATE, DB2 expects the Java implementation of the UDF to have the following signature:

```
import com.ibm.db2.app.*;
public class sample extends UDF {
   public void test3(String arg1, Blob arg2, String arg3, int result) { ... }
}
```

The parameters of a Java method must be SQL types. For example, if a UDF is declared as taking arguments of SQL types t1, t2, and t3, returning type t4, it is called as a Java method with the expected Java signature:

```
public void name (T1 a, T2 b, T3 c, T4 d)  { .....}
```

where:

- *name* is the method name
- T1 through T4 are the Java types that correspond to SQL types t1 through t4.
- *a*, *b*, and *c* are arbitrary variable names for the input arguments.
- *d* is an arbitrary variable name that represents the UDF result being computed.

The correlation between SQL types and Java types is given in the section, Parameter passing conventions for stored procedures and UDFs.

SQL NULL values are represented by Java variables that are not initialized. These variables have a value of zero if they are primitive types, and Java null if they are object types, in accordance with Java rules. To tell an SQL NULL apart from an ordinary zero, the isNull method can be called for any input argument:

```
{ ....
  if (isNull(1)) { /* argument #1 was a SQL NULL */ }
  else           { /* not NULL */ }
}
```

In the previous example, the argument numbers start at one. The isNull() function, like the other functions that follow, are inherited from the com.ibm.db2.app.UDF class. To return a result from a Java UDF when using the DB2GENERAL parameter style, use the set() method in the UDF, as follows:

```
{ ....
  set(2, value);
}
```

Where 2 is the index of an output argument, and *value* is a literal or variable of a compatible type. The argument number is the index in the argument list of the selected output. In the first example in this section, the int result variable has an index of 4. An output argument that is not set before the UDF returns has a NULL value.

Like C modules used in UDFs and stored procedures, you cannot use the Java standard I/O streams (System.in, System.out, and System.err) in Java UDFs.

Typically, DB2 calls a UDF many times, once for each row of an input or result set in a query. If SCRATCHPAD is specified in the CREATE FUNCTION statement of the UDF, DB2 recognizes that some "continuity" is needed between successive invocations of the UDF, and therefore, for DB2GENERAL parameter style functions, the implementing Java class is not instantiated for each call, but generally speaking once per UDF reference per statement. If, however, NO SCRATCHPAD is specified for a UDF, then a clean instance is instantiated for each call to the UDF, by means of a call to the class constructor.

A scratchpad may be useful for saving information across calls to a UDF. Java UDFs can either use instance variables or set the scratchpad to achieve continuity between calls. Java UDFs access the scratchpad with the getScratchPad and setScratchPad methods available in com.ibm.db2.app.UDF. At the end of a query, if you specify the FINAL CALL option on the CREATE FUNCTION statement, the object's public void close() method is called (for DB2GENERAL parameter style functions). If you do not define this method, a stub function takes over and the event is ignored. The com.ibm.db2.app.UDF class contains useful variables and methods that you can use within a DB2GENERAL parameter style UDF. These variables and methods are explained in the following table.

Variables and Methods	**Description**
• public static final int SQLUDF_FIRST_CALL = -1; • public static final int SQLUDF_NORMAL_CALL = 0; • public static final int SQLUDF_TF_FIRST = -2; • public static final int SQLUDF_TF_OPEN = -1; • public static final int SQLUDF_TF_FETCH = 0; • public static final int SQLUDF_TF_CLOSE = 1; • public static final int SQLUDF_TF_FINAL = 2;	For scalar UDFs, these are constants to determine if the call is a first call or a normal call. For table UDFs, these are constants to determine if the call is a first call, open call, fetch call, close call, or final call.

Variables and Methods	Description
public Connection getConnection();	The method obtains the JDBC connection handle for this stored procedure call and returns a JDBC object that represents the calling application's connection to the database. It is analogous to the result of a null SQLConnect() call in a C stored procedure.
public void close();	This method is called by the database at the end of a UDF evaluation, if the UDF was created with the FINAL CALL option. It is analogous to the final call for a C UDF. If a Java UDF class does not implement this method, this event is ignored.
public boolean isNull(int i)	This method tests whether an input argument with the given index is an SQL NULL.
• public void set(int i, short s); • public void set(int i, int j); • public void set(int i, long j); • public void set(int i, double d); • public void set(int i, float f); • public void set(int i, BigDecimal bigDecimal); • public void set(int i, String string); • public void set(int i, Blob blob); • public void set(int i, Clob clob); • public boolean needToSet(int i);	These methods set an output argument to the given value. An exception is thrown if anything goes wrong, including the following: • UDF call is not in progress • Index does not refer to valid output argument • Data type does not match • Data length does not match • Code page conversion error occurs
public void setSQLstate(String string);	This method may be called from a UDF to set the SQLSTATE to be returned from this call. If the string is not acceptable as an SQLSTATE, an exception is thrown. The user may set the SQLSTATE in the external program to return an error or warning from the function. In this case, the SQLSTATE must contain one of the following: • '00000' to indicate success • '01Hxx', where xx is any two digits or uppercase letters, to indicate a warning • '38yxx', where y is an uppercase letter between 'I' and 'Z' and xx is any two digits or uppercase letters, to indicate an error
public void setSQLmessage(String string);	This method is similar to the setSQLstate method. It sets the SQL message result. If the string is not acceptable (for example, longer than 70 characters), an exception is thrown.
public String getFunctionName();	This method returns the name of the processing UDF.
public String getSpecificName();	This method returns the specific name of the processing UDF.
public byte[] getDBinfo();	This method returns a raw, unprocessed DBINFO structure for the processing UDF, as a byte array. The UDF must have been registered (using CREATE FUNCTION) with the DBINFO option.

Variables and Methods	Description
• public String getDBname(); • public String getDBauthid(); • public String getDBver_rel(); • public String getDBplatform(); • public String getDBapplid(); • public String getDBapplid(); • public String getDBtbschema(); • public String getDBtbname(); • public String getDBcolname();	These methods return the value of the appropriate field from the DBINFO structure of the processing UDF. The UDF must have been registered (using CREATE FUNCTION) with the DBINFO option. The getDBtbschema(), getDBtbname(), and getDBcolname() methods only return meaningful information if a user-defined function is specified on the right side of a SET clause in an UPDATE statement.
public int getCCSID();	This method returns the CCSID of the job.
public byte[] getScratchpad();	This method returns a copy of the scratchpad of the currently processing UDF. You must first declare the UDF with the SCRATCHPAD option.
public void setScratchpad(byte ab[]);	This method overwrites the scratchpad of the currently processing UDF with the contents of the given byte array. You must first declare the UDF with the SCRATCHPAD option. The byte array must have the same size as getScratchpad() returns.
public int getCallType();	This method returns the type of call that is currently being made. These values correspond to the C values defined in sqludf.h. Possible return values include the following: • SQLUDF_FIRST_CALL • SQLUDF_NORMAL_CALL • SQLUDF_TF_FIRST • SQLUDF_TF_OPEN • SQLUDF_TF_FETCH • SQLUDF_TF_CLOSE • SQLUDF_TF_FINAL

Related concepts

Java stored procedures
When using Java to write stored procedures, you can use two possible parameter passing styles.

SQLJ procedures that manipulate JAR files
Both Java stored procedures and Java UDFs can use Java classes that are stored in Java JAR files.

Parameter passing conventions for Java stored procedures and UDFs
The following table lists how SQL data types are represented in Java stored procedures and UDFs.

Related tasks

Using Java SQL routines

You can access Java programs from SQL statements and programs. This can be done using Java stored procedures and Java user-defined functions (UDFs).

Restrictions on Java user-defined functions

These restrictions apply to Java user-defined functions (UDFs).

- A Java UDF should not create additional threads. An additional thread may be created in a job only if the job is multithread capable. Since it cannot be guaranteed that a job that calls an SQL stored procedure is multithread capable, a Java stored procedure should not create additional threads.
- The complete name of the Java stored procedure defined to the database is limited to 279 characters. This limit is a consequence of the EXTERNAL_NAME column, which has a maximum width of 279 characters.
- Adopted authority cannot be used to access Java class files.
- A Java UDF always uses the latest version of the JDK that is installed on the system.
- Since Blob and Clob classes reside in both the java.sql and com.ibm.db2.app packages, the programmer must use the entire name of these classes if both classes are used in the same program. The program must ensure that the Blob and Clob classes from the com.ibm.db2.app are used as the parameters passed to the stored procedure.
- Like sourced functions, when a Java UDF is created, a service program in the library is used to store the function definition. The name of the service program is generated by the system and can be found in the job log of the job that created the function. If this object is saved and then restored to another system, then the function definition is restored. If a Java UDF is to be moved from one system to another, you are responsible for moving the service program that contains the function definition as well as the integrated file system file that contains the Java class.
- A Java UDF cannot set the properties (for example, system naming) of the JDBC connection that is used to connect to the database. The default JDBC connection properties are always used, except when prefetching is disabled.

Related concepts

Java user-defined table functions
DB2 provides the ability for a function to return a table. This is useful for exposing information from outside the database to the database in table form. For example, a table can be created that exposes the properties set in the Java virtual machine (JVM) used for Java stored procedures and Java UDFs (both table and scalar).

Java user-defined table functions

DB2 provides the ability for a function to return a table. This is useful for exposing information from outside the database to the database in table form. For example, a table can be created that exposes the properties set in the Java virtual machine (JVM) used for Java stored procedures and Java UDFs (both table and scalar).

The *SQLJ Part 1: SQL Routines* standard does support table functions. Consequently, table functions are only available using parameter style DB2GENERAL.

Five different types of calls are made to a table function. The following table explains these calls. These assume that scratchpad has been specified on the create function SQL statement.

Point in scan time	NO FINAL CALL LANGUAGE JAVA SCRATCHPAD	FINAL CALL LANGUAGE JAVA SCRATCHPAD
Before the first OPEN of the table function	No calls	Class constructor is called (means new scratchpad). UDF method is called with FIRST call.
At each OPEN of the table function.	Class constructor is called (means new scratchpad). UDF method is called with OPEN all.	UDF method is called with OPEN call.

Point in scan time	NO FINAL CALL LANGUAGE JAVA SCRATCHPAD	FINAL CALL LANGUAGE JAVA SCRATCHPAD
At each FETCH for a new row of table function data.	UDF method is called with FETCH call.	UDF method is called with FETCH call.
At each CLOSE of the table function	UDF method is called with CLOSE call. The close() method, if it exists, is also called.	UDF method is called with CLOSE call.
After the last CLOSE of the table function.	No calls	UDF method is called with FINAL call. The close() method, if it exists, is also called.

Example: Java table function

The following is an example of a Java table function that determines the properties set in the JVM used to run the Java user-defined table function.

Note: Read the Code example disclaimer for important legal information.

```
import com.ibm.db2.app.*;
import java.util.*;

public class JVMProperties extends UDF {
 Enumeration propertyNames;
 Properties properties ;

 public void dump (String property, String value) throws Exception
 {
   int callType = getCallType();
   switch(callType) {
     case SQLUDF_TF_FIRST:
       break;
     case SQLUDF_TF_OPEN:
       properties = System.getProperties();
       propertyNames = properties.propertyNames();
       break;
     case SQLUDF_TF_FETCH:
       if (propertyNames.hasMoreElements()) {
           property = (String) propertyNames.nextElement();
           value = properties.getProperty(property);
           set(1, property);
           set(2, value);
       } else {
          setSQLstate("02000");
       }
       break;
     case SQLUDF_TF_CLOSE:
       break;
     case SQLUDF_TF_FINAL:
       break;
     default:
       throw new Exception("UNEXPECT call type of "+callType);
   }
 }
}
```

After the table function is compiled, and its class file copied to /QIBM/UserData/OS400/SQLLib/Function, the function can be registered to the database by using the following SQL statement.

```
create function properties()
returns table (property varchar(500), value varchar(500))
external name 'JVMProperties.dump' language java
parameter style db2general fenced no sql
disallow parallel scratchpad
```

After the function has been registered, it can be used as part of an SQL statement. For example, the following SELECT statement returns the table generated by the table function.

```
SELECT * FROM TABLE(PROPERTIES())
```

Related concepts

Restrictions on Java user-defined functions
These restrictions apply to Java user-defined functions (UDFs).

SQLJ procedures that manipulate JAR files

Both Java stored procedures and Java UDFs can use Java classes that are stored in Java JAR files.

To use a JAR file, a *jar-id* must be associated with the JAR file. The system provides stored procedures in the SQLJ schema that allow *jar-ids* and JAR files to be manipulated. These procedures allow JAR files to be installed, replaced, and removed. They also provide the ability to use and update the SQL catalogs associated with JAR files.

Related concepts

Java stored procedures
When using Java to write stored procedures, you can use two possible parameter passing styles.

Java user-defined scalar functions
A Java scalar function returns one value from a Java program to the database. For example, a scalar function could be created that returns the sum of two numbers.

Parameter passing conventions for Java stored procedures and UDFs
The following table lists how SQL data types are represented in Java stored procedures and UDFs.

Related tasks

Using Java SQL routines
You can access Java programs from SQL statements and programs. This can be done using Java stored procedures and Java user-defined functions (UDFs).

SQLJ.INSTALL_JAR

The SQLJ.INSTALL_JAR stored procedure installs a JAR file into the database system. This JAR file can be used in subsequent CREATE FUNCTION and CREATE PROCEDURE statements.

Authorization

The privilege held by the authorization ID of the CALL statement must include at least one of the following for the SYSJAROBJECTS and SYSJARCONTENTS catalog tables:

- The following system authorities:
  - The INSERT and SELECT privileges on the table
  - The system authority *EXECUTE on library QSYS2
- Administrative authority

The privilege held by the authorization ID of the CALL statement must also have the following authorities:

- Read (*R) access to the JAR file specified in the *jar-url* parameter being installed.
- Write, Execute, and Read (*RWX) access to the directory where the JAR file is installed. This directory is /QIBM/UserData/OS400/SQLLib/Function/jar/*schema*, where *schema* is the schema of the *jar-id*.

Adopted authority cannot be used for these authorities.

SQL syntax

```
>>-CALL--SQLJ.INSTALL_JAR-- (--'jar-url'--,--'jar-id'--,--deploy--)-->

>--------------------------------------------------------------><
```

Description

jar-url
The URL containing the JAR file to be installed or replaced. The only URL scheme supported is 'file:'.

jar-id
: The JAR identifier in the database to be associated with the file specified by the *jar-url*. The *jar-id* uses SQL naming and the JAR file is installed in the schema or library specified by the implicit or explicit qualifier.

deploy
: Value used to describe the install_action of the deployment descriptor file. If this integer is a nonzero value, then the install_actions of a deployment descriptor file should be run at the end of the install_jar procedure. The current version of Db2 for i only supports a value of zero.

Usage notes

When a JAR file is installed, Db2 for i registers the JAR file in the SYSJAROBJECTS system catalog. It also extracts the names of the Java class files from the JAR file and registers each class in the SYSJARCONTENTS system catalog. Db2 for i copies the JAR file to a jar/schema subdirectory of the /QIBM/UserData/OS400/SQLLib/Function directory. Db2 for i gives the new copy of the JAR file the name given in the *jar-id* clause. A JAR file that has been installed by Db2 for i into a subdirectory of /QIBM/UserData/OS400/SQLLib/Function/jar should not be changed. Instead, the CALL SQLJ.REMOVE_JAR and CALL SQLJ.REPLACE_JAR SQL commands should be used to remove or replace an installed JAR file.

Example

The following command is issued from an SQL interactive session.

```
CALL SQLJ.INSTALL_JAR('file:/home/db2inst/classes/Proc.jar' , 'myproc_jar', 0)
```

The Proc.jar file located in the file:/home/db2inst/classes/ directory is installed into Db2 for i with the name myproc_jar. Subsequent SQL commands that use the Procedure.jar file refer to it with the name myproc_jar.

Related reference

SQLJ.REMOVE_JAR
The SQLJ.REMOVE_JAR stored procedure removes a JAR file from the database system.

SQLJ.REPLACE_JAR
The SQLJ.REPLACE_JAR stored procedure replaces a JAR file into the database system.

SQLJ.UPDATEJARINFO
The SQLJ.UPDATEJARINFO updates the CLASS_SOURCE column of the SYSJARCONTENTS catalog table. This procedure is not part of the SQLJ standard but is used by the Db2 for i stored procedure builder.

SQLJ.RECOVERJAR
The SQLJ.RECOVERJAR procedure takes the JAR file that is stored in the SYSJAROBJECTS catalog and restores it to the /QIBM/UserData/OS400/SQLLib/Function/jar/*jarschema*/*jar_id*.jar file.

SQLJ.REFRESH_CLASSES
The SQLJ.REFRESH_CLASSES stored procedure causes the reloading of user defined classes used by Java stored procedures or Java UDFs in the current database connection. This stored procedure must be called by existing database connections to obtain changes made by a call to the SQLJ.REPLACE_JAR stored procedure.

SQLJ.REMOVE_JAR

The SQLJ.REMOVE_JAR stored procedure removes a JAR file from the database system.

Authorization

The privilege held by the authorization ID of the CALL statement must include at least one of the following for the SYSJARCONTENTS and SYSJAROBJECTS catalog tables:

- The following system authorities:
  - The SELECT and DELETE privileges on the table

- The system authority *EXECUTE on library QSYS2
- Administrative authority

The privilege held by the authorization ID of the CALL statement must also have the following authority.

- *OBJMGT authority to the JAR file being removed. The JAR file is named /QIBM/UserData/OS400/SQLLib/Function/jar/schema/jarfile.

Adopted authority cannot be used for this authority.

Syntax

```
>>-CALL--SQLJ.REMOVE_JAR--(--'jar-id'--,--undeploy--)----------><
```

Description

jar-id
The JAR identifier of the JAR file that is to be removed from the database.

undeploy
The value used to describe the remove_action of the deployment descriptor file. If this integer is a non-zero value, then the remove_actions of a deployment descriptor file should be run at the end of the install_jar procedure. The current version of Db2 for i only supports a value of zero.

Example

The following command is issued from an SQL interactive session:

```
CALL SQLJ.REMOVE_JAR('myProc_jar', 0)
```

The JAR file myProc_jar is removed from the database.

Related reference

SQLJ.INSTALL_JAR
The SQLJ.INSTALL_JAR stored procedure installs a JAR file into the database system. This JAR file can be used in subsequent CREATE FUNCTION and CREATE PROCEDURE statements.

SQLJ.REPLACE_JAR
The SQLJ.REPLACE_JAR stored procedure replaces a JAR file into the database system.

SQLJ.UPDATEJARINFO
The SQLJ.UPDATEJARINFO updates the CLASS_SOURCE column of the SYSJARCONTENTS catalog table. This procedure is not part of the SQLJ standard but is used by the Db2 for i stored procedure builder.

SQLJ.RECOVERJAR
The SQLJ.RECOVERJAR procedure takes the JAR file that is stored in the SYSJAROBJECTS catalog and restores it to the /QIBM/UserData/OS400/SQLLib/Function/jar/*jarschema*/*jar_id*.jar file.

SQLJ.REFRESH_CLASSES
The SQLJ.REFRESH_CLASSES stored procedure causes the reloading of user defined classes used by Java stored procedures or Java UDFs in the current database connection. This stored procedure must be called by existing database connections to obtain changes made by a call to the SQLJ.REPLACE_JAR stored procedure.

SQLJ.REPLACE_JAR

The SQLJ.REPLACE_JAR stored procedure replaces a JAR file into the database system.

Authorization

The privilege held by the authorization ID of the CALL statement must include at least one of the following for the SYSJAROBJECTS and SYSJARCONTENTS catalog tables:

- The following system authorities:
  - The SELECT, INSERT, and DELETE privileges on the table
  - The system authority *EXECUTE on library QSYS2
- Administrative authority

The privilege held by the authorization ID of the CALL statement must also have the following authorities:

- Read (*R) access to the JAR file specified by the *jar-url* parameter being installed.
- *OBJMGT authority to the JAR file being removed. The JAR file is named /QIBM/UserData/OS400/SQLLib/Function/jar/schema/jarfile.

Adopted authority cannot be used for these authorities.

Syntax

```
>>-CALL--SQLJ.REPLACE_JAR--(--'jar-url'--,--'jar-id'--)--------><
```

Description

jar-url
The URL containing the JAR file to be replaced. The only URL scheme supported is 'file:'.

jar-id
The JAR identifier in the database to be associated with the file specified by the *jar-url*. The *jar-id* uses SQL naming and the JAR file is installed in the schema or library specified by the implicit or explicit qualifier.

Usage notes

The SQLJ.REPLACE_JAR stored procedure replaces a JAR file that was previously installed in the database using SQLJ.INSTALL_JAR.

Example

The following command is issued from an SQL interactive session:

```
CALL SQLJ.REPLACE_JAR('file:/home/db2inst/classes/Proc.jar' , 'myproc_jar')
```

The current JAR file referred to by the *jar-id* myproc_jar is replaced with the Proc.jar file located in the file:/home/db2inst/classes/ directory.

Related reference

SQLJ.INSTALL_JAR
The SQLJ.INSTALL_JAR stored procedure installs a JAR file into the database system. This JAR file can be used in subsequent CREATE FUNCTION and CREATE PROCEDURE statements.

SQLJ.REMOVE_JAR
The SQLJ.REMOVE_JAR stored procedure removes a JAR file from the database system.

SQLJ.UPDATEJARINFO
The SQLJ.UPDATEJARINFO updates the CLASS_SOURCE column of the SYSJARCONTENTS catalog table. This procedure is not part of the SQLJ standard but is used by the Db2 for i stored procedure builder.

SQLJ.RECOVERJAR
The SQLJ.RECOVERJAR procedure takes the JAR file that is stored in the SYSJAROBJECTS catalog and restores it to the /QIBM/UserData/OS400/SQLLib/Function/jar/*jarschema*/*jar_id*.jar file.

SQLJ.REFRESH_CLASSES
The SQLJ.REFRESH_CLASSES stored procedure causes the reloading of user defined classes used by Java stored procedures or Java UDFs in the current database connection. This stored procedure must be called

by existing database connections to obtain changes made by a call to the SQLJ.REPLACE_JAR stored procedure.

SQLJ.UPDATEJARINFO

The SQLJ.UPDATEJARINFO updates the CLASS_SOURCE column of the SYSJARCONTENTS catalog table. This procedure is not part of the SQLJ standard but is used by the Db2 for i stored procedure builder.

Authorization

The privilege held by the authorization ID of the CALL statement must include at least one of the following for the SYSJARCONTENTS catalog table:

- The following system authorities:
  - The SELECT and UPDATEINSERT privileges on the table
  - The system authority *EXECUTE on library QSYS2
- Administrative authority

The user running the CALL statement must also have the following authorities:

- Read (*R) access to the JAR file specified in the *jar-url* parameter. Read (*R) access to the JAR file being installed.
- Write, Execute, and Read (*RWX) access to the directory where the JAR file is installed. This directory is /QIBM/UserData/OS400/SQLLib/Function/jar/*schema*, where *schema* is the schema of the *jar-id*.

Adopted authority cannot be used for these authorities.

Syntax

```
>>-CALL--SQLJ.UPDATEJARINFO--(--'jar-id'--,--'class-id'--,--'jar-url'--)-->

>--------------------------------------------------------------><
```

Description

jar-id
: The JAR identifier in the database that is to be updated.

class-id
: The package qualified class name of the class to be updated.

jar-url
: The URL containing the classfile to update the JAR file with. The only URL scheme supported is 'file:'.

Example

The following command is issued from an SQL interactive session:

```
CALL SQLJ.UPDATEJARINFO('myproc_jar', 'mypackage.myclass',
                        'file:/home/user/mypackage/myclass.class')
```

The JAR file associated with the *jar-id* myproc_jar, is updated with a new version of the mypackage.myclass class. The new version of the class is obtained from the file /home/user/mypackage/myclass.class.

Related reference

SQLJ.INSTALL_JAR
The SQLJ.INSTALL_JAR stored procedure installs a JAR file into the database system. This JAR file can be used in subsequent CREATE FUNCTION and CREATE PROCEDURE statements.

SQLJ.REMOVE_JAR

The SQLJ.REMOVE_JAR stored procedure removes a JAR file from the database system.

SQLJ.REPLACE_JAR
The SQLJ.REPLACE_JAR stored procedure replaces a JAR file into the database system.

SQLJ.RECOVERJAR
The SQLJ.RECOVERJAR procedure takes the JAR file that is stored in the SYSJAROBJECTS catalog and restores it to the /QIBM/UserData/OS400/SQLLib/Function/jar/*jarschema*/*jar_id*.jar file.

SQLJ.REFRESH_CLASSES
The SQLJ.REFRESH_CLASSES stored procedure causes the reloading of user defined classes used by Java stored procedures or Java UDFs in the current database connection. This stored procedure must be called by existing database connections to obtain changes made by a call to the SQLJ.REPLACE_JAR stored procedure.

SQLJ.RECOVERJAR

The SQLJ.RECOVERJAR procedure takes the JAR file that is stored in the SYSJAROBJECTS catalog and restores it to the /QIBM/UserData/OS400/SQLLib/Function/jar/*jarschema*/*jar_id*.jar file.

Authorization

The privilege held by the authorization ID of the CALL statement must include at least one of the following for the SYSJAROBJECTS catalog table:

- The following system authorities:
  - The SELECT and UPDATEINSERT privileges on the table
  - The system authority *EXECUTE on library QSYS2
- Administrative authority

The user running the CALL statement must also have the following authorities:

- Write, Execute, and Read (*RWX) access to the directory where the JAR file is installed. This directory is /QIBM/UserData/OS400/SQLLib/Function/jar/*schema*, where *schema* is the schema of the *jar-id*.
- *OBJMGT authority to the JAR file being removed. The JAR file is named /QIBM/UserData/OS400/SQLLib/Function/jar/schema/jarfile.

Syntax

```
>>-CALL--SQLJ.RECOVERJAR--(--'jar-id'--)-----------------------><
```

Description

jar-id
The JAR identifier in the database that is to be recovered.

Example

The following command is issued from a SQL interactive session:

```
CALL SQLJ.UPDATEJARINFO('myproc_jar')
```

The JAR file associated with the myproc_jar is updated with the contents from SYSJARCONTENT table. The file is copied to /QIBM/UserData/OS400/SQLLib/Function/jar/jar_schema myproc_jar.jar.

Related reference

SQLJ.INSTALL_JAR
The SQLJ.INSTALL_JAR stored procedure installs a JAR file into the database system. This JAR file can be used in subsequent CREATE FUNCTION and CREATE PROCEDURE statements.

SQLJ.REMOVE_JAR

The SQLJ.REMOVE_JAR stored procedure removes a JAR file from the database system.

SQLJ.REPLACE_JAR
The SQLJ.REPLACE_JAR stored procedure replaces a JAR file into the database system.

SQLJ.UPDATEJARINFO
The SQLJ.UPDATEJARINFO updates the CLASS_SOURCE column of the SYSJARCONTENTS catalog table. This procedure is not part of the SQLJ standard but is used by the Db2 for i stored procedure builder.

SQLJ.REFRESH_CLASSES
The SQLJ.REFRESH_CLASSES stored procedure causes the reloading of user defined classes used by Java stored procedures or Java UDFs in the current database connection. This stored procedure must be called by existing database connections to obtain changes made by a call to the SQLJ.REPLACE_JAR stored procedure.

SQLJ.REFRESH_CLASSES

The SQLJ.REFRESH_CLASSES stored procedure causes the reloading of user defined classes used by Java stored procedures or Java UDFs in the current database connection. This stored procedure must be called by existing database connections to obtain changes made by a call to the SQLJ.REPLACE_JAR stored procedure.

Authorization

NONE

Syntax

```
>>-CALL--SQLJ.REFRESH_CLASSES-- ()-->
>--------------------------------------------------------------><
```

Example

Call a Java stored procedure, MYPROCEDURE, that uses a class in a JAR file registered with the MYJAR jarid:

```
CALL MYPROCEDURE()
```

Replace the JAR file using the following call:

```
CALL SQLJ.REPLACE_JAR('MYJAR', '/tmp/newjarfile.jar')
```

In order for subsequence calls to the MYPROCEDURE stored procedure to use the updated JAR file, SQLJ.REFRESH_CLASSES must be called:

```
CALL SQLJ.REFRESH_CLASSES()
```

Call the stored procedure again. The updated class files are used when the procedure is called.

```
CALL MYPROCEDURE()
```

Related reference

SQLJ.INSTALL_JAR
The SQLJ.INSTALL_JAR stored procedure installs a JAR file into the database system. This JAR file can be used in subsequent CREATE FUNCTION and CREATE PROCEDURE statements.

SQLJ.REMOVE_JAR
The SQLJ.REMOVE_JAR stored procedure removes a JAR file from the database system.

SQLJ.REPLACE_JAR

The SQLJ.REPLACE_JAR stored procedure replaces a JAR file into the database system.

SQLJ.UPDATEJARINFO
The SQLJ.UPDATEJARINFO updates the CLASS_SOURCE column of the SYSJARCONTENTS catalog table. This procedure is not part of the SQLJ standard but is used by the Db2 for i stored procedure builder.

SQLJ.RECOVERJAR
The SQLJ.RECOVERJAR procedure takes the JAR file that is stored in the SYSJAROBJECTS catalog and restores it to the /QIBM/UserData/OS400/SQLLib/Function/jar/*jarschema*/*jar_id*.jar file.

Parameter passing conventions for Java stored procedures and UDFs

The following table lists how SQL data types are represented in Java stored procedures and UDFs.

SQL data type	Java parameter style JAVA	Java parameter style DB2GENERAL
SMALLINT	short	short
INTEGER	int	int
BIGINT	long	long
DECIMAL(p,s)	BigDecimal	BigDecimal
NUMERIC(p,s)	BigDecimal	BigDecimal
REAL or FLOAT(p)	float	float
DOUBLE PRECISION or FLOAT or FLOAT(p)	double	double
CHARACTER(n)	String	String
CHARACTER(n) FOR BIT DATA	byte[]	com.ibm.db2.app.Blob
VARCHAR(n)	String	String
VARCHAR(n) FOR BIT DATA	byte[]	com.ibm.db2.app.Blob
GRAPHIC(n)	String	String
VARGRAPHIC(n)	String	String
DATE	Date	String
TIME	Time	String
TIMESTAMP	Timestamp	String
Indicator Variable	-	-
CLOB	java.sql.Clob	com.ibm.db2.app.Clob
BLOB	java.sql.Blob	com.ibm.db2.app.Blob
DBCLOB	java.sql.Clob	com.ibm.db2.app.Clob
DataLink	-	-
ARRAY	java.sql.Array	-

Related concepts

Java stored procedures
When using Java to write stored procedures, you can use two possible parameter passing styles.

Java user-defined scalar functions

A Java scalar function returns one value from a Java program to the database. For example, a scalar function could be created that returns the sum of two numbers.

SQLJ procedures that manipulate JAR files
Both Java stored procedures and Java UDFs can use Java classes that are stored in Java JAR files.

Related tasks

Using Java SQL routines
You can access Java programs from SQL statements and programs. This can be done using Java stored procedures and Java user-defined functions (UDFs).

Java with other programming languages

With Java, you have multiple ways to call code that was written in languages other than Java.

The IBM i Java environment is separate from the integrated language environment (ILE). Java is not an ILE language, and it cannot bind to ILE object modules to create programs or service programs. The following table shows some of the differences between ILE-based programs and Java programs:

ILE	Java
Members that are part of the library or file structure on an IBM i server store source codes.	Stream files in the integrated file system contain source code.
Source entry utility (SEU) edits extended binary-coded decimal interchange code (EBCDIC) source files.	American Standard Code for Information Interchange (ASCII) source files are usually edited using a workstation editor.
Source files compile into object code modules, which are stored in libraries on an IBM i server.	Source code compiles into class files, which the integrated file system stores.
Object modules are statically bound together in programs or service programs.	Classes are dynamically loaded, as needed, at runtime.
You can directly call to functions that are written in other ILE programming languages.	Java Native Interface must be used to call other languages from Java.
ILE languages are always compiled and run as machine instructions.	Java programs can be interpreted or compiled.

Note: If portability is a concern, avoid using a solution that is not "pure" Java.

Related concepts

“Java Invocation API” on page 211
The Invocation API, which is part of the Java Native Interface (JNI), allows non-Java code to create a Java virtual machine, and load and use Java classes. This function lets a multithreaded program make use of Java classes that are running in a single Java virtual machine in multiple threads.

“Using sockets for interprocess communication” on page 219
Sockets streams communicate between programs that are running in separate processes.

“Using input and output streams for interprocess communication” on page 222
Input and output streams communicate between programs that are running in separate processes.

Related reference

“Example: Calling Java from ILE C” on page 223
This is an example of an integrated language environment (ILE) C program that uses the `system()` function to call the Java Hello program.

“Example: Calling Java from RPG” on page 224
This is an example of an RPG program that uses the QCMDEXC API to call the Java Hello program.

Related information

IBM Toolbox for Java

Native methods and the Java Native Interface

Native methods are Java methods that start in a language other than Java. Native methods can access system-specific functions and APIs that are not available directly in Java.

The use of native methods limits the portability of an application, because it involves system-specific code. Native methods can either be new native code statements or native code statements that call existing native code.

Once you decide that a native method is required, it may have to interoperate with the Java virtual machine where it runs. The Java Native Interface (JNI) facilitates this interoperability in a platform-neutral way.

The JNI is a set of interfaces that permit a native method to interoperate with the Java virtual machine in numerous ways. For example, the JNI includes interfaces that create new objects and call methods, get fields and set fields, process exceptions, and manipulate strings and arrays.

For a complete description of the JNI, refer to Java Native Interface by Oracle.

Related concepts

Java Invocation API
The Invocation API, which is part of the Java Native Interface (JNI), allows non-Java code to create a Java virtual machine, and load and use Java classes. This function lets a multithreaded program make use of Java classes that are running in a single Java virtual machine in multiple threads.

Using java.lang.Runtime.exec()
Use the `java.lang.Runtime.exec()` method to call programs or commands from within your Java program. Using `java.lang.Runtime.exec()` method creates one or more additional thread-enabled jobs. The additional jobs process the command string that you pass on the method.

Interprocess communications
When communicating with programs that are running in another process, there are a number of options.

Related information

Java Native Interface by Oracle.

Getting started with Java native methods

You should only use native methods in cases where pure Java cannot meet your programming needs.

Limit the use of native methods by only using them under these circumstances:

- To access system functions that are not available using pure Java.
- To implement performance-sensitive methods that can benefit significantly from a native implementation.
- To interface to existing application programming interfaces (API) that allow Java to call other APIs.

The following instructions apply to using the Java Native Interface (JNI) with the C language. For information about using JNI with the RPG language, see Chapter 11 of the WebSphere Development Studio: ILE RPG Programmer's Guide, SC09-2507.

Note: The term native library or native method library refers to integrated language environment (ILE) service programs when used in the context of ILE native methods, and AIX static or shared libraries when used in the context of PASE for i native methods.

To create Java native methods, do these steps:

1. Create the Java class and specify which methods are native methods using the standard Java language syntax.

   In the static initializer for the class, you need to add code that loads the native library which contains the C implementation of the native methods. You can use either the `System.load()` or `System.loadLibrary()` Java methods to load the native library. The `System.load()` method takes as a parameter a fully qualified path to the native library and loads the specified native library. The `System.loadLibrary()` takes as parameter a library name, locates a native library that corresponds to that name, and loads the native library. For information about how a native library is located by the `System.loadLibrary()` method, see "Managing native method libraries" on page 208.

You need to be aware of the following library naming conventions:

- If the native methods are ILE native methods and the Java code loads a library named `Sample`, the corresponding executable file must be an ILE service program named SAMPLE. The following shows how you would load the ILE native library:

  ```
  System.loadLibrary("Sample");

  System.load("/qsys.lib/mylib.lib/Sample.srvpgm");
  ```

  Note: A symbolic link to a service program can be used in these library loading methods.

- If the native methods are PASE for i native methods and the Java code loads a library named `Sample`, the corresponding executable file must be an AIX library named either `libSample.a` or `libSample.so`. The following shows how you would load the PASE for i native library:

  ```
  System.loadLibrary("Sample");

  System.load("/somedir/libSample.so");
  ```

2. Use the `javac` tool to compile the Java source into a class file.
3. Use the `javah` tool to create the header file (.h). This header file contains the exact prototypes for creating the native method implementations. The -d option specifies the directory where you should create the header file.
4. Write the C implementation code for the native method. See the "Java native methods and threads considerations" on page 210 topic for details about the languages and functions that are used for native methods.
   a) Include the header file that was created in the previous steps.
   b) Match the prototypes in the header file exactly.
   c) If your native method must interact with the Java virtual machine, use the functions that are provided with JNI.
   d) For ILE native methods only, convert strings to American Standard Code for Information Interchange (ASCII) if the strings are to be passed to the Java virtual machine. For more information, see "Strings in ILE native methods" on page 205.
5. Compile your C implementation code for the native method into a native library.

   - For ILE native methods, use the Create C Module (CRTCMOD) command to compile source files into module objects. Then bind one or more module objects into a service program by using the Create Service Program (CRTSRVPGM) command. The name of this service program must match the name that you supplied in your Java code that is in the `System.load()` or `System.loadLibrary()` Java method calls.

     Note: The implementation code for the native method must be compiled with teraspace storage enabled. For more information about teraspace and native methods, see "Teraspace storage model native methods for Java" on page 204.

   - For PASE for i native methods, use the `xlc` or `xlc_r` commands to compile and build an AIX library. For more information about compiling and building libraries for PASE for i, see Compiling your AIX source topic.
6. If you used the `System.loadLibrary()` call in your Java code to load the native library, perform one of the following tasks:

   - Include the list of the native library paths that you need in the `LIBPATH` environment variable. You can change the `LIBPATH` environment variable in QShell and from the IBM i command line.
     – From the Qshell command prompt, type in:

```
export LIBPATH=/QSYS.LIB/MYLIB.LIB
java  myclass
```

– Or, from the command line:

```
ADDENVVAR LIBPATH '/QSYS.LIB/MYLIB.LIB'
JAVA  myclass
```

- Or, supply the list in the **java.library.path** property. You can change the `java.library.path` property in QShell and from the IBM i command line.
  – From the Qshell command prompt, enter:

```
java -Djava.library.path=/QSYS.LIB/MYLIB.LIB  myclass
```

  – Or, from the IBM i command line, type in:

```
JAVA PROP((java.library.path '/QSYS.LIB/MYLIB.LIB')) myclass
```

Where */QSYS.LIB/MYLIB.LIB* is the path that contains the native library you want to load using the `System.loadLibrary()` call, and *myclass* is the name of your Java application.

For information about how a native library is located by the `System.loadLibrary()` method, see "Managing native method libraries" on page 208.

For an example of an ILE native method, see "Example: ILE native method for Java" on page 206. For an example of a PASE for i native method, see "Example: IBM PASE for i native method for Java" on page 207.

Websphere Development Studio: ILE RPG Programmer's Guide, SC09-2507.

"Java native methods and threads considerations" on page 210
You can use native methods to access functions that are not available in Java. To better use Java with native methods, you need to understand these concepts.

Java Native Interface by Oracle.

"Example: ILE native method for Java" on page 206
The integrated language environment (ILE) native method for Java example calls an instance of a native C method that then uses Java Native Interface (JNI) to call back into Java code to set the value of a Java string variable. The Java string variable is then written to standard out by the Java code.

"Strings in ILE native methods" on page 205
Many Java Native Interface (JNI) functions accept C language-style strings as parameters. For example, the `FindClass()` JNI function accepts a string parameter that specifies the fully-qualified name of a class file. If the class file is found, it is loaded by `FindClass()`, and a reference to it is returned to the caller of `FindClass()`.

"Java character encodings" on page 20
Java programs can convert data in different formats, enabling your applications to transfer and use information from many kinds of international character sets.

ILE native methods for Java

The IBM i Java virtual machine (JVM) supports the use of native methods running in the integrated language environment (ILE).

Support for ILE native methods includes:

- Full use of the native IBM i Java Native Interface (JNI) from ILE native methods
- The ability to call ILE native methods from the native IBM i JVM

When using ILE native methods, you should take the following in consideration:

- ILE programs or service programs that use JNI functions must be compiled with teraspace storage enabled. This is necessary because the Java object is in PASE for i storage, which is mapped on top of teraspace storage, and a teraspace storage pointer is returned. Also, the JNI functions, such as `GetxxxArrayRegion`, have a parameter to a buffer where the data is placed. This pointer must point to teraspace storage to enable the JNI function in PASE for i to copy the data into this storage. If you do not compile your program with teraspace storage enabled, you will receive the escape message MCH4443 (Invalid storage model for target program LOADLIB).

- All JNI functions expect their string parameters to be encoded in UTF-8. For more information about strings and JNI functions, see "Strings in ILE native methods" on page 205.

Teraspace storage model native methods for Java

The IBM i Java virtual machine (JVM) supports the use of teraspace storage model native methods. The teraspace storage model provides a large, process-local address environment for ILE programs. Using the teraspace storage model allows you to port native method code from other operating systems to IBM i with little or no source code changes.

Note: Teraspace storage model provides an environment in which static storage, local variables, and heap allocations are automatically located in teraspace storage. If all you need to do is enable access to teraspace storage, then you **do not** need to use teraspace storage model. It is enough to teraspace enable your native method code. To teraspace enable your native method, use the TERASPACE(*YES) parameter on the Create C Module (CRTCMOD), Create C++ Module (CRTCPPMOD), or other module creation command.

For details about programming with the teraspace storage model, see the following information:

- Chapter 4 of ILE Concepts
- Chapter 17 of WebSphere Development Studio ILE C/C++ Programmer's Guide

The concept for Java native methods created for the teraspace storage model is very similar to that of native methods that use single-level storage. The JVM passes the teraspace storage model native methods a pointer to the Java Native Interface (JNI) environment that the methods can use to call JNI functions.

For teraspace storage model native methods, the JVM provides JNI function implementations that utilize teraspace storage model and 8-byte pointers.

Creating teraspace storage model native methods

To successfully create a teraspace storage model native method, your teraspace storage model module creation command needs to use the following options:

```
TERASPACE(*YES) STGMDL(*TERASPACE) DTAMDL(*LLP64)
```

The following option (`*TSIFC`), to use teraspace storage functions, is optional:

```
TERASPACE(*YES *TSIFC)
```

Note: If you do not use `DTAMDL(*LLP64)` when using teraspace storage model Java native methods, calling a native method throws a runtime exception.

Creating teraspace storage model service programs that implement native methods

In order to create a teraspace storage model service program, use the following option on the Create Service Program (CRTSRVPGM) control language (CL) command:

```
CRTSRVPGM STGMDL(*TERASPACE)
```

In addition, you should use the `ACTGRP(*CALLER)` option, which allows the JVM to activate all teraspace storage model native method service programs into the same teraspace activation group. Using a teraspace activation group this way can be important for native methods to efficiently handle exceptions.

For additional details on program activation and activation groups, see Chapter 3 of ILE Concepts.

Using Java Invocation APIs with teraspace storage model native methods

Use the Invocation API GetEnv function when the JNI environment pointer does not match the storage model of the service program. The Invocation API GetEnv function always returns the correct JNI environment pointer.

The JVM supports both single-level and teraspace storage model native methods, but the two storage models use different JNI environments. Because the two storage models use different JNI environments, do not pass the JNI environment pointer as a parameter between native methods in the two storage models.

Related concepts

Strings in ILE native methods
Many Java Native Interface (JNI) functions accept C language-style strings as parameters. For example, the `FindClass()` JNI function accepts a string parameter that specifies the fully-qualified name of a class file. If the class file is found, it is loaded by `FindClass()`, and a reference to it is returned to the caller of `FindClass()`.

Example: ILE native method for Java
The integrated language environment (ILE) native method for Java example calls an instance of a native C method that then uses Java Native Interface (JNI) to call back into Java code to set the value of a Java string variable. The Java string variable is then written to standard out by the Java code.

"Java Invocation API" on page 211
The Invocation API, which is part of the Java Native Interface (JNI), allows non-Java code to create a Java virtual machine, and load and use Java classes. This function lets a multithreaded program make use of Java classes that are running in a single Java virtual machine in multiple threads.

Related information

Java Native Interface by Oracle.

Create C Module (CRTCMOD) CL command

Create C++ Module (CRTCPPMOD) CL command

Websphere Development Studio ILE C/C++ Programmer's Guide

Strings in ILE native methods

Many Java Native Interface (JNI) functions accept C language-style strings as parameters. For example, the `FindClass()` JNI function accepts a string parameter that specifies the fully-qualified name of a class file. If the class file is found, it is loaded by `FindClass()`, and a reference to it is returned to the caller of `FindClass()`.

All JNI functions expect their string parameters to be encoded in UTF-8. For details on UTF-8, you can refer to the JNI Specification, but in most cases it is enough to observe that 7-bit American Standard Code for Information Interchange (ASCII) characters are equivalent to their UTF-8 representation. 7-bit ASCII characters are actually 8-bit characters but their first bit is always 0. So, most ASCII C strings are actually already in UTF-8.

The integrated language environment (ILE) C compiler on the server operates in extended binary-coded decimal interchange code (EBCDIC) by default, so strings passed to JNI functions need to be converted to UTF-8. There are two ways to do this. You can use literal strings, or you can use dynamic strings. "Literal strings" on page 205 are strings whose value is known when the source code is compiled. Dynamic strings are strings whose value is not known at compile time, but is actually computed at run time.

Literal strings

If the string can be represented in ASCII, as most are, then the string can be bracketed by `pragma` statements that change the current codepage of the compiler. Then, the compiler stores the string internally in the UTF-8 form that is required by the JNI. If the string cannot be represented in ASCII, it is easier to treat the original extended binary-coded decimal interchange code (EBCDIC) string as a dynamic string, and process it using `iconv()` before passing it to the JNI.

For example, to find the class named `java/lang/String`, the code looks like this:

```
#pragma convert(819)
myClass = (*env)->FindClass(env,"java/lang/String");
#pragma convert(0)
```

The first pragma, with the number 819, informs the compiler to store all subsequent double-quoted strings (literal strings) in ASCII. The second pragma, with the number 0, tells the compiler to revert to the default code page of the compiler for double-quoted strings, which is usually the EBCDIC code page 37. So, by bracketing this call with these pragmas, we satisfy the JNI requirement that string parameters are encoded in UTF-8.

Caution: Be careful with text substitutions. For example, if your code looks like this:

```
#pragma convert(819)
#define MyString "java/lang/String"
#pragma convert(0)
myClass = (*env)->FindClass(env,MyString);
```

Then, the resulting string is EBCDIC, because the value of `MyString` is substituted into the `FindClass()` call during compilation. At the time of this substitution, the pragma, number 819, is not in effect. Thus, literal strings are not stored in ASCII.

Converting dynamic strings to and from EBCDIC, Unicode, and UTF-8

To manipulate string variables that are computed at run time, it may be necessary to convert strings to and from EBCDIC, Unicode, and UTF-8. Conversions can be done using the `iconv()` API. In Example 3 of the using the Java Native Interface for native methods examples, the routine creates, uses, and then destroys the `iconv()` conversion descriptor. This scheme avoids the problems with multithreaded use of an `iconv_t` descriptor, but for performance sensitive code it is better to create a conversion descriptor in static storage, and moderate multiple access to it using a mutual exclusion (mutex) or other synchronization facility.

Related concepts

Teraspace storage model native methods for Java
The IBM i Java virtual machine (JVM) supports the use of teraspace storage model native methods. The teraspace storage model provides a large, process-local address environment for ILE programs. Using the teraspace storage model allows you to port native method code from other operating systems to IBM i with little or no source code changes.

Example: ILE native method for Java
The integrated language environment (ILE) native method for Java example calls an instance of a native C method that then uses Java Native Interface (JNI) to call back into Java code to set the value of a Java string variable. The Java string variable is then written to standard out by the Java code.

Example: ILE native method for Java

The integrated language environment (ILE) native method for Java example calls an instance of a native C method that then uses Java Native Interface (JNI) to call back into Java code to set the value of a Java string variable. The Java string variable is then written to standard out by the Java code.

To see HTML versions of the example source files, use the following links:

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

- "Example: NativeHello.java" on page 468
- "Example: NativeHello.c" on page 469

Before you can run the ILE native method example, you must complete the tasks in the following topics:

1. "Example: Preparing the ILE native method source code" on page 472
2. "Example: Creating the ILE native method program objects" on page 473

Running the ILE native method for Java example

After you complete the previous tasks, you can run the example. Use either of the following commands to run the example program:

- From an IBM i command prompt:

```
JAVA CLASS(NativeHello) CLASSPATH('/ileexample')
```

- From a Qshell command prompt:

```
cd /ileexample
java NativeHello
```

Related concepts

Teraspace storage model native methods for Java
The IBM i Java virtual machine (JVM) supports the use of teraspace storage model native methods. The teraspace storage model provides a large, process-local address environment for ILE programs. Using the teraspace storage model allows you to port native method code from other operating systems to IBM i with little or no source code changes.

Strings in ILE native methods
Many Java Native Interface (JNI) functions accept C language-style strings as parameters. For example, the `FindClass()` JNI function accepts a string parameter that specifies the fully-qualified name of a class file. If the class file is found, it is loaded by `FindClass()`, and a reference to it is returned to the caller of `FindClass()`.

PASE for i native methods for Java

The IBM i Java virtual machine (JVM) supports the use of native methods running in the PASE for i environment.

Support for PASE for i native methods includes:

- Full use of the native IBM i Java Native Interface (JNI) from PASE for i native methods
- The ability to call PASE for i native methods from the native IBM i JVM

This support enables you to easily port your Java applications that run in AIX to your server. You can copy the class files and AIX native method libraries to the integrated file system on the server and run them from any of the control language (CL), Qshell or PASE for i terminal session command prompts.

When using PASE for i native methods, you show take the following in consideration:

- The architecture of the native code needs to match the architecture of the JVM. That is, object binaries need to be compiled as 32-bit binaries for a 32-bit JVM, or as 64-bit binaries for a 64-bit JVM. This also applies to agents (such as user-provided JVMTI agents).

Related information

PASE for i

Example: IBM PASE for i native method for Java

The PASE for i native method for Java example calls an instance of a native C method that then uses Java Native Interface (JNI) to call back into Java code. Rather than accessing the string directly from Java code, the example calls a native method that then calls back into Java through JNI to get the string value.

To see HTML versions of the example source files, use the following links:

Note: By using the code examples, you agree to the terms of the “Code license and disclaimer information” on page 489.

- “Example: PaseExample1.java” on page 464
- “Example: PaseExample1.c” on page 465

Before you can run the PASE for i native method example, you must complete the tasks in the following topics:

1. "Example: Downloading the example source code to your AIX workstation" on page 465
2. "Example: Preparing the example source code" on page 466
3. "Example: Preparing your IBM i server to run the PASE for i native method for Java example" on page 467

Running the PASE for i native method for Java example

After you complete the previous tasks, you can run the example. Use either of the following commands to run the example program:

- From an IBM i command prompt:

```
JAVA CLASS(PaseExample1) CLASSPATH('/home/example')
```

- From a Qshell command prompt or PASE for i terminal session:

```
cd /home/example
java PaseExample1
```

Managing native method libraries

To use native method libraries, especially when you want to manage multiple versions of a native method library on your IBM i server, you need to understand both the Java library naming conventions and the library search algorithm.

Note: The term native library or native method library refers to integrated language environment (ILE) service programs when used in the context of ILE native methods, and AIX static or shared libraries when used in the context of PASE for i native methods.

The Java method `System.loadLibrary()` is used to load a native library given a library name. IBM i uses the first native method library that matches the name of the library that the Java virtual machine (JVM) loads. In order to ensure that IBM i finds the correct native methods, you must avoid library name clashes and confusion about which native method library the JVM uses.

Native library naming conventions

You need to be aware of the following library naming conventions:

- If the native methods are ILE native methods and the Java code loads a library named `Sample`, the corresponding executable file must be an ILE service program named SAMPLE.
- If the native methods are PASE for i native methods and the Java code loads a library named `Sample`, the corresponding executable file must be an AIX library named either `libSample.a` or `libSample.so`.

Java library search order

To locate a native library, Java uses the `java.library.path` property to determine the search path. By default, the `java.library.path` property is set to a value that is a result of the concatenation of two lists (in the following order):

1. IBM i library list
2. The value of the `LIBPATH` environment variable

In order to perform the search, IBM i converts the library list to the integrated file system format. QSYS file system objects have equivalent names in the integrated file system, but some integrated file system objects do not have equivalent QSYS file system names. Because the library loader looks for objects in both the QSYS file system and in the integrated file system, IBM i uses the integrated file system format to search for native method libraries.

The following table shows how IBM i converts entries in the library list to the integrated file system format:

Library list entry	Integrated file system format
QSYS	`/qsys.lib`
QSYS2	`/qsys.lib/qsys2.lib`
QGPL	`/qsys.lib/qgpl.lib`
QTEMP	`/qsys.lib/qtemp.lib`

Example: Searching for the Sample2 library

In this example, the `LIBPATH` environment variable is set to `/home/user1/lib32:/samples/lib32`. The following table, when read from top to bottom, indicates the full search path:

Source	Integrated file system directories

Library list	`/qsys.lib`
	`/qsys.lib/qsys2.lib`
	`/qsys.lib/qgpl.lib`
	`/qsys.lib/qtemp.lib`

LIBPATH	`/home/user1/lib32` `/samples/lib32`

Note: Uppercase and lowercase characters are significant only in the `/QOpenSys` path.

In order to search for library `Sample2`, the Java library loader searches for file candidates in the following order:

1. /qsys.lib/sample2.srvpgm
2. /qsys.lib/libSample2.a
3. /qsys.lib/libSample2.so
4. /qsys.lib/qsys2.lib/sample2.srvpgm
5. /qsys.lib/qsys2.lib/libSample2.a
6. /qsys.lib/qsys2.lib/libSample2.so
7. /qsys.lib/qgpl.lib/sample2.srvpgm
8. /qsys.lib/qgpl.lib/libSample2.a
9. /qsys.lib/qgpl.lib/libSample2.so
10. /qsys.lib/qtemp.lib/sample2.srvpgm
11. /qsys.lib/qtemp.lib/libSample2.a
12. /qsys.lib/qtemp.lib/libSample2.so
13. /home/user1/lib32/sample2.srvpgm
14. /home/user1/lib32/libSample2.a
15. /home/user1/lib32/libSample2.so
16. /samples/lib32/sample2.srvpgm
17. /samples/lib32/libSample2.a
18. /samples/lib32/libSample2.so

IBM i loads the first candidate in the list that actually exists into the JVM as a native method library.

Note: You can create arbitrary symbolic links from integrated file system directories to IBM i objects in the QSYS file system. As a result, valid file candidates can include files such as `/home/user1/lib32/sample2.srvpgm`.

Java native methods and threads considerations

You can use native methods to access functions that are not available in Java. To better use Java with native methods, you need to understand these concepts.

- A Java thread, whether created by Java or an attached native thread, has all floating point exceptions disabled. If the thread runs a native method that reenables floating point exceptions, Java does not turn them off a second time. If the user application does not disable them before returning to run Java code, then the Java code may not behave correctly if a floating point exception occurs. When a native thread detaches from the Java virtual machine, its floating point exception mask is restored to the value that was in effect when it was attached.
- When a native thread attaches to the Java virtual machine, the Java virtual machine changes the threads priority, if necessary, to conform to the one to ten priority schemes that Java defines. When the thread detaches, the priority is restored. After attaching, the thread can change the thread priority by using a native method interface (for example, a POSIX API). Java does not change the thread priority on transitions back to the Java virtual machine.
- The Invocation API component of the Java Native Interface (JNI) permits a user to embed a Java virtual machine within their application. If an application creates a Java virtual machine and the Java virtual machine ends abnormally, the MCH74A5 "Java Virtual Machine Terminated" IBM i exception is signalled to the initial thread of the process if that thread was attached to the Java virtual machine when the Java virtual machine ended. The Java virtual machine could end abnormally for any of these reasons:
  - The user calls the java.lang.System.exit() method.
  - A thread that the Java virtual machine requires ends.
  - An internal error occurs in the Java virtual machine.

  This behavior differs from most other Java platforms. On most other platforms, the process that automatically creates the Java virtual machine ends abruptly as soon as the Java virtual machine ends. If the application monitors and handles a signalled MCH74A5 exception, it may continue to run. Otherwise, the process ends when the exception goes unhandled. By adding the code that deals with the IBM i system-specific MCH74A5 exception, you can make the application less portable to other platforms.

Because native methods always run in a multithreaded process, the code that they contain must be thread safe. This places these restrictions on the languages and functions that are used for native methods:

- You should not use ILE CL for native methods, because this language is not thread safe. To run thread safe CL commands, you can use the C language system() function or the java.lang.Runtime.exec() method.
  - Use the C language system() function to run thread safe CL commands from within a C or C++ native method.
  - Use the java.lang.Runtime.exec() method to run thread safe CL commands directly from Java.
- You can use AIX C/C++, ILE C, ILE C++, ILE COBOL, and ILE RPG to write a native method, but all of the functions that are called from within the native method must be thread safe.

  Note: Compile-time support for writing native methods is currently only supplied for the C, C++, and RPG languages. While possible, writing native methods in other languages may be much more complicated.

  Caution: ***Not all standard C, C++, COBOL, or RPG functions are thread safe.***
- The C and C++ exit() and abort() functions should never be used within a native method. These functions cause the entire process that runs the Java virtual machine to stop. This includes all of the threads in the process, regardless of if they were originated by Java or not.

Note: The exit() function referred to is the C and C++ function, and is not the same as the java.lang.Runtime.exit() method.

For more information about threads on the server, see Multithreaded applications.

Java Invocation API

The Invocation API, which is part of the Java Native Interface (JNI), allows non-Java code to create a Java virtual machine, and load and use Java classes. This function lets a multithreaded program make use of Java classes that are running in a single Java virtual machine in multiple threads.

The IBM Developer Kit for Java supports the Java Invocation API for the following types of callers:

- An ILE program or service program that is enabled to work with teraspace storage. The storage model can either be single-level storage or teraspace storage. For more information about JNI and teraspace storage, see "Teraspace storage model native methods for Java" on page 204.
- An PASE for i executable created for either 32-bit or 64-bit AIX.

  Note: The `LIBPATH` and `LDR_CNTRL` environment variables may need to be set appropriately when running PASE for i executables.

The application controls the Java virtual machine. The application can create the Java virtual machine, call Java methods (similar to the way in which an application calls subroutines), and destroy the Java virtual machine. Once you create the Java virtual machine, it remains ready to run within the process until the application explicitly destroys it. While being destroyed, the Java virtual machine performs clean-up, such as running finalizers, ending Java virtual machine threads, and releasing Java virtual machine resources.

With a Java virtual machine that is ready to run, an application written in ILE languages, such as C and RPG, can call into the Java virtual machine to perform any function. It also can return from the Java virtual machine to the C application, call into the Java virtual machine again, and so on. The Java virtual machine is created once and does not have to be recreated before calling into the Java virtual machine to run a little or a lot of Java code.

When using the Invocation API to run Java programs, the destination for STDOUT and STDERR is controlled by the use of an environment variable called `QIBM_USE_DESCRIPTOR_STDIO`. If this environment variable is set to Y or I (for example, `QIBM_USE_DESCRIPTOR_STDIO=Y`), the Java virtual machine uses file descriptors for STDIN (fd 0), STDOUT (fd 1), and STDERR (fd 2). In this case, the program must set these file descriptors to valid values by opening them as the first three files or pipes in this job. The first file opened in the job is given fd of 0, the second fd of 1, and third is fd of 2. For jobs initiated with the spawn API, these descriptors can be preassigned using a file descriptor map (see documentation on Spawn API). If the environment variable `QIBM_USE_DESCRIPTOR_STDIO` is not set or is set to any other value, file descriptors are not used for STDIN, STDOUT, or STDERR. Instead, STDOUT and STDERR are routed to a spooled file that is owned by the current job, and use of STDIN results in an IO exception.

Note: Message `CPFB9C8 (File descriptors 0, 1, and 2 must be open to run the PASE for i program.)` will be issued if it is determined that the file descriptors for STDIN, STDOUT, and STDERR are not set but are required to be set.

Related concepts

Native methods and the Java Native Interface
Native methods are Java methods that start in a language other than Java. Native methods can access system-specific functions and APIs that are not available directly in Java.

Using java.lang.Runtime.exec()
Use the `java.lang.Runtime.exec()` method to call programs or commands from within your Java program. Using `java.lang.Runtime.exec()` method creates one or more additional thread-enabled jobs. The additional jobs process the command string that you pass on the method.

Interprocess communications

When communicating with programs that are running in another process, there are a number of options.

Invocation API functions

The IBM Developer Kit for Java supports these Invocation API functions.

Note: Before using this API, you must ensure that you are in a multithread-capable job. See Multithreaded applications for more information about multithread-capable jobs.

- **JNI_GetCreatedJavaVMs**

  Returns information about all Java virtual machines that were created. Even though this API is designed to return information for multiple Java virtual machines (JVMs), only one JVM can exist for a process. Therefore, this API will return a maximum of one JVM.

  Signature:

```
    jint JNI_GetCreatedJavaVMs(JavaVM **vmBuf,
                               jsize bufLen,
                               jsize *nVMs);
```

  vmBuf is an output area whose size is determined by bufLen, which is the number of pointers. Each Java virtual machine has an associated JavaVM structure that is defined in java.h. This API stores a pointer to the JavaVM structure that is associated with each created Java virtual machine into vmBuf, unless vmBuf is 0. Pointers to JavaVM structures are stored in the order of the corresponding Java virtual machines that are created. nVMs returns the number of virtual machines that are currently created. Your server supports the creation of more than one Java virtual machine, so you may expect a value higher than one. This information, along with the size of the vmBuf, determines whether pointers to JavaVM structures for each created Java virtual machine are returned.

- **JNI_CreateJavaVM**

  Allows you to create a Java virtual machine and subsequently use it in an application.

  Signature:

```
jint JNI_CreateJavaVM(JavaVM **p_vm,
                      void **p_env,
                      void *vm_args);
```

  p_vm is the address of a JavaVM pointer for the newly created Java virtual machine. Several other JNI Invocation APIs use p_vm to identify the Java virtual machine. p_env is the address of a JNI Environment pointer for the newly created Java virtual machine. It points to a table of JNI functions that start those functions. vm_args is a structure that contains Java virtual machine initialization parameters.

  If you start a Run Java (RUNJVA) command or JAVA command and specify a property that has an equivalent command parameter, then the command parameter takes precedence. The property is ignored.

  For a list of unique properties that are supported by the JNI_CreateJavaVM API, see "Java system properties" on page 14.

  Note: Java on IBM i supports creating only one Java virtual machine (JVM) within a single job or process. For more information, see "Support for multiple Java virtual machines" on page 213

- **DestroyJavaVM**

  Destroys the Java virtual machine.

  Signature:

```
jint DestroyJavaVM(JavaVM *vm)
```

  When the Java virtual machine is created, vm is the JavaVM pointer that is returned.

- **AttachCurrentThread**

Attaches a thread to a Java virtual machine, so it can use Java virtual machine services.

Signature:

```
jint AttachCurrentThread(JavaVM *vm,
                   void **p_env,
                   void *thr_args);
```

The JavaVM pointer, vm, identifies the Java virtual machine to which the thread is being attached. p_env is the pointer to the location where the JNI Interface pointer of the current thread is placed. thr_args contains VM specific thread attachment arguments.

- **DetachCurrentThread**

  Signature:

```
jint DetachCurrentThread(JavaVM *vm);
```

  vm identifies the Java virtual machine from which the thread is being detached.

Java Native Interface by Oracle.

Support for multiple Java virtual machines

Java on the IBM i platform no longer supports creating more than one Java virtual machine (JVM) within a single job or process. This restriction affects only those users who create JVMs by using the Java Native Interface Invocation (JNI) API. This change in support does not affect how you use the java command to run your Java programs.

You cannot successfully call JNI_CreateJavaVM() more than once in a job, and JNI_GetCreatedJavaVMs() cannot return more than one JVM in a list of results.

Support for creating only a single JVM within a single job or process follows the standards of the Oracle America, Inc. reference implementation of Java.

Example: Java Invocation API

This integrated language environment (ILE) C example follows the standard Invocation API paradigm.

It does the following:

- Creates a Java virtual machine by using `JNI_CreateJavaVM()`.
- Uses the Java virtual machine to find the class file that you want to run.
- Finds the methodID for the main method of the class.
- Calls the main method of the class.
- Reports errors if an exception occurs.

When you create the program, the QJVAJNI or QJVAJNI64 service program provides the `JNI_CreateJavaVM()` API function. `JNI_CreateJavaVM()` creates the Java virtual machine.

Note: QJVAJNI64 is a ervice program for teraspace/LLP64 native method and Invocation API support.

These service programs reside in the system binding directory and you do not need to explicitly identify them on a control language (CL) create command. For example, you would not explicitly identify the previously mentioned service programs when using the Create Program (CRTPGM) command or the Create Service Program (CRTSRVPGM) command.

One way to run this program is to use the following control language command:

```
SBMJOB CMD(CALL PGM(YOURLIB/PGMNAME)) ALWMLTTHD(*YES)
```

Any job that creates a Java virtual machine must be multithread-capable. The output from the main program, as well as any output from the program, ends up in QPRINT spooled files. These spooled files are visible when you use the Work with Submitted Jobs (WRKSBMJOB) control language (CL) command and view the job that you started by using the Submit Job (SBMJOB) CL command.

Example: Using the Java Invocation API within ILE C

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
#define OS400_JVM_12
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#include <jni.h>

/* Specify the pragma that causes all literal strings in the
 * source code to be stored in ASCII (which, for the strings
 * used, is equivalent to UTF-8)
 */

#pragma convert(819)

/* Procedure: Oops
 *
 * Description: Helper routine that is called when a JNI function
 *              returns a zero value, indicating a serious error.
 *              This routine reports the exception to stderr and
 *              ends the JVM abruptly with a call to FatalError.
 *
 * Parameters:  env -- JNIEnv* to use for JNI calls
 *              msg -- char* pointing to error description in UTF-8
 *
 * Note:        Control does not return after the call to FatalError
 *              and it does not return from this procedure.
 */

void Oops(JNIEnv* env, char *msg) {
    if ((*env)->ExceptionOccurred(env)) {
        (*env)->ExceptionDescribe(env);
    }
    (*env)->FatalError(env, msg);
}

/* This is the program's "main" routine. */
int main (int argc, char *argv[])
{

    JavaVMInitArgs initArgs; /* Virtual Machine (VM) initialization structure, passed by
                              * reference to JNI_CreateJavaVM(). See jni.h for details
                              */
    JavaVM* myJVM;           /* JavaVM pointer set by call to JNI_CreateJavaVM */
    JNIEnv* myEnv;           /* JNIEnv pointer set by call to JNI_CreateJavaVM */
    char*   myClasspath;     /* Changeable classpath 'string' */
    jclass  myClass;         /* The class to call, 'NativeHello'. */
    jmethodID mainID;        /* The method ID of its 'main' routine. */
    jclass  stringClass;     /* Needed to create the String[] arg for main */
    jobjectArray args;       /* The String[] itself */
    JavaVMOption options[1]; /* Options array -- use options to set classpath */
    int     fd0, fd1, fd2;   /* file descriptors for IO */

    /* Open the file descriptors so that IO works. */
    fd0 = open("/dev/null", O_CREAT|O_TRUNC|O_RDWR,   S_IRUSR|S_IROTH);
    fd1 = open("/dev/null", O_CREAT|O_TRUNC|O_WRONLY, S_IWUSR|S_IWOTH);
    fd2 = open("/dev/null", O_CREAT|O_TRUNC|O_WRONLY, S_IWUSR|S_IWOTH);

    /*  Set the version field of the initialization arguments for JNI v1.4. */
    initArgs.version = 0x00010004;


    /* Now, you want to specify the directory for the class to run in the classpath.
     * with  Java2, classpath is passed in as an option.
     * Note: You must specify the directory name in UTF-8 format. So, you wrap
     *       blocks of code in #pragma convert statements.
     */
    options[0].optionString="-Djava.class.path=/CrtJvmExample";

    initArgs.options=options;  /* Pass in the classpath that has been set up. */
    initArgs.nOptions = 1;     /* Pass in classpath and version options */
```

```
    /*  Create the JVM -- a nonzero return code indicates there was
     *  an error. Drop back into EBCDIC and write a message to stderr
     *  before exiting the program.
     *  Note:  This will run the default JVM and JDK which is 32bit JDK 6.0.
     *  If you want to run a different JVM and JDK, set the JAVA_HOME environment
     *  variable to the home directory of the JVM you want to use
     *  (prior to the CreateJavaVM() call).
     */
    if (JNI_CreateJavaVM(&myJVM, (void **)&myEnv, (void *)&initArgs)) {
        #pragma convert(0)
            fprintf(stderr, "Failed to create the JVM\n");
        #pragma convert(819)
            exit(1);
    }

    /*  Use the newly created JVM to find the example class,
     *  called 'NativeHello'.
     */
    myClass = (*myEnv)->FindClass(myEnv, "NativeHello");
    if (! myClass) {
        Oops(myEnv, "Failed to find class 'NativeHello'");
    }

    /* Now, get the method identifier for the 'main' entry point
     * of the class.
     * Note: The signature of 'main' is always the same for any
     *       class called by the following java command:
     *           "main" , "([Ljava/lang/String;)V"
     */
    mainID = (*myEnv)->GetStaticMethodID(myEnv,myClass,"main",
                                     "([Ljava/lang/String;)V");
    if (! mainID) {
        Oops(myEnv, "Failed to find jmethodID of 'main'");
    }

    /* Get the jclass for String to create the array
     * of String to pass to 'main'.
     */
    stringClass = (*myEnv)->FindClass(myEnv, "java/lang/String");
    if (! stringClass) {
        Oops(myEnv, "Failed to find java/lang/String");
    }

    /*  Now, you need to create an empty array of strings,
     *  since main requires such an array as a parameter.
     */
    args = (*myEnv)->NewObjectArray(myEnv,0,stringClass,0);
    if (! args) {
        Oops(myEnv, "Failed to create args array");
    }

    /* Now, you have the methodID of main and the class, so you can
     * call the main method.
     */
    (*myEnv)->CallStaticVoidMethod(myEnv,myClass,mainID,args);

    /* Check for errors. */
    if ((*myEnv)->ExceptionOccurred(myEnv)) {
        (*myEnv)->ExceptionDescribe(myEnv);
    }

    /* Finally, destroy the JavaVM that you created. */
    (*myJVM)->DestroyJavaVM(myJVM);

    /* All done. */
    return 0;
}
```

For more information, see "Java Invocation API" on page 211.

Using java.lang.Runtime.exec()

Use the `java.lang.Runtime.exec()` method to call programs or commands from within your Java program. Using `java.lang.Runtime.exec()` method creates one or more additional thread-enabled jobs. The additional jobs process the command string that you pass on the method.

The `java.lang.Runtime.exec()` method runs programs in a separate job, which is different than the C `system()` function. The C `system()` function runs programs in the same job. The actual processing that occurs depends on the kind of command that you pass in on `java.lang.Runtime.exec()`. The following table indicates how `java.lang.Runtime.exec()` processes different kinds of commands.

Type of command	How command is processed
java command	Starts a second job that runs the JVM. The JVM starts a third job that runs the Java application.
program	Starts a second job that runs an executable program (IBM i ILE program or PASE for i program).
CL command	Starts a second job that runs an IBM i ILE program. The IBM i ILE program runs the CL command in the second job.

Related concepts

Native methods and the Java Native Interface
Native methods are Java methods that start in a language other than Java. Native methods can access system-specific functions and APIs that are not available directly in Java.

Java Invocation API
The Invocation API, which is part of the Java Native Interface (JNI), allows non-Java code to create a Java virtual machine, and load and use Java classes. This function lets a multithreaded program make use of Java classes that are running in a single Java virtual machine in multiple threads.

Interprocess communications
When communicating with programs that are running in another process, there are a number of options.

Example: Calling another Java program with java.lang.Runtime.exec()

This example shows how to call another Java program with `java.lang.Runtime.exec()`. This class calls the Hello program that is shipped as part of the IBM Developer Kit for Java. When the Hello class writes to System.out, this program gets a handle to the stream and can read from it.

Source code for CallHelloPgm Java class

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
import java.io.*;

public class CallHelloPgm
{
   public static void main(String args[])
   {
      Process theProcess = null;
      BufferedReader inStream = null;

      System.out.println("CallHelloPgm.main() invoked");

      // call the Hello class
      try
      {
          theProcess = Runtime.getRuntime().exec("java QIBMHello");
      }
      catch(IOException e)
      {
         System.err.println("Error on exec() method");
         e.printStackTrace();
      }
```

```
      // read from the called program's standard output stream
      try
      {
         inStream = new BufferedReader(
                              new InputStreamReader( theProcess.getInputStream() ));
         System.out.println(inStream.readLine());
      }
      catch(IOException e)
      {
         System.err.println("Error on inStream.readLine()");
         e.printStackTrace();
      }
   }
}
```

Related reference

Example: Calling a CL program with java.lang.Runtime.exec()
This example shows how to run CL programs from within a Java program. In this example, the Java class CallCLPgm runs a CL program.

Example: Calling a CL command with java.lang.Runtime.exec()
This example shows how to run a control language (CL) command from within a Java program.

Example: Calling a CL program with java.lang.Runtime.exec()

This example shows how to run CL programs from within a Java program. In this example, the Java class CallCLPgm runs a CL program.

The CL program uses the Display JVM Jobs (DSPJVMJOB) CL command to display all of the jobs on the system that contain an active Java Virtual Machine. This example assumes that the CL program has been compiled and exists in a library that is called JAVSAMPLIB. The output from the CL program is in the QSYSPRT spooled file.

See “Example: Calling a CL command with java.lang.Runtime.exec()” on page 218 for an example of how to call a CL command from within a Java program.

Note: The JAVSAMPLIB is not created as part of the IBM Developer Kit licensed program (LP) number 5770-JV1 installation process. You must explicitly create the library.

Source code for CallCLPgm Java class

Note: By using the code examples, you agree to the terms of the “Code license and disclaimer information” on page 489.

```
import java.io.*;

public class CallCLPgm
{
   public static void main(String[] args)
   {
      try
      {
         Process theProcess =
                 Runtime.getRuntime().exec("/QSYS.LIB/JAVSAMPLIB.LIB/DSPJVA.PGM");
      }
      catch(IOException e)
      {
         System.err.println("Error on exec() method");
         e.printStackTrace();
      }
   }
}
```

Source code for DSPJVA CL program

```
PGM
    DSPJVMJOB OUTPUT(*PRINT)
ENDPGM
```

Related reference

Example: Calling another Java program with java.lang.Runtime.exec()
This example shows how to call another Java program with `java.lang.Runtime.exec()`. This class calls the Hello program that is shipped as part of the IBM Developer Kit for Java. When the Hello class writes to System.out, this program gets a handle to the stream and can read from it.

Example: Calling a CL command with java.lang.Runtime.exec()
This example shows how to run a control language (CL) command from within a Java program.

Example: Calling a CL command with java.lang.Runtime.exec()

This example shows how to run a control language (CL) command from within a Java program.

In this example, the Java class runs a CL command. The CL command uses the Display JVM Jobs (DSPJVMJOB) CL command to display all of the jobs on the system that contain an active Java Virtual Machine. The output from the CL command is in the QSYSPRT spooled file.

CL commands that you pass into the `Runtime.getRuntime().exec()` function use the following format:

```
Runtime.getRuntime().exec("system CLCOMMAND");
```

where *CLCOMMAND* is the CL command you want to run.

Source code for Java class for calling a CL command

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
import java.io.*;

public class CallCLCom
{
   public static void main(String[] args)
   {
      try
      {
         Process theProcess =
            Runtime.getRuntime().exec("system DSPJVMJOB OUTPUT(*PRINT)");
      }
      catch(IOException e)
      {
         System.err.println("Error on exec() method");
         e.printStackTrace();
      }
   }
}
```

Related concepts

"Using java.lang.Runtime.exec()" on page 216
Use the `java.lang.Runtime.exec()` method to call programs or commands from within your Java program. Using `java.lang.Runtime.exec()` method creates one or more additional thread-enabled jobs. The additional jobs process the command string that you pass on the method.

"List of Java system properties" on page 15
Java system properties determine the environment in which the Java programs run. They are like system values or environment variables in IBM i.

Related reference

Example: Calling another Java program with java.lang.Runtime.exec()
This example shows how to call another Java program with `java.lang.Runtime.exec()`. This class calls the Hello program that is shipped as part of the IBM Developer Kit for Java. When the Hello class writes to System.out, this program gets a handle to the stream and can read from it.

Example: Calling a CL program with java.lang.Runtime.exec()

This example shows how to run CL programs from within a Java program. In this example, the Java class CallCLPgm runs a CL program.

Interprocess communications

When communicating with programs that are running in another process, there are a number of options.

One option is to use sockets for interprocess communication. One program can act as the server program that listens on a socket connection for input from the client program. The client program connects to the server with a socket. Once the socket connection is established, either program can send or receive information.

Another option is to use stream files for communication between programs. To do this, use the System.in, System.out, and System.err classes.

A third option is to use the IBM Toolbox for Java, which provides data queues and IBM i message objects.

You can also call Java from other languages, as demonstrated in the examples below.

Related concepts

Native methods and the Java Native Interface
Native methods are Java methods that start in a language other than Java. Native methods can access system-specific functions and APIs that are not available directly in Java.

Java Invocation API
The Invocation API, which is part of the Java Native Interface (JNI), allows non-Java code to create a Java virtual machine, and load and use Java classes. This function lets a multithreaded program make use of Java classes that are running in a single Java virtual machine in multiple threads.

Using java.lang.Runtime.exec()
Use the `java.lang.Runtime.exec()` method to call programs or commands from within your Java program. Using `java.lang.Runtime.exec()` method creates one or more additional thread-enabled jobs. The additional jobs process the command string that you pass on the method.

Related information

IBM Toolbox for Java

Using sockets for interprocess communication

Sockets streams communicate between programs that are running in separate processes.

The programs can either start separately or start by using the java.lang.Runtime.exec() method from within the main Java program. If a program is written in a language other than Java, you must ensure that any American Standard Code for Information Interchange (ASCII) or extended binary-coded decimal interchange code (EBCDIC) conversion takes place. See the Java character encodings topics for more details.

Related concepts

Using input and output streams for interprocess communication
Input and output streams communicate between programs that are running in separate processes.

"Using java.lang.Runtime.exec()" on page 216
Use the `java.lang.Runtime.exec()` method to call programs or commands from within your Java program. Using `java.lang.Runtime.exec()` method creates one or more additional thread-enabled jobs. The additional jobs process the command string that you pass on the method.

"Java character encodings" on page 20
Java programs can convert data in different formats, enabling your applications to transfer and use information from many kinds of international character sets.

Related reference

Example: Calling Java from ILE C

This is an example of an integrated language environment (ILE) C program that uses the `system()` function to call the Java Hello program.

Example: Calling Java from RPG
This is an example of an RPG program that uses the QCMDEXC API to call the Java Hello program.

Example: Using sockets for interprocess communication

This example uses sockets to communicate between a Java program and a C program.

You should start the C program first, which listens on a socket. Once the Java program connects to the socket, the C program sends it a string by using that socket connection. The string that is sent from the C program is an American Standard Code for Information Interchange (ASCII) string in codepage 819.

The Java program should be started using this command, `java TalkToC xxxxx` nnnn on the Qshell Interpreter command line or on another Java platform. Or, enter `JAVA TALKTOC PARM(xxxxx nnnn)` on the IBM i command line to start the Java program. `xxxxx` is the domain name or Internet Protocol (IP) address of the system on which the C program is running. nnnn is the port number of the socket that the C program is using. You should also use this port number as the first parameter on the call to the C program.

Source code for TalkToC client Java class

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
import java.net.*;
import java.io.*;

class TalkToC
{
   private String host = null;
   private int port = -999;
   private Socket socket = null;
   private BufferedReader inStream = null;

   public static void main(String[] args)
   {
      TalkToC caller = new TalkToC();
      caller.host = args[0];
      caller.port = new Integer(args[1]).intValue();
      caller.setUp();
      caller.converse();
      caller.cleanUp();
   }

   public void setUp()
   {
      System.out.println("TalkToC.setUp() invoked");

      try
      {
         socket = new Socket(host, port);
         inStream = new BufferedReader(new InputStreamReader(
                                       socket.getInputStream()));
      }
      catch(UnknownHostException e)
      {
         System.err.println("Cannot find host called: " + host);
         e.printStackTrace();
         System.exit(-1);
      }
      catch(IOException e)
      {
         System.err.println("Could not establish connection for " + host);
         e.printStackTrace();
         System.exit(-1);
      }
   }

   public void converse()
   {
      System.out.println("TalkToC.converse() invoked");

      if (socket != null && inStream != null)
      {
         try
```

```
         {
            System.out.println(inStream.readLine());
         }
         catch(IOException e)
         {
            System.err.println("Conversation error with host " + host);
            e.printStackTrace();
         }
      }
   }

   public void cleanUp()
   {
      try
      {
         if (inStream != null)
            inStream.close();
         if (socket != null)
            socket.close();
      }
      catch(IOException e)
      {
         System.err.println("Error in cleanup");
         e.printStackTrace();
         System.exit(-1);
      }
   }
}
```

SockServ.C starts by passing in a parameter for the port number. For example, `CALL SockServ '2001'`.

Source code for SockServ.C server program

Note: Read the Code example disclaimer for important legal information.

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <unistd.h>
#include <sys/time.h>

void main(int argc, char* argv[])
{
   int    portNum = atoi(argv[1]);
   int    server;
   int    client;
   int    address_len;
   int    sendrc;
   int    bndrc;
   char*  greeting;
   struct sockaddr_in  local_Address;
   address_len = sizeof(local_Address);

   memset(&local_Address,0x00,sizeof(local_Address));
   local_Address.sin_family = AF_INET;
   local_Address.sin_port = htons(portNum);
   local_Address.sin_addr.s_addr = htonl(INADDR_ANY);

   #pragma convert (819)
   greeting = "This is a message from the C socket server.";
   #pragma convert (0)

   /*  allocate socket    */
   if((server = socket(AF_INET, SOCK_STREAM, 0))<0)
   {
      printf("failure on socket allocation\n");
      perror(NULL);
      exit(-1);
   }

   /* do bind   */
   if((bndrc=bind(server,(struct sockaddr*)&local_Address, address_len))<0)
   {
     printf("Bind failed\n");
     perror(NULL);
```

```
      exit(-1);
    }

    /* invoke listen   */
    listen(server, 1);

    /* wait for client request */
    if((client = accept(server,(struct sockaddr*)NULL, 0))<0)
    {
      printf("accept failed\n");
      perror(NULL);
      exit(-1);
    }

    /* send greeting to client    */
    if((sendrc = send(client, greeting, strlen(greeting),0))<0)
    {
       printf("Send failed\n");
       perror(NULL);
       exit(-1);
    }

    close(client);
    close(server);
 }
```

Using input and output streams for interprocess communication

Input and output streams communicate between programs that are running in separate processes.

The java.lang.Runtime.exec() method runs a program. The parent program can get handles to the child process input and output streams and can write to or read from those streams. If the child program is written in a language other than Java, you must ensure that any American Standard Code for Information Interchange (ASCII) or extended binary-coded decimal interchange code (EBCDIC) conversion takes place. See Java character encodings for more details.

Related concepts

Using sockets for interprocess communication
Sockets streams communicate between programs that are running in separate processes.

"Using java.lang.Runtime.exec()" on page 216
Use the `java.lang.Runtime.exec()` method to call programs or commands from within your Java program. Using `java.lang.Runtime.exec()` method creates one or more additional thread-enabled jobs. The additional jobs process the command string that you pass on the method.

"Java character encodings" on page 20
Java programs can convert data in different formats, enabling your applications to transfer and use information from many kinds of international character sets.

Related reference

Example: Calling Java from ILE C
This is an example of an integrated language environment (ILE) C program that uses the `system()` function to call the Java Hello program.

Example: Calling Java from RPG
This is an example of an RPG program that uses the QCMDEXC API to call the Java Hello program.

Example: Using input and output streams for interprocess communication

This example shows how to call a C program from Java and use input and output streams for interprocess communication.

In this example, the C program writes a string to its standard output stream, and the Java program reads this string and displays it. This example assumes that a library, which is named JAVSAMPLIB, has been created and that the CSAMP1 program has been created in it.

Note: The JAVSAMPLIB is not created as part of the IBM Developer Kit licensed program (LP) number 5770-JV1 installation process. You must explicitly create it.

Source code for CallPgm Java class

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
import java.io.*;

public class CallPgm
{
   public static void main(String args[])
   {
      Process theProcess = null;
      BufferedReader inStream = null;

      System.out.println("CallPgm.main() invoked");

      // call the CSAMP1 program
      try
      {
          theProcess = Runtime.getRuntime().exec(
                       "/QSYS.LIB/JAVSAMPLIB.LIB/CSAMP1.PGM");
      }
      catch(IOException e)
      {
         System.err.println("Error on exec() method");
         e.printStackTrace();
      }

      // read from the called program's standard output stream
      try
      {
         inStream = new BufferedReader(new InputStreamReader
                          (theProcess.getInputStream()));
         System.out.println(inStream.readLine());
      }
      catch(IOException e)
      {
         System.err.println("Error on inStream.readLine()");
         e.printStackTrace();
      }
   }
 }
```

Source code for CSAMP1 C Program

Note: Read the Code example disclaimer for important legal information.

```
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char* args[])
{
     /* Convert the string to ASCII at compile time */
#pragma convert(819)
     printf("Program JAVSAMPLIB/CSAMP1 was invoked\n");
#pragma convert(0)
     /* Stdout may be buffered, so flush the buffer */

     fflush(stdout);
}
```

Example: Calling Java from ILE C

This is an example of an integrated language environment (ILE) C program that uses the `system()` function to call the Java Hello program.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
#include <stdlib.h>

 int main(void)
 {
   int result;

   /* The system function passes the given string
    * to the CL command processor for processing.
    */
```

```
  result = system("JAVA CLASS('QIBMHello')");
}
```

Related concepts

Using sockets for interprocess communication
Sockets streams communicate between programs that are running in separate processes.

Using input and output streams for interprocess communication
Input and output streams communicate between programs that are running in separate processes.

Related reference

Example: Calling Java from RPG
This is an example of an RPG program that uses the QCMDEXC API to call the Java Hello program.

Example: Calling Java from RPG

This is an example of an RPG program that uses the QCMDEXC API to call the Java Hello program.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
D*              DEFINE  THE PARAMETERS FOR THE QCMDEXC API
D*
DCMDSTRING         S             25    INZ('JAVA CLASS(''QIBMHello'')')
DCMDLENGTH         S             15P 5 INZ(23)
D*              NOW THE CALL TO QCMDEXC WITH THE 'JAVA' CL COMMAND
C                    CALL      'QCMDEXC'
C                    PARM                    CMDSTRING
C                    PARM                    CMDLENGTH
C*        This next line displays 'DID IT' after you exit the
C*        Java Shell via F3 or F12.
C      'DID IT'      DSPLY
C*        Set On LR to exit the RPG program
C                    SETON                                        LR
C
```

Related concepts

Using sockets for interprocess communication
Sockets streams communicate between programs that are running in separate processes.

Using input and output streams for interprocess communication
Input and output streams communicate between programs that are running in separate processes.

Related reference

Example: Calling Java from ILE C
This is an example of an integrated language environment (ILE) C program that uses the `system()` function to call the Java Hello program.

Java platform

The Java platform is the environment for developing and managing Java applets and applications. It consists of three primary components: the Java language, the Java packages, and the Java virtual machine.

The Java language and packages are similar to C++ and its class libraries. The Java packages contain classes, which are available in any compliant Java implementation. The application programming interface (API) should be the same on any system that supports Java.

Java differs from a traditional language, like C++, in the way it compiles and runs. In a traditional programming environment, you write and compile source code of a program into object code for a specific hardware and operating system. The object code binds to other object code modules to create a running program. The code is specific for a particular set of computer hardware and does not run on other systems without being changed. This figure illustrates the traditional language deployment environment.

Java applets and applications

An applet is a Java program designed to be included in an HTML Web document. You can write your Java applet and include it in an HTML page, much in the same way an image is included. When you use a Java-enabled browser to view an HTML page that contains an applet, the applet's code is transferred to your system and is run by the browser's Java virtual machine.

The HTML document contains tags, which specify the name of the Java applet and its Uniform Resource Locator (URL). The URL is the location at which the applet bytecodes reside on the Internet. When an HTML document containing a Java applet tag is displayed, a Java-enabled Web browser downloads the Java bytecodes from the Internet and uses the Java virtual machine to process the code from within the Web document. These Java applets are what enable Web pages to contain animated graphics or interactive content.

You can also write a Java application that does not require the use of a Web browser.

For more information, see Writing Applets, Oracle's tutorial for Java applets. It includes an overview of applets, directions for writing applets, and some common applet problems.

Applications are stand-alone programs that do not require the use of a browser. Java applications run by starting the Java interpreter from the command line and by specifying the file that contains the compiled application. Applications usually reside on the system on which they are deployed. Applications access resources on the system, and are restricted by the Java security model.

Related concepts

Java virtual machine
The Java virtual machine is a runtime environment that you can add into a web browser or any operating system, such as IBM i. The Java virtual machine runs instructions that a Java compiler generates. It consists of a bytecode interpreter and runtime that allow Java class files to run on any platform, regardless of the platform on which they were originally developed.

Java JAR and class files
A Java ARchive (JAR) file is a file format that combines many files into one. The Java environment differs from other programming environments in that the Java compiler does not generate machine code for a hardware-specific instruction set. Instead, the Java compiler converts Java source code into Java virtual machine instructions, which Java class files store. You can use JAR files to store class files. The class file does not target a specific hardware platform, but instead targets the Java virtual machine architecture.

Java threads
A thread is a single independent stream that runs within a program. Java is a multithreaded programming language, so more than one thread may be running within the Java virtual machine at one time. Java threads provide a way for a Java program to perform multiple tasks at the same time. A thread is essentially a flow of control in a program.

Java Development Kit
The Java Development Kit (JDK) is software for Java developers. It includes the Java interpreter, Java classes, and Java development tools: compiler, debugger, disassembler, appletviewer, stub file generator, and documentation generator.

Java virtual machine

The Java virtual machine is a runtime environment that you can add into a web browser or any operating system, such as IBM i. The Java virtual machine runs instructions that a Java compiler generates. It consists of a bytecode interpreter and runtime that allow Java class files to run on any platform, regardless of the platform on which they were originally developed.

The class loader and security manager, which are part of the Java runtime, insulate code that comes from another platform. They can also restrict which system resources are allowed to be accessed by each class that is loaded.

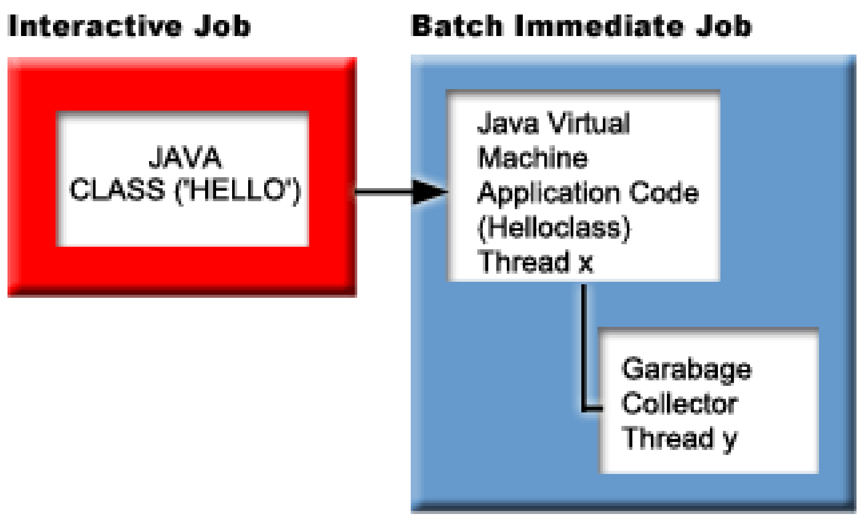
Note: Java applications are not restricted; only applets are restricted. Applications can freely access system resources and use native methods. Most IBM Developer Kit for Java programs are applications.

In addition to loading and running the bytecodes, the Java virtual machine includes a garbage collector that manages memory. "Java garbage collection" on page 351 runs at the same time as the loading and interpretation of the bytecodes.

Java runtime environment

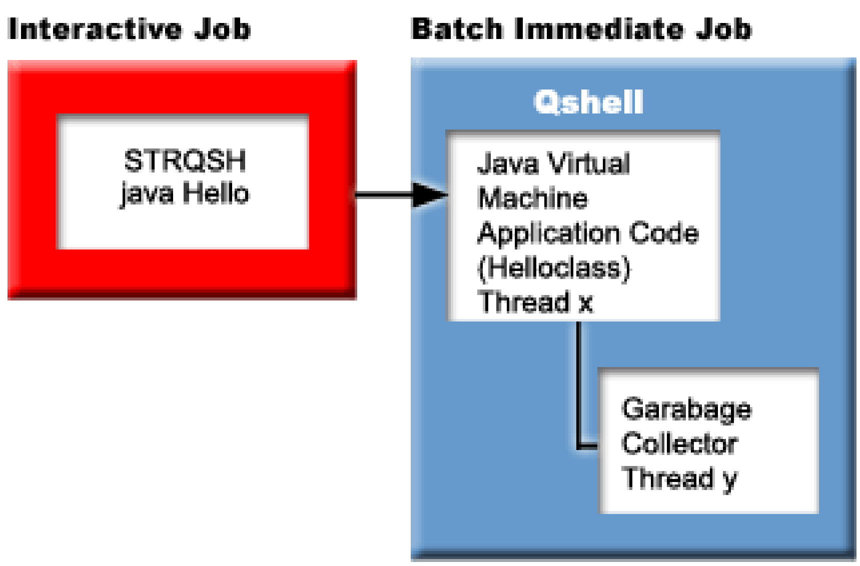
The Java runtime environment starts whenever you enter the Run Java (RUNJVA) command or JAVA command on the IBM i command line. Because the Java environment is multithreaded, it is necessary to run the Java virtual machine in a job that supports threads, such as a batch immediate (BCI) job. As illustrated in the following figure, after the Java virtual machine starts, additional threads may start in the job in which the garbage collector runs.

Figure 1: The typical Java environment when using the RUNJVA or JAVA CL command



It is also possible to start the Java runtime environment by using the `java` command in Qshell from the Qshell Interpreter. In this environment, the Qshell Interpreter is running in a BCI job that is associated with an interactive job. The Java runtime environment starts in the job that is running the Qshell Interpreter.

Figure 2: The Java environment when using the java command in Qshell



When the Java runtime environment starts from an interactive job, the Java Shell Display is shown. This display provides an input line for entering data into the System.in stream, as well as displaying data that is written to the System.out stream and System.err stream.

Java interpreter

The Java interpreter is the part of the Java virtual machine that interprets Java class files for a particular hardware platform. The Java interpreter decodes each bytecode and performs the corresponding operation.

Related concepts

Java applets and applications
An applet is a Java program designed to be included in an HTML Web document. You can write your Java applet and include it in an HTML page, much in the same way an image is included. When you use a Java-enabled browser to view an HTML page that contains an applet, the applet's code is transferred to your system and is run by the browser's Java virtual machine.

Java JAR and class files
A Java ARchive (JAR) file is a file format that combines many files into one. The Java environment differs from other programming environments in that the Java compiler does not generate machine code for a hardware-specific instruction set. Instead, the Java compiler converts Java source code into Java virtual machine instructions, which Java class files store. You can use JAR files to store class files. The class file does not target a specific hardware platform, but instead targets the Java virtual machine architecture.

Java threads
A thread is a single independent stream that runs within a program. Java is a multithreaded programming language, so more than one thread may be running within the Java virtual machine at one time. Java threads provide a way for a Java program to perform multiple tasks at the same time. A thread is essentially a flow of control in a program.

Java Development Kit
The Java Development Kit (JDK) is software for Java developers. It includes the Java interpreter, Java classes, and Java development tools: compiler, debugger, disassembler, appletviewer, stub file generator, and documentation generator.

Running an PASE for i program with QP2TERM()

"Invocation API functions" on page 212
The IBM Developer Kit for Java supports these Invocation API functions.

Java JAR and class files

A Java ARchive (JAR) file is a file format that combines many files into one. The Java environment differs from other programming environments in that the Java compiler does not generate machine code for a hardware-specific instruction set. Instead, the Java compiler converts Java source code into Java virtual machine instructions, which Java class files store. You can use JAR files to store class files. The class file does not target a specific hardware platform, but instead targets the Java virtual machine architecture.

You can use JAR as a general archiving tool and also to distribute Java programs of all types, including applets. Java applets download into a browser in a single Hypertext Transfer Protocol (HTTP) transaction rather than by opening a new connection for each piece. This method of downloading improves the speed at which an applet loads on a Web page and begins functioning.

The JAR format supports compression, which reduces the size of the file and decreases download time. Additionally, an applet author may digitally sign individual entries in a JAR file to authenticate their origin.

To update classes in JAR files, use the `jar` tool.

Java class files are stream files that are produced when a source file is compiled by the Java compiler. The class file contains tables that describe each field and method of the class. The file also contains the bytecodes for each method, static data, and descriptions that are used to represent Java objects.

Related concepts

Java applets and applications

An applet is a Java program designed to be included in an HTML Web document. You can write your Java applet and include it in an HTML page, much in the same way an image is included. When you use a Java-enabled browser to view an HTML page that contains an applet, the applet's code is transferred to your system and is run by the browser's Java virtual machine.

Java virtual machine
The Java virtual machine is a runtime environment that you can add into a web browser or any operating system, such as IBM i. The Java virtual machine runs instructions that a Java compiler generates. It consists of a bytecode interpreter and runtime that allow Java class files to run on any platform, regardless of the platform on which they were originally developed.

Java threads
A thread is a single independent stream that runs within a program. Java is a multithreaded programming language, so more than one thread may be running within the Java virtual machine at one time. Java threads provide a way for a Java program to perform multiple tasks at the same time. A thread is essentially a flow of control in a program.

Java Development Kit
The Java Development Kit (JDK) is software for Java developers. It includes the Java interpreter, Java classes, and Java development tools: compiler, debugger, disassembler, appletviewer, stub file generator, and documentation generator.

Related information

Java jar tool by Oracle.

Java threads

A thread is a single independent stream that runs within a program. Java is a multithreaded programming language, so more than one thread may be running within the Java virtual machine at one time. Java threads provide a way for a Java program to perform multiple tasks at the same time. A thread is essentially a flow of control in a program.

Threads are a modern programming construct that are used to support concurrent programs and to improve the performance and scalability of applications. Most programming languages support threads through the use of add-in programming libraries. Java supports threads as built-in application programming interfaces (APIs).

Note: The use of threads provides the support to increase the interactivity, meaning a shorter wait at the keyboard because more tasks are running in parallel. But, the program is not necessarily more interactive just because it has threads.

Threads are the mechanism for waiting on long running interactions, while still allowing the program to handle other work. Threads have the ability to support multiple flows through the same code stream. They are sometimes called **lightweight processes**. The Java language includes direct support for threads. But, by design, it does not support asynchronous non-blocking input and output with interrupts or multiple wait.

Threads allow the development of parallel programs that scale well in an environment where a machine has multiple processors. If properly constructed, they also provide a model for handling multiple transactions and users.

You can use threads in a Java program for a number of situations. Some programs must be able to engage in multiple activities and still be able to respond to additional input from the user. For example, a Web browser should be able to respond to user input while playing a sound.

Threads can also use asynchronous methods. When you call a second method, you do not have to wait for the first method to complete before the second method continues with its own activity.

There are also many reasons not to use threads. If a program uses inherently sequential logic, one thread can accomplish the entire sequence. Using multiple threads in such a case results in a complex program with no benefits. There is considerable work in creating and starting a thread. If an operation involves only a few statements, it is faster to handle it in a single thread. This can be true even when the operation is conceptually asynchronous. When multiple threads share objects, the objects must synchronize to coordinate thread access and maintain consistency. Synchronization adds complexity to a program, is difficult to tune for optimal performance, and can be a source of programming errors.

For more threads information, see Developing multithreaded applications.

Related concepts

Java applets and applications
An applet is a Java program designed to be included in an HTML Web document. You can write your Java applet and include it in an HTML page, much in the same way an image is included. When you use a Java-enabled browser to view an HTML page that contains an applet, the applet's code is transferred to your system and is run by the browser's Java virtual machine.

Java virtual machine
The Java virtual machine is a runtime environment that you can add into a web browser or any operating system, such as IBM i. The Java virtual machine runs instructions that a Java compiler generates. It consists of a bytecode interpreter and runtime that allow Java class files to run on any platform, regardless of the platform on which they were originally developed.

Java JAR and class files
A Java ARchive (JAR) file is a file format that combines many files into one. The Java environment differs from other programming environments in that the Java compiler does not generate machine code for a hardware-specific instruction set. Instead, the Java compiler converts Java source code into Java virtual machine instructions, which Java class files store. You can use JAR files to store class files. The class file does not target a specific hardware platform, but instead targets the Java virtual machine architecture.

Java Development Kit
The Java Development Kit (JDK) is software for Java developers. It includes the Java interpreter, Java classes, and Java development tools: compiler, debugger, disassembler, appletviewer, stub file generator, and documentation generator.

Java Development Kit

The Java Development Kit (JDK) is software for Java developers. It includes the Java interpreter, Java classes, and Java development tools: compiler, debugger, disassembler, appletviewer, stub file generator, and documentation generator.

The JDK enables you to write applications that are developed once and run anywhere on any Java virtual machine. Java applications that are developed with the JDK on one system can be used on another system without changing or recompiling the code. The Java class files are portable to any standard Java virtual machine.

To find more information about the current JDK, check the version of the IBM Developer Kit for Java on your server.

You can check the version of the default IBM Developer Kit for Java Java virtual machine on your server by entering either of the following commands:

- `java -version` on the Qshell command prompt.
- `RUNJVA CLASS(*VERSION)` on the CL command line.

Then, look for the same version of Oracle. JDK at The Source for Java Technology for specific documentation. The IBM Developer Kit for Java is a compatible implementation of the Oracle Java Technology, so you should be familiar with their JDK documentation.

Java packages

A Java package is a way of grouping related classes and interfaces in Java. Java packages are similar to class libraries that are available in other languages.

The Java packages, which provide the Java APIs, are available as part of Oracle Java Development Kit (JDK). For a complete list of Java packages and information on Java APIs, see Java 2 Platform Packages.

Java tools

For a complete list of tools that Oracle. Java Development Kit supplies, see the Tools Reference by Oracle. For more information about each individual tool that the IBM Developer Kit for Java supports, see Java tools that are supported by the IBM Developer Kit for Java.

Related concepts

Java applets and applications
An applet is a Java program designed to be included in an HTML Web document. You can write your Java applet and include it in an HTML page, much in the same way an image is included. When you use a Java-enabled browser to view an HTML page that contains an applet, the applet's code is transferred to your system and is run by the browser's Java virtual machine.

Java virtual machine
The Java virtual machine is a runtime environment that you can add into a web browser or any operating system, such as IBM i. The Java virtual machine runs instructions that a Java compiler generates. It consists of a bytecode interpreter and runtime that allow Java class files to run on any platform, regardless of the platform on which they were originally developed.

Java JAR and class files
A Java ARchive (JAR) file is a file format that combines many files into one. The Java environment differs from other programming environments in that the Java compiler does not generate machine code for a hardware-specific instruction set. Instead, the Java compiler converts Java source code into Java virtual machine instructions, which Java class files store. You can use JAR files to store class files. The class file does not target a specific hardware platform, but instead targets the Java virtual machine architecture.

Java threads
A thread is a single independent stream that runs within a program. Java is a multithreaded programming language, so more than one thread may be running within the Java virtual machine at one time. Java threads provide a way for a Java program to perform multiple tasks at the same time. A thread is essentially a flow of control in a program.

"Support for multiple Java Development Kits (JDKs)" on page 6
The IBM i platform supports multiple versions of the Java Development Kits (JDKs) and the Java 2 Platform, Standard Edition.

"Native methods and the Java Native Interface" on page 201
Native methods are Java methods that start in a language other than Java. Native methods can access system-specific functions and APIs that are not available directly in Java.

"Java tools and utilities" on page 355
The Qshell environment includes the Java development tools that are typically required for program development.

Java 2 Platform Packages

Tools Reference by Oracle.

Advanced topics

This topic provides instructions on how to run Java in a batch job and describes the Java file authorities required in the integrated file system to display, run, or debug a Java program.

Java classes, packages, and directories

Each Java class is part of a package. The first statement in a Java source file indicates which class is in what package. If the source file does not contain a package statement, the class is part of an unnamed default package.

The package name relates to the directory structure in which the class resides. The integrated file system supports Java classes in a hierarchical file structure that is similar to what you find on most PC and

UNIX systems. You must store a Java class in a directory with a relative directory path that matches the package name for that class. For example, consider the following Java class:

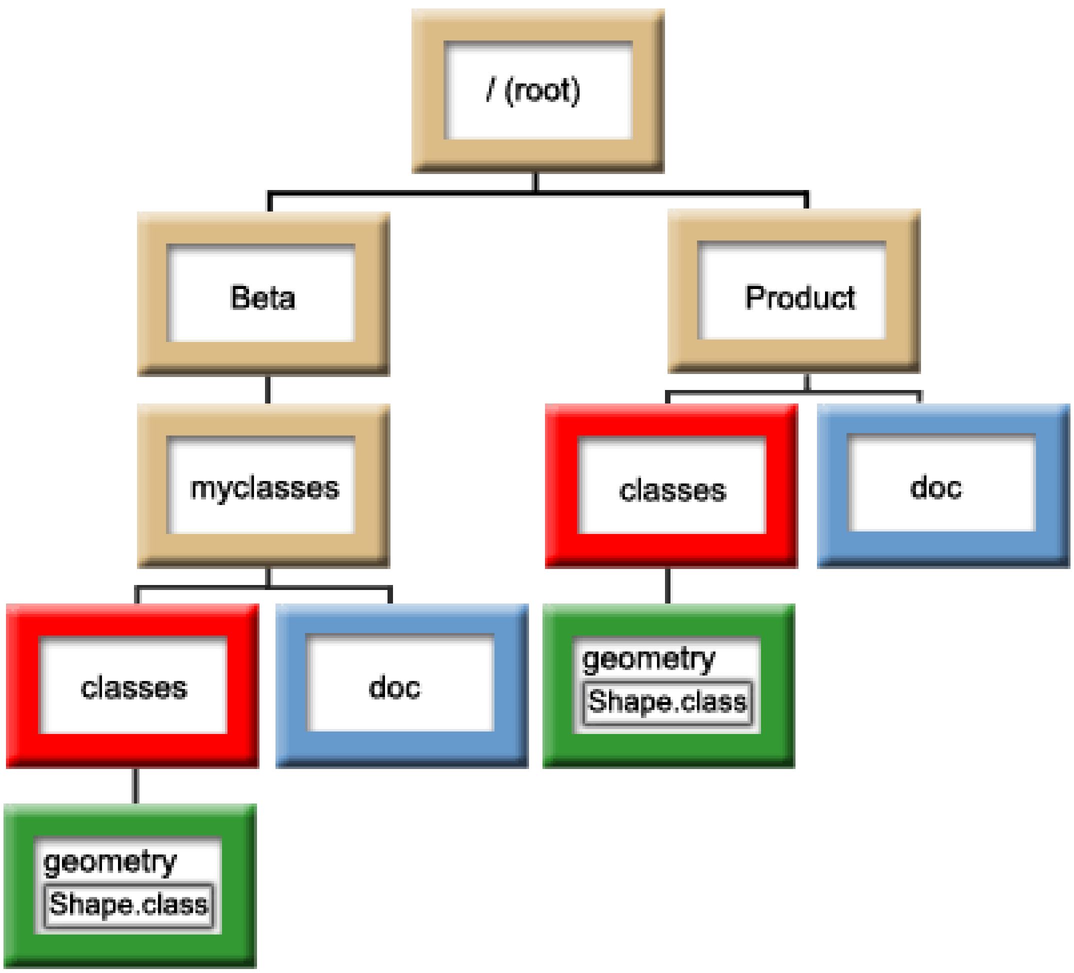
```
package classes.geometry;
import java.awt.Dimension;

public class Shape {

Dimension metrics;

   // The implementation for the Shape class would be coded here ...

}
```

The package statement in the previous code indicates that the Shape class is part of the classes.geometry package. For the Java runtime to find the Shape class, store the Shape class in the relative directory structure classes/geometry.

Note: The package name corresponds to the relative directory name in which the class resides. The Java virtual machine class loader finds the class by appending the relative path name to each directory that you specify in the classpath. The Java virtual machine class loader can also find the class by searching the ZIP files or JAR files that you specify in the classpath.

For example, when you store the Shape class in the /Product/classes/geometry directory in the "root" (/) file system, you need to specify /Product in the classpath.

Figure 1: Example directory structure for Java classes of the same name in different packages



Note: Multiple versions of the Shape class can exist in the directory structure. To use the Beta version of the Shape class, place /Beta/myclasses in the classpath before any other directories or ZIP files that contain the Shape class.

The Java compiler uses the Java classpath, package name, and directory structure to find packages and classes when compiling Java source code. For more information, see "Java classpath" on page 12.

Related concepts

Java-related files in the IFS
The integrated file system (IFS) stores Java-related class, source, ZIP, and JAR files in a hierarchical file structure. IBM Developer Kit for Java supports using the threadsafe file systems in the IFS to store and work with your Java-related class files, source files, ZIP files, and JAR files.

Java file authorities in the integrated file system
To run or debug a Java program, the class file, JAR file, or ZIP file needs to have read authority (*R). Directories need read and execute authorities (*RX).

Related tasks

Running Java in a batch job
Java programs run in a batch job by using the Submit Job (SBMJOB) command. In this mode, the Java Qshell Command Entry display is not available to handle the System.in, System.out, nor System.err streams.

Java-related files in the IFS

The integrated file system (IFS) stores Java-related class, source, ZIP, and JAR files in a hierarchical file structure. IBM Developer Kit for Java supports using the threadsafe file systems in the IFS to store and work with your Java-related class files, source files, ZIP files, and JAR files.

Related concepts

Java classes, packages, and directories
Each Java class is part of a package. The first statement in a Java source file indicates which class is in what package. If the source file does not contain a package statement, the class is part of an unnamed default package.

Java file authorities in the integrated file system
To run or debug a Java program, the class file, JAR file, or ZIP file needs to have read authority (*R). Directories need read and execute authorities (*RX).

Related tasks

Running Java in a batch job
Java programs run in a batch job by using the Submit Job (SBMJOB) command. In this mode, the Java Qshell Command Entry display is not available to handle the System.in, System.out, nor System.err streams.

Related information

File system considerations for multithreaded programming

File system comparison

Java file authorities in the integrated file system

To run or debug a Java program, the class file, JAR file, or ZIP file needs to have read authority (*R). Directories need read and execute authorities (*RX).

Note: Files and directories that do not have execute authority (*X) always appear to have execute authority (*X) to a user with QSECOFR authority. Different users can get different results in certain situations, even though both users appear to have the same access to the same files. This is important to know when running shell scripts using the Qshell Interpreter or java.Runtime.exec().

For example, one user writes a Java program that uses java.Runtime.exec() to call a shell script, then tests it using a user ID with QSECOFR authority. If the file mode of the shell script has read and write authority (*RW), the integrated file system allows the user ID with QSECOFR authority to run it. However, a non-QSECOFR authority user could try to run the same Java program, and the integrated file system would tell the java.Runtime.exec() code that the shell script cannot be run, because *X is missing. In this case, java.Runtime.exec() throws an input and output exception.

You can also assign authorities to new files created by Java programs in an integrated file system. By using the os400.file.create.auth system property for files and os400.dir.create.auth for directories, any combination of read, write, and execute authorities may be used.

For more information, see the Program and CL Command APIs or the Integrated file system.

Related concepts

Java classes, packages, and directories
Each Java class is part of a package. The first statement in a Java source file indicates which class is in what package. If the source file does not contain a package statement, the class is part of an unnamed default package.

Java-related files in the IFS
The integrated file system (IFS) stores Java-related class, source, ZIP, and JAR files in a hierarchical file structure. IBM Developer Kit for Java supports using the threadsafe file systems in the IFS to store and work with your Java-related class files, source files, ZIP files, and JAR files.

Related tasks

Running Java in a batch job
Java programs run in a batch job by using the Submit Job (SBMJOB) command. In this mode, the Java Qshell Command Entry display is not available to handle the System.in, System.out, nor System.err streams.

Running Java in a batch job

Java programs run in a batch job by using the Submit Job (SBMJOB) command. In this mode, the Java Qshell Command Entry display is not available to handle the System.in, System.out, nor System.err streams.

You may redirect these streams to other files. Default handling sends the System.out and System.err streams to a spooled file. The batch job, which results in an input and output exception for read requests from System.in, owns the spooled file. You can redirect System.in, System.out, and System.err within your Java program. You can also use the os400.stdin, os400.stdout, and os400.stderr system properties to redirect System.in, System.out, and System.err.

Note: SBMJOB sets the current working directory (CWD) to the HOME directory that is specified in the user profile.

Example: Running Java in a Batch Job

```
SBMJOB CMD(JAVA QIBMHello OPTION(*VERBOSE)) CPYENVVAR(*YES)
```

Running the JAVA command in the previous example spawns a second job. Therefore, the subsystem that the batch job runs in must be capable of running more than one job.

You can verify that your batch job is capable of running more than one job by following these steps:

1. On the CL command line, enter `DSPSBSD(MYSBSD)`, where `MYSBSD` is the subsystem description of your batch job.
2. Choose option 6, Job queue entries.
3. Look at the Max Active field for your job queue.

Related concepts

Java classes, packages, and directories
Each Java class is part of a package. The first statement in a Java source file indicates which class is in what package. If the source file does not contain a package statement, the class is part of an unnamed default package.

Java-related files in the IFS
The integrated file system (IFS) stores Java-related class, source, ZIP, and JAR files in a hierarchical file structure. IBM Developer Kit for Java supports using the threadsafe file systems in the IFS to store and work with your Java-related class files, source files, ZIP files, and JAR files.

Java file authorities in the integrated file system

To run or debug a Java program, the class file, JAR file, or ZIP file needs to have read authority (*R). Directories need read and execute authorities (*RX).

Running your Java application on a host that does not have a GUI

If you want to run your Java application on a host that does not have a graphical user interface (GUI), such as an IBM i server, you can use the Native Abstract Windowing Toolkit (NAWT).

Use NAWT to provide your Java applications and servlets with the full capability of the Java 2 Platform, Standard Edition (J2SE) AWT graphics functionality.

Native Abstract Windowing Toolkit

Native Abstract Windowing Toolkit (NAWT) is a not really a toolkit but rather a term that has evolved to refer to the native IBM i support that provides Java applications and servlets with the capability to use the Abstract Windowing Toolkit (AWT) graphics functions offered by the Java Platform, Standard Edition (J2SE).

Most of the information needed to use AWT can be obtained by starting with the following links:

- Abstract Window Toolkit
- Java Internationalization FAQ

Selecting an AWT mode

There are two modes available when using AWT: normal and headless. One factor in choosing which mode to use is whether you will need to use heavyweight AWT components.

Normal Mode

Normal mode must be used if your application uses the Java AWT API to display windows, frames, dialog boxes, or similar heavyweight components. Use normal mode if you expect your application to receive mouse operation events or keyboard input. Normal mode is the default mode and nothing needs to be specified to enable it.

Headless mode

Headless mode can be used if your Java application does not interact directly with a user. This means that your Java application does not display windows or dialog boxes, does not accept keyboard or mouse input, and does not use any heavyweight AWT components. This mode is selected by specifying Java property *java.awt.headless=true* on the Java invocation. By using headless mode, you avoid the need for a VNC/X-server.

Examples of applications that can use headless mode include:

- Servlets or other server based programs that only use the AWT API to create images to be included in a data stream returned to a remote user
- Any program that only creates or manipulates images or image files without actually displaying them using heavyweight AWT components

The default value for the Java property *java.awt.headless* is false.

Heavyweight AWT components

The following items are considered heavyweight AWT components. If your application requires these, use normal mode:

Table 8. Heavyweight AWT components			
Heavyweight AWT components			
Applet	Frame	List	Robot
Button	JApplet	Menu	Scrollbar
Checkbox	JDialog	MenuBar	ScrollPane
Choice	JFrame	MenuComponent	TextArea
Dialog	JWindow	MenuItem	TextComponent
FileDialog	Label	PopupMenu	Window

Using AWT in normal mode with full graphical user interface support

In order to support a graphical user interface, a windowing system is required. The supported choice for IBM i Java is the Virtual Network Computing (VNC) server. The VNC server is well-suited to the system because it does not require a dedicated mouse, keyboard, and graphics-capable monitor. IBM provides a version of the VNC server that runs in PASE for i. Follow these instructions to ensure that VNC is installed, started, and your Java session is configured to use it.

Before you can test or begin using AWT, perform the following required and optional steps:

- Create a VNC password. This is required one time for each user profile that will be used to start a VNC server.
- Start the VNC server, typically after each system IPL.
- Configure AWT environment variables, once in each session before the first time you run Java and use the AWT API.
- Configure Java system properties. This must be done each time you run Java.
- Optional, for interactive use: Configure the iceWM window manager
- Optional, for direct interaction with a user: Use a VNC viewer or web browser to connect to VNC
- Optional: Verify your AWT configuration

Creating a VNC password file

To use the Native Abstract Windowing Toolkit (NAWT) with a Virtual Network Computing (VNC) server, you need to create a VNC password file.

The VNC server default setting requires a password file that it uses to protect the VNC display against unauthorized user access. You should create the VNC password file under the profile to be used to start the VNC server. Enter the following at an IBM i command prompt:

1. `MKDIR DIR('/home/VNCprofile/.vnc')`
2. `QAPTL/VNCPASSWD USEHOME(*NO) PWDFILE('/home/VNCprofile/.vnc/passwd')` where *VNCprofile* is the profile that started the VNC server.

To gain interactive access to the VNC server using a VNCviewer or a Web browser from a remote system, users must use the password that you specify in this step.

Starting the VNC Server

To the Virtual Network Computing (VNC) server, complete the following steps.

where *n* is the display number that you want to use. Display numbers can be any integer in the range 1-99.

The .Xauthority file

The process of starting the VNC server either creates a new .Xauthority file or modifies an existing .Xauthority file. VNC server authorization uses the .Xauthority file, which contains encrypted key information, to prevent applications of other users from intercepting your X server requests. Secure communications between the Java virtual machine (JVM) and VNC **REQUIRES** that both the JVM and VNC have access to the encrypted key information in the .Xauthority file.

The .Xauthority file belongs to the profile that started VNC. The simplest way to allow both the JVM and the VNC server to share access to the .Xauthority file is to run the VNC server and the JVM under the same user profile. If you cannot run both the VNC server and the JVM under the same user profile, you can configure the XAUTHORITY environment variable to point to the correct .Xauthority file.

To start the Virtual Network Computing (VNC) server, type the following command at the command line and press **ENTER**: `CALL PGM(QSYS/QP2SHELL) PARM('/QOpenSys/QIBM/ProdData/DeveloperTools/vnc/vncserver_java' ':n'))` where *n* is the display number that you want to use. Display numbers can be any integer in the range 1-99.

Starting the VNC server displays a message that identifies the IBM i server system name and display number; for example, `New 'X'desktop is systemname:1`. Make a note of the system name and display number as you will use it later to configure the DISPLAY environment variable when you run your Java application that uses AWT.

If you have more than one VNC server running at the same time, each VNC server will require a unique display number. Explicitly specifying the display value when you start the VNC server as shown lets you control which display number is used by each application. Alternatively, if you do not want to specify the display number, remove `':n'` from the previous command, let the vncserver_java program find an available display number, and note that display number.

.Xauthority file

The process of starting the VNC server either creates a new .Xauthority file or modifies an existing .Xauthority file in the home directory of the user starting the server. The .Xauthority file contains encrypted key authorization information which the VNC server uses to prevent applications of other users from intercepting your X server requests. Secure communications between the Java virtual machine (JVM) and VNC **requires** that both the JVM and VNC have access to the same .Xauthority file.

The .Xauthority file belongs to the profile that started VNC. To allow both the JVM and the VNC server to share access, run the VNC server and the JVM under the same user profile. If this is not possible, you can configure the XAUTHORITY environment variable to point to the correct .Xauthority file.

Configuring NAWT environment variables
When you run Java and wish to use full AWT graphical user interface support, you must have the DISPLAY and XAUTHORITY environment variables defined to tell Java which X-server display to use and where to find the correct .Xauthority file.

DISPLAY environment variable

In the session where you want to run Java programs, set the DISPLAY environment variable to your system name and display number. Type the following command at an IBM i command prompt and press ENTER:

```
ADDENVVAR ENVVAR(DISPLAY) VALUE('systemname:n')
```

where *systemname* is the host name or IP address of your system and *n* is the display number of the VNC server to be used.

XAUTHORITY environment variable

In the session where you want to run Java programs, set the XAUTHORITY environment variable to `/home/VNCprofile/.Xauthority`, where *VNCprofile* is the profile that started the VNC server. From an IBM i command prompt, run the command:

```
ADDENVVAR ENVVAR(XAUTHORITY) VALUE('/home/VNCprofile/.Xauthority')
```

replacing *VNCprofile* with the appropriate profile name.

Configuring the iceWM window manager

Configure the iceWM window manager (iceWM), as an optional step during NAWT set up, when you want to interactively use the Virtual Network Computing (VNC) server. For example, you may want to run a Java application that features a graphical user interface (GUI). iceWM is a small but powerful window manager included in the IBM i Tools For Developers PRPQ.

Running in the background, iceWM controls the look and feel of windows running within the X Window environment of the VNC server. iceWM provides an interface and a set of features that are similar to many popular window managers. The default behavior of the included vncserver_java script starts the VNC server and runs iceWM.

Completing this step creates several configuration files that iceWM requires. If you want, you can also disable iceWM.

Configuring iceWM

To configure the iceWM window manager, complete the following steps at an IBM i command prompt. Be sure to perform these steps under the profile that you use to start the VNC server.

1. Type the following command and press **ENTER** to start the installation:

   ```
   STRPTL CLIENT(IGNORE)
   ```

   The IGNORE value functions as a placeholder that ensures the command activates only the configuration features of STRPTL that NAWT requires.
2. Type the following command and press **ENTER** to sign off:

   ```
   SIGNOFF
   ```

   Signing off ensures that any session-specific results of the STRPTL command do not affect subsequent actions that you perform to use or configure NAWT.

Note: Run the STRPTL command only once for each profile that starts a VNC server. NAWT does not require any of the available optional arguments for the command. These statements override any setup instructions for STRPTL associated with the 5799-PTL IBM i Tools For Developers PRPQ.

Disabling iceWM

Starting the VNC server creates or modifies an existing script file called xstartup_java that contains the command to run iceWM. The xstartup_java script file resides in the following integrated file system directory:

```
/home/VNCprofile/.vnc/
```

where *VNCprofile* is the name of the profile that started the VNC server.

To completely disable iceWM, use a text editor to either comment out or remove the line in the script that starts iceWM. To comment out the line, insert a pound sign character (#) at the beginning of the line.

Using a VNCviewer or Web browser

To run an application that features a graphical user interface (GUI) on an IBM i server, you must use either a VNCviewer or a Web browser to connect to the Virtual Network Computing (VNC) server. You must run the VNCviewer or Web browser on a graphics-capable platform, such as a personal computer.

Note: The following steps require you to know your display number and VNC password. Starting the Virtual Network Computing (VNC) server determines the value for the display number. Creating a VNC password file sets the VNC password.

Using a VNCviewer to access the VNC server

To use a VNCviewer to connect to the VNC server, complete the following steps:

1. Download and install the VNCviewer application:

   - VNC viewers are available for most platforms from the RealVNC Web site

2. Start the VNCviewer that you downloaded. At the prompt, enter the system name and display number and click **OK**.
3. At the password prompt, type the VNC password to gain access to the VNC server display.

Using a Web browser to access the VNC server

To use a Web browser to connect to the VNC server, complete the following steps:

1. Start the browser and access the following URL:

   ```
   http://systemname:58nn
   ```

   where:

   - *systemname* is the name or IP address of the system that is running the VNC server
   - *nn* is the 2-digit representation of the VNC server display number

     For example, when the system name is `system_one` and the display number is 2, the URL is:

     ```
     http://system_one:5802
     ```

2. Successfully accessing the URL displays a prompt for the VNC server password. At the password prompt, type the VNC password to gain access to the VNC server display.

Tips on using VNC
Use IBM i control language (CL) commands to start and stop a Virtual Network Computing (VNC) server, and to display information about the currently running VNC servers.

Starting a VNC display server from a CL program

The following example is one way to set the DISPLAY environment variable and start VNC automatically by using control language (CL) commands:

```
CALL QP2SHELL PARM('/QOpenSys/QIBM/ProdData/DeveloperTools/vnc/vncserver_java' ':n')
ADDENVVAR ENVVAR(DISPLAY) VALUE('systemname:n')
```

where:

- *systemname* is the host name or IP address of the system where VNC is running
- *n* is the numeric value that represents the display number that you want to start

Note: The example assumes that you are not already running display :*n* and that you have successfully created the required VNC password file. For more information about creating a password file, see Creating a VNC password file.

Stopping a VNC display server from a CL program

The following code shows one way to stop a VNC server from a CL program:

```
    CALL QP2SHELL PARM('/QOpenSys/QIBM/ProdData/DeveloperTools/vnc/vncserver_java' '-kill'
':n')
```

where *n* is the numeric value that represents the display number that you want to terminate.

Checking for running VNC display servers

To determine what (if any) VNC servers are currently running on the system, complete the following steps:

1. From an IBM i command line, start a PASE for i shell:

   ```
   CALL QP2TERM
   ```

2. From the PASE for i shell prompt, use the ps command to list the VNC servers:

```
ps gaxuw | grep Xvnc
```

The resulting output from this command will reveal running VNC servers in the following format:

```
john  418  0.9  0.0  5020  0  - A  Jan 31 222:26
  /QOpenSys/QIBM/ProdData/DeveloperTools/vnc/Xvnc :1 -desktop X -httpd
jane   96  0.2  0.0   384  0  - A  Jan 30 83:54
  /QOpenSys/QIBM/ProdData/DeveloperTools/vnc/Xvnc :2 -desktop X -httpd
```

Where:

- The first column is the profile which started the server.
- The second column is the process ID of the server.
- The information starting with */QOpensys/* is the command that started the VNC server (including arguments). The display number typically is the first item in the argument list for the Xvnc command.

  Note: The Xvnc process, shown in the previous example output, is the name of the actual VNC server program. You start Xvnc when you run the vncserver_java script, which prepares the environment and parameters for Xvnc and then starts Xvnc.

Tips for using AWT with WebSphere Application Server
If you must run your WebSphere-based applications in full GUI mode, rather than using headless mode, use these tips for avoiding problems with the connection between WebSphere and the VNC server.

Ensuring secure communications

The VNC server uses a method called X Authority checking to ensure secure communications between itself and the using application, such as WebSphere.

The process of starting the VNC server creates an .Xauthority file that contains encrypted key information. In order for the WebSphere Application Server to access VNC, it **must** have access to, and use, the same .Xauthority file that the VNC server is using.

To accomplish this, use one of the following methods:

Run WebSphere Application Server and VNC using the same profile

If both WebSphere Application Server and the VNC server it is to use are started by the same user profile, then by default they will both be using the same .Xauthority file. This requires you to either start the VNC server from the WebSphere default user (QEJBSVR) or change the WebSphere default user to the profile used to start the VNC server.

To switch the user profile for the application server from the default user (QEJBSVR) to a different profile, you must perform the following actions:

1. Use the WebSphere Application Server administrative console to change the application server configuration
2. Use System i Navigator to enable the new profile

Run WebSphere Application Server and VNC using different profiles

In this case, the WebSphere Application Server is started by one user profile and the .Xauthority file is owned by a different user profile. To enable the WebSphere Application Server to start the VNC server, complete the following steps:

1. Create a new .Xauthority file (or update an existing .Xauthority file) by starting the VNC server from the desired user profile. For example, from an IBM i control language (CL) command line, type the following command and press ENTER: :

   ```
   CALL QP2SHELL PARM('/QOpenSys/QIBM/ProdData/DeveloperTools/vnc/vncserver_java' ':n')
   ```

   where *n* is the display number (a numeric value in the range of 1-99).

Note: The .Xauthority file resides in the directory for the profile under which you are running the VNC server.

2. Use the following CL commands to grant the profile under which you run WebSphere Application Server the authority to read the .Xauthority file:

```
CHGAUT OBJ('/home') USER(WASprofile) DTAAUT(*RX)
CHGAUT OBJ('/home/VNCprofile') USER(WASprofile) DTAAUT(*RX)
CHGAUT OBJ('/home/VNCprofile/.Xauthority') USER(WASprofile) DTAAUT(*R)
```

   where *VNCprofile* and *WASprofile* are the appropriate profiles under which you are running the VNC server and WebSphere Application Server.

   Note: You should only follow these steps when the *VNCprofile* and *WASprofile* are different profiles. Following these steps when *VNCprofile* and *WASprofile* are the same profile can cause VNC to not function correctly.

3. From the WebSphere Application Server administrative console, define the DISPLAY and XAUTHORITY environment variables for your application:
   - For DISPLAY, use either: `system:n` or `localhost:n`

     where *system* is the name or IP address of your system and *n* is the display number that you used to start the VNC server.
   - For XAUTHORITY, use: `/home/VNCprofile/.Xauthority`

     where *VNCprofile* is the profile that started the VNC server.
4. Pick up the configuration changes by restarting WebSphere Application Server.

WebSphere Application Server for IBM i

Verifying your AWT configuration

You can verify your AWT configuration by running a Java test program.

To run the test program from an IBM i command line, enter one of the following commands, depending on the mode that you wish to test:

```
JAVA CLASS(NAWTtest) CLASSPATH('/QIBM/ProdData/Java400')
```

Or

```
JAVA CLASS(NAWTtest) CLASSPATH('/QIBM/ProdData/Java400') PROP((java.awt.headless true))
```

The test program creates a JPEG-encoded image and saves it to the following path in the integrated file system:

```
/tmp/NAWTtest.jpg
```

After you run the test program, check to ensure that the test program created the file and produced no Java exceptions. To display the image, use binary mode to upload the image file to a graphics-capable system and display it with a browser, paint program, or similar tool.

Java security

This topic provides details on adopted authority and explains how you can use SSL to make socket streams secure in your Java application.

Java applications are subject to the same security restrictions as any other program on the IBM i platform. To run a Java program on a IBM i server, you must have authority to the class file in the integrated file system. Once the program starts, it runs under the user's authority.

The majority of the Java programs that run on a IBM i server are applications, not applets, so the "sandbox" security model does not restrict them.

Note: JAAS, JCE, JGSS, and JSSE are part of the base JDK and are not considered to be extensions.

Changes to adopted authority in IBM i 7.2

Support for adopted user profile authority through Java programs is not supported in IBM i 7.2. This topic describes how to determine if your applications are using adopted authority and how to modify your applications to accommodate this change.

In IBM i 7.2, Java applications are no longer able to adopt user profile authority through Java programs.

These topics describes some common situations where Java adopted authority is used and how Java applications may be modified to remove dependence on Java adopted authority.

Determining if your applications use adopted authority

You can use a tool in IBM i 6.1, and 7.1 to help you determine if you have Java applications that will be affected by the changes to adopted authority. The tool works with the JVM to report Java programs that the JVM has logged as using adopted authority.

This tool also helps identify Java applications that may rely on adopted authority by scanning the system for Java programs created with support for adopted authority.

By default, this tool displays the JVM's logged uses of adopted authority and also scans the entire system. However, it also supports several options from Qshell:

```
usage: /qsys.lib/qjava.lib/qjvaadptl.pgm [option]...
        Valid options include
          -h              : Show this usage statement.
          -o <file>       : Write output to the specified file.
          -d <directory> : Scan only the specified directory tree.
          -noscan         : Do not scan system. Report only logged uses.
```

The output from the tool can help you determine what Java applications on your system are using adopted authority. Using this information, you may need to do the following:

- If the usage is in code that you have purchased, contact the vendor to find out what their plans are relative to adopted authority.
- If the usage is in your code, read through the solutions outlined in this document, and see if you are willing and able to modify your code to no longer use adopted authority.

Use of adopted authority

Because adopted authority is only useful for performing operations on IBM i objects, and database records, or for accessing native methods, the examples in this topic collection will focus on those areas. For a basic explanation of adopted authority, see Objects that adopt the owner's authority in the Security reference topic.

Using adopted authority, a method could adopt the authority of the owner of the program being run rather than the authority of the person running the program in order to accomplish some operation. Conceptually this is very similar to `Set UID` and `Set GID` on UNIX, and the canonical example of use is `Change Password` as built in UNIX. While it is not reasonable for each user to have authority to change the password file, it is advantageous for the program to be trusted and allowed to change the password for that user.

System i was able to provide the adopted authority feature for Java programs because the JVM was part of the Trusted Computing Base in system licensed internal code (SLIC). As IBM i moves to a Java implementation where the JVM is a user level program, Java can no longer provide this feature.

In order to achieve the same functionality, the most common approach will be to add an ILE native method to the Java program that adopts equivalent user profile authority and performs the required operation. It may also be possible to achieve the same effects as adopted authority without adding native methods by running the function in a separate process that has increased authority, and sending requests to that program as needed.

Related concepts

Java security model
You can download Java applets from any system; thus, security mechanisms exist within the Java virtual machine to protect against malicious applets. The Java runtime system verifies the bytecodes as the Java virtual machine loads them. This ensures that they are valid bytecodes and that the code does not violate any of the restrictions that the Java virtual machine places on Java applets.

Java Cryptography Extension
The Java Cryptography Extension (JCE) provides a framework and implementations for encryption, key generation, and key agreement, as well as Message Authentication Code (MAC) algorithms. Support for encryption includes symmetric, asymmetric, block, and stream ciphers. The software also supports secure streams and sealed objects. JCE supplements the Java 2 platform, which already includes interfaces and implementations of message digests and digital signatures.

Java Secure Socket Extension
Java Secure Socket Extension (JSSE) is like a framework that abstracts the underlying mechanisms of both Secure Sockets Layer (SSL) and Transport Layer Security (TLS). By abstracting the complexity and peculiarities of the underlying protocols, JSSE enables programmers to use secure encrypted communications while at the same time minimizing possible security vulnerabilities. Java Secure Socket Extension (JSSE) uses both the SSL protocol and the TLS protocol to provide secure encrypted communications between your clients and servers.

Java Authentication and Authorization Service
The Java Authentication and Authorization Service (JAAS) is a standard extension to the Java 2 Platform, Standard Edition (J2SE). J2SE provides access controls that are based on where the code originated and who signed the code (code source-based access controls). It lacks, however, the ability to enforce additional access controls based on who runs the code. JAAS provides a framework that adds this support to the Java 2 security model.

IBM Java Generic Security Service (JGSS)
The Java Generic Security Service (JGSS) provides a generic interface for authentication and secure messaging. Under this interface you can plug a variety of security mechanisms based on secret-key, public-key, or other security technologies.

Examples: Adopted authority alternatives

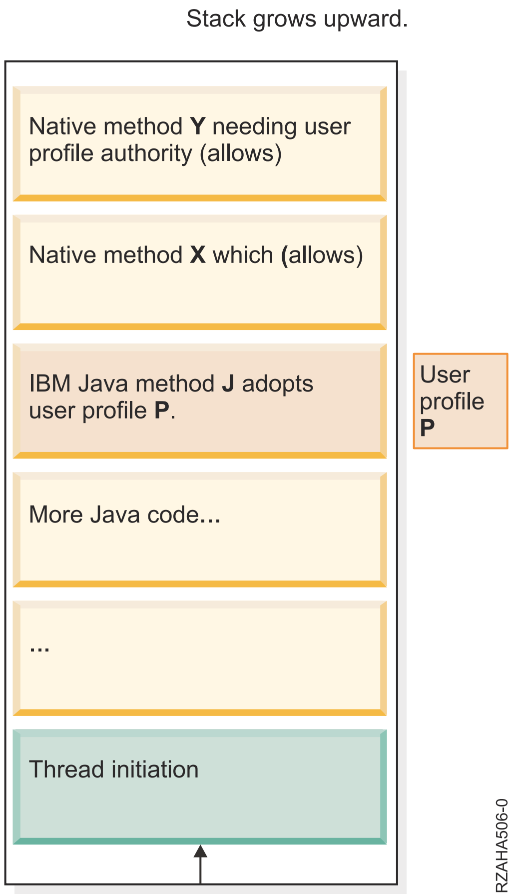
This topic contains some examples of how Java adopted authority was used and some suggested alternatives. These alternatives use native methods in ILE service programs to adopt authority in a manner similar to the Java examples.

Other alternatives are possible and may be preferable in certain circumstances. One possibility is to swap the user profile of the process in order to acquire additional authority. This topic does not describe swapping the user profile, which has its own set of issues and risks. The examples included in this topic collection will demonstrate two common situations where Java adopted authority is used and offer possible alternatives.

- “Example 1: Java method adopts authority immediately before calling a native method” on page 243
- “Alternative 1A: Repackaging native method X” on page 245
- “Alternative 1B: New native method N” on page 247
- “Example 2: Java method adopts authority and calls other Java methods before calling a native method” on page 249
- “Alternative 2: New native method N” on page 251
- “Compile commands for the examples” on page 252

Note: By using the code examples, you agree to the terms of the “Code license and disclaimer information” on page 489.

Example 1: Java method adopts authority immediately before calling a native method



In this example, IBM Java method **J** is contained in a Java program, which adopts user profile **P** and directly calls native method **X**. Native method **X** calls ILE method **Y**, which requires the adopted authority.

JA61Example1.java

```
public class JA61Example1 {
    public static void main(String args[]) {
    int returnVal = J();
    if (returnVal > 0) {
        System.out.println("Adopted authority successfully.");
    }
    else {
        System.out.println("ERROR: Unable to adopt authority.");
    }
    }

    static int J() {
    return X();
    }

    // Returns: 1 if able to successfully access *DTAARA JADOPT61/DATAAREA
    static native int X();

    static {
    System.loadLibrary("EX1");
    }
}
```

JA61Example1.h

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class JA61Example1 */

#ifndef _Included_JA61Example1
#define _Included_JA61Example1
```

```
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     JA61Example1
 * Method:    X
 * Signature: ()I
 */
JNIEXPORT jint JNICALL Java_JA61Example1_X
  (JNIEnv *, jclass);

#ifdef __cplusplus
}
#endif
#endif
```

JA61Example1.c

```
/* This contains the source code for native method Java_JA61Example1_X. This
   module is bound into service program JADOPT61/EX1. */

#include "JA61Example1.h"
#include "JA61ModuleY.h"

/*
 * Class:     JA61Example1
 * Method:    X
 * Signature: ()I
 */
JNIEXPORT jint JNICALL Java_JA61Example1_X(JNIEnv* env, jclass klass) {
   return methodY();
}
```

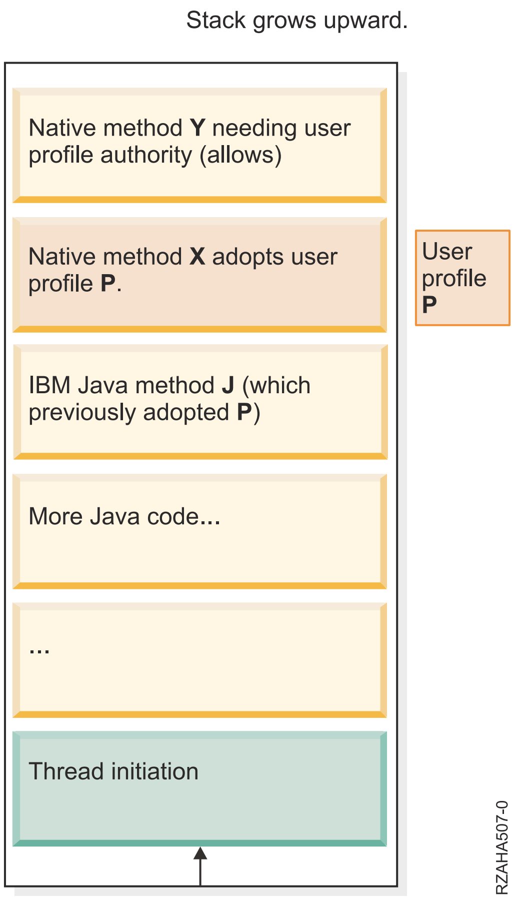
JA61ModuleY.h

```
/* This method tries to change *DTAARA JADOPT61/DATAAREA. This
   will only be possible if the JADOPT61UP user profile has been adopted. */
int methodY(void);
```

JA61ModuleY.c

```
#include <except.h>
#include <stdio.h>
#include "JA61ModuleY.h"
#include <xxdtaa.h>

#define START 1
#define LENGTH 8


/* This method tries to operate on *DTAARA JADOPT61/DATAAREA. This
 will only be possible if the JADOPT61UP user profile has been adopted. */
int methodY(void) {
   int returnValue;
   volatile int com_area;
   char newdata[LENGTH] = "new data";
   _DTAA_NAME_T dtaname = {"DATAAREA  ", "JADOPT61  "};

   /* Monitor for exception in this range */
#pragma exception_handler(ChangeFailed, 0, _C1_ALL, _C2_MH_ESCAPE)
   /* change the *DTAARA JADOPT61/DATAAREA */
   QXXCHGDA(dtaname, START, LENGTH, newdata);
#pragma disable_handler

ChangeCompleted:
   printf("Successfully updated data area\n");
   returnValue = 1;
   goto TestComplete;

ChangeFailed:      /* Control goes here for an exception */
   printf("Got an exception.\n");
   returnValue = 0;

TestComplete:
   printf("methodY completed\n");
   return returnValue;
}
```

Alternative 1A: Repackaging native method X



One way to preserve authority adoption is to separate native method **X** into a new service program. That new service program may then adopt user profile **P**, which was previously adopted by Java method **J**.

JA61Alternative1A.java

```
public class JA61Alternative1A {
    public static void main(String args[]) {
    int returnVal = J();
    if (returnVal > 0) {
        System.out.println("Adopted authority successfully.");
    }
    else {
        System.out.println("ERROR: Unable to adopt authority.");
    }
    }

    static int J() {
    return X();
    }

    // Returns: 1 if able to successfully access *DTAARA JADOPT61/DATAAREA
    static native int X();

    static {
    System.loadLibrary("ALT1A");
    }
}
```

JA61Alternative1A.h

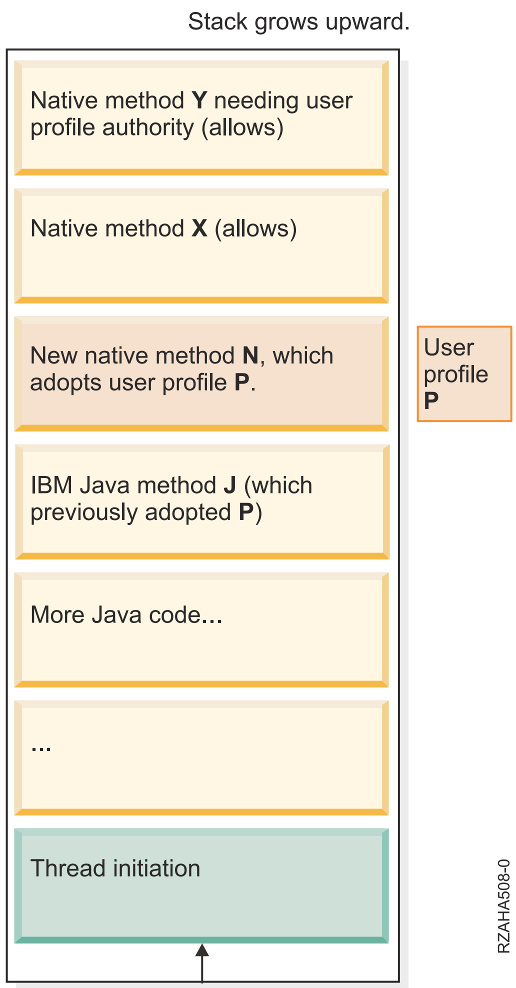
```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class JA61Alternative1A */

#ifndef _Included_JA61Alternative1A
#define _Included_JA61Alternative1A
```

```
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     JA61Alternative1A
 * Method:    X
 * Signature: ()I
 */
JNIEXPORT jint JNICALL Java_JA61Alternative1A_X
  (JNIEnv *, jclass);

#ifdef __cplusplus
}
#endif
#endif
```

JA61Alternative1A.c

```
/* This contains the source code for native method Java_JA61Alternative1A_X. This
   module is bound into service program JADOPT61/ALT1A.*/

#include "JA61Alternative1A.h"
#include "JA61ModuleY.h"

/*
 * Class:     JA61Alternative1A
 * Method:    X
 * Signature: ()I
 */
JNIEXPORT jint JNICALL Java_JA61Alternative1A_X(JNIEnv* env, jclass klass) {
   return methodY();
}
```

Alternative 1B: New native method N



Another way to preserve user profile adoption is to create an entirely new native method **N** contained in a service program which adopts user profile **P**. This new method is called by Java method **J** and calls native method **X**. Java method **J** would need to be changed to call **N** instead of **X**, but native method **X** would not need to be changed or repackaged.

JA61Alternative1B.java

```
public class JA61Alternative1B {
    public static void main(String args[]) {
    int returnVal = J();
    if (returnVal > 0) {
        System.out.println("Adopted authority successfully.");
    }
    else {
        System.out.println("ERROR: Unable to adopt authority.");
    }
    }

    static int J() {
    return N();
    }

    // Returns: 1 if able to successfully access *DTAARA JADOPT61/DATAAREA
    static native int N();

    static {
    System.loadLibrary("ALT1B");
    }
}
```

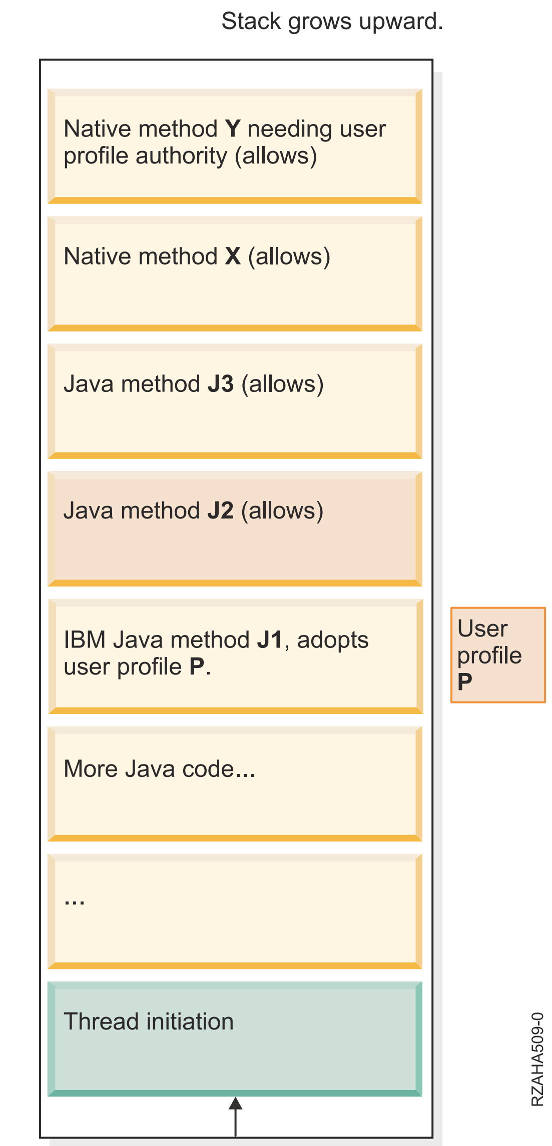
JA61Alternative1B.h

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class JA61Alternative1B */

#ifndef _Included_JA61Alternative1B
#define _Included_JA61Alternative1B
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     JA61Alternative1B
 * Method:    N
 * Signature: ()I
 */
JNIEXPORT jint JNICALL Java_JA61Alternative1B_N
  (JNIEnv *, jclass);

#ifdef __cplusplus
}
#endif
#endif
```

JA61Alternative1B.c

```
/* This contains the source code for native method Java_JA61Alternative1B_N. This
   module is bound into service program JADOPT61/ALT1B. */

#include "JA61Alternative1B.h"
#include "JA61Example1.h"

/*
 * Class:     JA61Alternative1B
 * Method:    N
 * Signature: ()I
 */
JNIEXPORT jint JNICALL Java_JA61Alternative1B_N(JNIEnv* env, jclass klass) {
   return Java_JA61Example1_X(env, klass);  /* from JA61Example1.h */
}
```

Example 2: Java method adopts authority and calls other Java methods before calling a native method



In this example, an IBM Java method **J1** is contained in a Java program which adopts user profile **P**. **J1** and calls Java method **J2**, which calls **J3**, which then calls native method **X**. Native method **X** calls ILE method **Y**, which requires the adopted authority.

JA61Example2.java

```
public class JA61Example2 {
    public static void main(String args[]) {
    int returnVal = J1();
    if (returnVal > 0) {
        System.out.println("Adopted authority successfully.");
    }
    else {
        System.out.println("ERROR: Unable to adopt authority.");
    }
    }

    static int J1() {
    return JA61Example2Allow.J2();
    }

}
```

JA61Example2Allow.java

```
public class JA61Example2Allow {
    public static int J2() {
    return J3();
    }

    static int J3() {
    return X();
    }

    // Returns: 1 if able to successfully access *DTAARA JADOPT61/DATAAREA
    static native int X();

    static {
    System.loadLibrary("EX2ALLOW");
    }
}
```

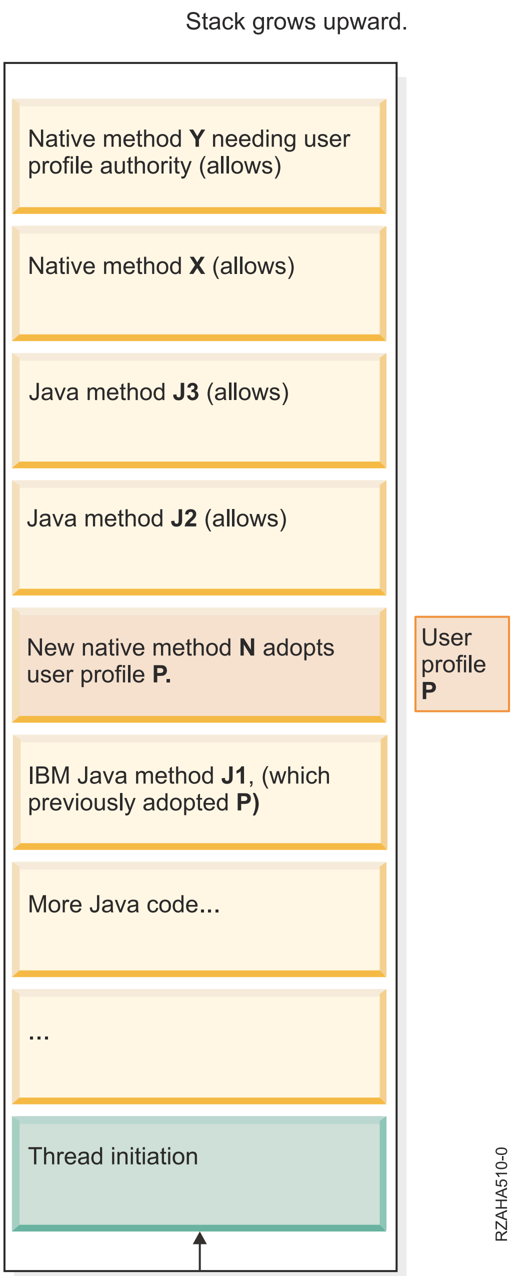
JA61Example2Allow.h

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class JA61Example2Allow */

#ifndef _Included_JA61Example2Allow
#define _Included_JA61Example2Allow
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     JA61Example2Allow
 * Method:    X
 * Signature: ()I
 */
JNIEXPORT jint JNICALL Java_JA61Example2Allow_X
  (JNIEnv *, jclass);

#ifdef __cplusplus
}
#endif
#endif
```

JA61Example2Allow.c

```
/* This contains the source code for native method Java_JA61Example2Allow_X. This
   module is bound into service program JADOPT61/EX2ALLOW. */

#include "JA61Example2Allow.h"
#include "JA61ModuleY.h"

/*
 * Class:     JA61Example2Allow
 * Method:    X
 * Signature: ()I
 */
JNIEXPORT jint JNICALL Java_JA61Example2Allow_X(JNIEnv* env, jclass klass) {
   return methodY();
}
```

Alternative 2: New native method N



In order to preserve adopted authority in this case, a new native method **N** may be created. This native method would be contained in a service program that adopts user profile **P**.

Native method **N** would then use JNI to call Java method **J2**, which is unchanged. Java method **J1** would need to be changed to call native method **N** instead of Java method **J2**.

JA61Alternative2.java

```
public class JA61Alternative2 {
    public static void main(String args[]) {
    int returnVal = J1();
    if (returnVal > 0) {
        System.out.println("Adopted authority successfully.");
    }
    else {
        System.out.println("ERROR: Unable to adopt authority.");
    }
    }
```

```
    static native int N();

    static int J1() {
    return N();
    }

    static {
    System.loadLibrary("ALT2");
    }

}
```

JA61Alternative2.h

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class JA61Alternative2 */

#ifndef _Included_JA61Alternative2
#define _Included_JA61Alternative2
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     JA61Alternative2
 * Method:    N
 * Signature: ()I
 */
JNIEXPORT jint JNICALL Java_JA61Alternative2_N
  (JNIEnv *, jclass);

#ifdef __cplusplus
}
#endif
#endif
```

JA61Alternative2.C

```
include "JA61Alternative2.h"

/*
 * Class:     JA61Alternative2
 * Method:    N
 * Signature: ()I
 */
JNIEXPORT jint JNICALL Java_JA61Alternative2_N(JNIEnv* env, jclass klass) {
#pragma convert(819)
    char* className = "JA61Example2Allow";
    char* methodName = "J2";
    char* methodSig = "()I";
#pragma convert(0)

    // Locate class JA61Example2Allow
    jclass cls = env->FindClass(className);
    // Get the method id for J2()I and call it.
    jmethodID methodID = env->GetStaticMethodID(cls, methodName, methodSig);
    int result = env->CallStaticIntMethod(cls, methodID);
    return result;
}
```

Compile commands for the examples

All of the following instructions are based on the source code being in a directory named `/home/javatests/adoptup/v6r1mig` on a IBM i server.

In Qshell:

```
> cd /home/javatests/adoptup/v6r1mig
> javac -g *.java
```

From CL:

```
> CRTLIB JADOPT61
> CRTUSRPRF USRPRF(JADOPT61UP) STATUS(*DISABLED)
> CRTUSRPRF USRPRF(JADOPT61) PASSWORD(j61adopt) INLPGM(QSYS/QCMD) SPCAUT(*NONE)
```

```
> CRTDTAARA DTAARA(JADOPT61/DATAAREA) TYPE(*CHAR) LEN(50) VALUE('Initial value')
> GRTOBJAUT OBJ(JADOPT61/DATAAREA) OBJTYPE(*DTAARA) USER(JADOPT61UP) AUT(*ALL)
> RVKOBJAUT OBJ(JADOPT61/DATAAREA) OBJTYPE(*DTAARA) USER(*PUBLIC) AUT(*ALL)
> RVKOBJAUT OBJ(JADOPT61/DATAAREA) OBJTYPE(*DTAARA) USER(YOUR_USER_ID) AUT(*ALL)
```

Create SRVPGMY, which is used by all the examples:

```
> CRTCMOD MODULE(JADOPT61/MODULEY) SRCSTMF('/home/javatests/adoptup/v6r1mig/JA61ModuleY.c')
DBGVIEW(*ALL)
> CRTSRVPGM SRVPGM(JADOPT61/SRVPGMY) MODULE(JADOPT61/MODULEY) EXPORT(*ALL)
```

Create "Example 1: Java method adopts authority immediately before calling a native method" on page 243:

```
> CRTCMOD MODULE(JADOPT61/EX1) SRCSTMF('/home/javatests/adoptup/v6r1mig/JA61Example1.c')
INCDIR('/home/javatests/adoptup/v6r1mig') DBGVIEW(*ALL)
> CRTSRVPGM SRVPGM(JADOPT61/EX1) EXPORT(*ALL) BNDSRVPGM(JADOPT61/SRVPGMY)
> QSH CMD('chown JADOPT61UP /home/javatests/adoptup/v6r1mig/JA61Example1.class')
> CRTJVAPGM CLSF('/home/javatests/adoptup/v6r1mig/JA61Example1.class') USRPRF(*OWNER)
```

Create "Alternative 1A: Repackaging native method X" on page 245:

```
> CRTCMOD MODULE(JADOPT61/ALT1A) SRCSTMF('/home/javatests/adoptup/v6r1mig/
JA61Alternative1A.c')
INCDIR('/home/javatests/adoptup/v6r1mig') DBGVIEW(*ALL)
> CRTSRVPGM SRVPGM(JADOPT61/ALT1A) EXPORT(*ALL) BNDSRVPGM(JADOPT61/SRVPGMY) USRPRF(*OWNER)
> CHGOBJOWN OBJ(JADOPT61/ALT1A) OBJTYPE(*SRVPGM) NEWOWN(JADOPT61UP)
> CRTJVAPGM CLSF('/home/javatests/adoptup/v6r1mig/JA61Alternative1A.class')
```

Create "Alternative 1B: New native method N" on page 247:

```
> CRTCMOD MODULE(JADOPT61/ALT1B) SRCSTMF('/home/javatests/adoptup/v6r1mig/
JA61Alternative1B.c')
INCDIR('/home/javatests/adoptup/v6r1mig') DBGVIEW(*ALL)
> CRTSRVPGM SRVPGM(JADOPT61/ALT1B) EXPORT(*ALL) BNDSRVPGM(JADOPT61/EX1) USRPRF(*OWNER)
> CHGOBJOWN OBJ(JADOPT61/ALT1B) OBJTYPE(*SRVPGM) NEWOWN(JADOPT61UP)
> CRTJVAPGM CLSF('/home/javatests/adoptup/v6r1mig/JA61Alternative1B.class')
```

Create "Example 2: Java method adopts authority and calls other Java methods before calling a native method" on page 249

```
> CRTCMOD MODULE(JADOPT61/EX2ALLOW) SRCSTMF('/home/javatests/adoptup/v6r1mig/
JA61Example2Allow.c')
INCDIR('/home/javatests/adoptup/v6r1mig') DBGVIEW(*ALL)
> CRTSRVPGM SRVPGM(JADOPT61/EX2ALLOW) EXPORT(*ALL) BNDSRVPGM(JADOPT61/SRVPGMY)
> QSH CMD('chown JADOPT61UP /home/javatests/adoptup/v6r1mig/JA61Example2.class')
> CRTJVAPGM CLSF('/home/javatests/adoptup/v6r1mig/JA61Example2.class') USRPRF(*OWNER)
> CRTJVAPGM CLSF('/home/javatests/adoptup/v6r1mig/JA61Example2Allow.class') USEADPAUT(*YES)
```

Create "Alternative 2: New native method N" on page 251:

```
> CRTCPPMOD MODULE(JADOPT61/ALT2) SRCSTMF('/home/javatests/adoptup/v6r1mig/
JA61Alternative2.C')
INCDIR('/home/javatests/adoptup/v6r1mig') DBGVIEW(*ALL)
> CRTSRVPGM SRVPGM(JADOPT61/ALT2) EXPORT(*ALL) USRPRF(*OWNER)
> CHGOBJOWN OBJ(JADOPT61/ALT2) OBJTYPE(*SRVPGM) NEWOWN(JADOPT61UP)
> CRTJVAPGM CLSF('/home/javatests/adoptup/v6r1mig/JA61Alternative2.class')
```

In order to run the examples, perform the following steps:

```
> sign on as JADOPT61
> ADDLIBLE JADOPT61
> ADDENVVAR ENVVAR(CLASSPATH) VALUE('/home/javatests/adoptup/v6r1mig')
> JAVA JA61Example1
> JAVA JA61Alternative1A
> JAVA JA61Alternative1B
```

```
> JAVA JA61Example2
> JAVA JA61Alternative2
```

Java security model

You can download Java applets from any system; thus, security mechanisms exist within the Java virtual machine to protect against malicious applets. The Java runtime system verifies the bytecodes as the Java virtual machine loads them. This ensures that they are valid bytecodes and that the code does not violate any of the restrictions that the Java virtual machine places on Java applets.

Just as with applets, the byte code loader and verifier check that the byte codes are valid and data types are used properly. They also check that registers and memory are accessed correctly, and that the stack does not overflow or underflow. These checks ensure that the Java virtual machine can safely run the class without compromising the integrity of the system.

Java applets are restricted in what operations they can perform, how they access memory, and how they use the Java virtual machine. The restrictions are in place to prevent a Java applet from gaining access to underlying operating system or data on the system. This is the "sandbox" security model, because the Java applet can only "play" in its own sandbox.

The "sandbox" security model is a combination of the class loader, class file verifier, and the java.lang.SecurityManager class.

Related concepts

Changes to adopted authority in IBM i 7.2
Support for adopted user profile authority through Java programs is not supported in IBM i 7.2. This topic describes how to determine if your applications are using adopted authority and how to modify your applications to accommodate this change.

Java Cryptography Extension
The Java Cryptography Extension (JCE) provides a framework and implementations for encryption, key generation, and key agreement, as well as Message Authentication Code (MAC) algorithms. Support for encryption includes symmetric, asymmetric, block, and stream ciphers. The software also supports secure streams and sealed objects. JCE supplements the Java 2 platform, which already includes interfaces and implementations of message digests and digital signatures.

Java Secure Socket Extension
Java Secure Socket Extension (JSSE) is like a framework that abstracts the underlying mechanisms of both Secure Sockets Layer (SSL) and Transport Layer Security (TLS). By abstracting the complexity and peculiarities of the underlying protocols, JSSE enables programmers to use secure encrypted communications while at the same time minimizing possible security vulnerabilities. Java Secure Socket Extension (JSSE) uses both the SSL protocol and the TLS protocol to provide secure encrypted communications between your clients and servers.

Java Authentication and Authorization Service
The Java Authentication and Authorization Service (JAAS) is a standard extension to the Java 2 Platform, Standard Edition (J2SE). J2SE provides access controls that are based on where the code originated and who signed the code (code source-based access controls). It lacks, however, the ability to enforce additional access controls based on who runs the code. JAAS provides a framework that adds this support to the Java 2 security model.

IBM Java Generic Security Service (JGSS)
The Java Generic Security Service (JGSS) provides a generic interface for authentication and secure messaging. Under this interface you can plug a variety of security mechanisms based on secret-key, public-key, or other security technologies.

Secure applications with SSL

Java SE Security by Oracle.

Java Cryptography Extension

The Java Cryptography Extension (JCE) provides a framework and implementations for encryption, key generation, and key agreement, as well as Message Authentication Code (MAC) algorithms. Support for encryption includes symmetric, asymmetric, block, and stream ciphers. The software also supports secure streams and sealed objects. JCE supplements the Java 2 platform, which already includes interfaces and implementations of message digests and digital signatures.

For a general overview of JCE, see Oracle's documentation of JCE. The documents at this Web site contain links to many other Web-based information sources.

IBM provides the following JCE providers on IBM i:

IBMJCE
The default JCE provider.

IBMJCEFIPS
A JCE provider implementation evaluated against FIPS 140. For more information about the IBMJCEFIPS JCE provider, see "Security information" on the IBM developerWorks® Web site.

IBMJCECCAI5OS
A JCE provider implementation that extends JCE to use cryptographic hardware via the IBM Common Cryptographic Architecture Interface. For more information about the IBMJCECCAI5OS JCE provider, see "Using hardware cryptography" on page 256.

Related concepts

Changes to adopted authority in IBM i 7.2
Support for adopted user profile authority through Java programs is not supported in IBM i 7.2. This topic describes how to determine if your applications are using adopted authority and how to modify your applications to accommodate this change.

Java security model
You can download Java applets from any system; thus, security mechanisms exist within the Java virtual machine to protect against malicious applets. The Java runtime system verifies the bytecodes as the Java virtual machine loads them. This ensures that they are valid bytecodes and that the code does not violate any of the restrictions that the Java virtual machine places on Java applets.

Java Secure Socket Extension
Java Secure Socket Extension (JSSE) is like a framework that abstracts the underlying mechanisms of both Secure Sockets Layer (SSL) and Transport Layer Security (TLS). By abstracting the complexity and peculiarities of the underlying protocols, JSSE enables programmers to use secure encrypted communications while at the same time minimizing possible security vulnerabilities. Java Secure Socket Extension (JSSE) uses both the SSL protocol and the TLS protocol to provide secure encrypted communications between your clients and servers.

Java Authentication and Authorization Service
The Java Authentication and Authorization Service (JAAS) is a standard extension to the Java 2 Platform, Standard Edition (J2SE). J2SE provides access controls that are based on where the code originated and who signed the code (code source-based access controls). It lacks, however, the ability to enforce additional access controls based on who runs the code. JAAS provides a framework that adds this support to the Java 2 security model.

IBM Java Generic Security Service (JGSS)

The Java Generic Security Service (JGSS) provides a generic interface for authentication and secure messaging. Under this interface you can plug a variety of security mechanisms based on secret-key, public-key, or other security technologies.

Using hardware cryptography

The IBMJCECCAI5OS implementation extends the Java Cryptography Extension (JCE) and Java Cryptography Architecture (JCA) to add the capability to use hardware cryptography via the IBM Common Cryptographic Architecture (CCA) interfaces.

The IBMJCECCAI5OS provider takes advantage of hardware cryptography within the existing JCE architecture and gives Java 2 programmers the significant security and performance advantages of hardware cryptography with minimal changes to existing Java applications. As the complexities of hardware cryptography are taken care of within the normal JCE, advanced security and performance using hardware cryptographic devices are made easily available. The IBMJCECCAI5OS provider plugs into the JCE framework in the same manner as the current providers. For hardware requests, the CCA APIs are invoked via native methods. The IBMJCECCAI5OS provider stores CCA RSA key labels in the JCECCAI5OSKS Java key store type.

Requirements for hardware cryptography

To use hardware cryptography, you must have the following installed on your system:

- A model 4764 Cryptographic Coprocessor
- IBM i (5770-SS1) Option 35 - CCA Cryptographic Service Provider
- Licensed Program Offering (LPO) 5733-CY3 - IBM Cryptographic Device Manager

Features of the IBM hardware cryptography provider

The IBMJCECCAI5OS provider supports the following algorithms:

Table 9. Algorithms supported by the IBMJCECCAI5OS provider

Signature algorithms	Cipher algorithms	Message authentication codes	Message digests
SHA1withRSA MD2WithRSA MD5WithRSA	RSA	HmacMD2 HmacMD5 HmacSHA1	MD2 MD5 SHA-1

The IBMJCECCAI5OS provider also includes a strong Pseudo Random Number Generator (PRNG), key generation via key factories, key/certificate generation and key/certificate management via a `keytool` application.

The hardware cryptographic access provider is available using the `hwkeytool` application.

Note: The IBMJCECCAI5OS provider cannot be added to the JVM by using the insertProviderAt() and the addProviderAt() methods.

Cryptography system properties

You can use the following system properties for working with cryptographic devices:

i5os.crypto.device
Specifies the cryptographic device to use. If this property is not set, the default device CRP01 is used.

i5os.crypto.keystore
Specifies the CCA keystore file to use. If this property is not set, the keystore file named in the cryptographic device description is used.

4764 Cryptographic Coprocessor

“Java hwkeytool” on page 359
The `hwkeytool` application enables you to use the cryptography capabilities of the model 4764 Cryptographic Coprocessor with the Java Cryptography Extension (JCE) and Java Cryptography Architecture (JCA).

“List of Java system properties” on page 15
Java system properties determine the environment in which the Java programs run. They are like system values or environment variables in IBM i.

Key pairs and hardware utilization

In the hardware cryptography environment, you can utilize the cryptographic coprocessor with two types of key pairs, RETAINED and Public Key Data Set (PKDS).

Both types of hardware key pairs supported by the IBMJCECAI5OS provider can be utilized by any application. The RETAINED and PKDS key pairs each return a label, which is treated by the IBMJCECAI5OS provider the same as a key. You can choose the mechanism by which the application will store the label.

The IBMJCECAI5OS provider stores RSA keys in a Common Cryptographic Architecture (CCA) keystore file encrypted under a master key, which is housed inside the 4764 Cryptographic Coprocessor. The JCECCAI5OSKS keystore stores labels of key records in the CCA keystore.

RETAINED hardware key pairs

The most secure implementation of cryptography supported by the IBMJCECAI5OS provider is to store the keys on the actual hardware cryptographic device and never allow the private key, the sensitive part of the key pair, to be retrieved or viewed. This is called a *retained* key pair as the private key is retained on the hardware device and is never allowed to be viewed or retrieved in the clear. What is returned to the application or stored in the key store at key pair generation is only a reference to the private key, called a *label*.

When the key is needed, the request is sent to the hardware card where the key is retained, accompanied by the key label. The cryptographic operation is performed on that card using the retained key, and the results are returned. Retained keys are the most secure of the key types. The disadvantage to using retained keys is that there is no backup and recovery possible. If the card fails, the keys are lost.

Public Key Data Set (PKDS) hardware key pairs

The other option for utilizing RSA keys is through PKDS key pair. When this type of key pair is generated, the private key is encrypted with the master key of the coprocessor so that the clear text version of this key can never be viewed or retrieved. The key pair is stored in a DB2 database file. What is returned to the application or stored in the key store at key pair generation is only a reference to the private key, called a *label*. By storing the key-pair in a file, you have a method to back up keys and recover from card failures.

Java Secure Socket Extension

Java Secure Socket Extension (JSSE) is like a framework that abstracts the underlying mechanisms of both Secure Sockets Layer (SSL) and Transport Layer Security (TLS). By abstracting the complexity and peculiarities of the underlying protocols, JSSE enables programmers to use secure encrypted communications while at the same time minimizing possible security vulnerabilities. Java Secure Socket Extension (JSSE) uses both the SSL protocol and the TLS protocol to provide secure encrypted communications between your clients and servers.

SSL/TLS provides a means of authenticating a server and a client to provide privacy and data integrity. All SSL/TLS communications begin with a "handshake" between the server and the client. During the handshake, SSL/TLS negotiates the cipher suite that the client and server use to communicate with each other. This cipher suite is a combination of the various security features available through SSL/TLS.

JSSE does the following to improve the security of your application:

- Protects communication data through encryption.
- Authenticates remote user IDs.

- Authenticates remote system names.

Note: JSSE uses a digital certificate to encrypt the socket communication of your Java application. Digital certificates are an Internet standard for identifying secure systems, users, and applications. You can control digital certificates using the IBM Digital Certificate Manager. For more information, see IBM Digital Certificate Manager.

To make your Java application more secure by using JSSE:

- Prepare the IBM i to support JSSE.
- Design your Java application to use JSSE by:
  - Changing your Java socket code to use socket factories if you do not use socket factories already.
  - Changing your Java code to use JSSE.
- Use a digital certificate to make your Java application more secure by:
  1. Selecting a type of digital certificate to use.
  2. Using the digital certificate when you run your application.

You can also register your Java application as a secure application by using the QsyRegisterAppForCertUse API.

Related concepts

Changes to adopted authority in IBM i 7.2
Support for adopted user profile authority through Java programs is not supported in IBM i 7.2. This topic describes how to determine if your applications are using adopted authority and how to modify your applications to accommodate this change.

Java security model
You can download Java applets from any system; thus, security mechanisms exist within the Java virtual machine to protect against malicious applets. The Java runtime system verifies the bytecodes as the Java virtual machine loads them. This ensures that they are valid bytecodes and that the code does not violate any of the restrictions that the Java virtual machine places on Java applets.

Java Cryptography Extension
The Java Cryptography Extension (JCE) provides a framework and implementations for encryption, key generation, and key agreement, as well as Message Authentication Code (MAC) algorithms. Support for encryption includes symmetric, asymmetric, block, and stream ciphers. The software also supports secure streams and sealed objects. JCE supplements the Java 2 platform, which already includes interfaces and implementations of message digests and digital signatures.

Java Authentication and Authorization Service
The Java Authentication and Authorization Service (JAAS) is a standard extension to the Java 2 Platform, Standard Edition (J2SE). J2SE provides access controls that are based on where the code originated and who signed the code (code source-based access controls). It lacks, however, the ability to enforce additional access controls based on who runs the code. JAAS provides a framework that adds this support to the Java 2 security model.

IBM Java Generic Security Service (JGSS)
The Java Generic Security Service (JGSS) provides a generic interface for authentication and secure messaging. Under this interface you can plug a variety of security mechanisms based on secret-key, public-key, or other security technologies.

Preparing your system for secure sockets layer support

To prepare your IBM i server to use secure sockets layer (SSL), you need to install the Digital Certificate Manager LP.

Install the Digital Certificate Manager LP, 5770-SS1 IBM i - Digital Certificate Manager.

You also need to make sure you can access or create a digital certificate on your system.

Related information

Digital Certificate Manager

Changing your Java code to use socket factories

To use secure sockets layer (SSL) with your existing code, you must first change your code to use socket factories.

To change your code to use socket factories, perform the following steps:

1. Add this line to your program to import the SocketFactory class:

   ```
   import javax.net.*;
   ```

2. Add a line that declares an instance of a SocketFactory object. For example:

   ```
   SocketFactory socketFactory
   ```

3. Initialize the SocketFactory instance by setting it equal to the method SocketFactory.getDefault(). For example:

   ```
   socketFactory = SocketFactory.getDefault();
   ```

   The whole declaration of the SocketFactory should look like this:

   ```
   SocketFactory socketFactory = SocketFactory.getDefault();
   ```

4. Initialize your existing sockets. Call the SocketFactory method createSocket(host,port) on your socket factory for each socket you declare.

   Your socket declarations should now look like this:

   ```
   Socket s = socketFactory.createSocket(host,port);
   ```

   Where:

   - *s* is the socket that is being created.
   - *socketFactory* is the SocketFactory that was created in step 2.
   - *host* is a string variable that represents the name of a host server.
   - *port* is an integer variable that represents the port number of the socket connection.

When you have completed all of these steps, your code uses socket factories. You do not need to make any other changes to your code. All of the methods that you call and all the syntax with your sockets still work.

Examples: Changing your Java code to use server socket factories

These examples show you how to change a simple socket class, named simpleSocketServer, so that it uses socket factories to create all of the sockets. The first example shows you the simpleSocketServer class without socket factories. The second example shows you the simpleSocketServer class with socket factories. In the second example, simpleSocketServer is renamed to factorySocketServer.

Example 1: Socket server program without socket factories

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
/* File simpleSocketServer.java*/

import java.net.*;
import java.io.*;
```

```
public class simpleSocketServer {
  public static void main (String args[]) throws IOException {

    int serverPort = 3000;

    if (args.length < 1) {
      System.out.println("java simpleSocketServer serverPort");
      System.out.println("Defaulting to port 3000 since serverPort not specified.");
    }
    else
        serverPort = new Integer(args[0]).intValue();

    System.out.println("Establishing server socket at port " + serverPort);

    ServerSocket    serverSocket =
      new ServerSocket(serverPort);

    // a real server would handle more than just one client like this...

    Socket s = serverSocket.accept();
    BufferedInputStream is = new BufferedInputStream(s.getInputStream());
    BufferedOutputStream os = new BufferedOutputStream(s.getOutputStream());

    // This server just echoes back what you send it...

    byte buffer[] = new byte[4096];

    int bytesRead;

    // read until "eof" returned
    while ((bytesRead = is.read(buffer)) > 0) {
      os.write(buffer, 0, bytesRead); // write it back
      os.flush();    // flush the output buffer
    }


    s.close();
    serverSocket.close();
  }       // end main()

}       // end class definition
```

Example 2: Simple socket server program with socket factories

```
/* File factorySocketServer.java */

// need to import javax.net to pick up the ServerSocketFactory class
import javax.net.*;
import java.net.*;
import java.io.*;

public class factorySocketServer {
  public static void main (String args[]) throws IOException {

    int serverPort = 3000;

    if (args.length < 1) {
      System.out.println("java simpleSocketServer serverPort");
      System.out.println("Defaulting to port 3000 since serverPort not specified.");
    }
    else
        serverPort = new Integer(args[0]).intValue();

    System.out.println("Establishing server socket at port " + serverPort);

    // Change this to create an SSLServerSocketFactory instead of a ServerSocketFactory.
    ServerSocketFactory serverSocketFactory =
      SSLServerSocketFactory.getDefault();
    // Now have the factory create the server socket.  This is the last
    // change from the original program.
    ServerSocket    serverSocket =
      serverSocketFactory.createServerSocket(serverPort);

    // a real server would handle more than just one client like this...

    Socket s = serverSocket.accept();
    BufferedInputStream is = new BufferedInputStream(s.getInputStream());
    BufferedOutputStream os = new BufferedOutputStream(s.getOutputStream());

    // This server just echoes back what you send it...
```

```
    byte buffer[] = new byte[4096];

    int bytesRead;

    while ((bytesRead = is.read(buffer)) > 0) {
      os.write(buffer, 0, bytesRead);
      os.flush();
    }


    s.close();
    serverSocket.close();
  }

}
```

Related reference

Examples: Changing your Java code to use client socket factories
These examples show you how to change a simple socket class, named simpleSocketClient, so that it uses socket factories to create all of the sockets. The first example shows you the simpleSocketClient class without socket factories. The second example shows you the simpleSocketClient class with socket factories. In the second example, simpleSocketClient is renamed to factorySocketClient.

Examples: Changing your Java code to use client socket factories

These examples show you how to change a simple socket class, named simpleSocketClient, so that it uses socket factories to create all of the sockets. The first example shows you the simpleSocketClient class without socket factories. The second example shows you the simpleSocketClient class with socket factories. In the second example, simpleSocketClient is renamed to factorySocketClient.

Example 1: Socket client program without socket factories

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
/* Simple Socket Client Program */

import java.net.*;
import java.io.*;

public class simpleSocketClient {
  public static void main (String args[]) throws IOException {

    int serverPort = 3000;

    if (args.length < 1) {
      System.out.println("java simpleSocketClient serverHost serverPort");
      System.out.println("serverPort defaults to 3000 if not specified.");
      return;
    }
    if (args.length == 2)
      serverPort = new Integer(args[1]).intValue();

    System.out.println("Connecting to host " + args[0] + " at port " +
                       serverPort);

    // Create the socket and connect to the server.
    Socket  s = new Socket(args[0], serverPort);
    .
    .
    .

    // The rest of the program continues on from here.
```

Example 2: Simple socket client program with socket factories

```
/* Simple Socket Factory Client Program */

// Notice that javax.net.* is imported to pick up the SocketFactory class.
import javax.net.*;
import java.net.*;
import java.io.*;

public class factorySocketClient {
```

```
  public static void main (String args[]) throws IOException {

    int serverPort = 3000;

    if (args.length < 1) {
      System.out.println("java factorySocketClient serverHost serverPort");
      System.out.println("serverPort defaults to 3000 if not specified.");
      return;
    }
    if (args.length == 2)
      serverPort = new Integer(args[1]).intValue();

    System.out.println("Connecting to host " + args[0] + " at port " +
                       serverPort);

    // Change the original simpleSocketClient program to create a
    // SocketFactory and then use the socket factory to create sockets.

    SocketFactory socketFactory = SocketFactory.getDefault();

    // Now the factory creates the socket.  This is the last change
    // to the original simpleSocketClient program.

    Socket  s = socketFactory.createSocket(args[0], serverPort);
    .
    .
    .

    // The rest of the program continues on from here.
```

Related reference

Examples: Changing your Java code to use server socket factories
These examples show you how to change a simple socket class, named simpleSocketServer, so that it uses socket factories to create all of the sockets. The first example shows you the simpleSocketServer class without socket factories. The second example shows you the simpleSocketServer class with socket factories. In the second example, simpleSocketServer is renamed to factorySocketServer.

Changing your Java code to use secure sockets layer

If your code already uses socket factories to create its sockets, then you can add secure socket layer (SSL) support to your program.

If your code does not already use socket factories, see Change your Java code to use socket factories.

To change your code to use SSL, perform the following steps:

1. Import javax.net.ssl.* to add SSL support:

```
import javax.net.ssl.*;
```

2. Declare a SocketFactory by using SSLSocketFactory to initialize it:

```
SocketFactory newSF = SSLSocketFactory.getDefault();
```

3. Use your new SocketFactory to initialize your sockets the same way that you used your old SocketFactory:

```
Socket s = newSF.createSocket(args[0], serverPort);
```

Your code now uses SSL support. You do not need to make any other changes to your code.

Examples: Changing your Java server to use secure sockets layer

These examples show you how to change one class, named factorySocketServer, to use secure sockets layer (SSL).

The first example shows you the factorySocketServer class not using SSL. The second example shows you the same class, renamed factorySSLSocketServer, using SSL.

Example 1: Simple factorySocketServer class without SSL support

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
/* File factorySocketServer.java  */
// need to import javax.net to pick up the ServerSocketFactory class
import javax.net.*;
import java.net.*;
import java.io.*;

public class factorySocketServer {
  public static void main (String args[]) throws IOException {

    int serverPort = 3000;

    if (args.length < 1) {
      System.out.println("java simpleSocketServer serverPort");
      System.out.println("Defaulting to port 3000 since serverPort not specified.");
    }
    else
        serverPort = new Integer(args[0]).intValue();

    System.out.println("Establishing server socket at port " + serverPort);

    // Change this to create an SSLServerSocketFactory instead of a ServerSocketFactory.
    ServerSocketFactory serverSocketFactory =
      SSLServerSocketFactory.getDefault();
    // Now have the factory create the server socket.  This is the last
    // change from the original program.
    ServerSocket    serverSocket =
      serverSocketFactory.createServerSocket(serverPort);

    // a real server would handle more than just one client like this...

    Socket s = serverSocket.accept();
    BufferedInputStream is = new BufferedInputStream(s.getInputStream());
    BufferedOutputStream os = new BufferedOutputStream(s.getOutputStream());

    // This server just echoes back what you send it.

    byte buffer[] = new byte[4096];

    int bytesRead;

    while ((bytesRead = is.read(buffer)) > 0) {
      os.write(buffer, 0, bytesRead);
      os.flush();
    }

    s.close();
    serverSocket.close();
  }
}
```

Example 2: Simple factorySocketServer class with SSL support

```
/* File factorySocketServer.java */

// need to import javax.net to pick up the ServerSocketFactory class
import javax.net.*;
import java.net.*;
import java.io.*;

public class factorySocketServer {
  public static void main (String args[]) throws IOException {

    int serverPort = 3000;

    if (args.length < 1) {
      System.out.println("java simpleSocketServer serverPort");
      System.out.println("Defaulting to port 3000 since serverPort not specified.");
    }
    else
        serverPort = new Integer(args[0]).intValue();

    System.out.println("Establishing server socket at port " + serverPort);

    // Change the original simpleSocketServer to use a
```

```
    // ServerSocketFactory to create server sockets.
    ServerSocketFactory serverSocketFactory =
      ServerSocketFactory.getDefault();
    // Now have the factory create the server socket.  This is the last
    // change from the original program.
    ServerSocket    serverSocket =
      serverSocketFactory.createServerSocket(serverPort);

    // a real server would handle more than just one client like this...

    Socket s = serverSocket.accept();
    BufferedInputStream is = new BufferedInputStream(s.getInputStream());
    BufferedOutputStream os = new BufferedOutputStream(s.getOutputStream());

    // This server just echoes back what you send it.

    byte buffer[] = new byte[4096];

    int bytesRead;

    while ((bytesRead = is.read(buffer)) > 0) {
      os.write(buffer, 0, bytesRead);
      os.flush();
    }

    s.close();
    serverSocket.close();
  }
}
```

Related reference

Examples: Changing your Java client to use secure sockets layer
These examples show you how to change one class, named factorySocketClient, to use secure sockets layer (SSL). The first example shows you the factorySocketClient class not using SSL. The second example shows you the same class, renamed factorySSLSocketClient, using SSL.

Examples: Changing your Java client to use secure sockets layer

These examples show you how to change one class, named factorySocketClient, to use secure sockets layer (SSL). The first example shows you the factorySocketClient class not using SSL. The second example shows you the same class, renamed factorySSLSocketClient, using SSL.

Example 1: Simple factorySocketClient class without SSL support

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
/* Simple Socket Factory Client Program */

import javax.net.*;
import java.net.*;
import java.io.*;

public class factorySocketClient {
  public static void main (String args[]) throws IOException {

    int serverPort = 3000;

    if (args.length < 1) {
      System.out.println("java factorySocketClient serverHost serverPort");
      System.out.println("serverPort defaults to 3000 if not specified.");
      return;
    }
    if (args.length == 2)
      serverPort = new Integer(args[1]).intValue();

    System.out.println("Connecting to host " + args[0] + " at port " +
                       serverPort);

    SocketFactory socketFactory = SocketFactory.getDefault();

    Socket  s = socketFactory.createSocket(args[0], serverPort);
    .
    .
    .
```

```
    // The rest of the program continues on from here.
```

Example 2: Simple factorySocketClient class with SSL support

```
// Notice that we import javax.net.ssl.* to pick up SSL support
import javax.net.ssl.*;
import javax.net.*;
import java.net.*;
import java.io.*;

public class factorySSLSocketClient {
  public static void main (String args[]) throws IOException {

    int serverPort = 3000;

    if (args.length < 1) {
      System.out.println("java factorySSLSocketClient serverHost serverPort");
      System.out.println("serverPort defaults to 3000 if not specified.");
      return;
    }
    if (args.length == 2)
      serverPort = new Integer(args[1]).intValue();

    System.out.println("Connecting to host " + args[0] + " at port " +
                       serverPort);

    // Change this to create an SSLSocketFactory instead of a SocketFactory.
    SocketFactory socketFactory = SSLSocketFactory.getDefault();

    // We do not need to change anything else.
    // That's the beauty of using factories!
    Socket  s = socketFactory.createSocket(args[0], serverPort);
    .
    .
    .

    // The rest of the program continues on from here.
```

Related reference

Examples: Changing your Java server to use secure sockets layer
These examples show you how to change one class, named factorySocketServer, to use secure sockets layer (SSL).

Selecting a digital certificate

You should consider several factors when deciding which digital certificate to use. You can use your system's default certificate or you can specify another certificate to use.

You want to use your system's default certificate if:

- You do not have any specific security requirements for your Java application.
- You do not know what kind of security you need for your Java application.
- Your system's default certificate meets the security requirements for your Java application.

Note: If you decide that you want to use your system's default certificate, check with your system administrator to make sure that a default system certificate has been created.

If you do not want to use your system's default certificate, you need to choose a different certificate to use. You can choose from two types of certificates:

- **User certificate** that identifies the user of the application.
- **System certificate** that identifies the system on which the application is running.

You should use a user certificate if:

- your application runs as a client application.
- you want the certificate to identify the user who is working with the application.

You should use a system certificate if:

- your application runs as a server application.
- you want the certificate to identify on which system the application is running.

Once you know what kind of certificate you need, you can choose from any of the digital certificates in any of the certificate containers that you are able to access.

Related information

Digital Certificate Manager

Using the digital certificate when running your Java application

To use secure sockets layer (SSL), you must run your Java application using a digital certificate.

To specify which digital certificate to use when using the native IBM i JSSE provider, use the following properties:

- os400.certificateContainer
- os400.certificateLabel

For example, if you want run the Java application MyClass.class using the digital certificate MYCERTIFICATE, and MYCERTIFICATE was in the digital certificate container YOURDCC, then the `java` command would look like this:

```
java -Dos400.certificateContainer=YOURDCC
     -Dos400.certificateLabel=MYCERTIFICATE MyClass
```

If you have not already decided which digital certificate to use, see “Selecting a digital certificate” on page 265. You may also decide to use your system's default certificate, which is stored in the system's default certificate container.

To use your system's default digital certificate, you do not need to specify a certificate or a certificate container anywhere. Your Java application uses your system's default digital certificate automatically.

Digital certificates and the -os400.certificateLabel property

Digital certificates are an Internet standard for identifying secure systems, users, and applications. Digital certificates are stored in digital certificate containers. If you want to use a digital certificate container's default certificate, you do not need to specify a certificate label. If you want to use a specific digital certificate, you must specify that certificate's label in the `java` command using this property:

```
os400.certificateLabel=
```

For example, if the name of the certificate you want to use is MYCERTIFICATE, then the `java` command you enter would look like this:

```
java -Dos400.certificateLabel=MYCERTIFICATE MyClass
```

In this example, the Java application MyClass would use the certificate MYCERTIFICATE. MYCERTIFICATE would need to be in the system's default certificate container to be used by MyClass.

Digital certificate containers and the -os400.certificateContainer property

Digital certificate containers store digital certificates. If you want to use the IBM i system default certificate container, you do not need to specify a certificate container. To use a specific digital certificate container, you need to specify that digital certificate container in the `java` command using this property:

```
os400.certificateContainer=
```

For example, if the name of the certificate container that contains the digital certificate you want to use is named MYDCC, then the `java` command you enter would look like this:

```
java -Dos400.certificateContainer=MYDCC MyClass
```

In this example, the Java application, named MyClass.class, would run on the system by using the default digital certificate that is in the digital certificate container named MYDCC. Any sockets that you create in the application use the default certificate that is in MYDCC to identify themselves and make all of their communications secure.

If you wanted to use the digital certificate MYCERTIFICATE in the digital certificate container, then the `java` command that you would enter would look like this:

```
java -Dos400.certificateContainer=MYDCC
     -Dos400.certificateLabel=MYCERTIFICATE MyClass
```

Related information

Digital Certificate Manager

Using Java Secure Socket Extension

This information applies only to using JSSE on system that run Java 2 Platform, Standard Edition (J2SE) and subsequent releases. JSSE is like a framework that abstracts the underlying mechanisms of both SSL and TLS. By abstracting the complexity and peculiarities of the underlying protocols, JSSE enables programmers to use secure, encrypted communications while at the same time minimizing possible security vulnerabilities. Java Secure Socket Extension (JSSE) uses both the Secure Sockets Layer (SSL) protocol and the Transport Layer Security (TLS) protocol to provide secure, encrypted communications between your clients and servers.

The IBM implementation of JSSE is called IBM JSSE. IBM JSSE includes a native IBM i JSSE provider and an IBM pure Java JSSE provider.

Configuring your server to support JSSE

Configure your IBM i server to use different JSSE implementations. This topic includes software requirements, how to change JSSE providers, and the necessary security properties and system properties. The default configuration uses the IBM pure Java JSSE provider known as IBMJSSE2.

When you use the Java 2 Platform, Standard Edition (J2SE) on your IBM i server, JSSE is already configured. The default configuration uses the IBM pure Java JSSE provider.

Changing JSSE providers

You can configure JSSE to use the native IBM i JSSE provider instead of the IBM pure Java JSSE provider. By changing some specific JSSE security properties and Java system properties, you can switch between the providers.

Security managers

If you are running your JSSE application with a Java security manager enabled, you may need to set the available network permissions. For more information, see SSL Permission in Permissions in the Java 2 SDK.

JSSE providers

IBM JSSE includes a native IBM i JSSE provider and an IBM pure Java JSSE provider. The provider that you choose to use depends on the needs of your application.

All providers adhere to the JSSE interface specification. They can communicate with each other and with any other SSL or TLS implementation, even non-Java implementations.

IBM pure Java JSSE provider

The IBM pure Java JSSE provider offers the following features:

- Works with any type of KeyStore object to control and configure digital certificates (for example, JKS, PKCS12, and so on).
- Allows you to use any combination of JSSE components from multiple implementations together.

IBMJSSEProvider2 is the provider name for the pure Java implementation. You need to pass this provider name, using the proper case, to the java.security.Security.getProvider() method or the various getInstance() methods for several of the JSSE classes.

Native IBM i JSSE provider

The native IBM i JSSE provider offers the following features:

- Uses the native IBM i SSL support.
- Allows the use of the Digital Certificate Manager to configure and control digital certificates. This is provided via a unique IBM i type of KeyStore (IbmISeriesKeyStore).
- Allows you to use any combination of JSSE components from multiple implementations together.

IBMi5OSJSSEProvider is the name for the native IBM i implementation. You need to pass this provider name, using the proper case, to the java.security.Security.getProvider() method or the various getInstance() methods for several of the JSSE classes.

Changing the default JSSE provider

You can change the default JSSE provider by making the appropriate changes to your security properties.

After changing the JSSE provider, ensure that your system properties specify the proper configuration for digital certificate information (keystore) required by the new provider.

For more information, see JSSE security properties.

JSSE Reference Guide by Oracle.
"JSSE security properties" on page 268
A Java virtual machine (JVM) uses many important security properties that you set by editing the Java master security properties file.

JSSE security properties
A Java virtual machine (JVM) uses many important security properties that you set by editing the Java master security properties file.

This file, named java.security, usually resides in the `$JAVA_HOME/jre/lib/security` directory on your server. ("JAVA_HOME" is an environment variable that is set to the home directory of the Java Development Kit that you want to use)

The following list describes several relevant security properties for using JSSE. Use the descriptions as a guide for editing the java.security file.

security.provider.<integer>

The JSSE provider that you want to use. Also statically registers cryptographic provider classes. Specify the different JSSE providers exactly like the following example:

```
security.provider.5=com.ibm.i5os.jsse.JSSEProvider
security.provider.6=com.ibm.jsse2.IBMJSSEProvider2
```

ssl.KeyManagerFactory.algorithm

Specifies the default KeyManagerFactory algorithm. For the native IBM i JSSE provider, use the following:

```
ssl.KeyManagerFactory.algorithm=IbmISeriesX509
```

For the IBM pure Java JSSE provider, use the following:

```
ssl.KeyManagerFactory.algorithm=IbmX509
```

For the Oracle America, Inc. pure Java JSSE provider, use the following:

```
ssl.KeyManagerFactory.algorithm=SunX509
```

For more information, see the Javadoc for javax.net.ssl.KeyManagerFactory.

ssl.TrustManagerFactory.algorithm

Specifies the default TrustManagerFactory algorithm. For the native IBM i JSSE provider, use the following:

```
ssl.TrustManagerFactory.algorithm=IbmISeriesX509
```

For the IBM pure Java JSSE provider, use the following:

```
ssl.TrustManagerFactory.algorithm=IbmX509
```

For more information, see the Javadoc for javax.net.ssl.TrustManagerFactory.

ssl.SocketFactory.provider

Specifies the default SSL socket factory. For the native IBM i JSSE provider, use the following:

```
ssl.SocketFactory.provider=com.ibm.i5os.jsse.JSSESocketFactory
```

For the IBM pure Java JSSE provider, use the following:

```
ssl.SocketFactory.provider=com.ibm.jsse2.SSLSocketFactoryImpl
```

For more information, see the Javadoc for javax.net.ssl.SSLSocketFactory.

ssl.ServerSocketFactory.provider

Specifies the default SSL server socket factory. For the native IBM i JSSE provider, use the following:

```
ssl.ServerSocketFactory.provider=com.ibm.i5os.jsse.JSSEServerSocketFactory
```

For the pure Java JSSE provider, use the following:

```
ssl.ServerSocketFactory.provider=com.ibm.jsse2.SSLServerSocketFactoryImpl
```

For more information, see the Javadoc for javax.net.ssl.SSLServerSocketFactory.

Related information

javax.net.ssl.KeyManagerFactory Javadoc
javax.net.ssl.TrustManagerFactory Javadoc
javax.net.ssl.SSLSocketFactory Javadoc
javax.net.ssl.SSLServerSocketFactory Javadoc

JSSE for Java system properties
To use JSSE in your applications, you need to specify several system properties that the default SSLContext objects needs in order to provide confirmation of the configuration. Some of the properties apply to all providers, while others apply to only the native IBM i provider.

When using the native IBM i JSSE provider, if you do not specify the properties, the os400.certificateContainer defaults to *SYSTEM, which means that JSSE uses the default entry in the system certificate store.

Properties that apply to the native IBM i JSSE provider and the IBM pure Java JSSE provider

The following properties apply to both JSSE providers. Each description includes the default property, if applicable.

javax.net.ssl.trustStore

The name of the file that contains the KeyStore object that you want the default TrustManager to use. The default value is jssecacerts, or cacerts (if jssecacerets does not exist).

javax.net.ssl.trustStoreType

The type of KeyStore object that you want the default TrustManager to use. The default value is the value returned by the KeyStore.getDefaultType method.

javax.net.ssl.trustStorePassword

The password for the KeyStore object that you want the default TrustManager to use.

javax.net.ssl.keyStore

The name of the file that contains the KeyStore object that you want the default KeyManager to use. The default value is jssecacerts, or cacerts (if jssecacerets does not exist).

javax.net.ssl.keyStoreType

The type of KeyStore object that you want the default KeyManager to use. The default value is the value returned by the KeyStore.getDefaultType method.

javax.net.ssl.keyStorePassword

The password for the KeyStore object that you want the default KeyManager to use.

Properties that work for the native IBM i JSSE provider only

The following properties apply to the native IBM i JSSE provider only.

os400.secureApplication

The application identifier. JSSE uses this property only when you do not specify any of the following properties:

- javax.net.ssl.keyStore
- javax.net.ssl.keyStorePassword
- javax.net.ssl.keyStoreType
- javax.net.ssl.trustStore
- javax.net.ssl.trustStorePassword
- javax.ssl.net.trustStoreType

os400.certificateContainer

The name of the keyring that you want to use. JSSE uses this property only when you do not specify any of the following properties:

- javax.net.ssl.keyStore
- javax.net.ssl.keyStorePassword
- javax.net.ssl.keyStoreType
- javax.net.ssl.trustStore
- javax.net.ssl.trustStorePassword
- javax.ssl.net.trustStoreType
- os400.secureApplication

os400.certificateLabel

The keyring label that you want to use. JSSE uses this property only when you do not specify any of the following properties:

- javax.net.ssl.keyStore
- javax.net.ssl.keyStorePassword
- javax.net.ssl.trustStore
- javax.net.ssl.trustStorePassword
- javax.ssl.net.trustStoreType
- os400.secureApplication

Related concepts

"List of Java system properties" on page 15
Java system properties determine the environment in which the Java programs run. They are like system values or environment variables in IBM i.

Related information

Oracle America, Inc. System Properties

Using the native IBM i JSSE provider

The native IBM i JSSE provider offers the full suite of JSSE classes and interfaces including implementations of the JSSE KeyStore class and the SSLConfiguration class.

Protocol values for the SSLContext.getInstance method

The following table identifies and describes the protocol values for the SSLContext.getInstance method of the native IBM i JSSE provider.

The supported SSL protocols may be limited by the system values set on your system. For more details, see the Security system values: Secure Sockets Layer protocols subtopic in the Systems management information.

Protocol value	Supported SSL protocols
SSL	SSL version 3 and TLS version 1.0. Will accept SSLv3 or TLSv1 hello encapsulated in an SSLv2 format hello.
SSLv3	SSL version 3 protocol. Will accept SSLv3 hello encapsulated in a SSLv2 format hello.
TLS	TLS version 1.0 protocol, defined in Request for Comments (RFC) 2246. Will accept TLSv1 hello encapsulated in an SSLv2 format hello.
TLSv1	TLS version 1.0 protocol, defined in Request for Comments (RFC) 2246. Will accept TLSv1 hello encapsulated in an SSLv2 format hello.
SSL_TLS	SSL version 3 and TLS version 1.0. Will accept SSLv3 or TLSv1 hello encapsulated in an SSLv2 format hello.
TLSv1.1	TLS version 1.1 protocol, defined in Request for Comments (RFC) 4346. Will accept TLSv1 hello encapsulated in an SSLv2 format hello
TLSv1.2	TLS version 1.2 protocol, defined in Request for Comments (RFC) 5246.

Protocol value	Supported SSL protocols
SSL_TLSv2	TLS version 1.2, 1.1, 1.0 and SSL version 3. Will accept SSLv3 or TLSv1 hello encapsulated in an SSLv2 format hello but not TLSv1.2.

Native IBM i KeyStore implementations

The native IBM i provider offers two implementations of the KeyStore class, IbmISeriesKeyStore or IBMi5OSKeyStore. Both KeyStore implementations provide a wrapper around the Digital Certificate Manager (DCM) support.

IbmISeriesKeyStore
The contents of the keystore are based on a particular application identifier or keyring file, password, and label. JSSE loads the keystore entries from the Digital Certificate Manager. To load the entries, JSSE uses the appropriate application identifier or keyring information when your application makes the first attempt to access keystore entries or keystore information. You cannot modify the keystore, and you must make all configuration changes by using the Digital Certificate Manager.

IBMi5OSKeyStore
The contents of this keystore are based on an i5OS certificate store file and the password to access that file. This KeyStore class allows the modification of the certificate store. You can make changes without using the Digital Certificate Manager.

The IBMi5OSKeyStore implementation conforms to the Oracle America, Inc. specification for the Java KeyStore API. You can find more information in the Keystore javadoc information by Oracle.

For more information on how to manage keystores through DCM, see the Digital Certificate Manager topic.

Related information

Security system values: Secure Sockets Layer protocols

i5OSLoadStoreParameter class Javadoc information

com.ibm.i5os.keystore

Class i5OSLoadStoreParameter

```
java.lang.Object
  |
  +--com.ibm.i5os.keystore.i5OSLoadStoreParameter
```

All Implemented Interfaces:

java.security.Keystore.LoadStoreParameter

```
public class i5OSLoadStoreParameter
extends java.lang.Object
implements java.security.Keystore.LoadStoreParameter
```

This class creates a KeyStore.ProtectionParameter object that can be used to load/store i5OS certificate stores. Once created, this class provides information about the certificate store to be accessed and the password used to protect that certificate store.

An example use of this class would be:

```
//initialize the keystore
    KeyStore ks = KeyStore.getInstance(“IBMi5OSKeyStore”);

 //Load an existing keystore
    File kdbFile = new File(“/tmp/certificateStore.kdb”);
    i5OSLoadStoreParameter lsp =
   new i5OSLoadStoreParameter (kdbFile, “password”.toCharArray());
    ks.load(lsp);

 //Get and Add entries to the certificate store
```

```
   …

//Save the certificate store
   Ks.store(lsp);
```

Since:
SDK 1.5

Constructor Summary

i5OSLoadStoreParameter(java.io.File ksFile, char[] password)

Creates a ProtectionParameter instance from the KeyStore file and the password to be used for loading/storing an i5OS certificate store.

i5OSLoadStoreParameter(java.io.File ksFile, java.security.KeyStore.PasswordProtection pwdProtParam)

Creates a ProtectionParameter instance from the KeyStore file and PasswordProtection to be used for loading/storing an i5OS certificate store.

*Table 10. **Method Summary***

`java.security.KeyStore.ProtectionParameter`	"getProtectionParameter" on page 274() Returns the KeyStore.KeyStoreParameter asscociated with this LoadStoreParameter

----

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

----

Constructor detail

i5OSLoadStoreParameter

```
public i5OSLoadStoreParameter(java.io.File ksFile,
                              char[] password)
                       throws java.lang.IllegalArgumentException
```

Creates a ProtectionParameter instance from the KeyStore file and the password to be used for loading/storing an i5OS certificate store.

Parameters:

ksFile - The File object of the KeyStore.
If keystore.load() was used with an i5OSLoadStoreParameter(ksFile = null, password), then a new keystore is created.
If keystore.store() was used with an i5OSLoadStoreParameter(ksFile = null, password), then an IllegalArgumentException is thrown.
password - The password to access the i5OS certificate store. It cannot be null nor empty.

Throws:

java.lang.IllegalArgumentException - If password is null or empty

----

i5OSLoadStoreParameter

```
public i5OSLoadStoreParameter(java.io.File ksFile,
                              java.security.KeyStore.PasswordProtection pwdProtParam)
                                           throws java.lang.IllegalArgumentException
```

Creates a ProtectionParameter instance from the KeyStore file and PasswordProtection to be used for loading/storing an i5OS certificate store.

If `keystore.load()` was used with an `i5OSLoadStoreParameter(ksFile = null, password)`, then a new keystore is created.

If `keystore.store()` was used with an `i5OSLoadStoreParameter(ksFile = null, password)`, then an IllegalArgumentException is thrown.

Parameters:

`ksFile` - The File object of the KeyStore.

`pwdProtParam` - PasswordProtection instance which will be used to acquire the password. It cannot be null.

Throws:

`java.lang.IllegalArgumentException` - If KeyStore.PasswordProtection is null, or if the password contained in pwdProtParam is either null or empty.

Method detail

----

getProtectionParameter

```
public java.security.KeyStore.ProtectionParameter getProtectionParameter()
```

Returns the KeyStore.KeyStoreParameter asscociated with this LoadStoreParameter.

Specified by:

```
getProtectionParameter in interface java.security.KeyStore.LoadStoreParameter
```

Returns:

An instance that implements KeyStore.KeyStoreParameter interface

See Also:

```
java.security.KeyStore.ProtectionParameter#getProtectionParameter()
```

i5OSSystemCertificateStoreFile class Javadoc information

com.ibm.i5os.keystore

Class i5OSSystemCertificateStoreFile

```
java.lang.Object
   java.io.File
       com.ibm.i5os.keystore.i5OSSystemCertificateStoreFile
```

All Implemented Interfaces:

java.io.Serializable, java.lang.Comparable<java.io.File>

```
public class i5OSSystemCertificateStoreFile
extends java.io.File
```

This class provides a new File implementation which points to the *SYSTEM certificate store file. It provides a mechanism for the user to load the *SYSTEM certificate store without forcing them to know the actual path to the store.

To load the *SYSTEM certificate store into a keystore, first create an i5OSSystemCertificateStoreFile.

From here, the keystore can be loaded in 2 ways:

- Using an i5OSLoadStoreParameter:

```
//create an i5OSSystemCertificateStoreFile
        File starSystemFile = new i5OSSystemCertificateStoreFile();

     //use that file to create an i5OSLoadStoreParameter
        i5OSLoadStoreParameter lsp = new i5OSLoadStoreParameter(starSystemFile, pwd);

     //load the certificate store into a keystore
        KeyStore ks = KeyStore.getInstance("IBMi5OSKeyStore");
        ks.load(lsp);
```

- Using a FileInputStream:

```
//create an i5OSSystemCertificateStoreFile
        File starSystemFile = new i5OSSystemCertificateStoreFile();

     //create an input stream to the starSystemFile
     FileInputStream fis = new FileInputStream(starSystemFile);

     //load the certificate store into a keystore
        KeyStore ks = KeyStore.getInstance("IBMi5OSKeyStore");
        ks.load(fis, pwd);
```

Since:
SDK 1.5

See also:
Serialized Form

Field Summary

Fields inherited from class java.io.File
pathSeparator, pathSeparatorChar, separator, separatorChar

Constructor Summary

i5OSSystemCertificateStoreFile()

Creates a File() pointing to the *System certificate store file.

Method Summary

Methods inherited from class java.io.File
canRead, canWrite, compareTo, createNewFile, createTempFile, createTempFile, delete, deleteOnExit, equals, exists, getAbsoluteFile, getAbsolutePath, getCanonicalFile, getCanonicalPath, getName, getParent, getParentFile, getPath, hashCode, isAbsolute, isDirectory, isFile, isHidden, lastModified, length, list, list, listFiles, listFiles, listFiles, listRoots, mkdir, mkdirs, renameTo, setLastModified, setReadOnly, toString, toURI, toURL

Methods inherited from class java.lang.Object
clone, finalize, getClass, notify, notifyAll, wait, wait, wait

Constructor detail

i5OSSystemCertificateStoreFile

```
public i5OSSystemCertificateStoreFile()
```

Creates a File() pointing to the *System certificate store file.

SSLConfiguration Javadoc information

com.ibm.i5os.jsse

Class SSLConfiguration

```
java.lang.Object
  |
  +--com.ibm.i5os.jsse.SSLConfiguration
```

All Implemented Interfaces:

java.lang.Cloneable, javax.net.ssl.ManagerFactoryParameters

public final class **SSLConfiguration**
extends java.lang.Object
implements javax.net.ssl.ManagerFactoryParameters, java.lang.Cloneable

This class provides for the specification of the configuration needed by the native IBM i JSSE implementation.

The native IBM i JSSE implementation works the most efficiently using a `KeyStore` object of type "IbmISeriesKeyStore". This type of `KeyStore` object contains key entries and trusted certificate entries based either on an application identifier registered with the Digital Certificate Manager (DCM) or on a keyring file (digital certificate container). A `KeyStore` object of this type can then be used to initialize an `X509KeyManger` and an `X509TrustManager` object from the "IBMi5OSJSSEProvider" `Provider`. The `X509KeyManager` and `X509TrustManager` objects can then be used to initialize an `SSLContext` object from the "IBMi5OSJSSEProvider". The `SSLContext` object then provides access to the native IBM i JSSE implementation based on the configuration information specified for the `KeyStore` object. Each time a load is performed for an "IbmISeriesKeyStore" `KeyStore`, the `KeyStore` is initialized based on the current configuration specified by the application identifier or keyring file.

This class can also be used to generate a `KeyStore` object of any valid type. The `KeyStore` is initialized based on the current configuration specified by the application identifier or keyring file. Any change made to the configuration specified by an application identifier or keyring file would require the `KeyStore` object to be regenerated to pick up the change. Note that a keyring password must be specified (for the *SYSTEM certificate store when using an application ID) to be able to successfully create a `KeyStore` of a type other than "IbmISeriesKeyStore". The keyring password must be specified to successfully gain access to any private key for any `KeyStore` of type "IbmISeriesKeyStore" which is created.

Since:
SDK 1.5

See Also:
KeyStore, X509KeyManager, X509TrustManager, SSLContext

Constructor Summary

SSLConfiguration() Creates a new SSLConfiguration. See "Constructor detail" on page 277 for more information.

*Table 11. **Method Summary***

void	"clear" on page 281() Clears all information in the object so that all of the get methods return null.
java.lang.Object	"clone" on page 282() Generates a new copy of this SSL configuration.
boolean	"equals" on page 282(java.lang.Objectobj) Indicates whether some other object is "equal to" this one.
protected void	"finalize" on page 281() Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
java.lang.String	"getApplicationId" on page 280() Returns the application ID.
java.lang.String	"getKeyringLabel" on page 280() Returns the keyring label.
java.lang.String	"getKeyringName" on page 280() Returns the keyring name.
char[]	"getKeyringPassword" on page 280() Returns the keyring password.
java.security.KeyStore	"getKeyStore" on page 282(char[]password) Returns a keystore of type "IbmISeriesKeyStore" using the given password.
java.security.KeyStore	"getKeyStore" on page 283(java.lang.Stringtype, char[]password) Returns a keystore of the requested type using the given password.
int	"hashCode" on page 282() Returns a hash code value for the object.
staticvoid	(java.lang.String[]args) Executes SSLConfiguration functions.
void	(java.lang.String[]args, java.io.PrintStreamout) Executes SSLConfiguration functions.
void	"setApplicationId" on page 281(java.lang.StringapplicationId) Sets the application ID.
void	"setApplicationId" on page 281(java.lang.StringapplicationId, char[]password) Sets the application ID and the keyring password.
void	"setKeyring" on page 281(java.lang.Stringname,java.lang.Stringlabel, char[]password) Sets the keyring information.

Methods inherited from class java.lang.Object
getClass, notify, notifyAll, toString, wait, wait, wait

Constructor detail

SSLConfiguration

```
public SSLConfiguration()
```

Creates a new SSLConfiguration. The application identifier and keyring information is initialized to default values.

The default value for the application identifier is the value specified for the "os400.secureApplication" property.

The default values for the keyring information is null if the "os400.secureApplication" property is specified. If the "os400.secureApplication" property is not specified, then the default value for the keyring name is the value specified for the"os400.certificateContainer" property. If the "os400.secureApplication" property is not specified, then the keyring label is initialized to the value of the "os400.certificateLabel" property. If neither of the "os400.secureApplication" or "os400.certificateContainer" properties are set, then the keyring name will be initialized to "*SYSTEM".

Method detail

----

main

```
public static void main(java.lang.String[]args)
```

Executes SSLConfiguration functions. There are four commands that can be performed: -help, -create, -display, and -update. The command must be the first parameter specified.

The following are the options which may be specified (in any order):

***-keystore* keystore-file-name**
: Specifies the name of the keystore file to be created, updated or displayed. This option is required for all commands.

***-storepass* keystore-file-password**
: Specifies the password associated with the keystore file to be created, updated, or displayed. This option is required for all commands.

***-storetype* keystore-type**
: Specifies the type of keystore file to be created, updated, or displayed. This option may be specified for any command. If this option is not specified, then a value of "IbmISeriesKeyStore" is used.

***-appid* application-identifier**
: Specifies the application identifier to be used to initialize a keystore file being created or updated. This option is optional for the *-create* and *-update* commands. Only one of the *-appid*, *keyring*, and *-systemdefault* options may be specified.

***-keyring* keyring-file-name**
: Specifies the keyring file name to be used to initialize a keystore file being created or updated. This option is optional for the *-create* and *-update* commands. Only one of the *-appid*, *keyring*, and *-systemdefault* options may be specified.

***-keyringpass* keyring-file-password**
: Specifies the keyring file password to be used to initialize a keystore file being created or updated. This option may be specified for the *-create* and *-update* commands and is required when a keystore type other than "IbmISeriesKeyStore" is specified. If this option is not specified, then the stashed keyring password is used.

***-keyringlabel* keyring-file-label**
: Specifies the keyring file label to be used to initialize a keystore file being created or updated. This option may only be specified when the *-keyring* option is also specified. If this option is not specified when the *keyring* option is specified, then the default label in the keyring is used.

-systemdefault
: Specifies the system default value is to be used to initialize a keystore file being created or updated. This option is optional for the *-create* and *-update* commands. Only one of the *-appid*, *keyring*, and *-systemdefault* options may be specified.

-v
: Specifies that verbose output is to be produced. This option may be specified for any command.

The help command displays usage information for specifying the paramters to this method. The parameters to invoke the help function is specified as follows:

```
-help
```

The create command creates a new keystore file. There are three variations of the create command. One variation to create a keystore based on a particular application identifier, another variation to create a keystore based on a keyring name, label, and password, and a third variation to create a keystore based on the system default configuration.

To create a keystore based on a particular application identifier, the *-appid* option must be specified. The following parameters would create a keystore file of type "IbmISeriesKeyStore" named "keystore.file" with a password of "keypass" which is initialized based on the application identifier "APPID":

```
-create -keystore keystore.file -storepass keypass -storetype IbmISeriesKeyStore
        -appid APPID
```

To create a keystore based on a particular keyring file, the *-keyring* option must be specified. The *-keyringpass* and *keyringlabel* options may also be specified. The following parameters would create a keystore file of type "IbmISeriesKeyStore" named "keystore.file" with a password of "keypass" which is initialized based on the keyring file named "keyring.file", keyring password "ringpass", and keyring label "keylabel":

```
-create -keystore keystore.file -storepass keypass -storetype IbmISeriesKeyStore
        -keyring keyring.file -keyringpass ringpass -keyringlabel keylabel
```

To create a keystore based on the system default configuration, the *-systemdefault* option must be specified. The following parameters would create a keystore file of type "IbmISeriesKeyStore" named "keystore.file" with a password of "keypass" which is initialized based on the system default configuration:

```
-create -keystore keystore.file -storepass keypass -systemdefault
```

The update command updates an existing keystore file of type "IbmISeriesKeyStore". There are three variations of the update command which are identical to the variations of the create command. The options for the update command are identical to the options used for the create command. The display command displays the configuration specified for an existing keystore file. The following parameters would display the configuration specified by a keystore file of type "IbmISeriesKeyStore" named "keystore.file" with a password of "keypass":

```
-display -keystore keystore.file -storepass keypass -storetype IbmISeriesKeyStore
```

Parameters:
: `args` - the command line arguments

----

run

```
public void run(java.lang.String[]args,
                java.io.PrintStreamout)
```

Executes SSLConfiguration functions. The parameters and functionality of this method are identical to the main() method.

Parameters:
`args` - the command arguments
`out` - output stream to which results are to be written

See Also:`com.ibm.i5os.jsse.SSLConfiguration.main()`

----

getApplicationId

```
public java.lang.String getApplicationId()
```

Returns the application ID.

Returns:
the application ID.

----

getKeyringName

```
public java.lang.String getKeyringName()
```

Returns the keyring name.

Returns:
the keyring name.

----

getKeyringLabel

```
public java.lang.String getKeyringLabel()
```

Returns the keyring label.

Returns:
the keyring label.

----

getKeyringPassword

```
public final char[] getKeyringPassword()
```

Returns the keyring password.

Returns:
the keyring password.

----

finalize

```
protected void finalize()
                 throws java.lang.Throwable
```

Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.

Overrides:
`finalize` in class `java.lang.Object`

Throws:
`java.lang.Throwable` - the exception raised by this method.

----

clear

```
public void clear()
```

Clears all information in the object so that all of the get methods return null.

----

setKeyring

```
public void setKeyring(java.lang.Stringname,
                       java.lang.Stringlabel,
                       char[]password)
```

Sets the keyring information.

Parameters:
`name` - the keyring name

`label` - the keyring label, or null if the default keyring entry is to be used.

`password` - the keyring password, or null if the stashed password is to be used.

----

setApplicationId

```
public void setApplicationId(java.lang.StringapplicationId)
```

Sets the application ID.

Parameters:
`applicationId` - the application ID.

----

setApplicationId

```
public void setApplicationId(java.lang.StringapplicationId,
                             char[]password)
```

Sets the application ID and the keyring password. Specifying the keyring password allows any keystore which is created to allow access to the private key.

Parameters:
: `applicationId` - the application ID.
: `password` - the keyring password.

----

equals

```
public boolean equals(java.lang.Objectobj)
```

Indicates whether some other object is "equal to" this one.

Overrides:
: `equals` in class `java.lang.Object`

Parameters:
: `obj` - object to be compared

Returns:
: indicator of whether the objects specify the same configuration information

----

hashCode

```
public int hashCode()
```

Returns a hash code value for the object.

Overrides:
: `hashCode` in class `java.lang.Object`

Returns:
: a hash code value for this object.

----

clone

```
public java.lang.Object clone()
```

Generate a new copy of this SSL configuration. Subsequent changes to the components of this SSL configuration will not affect the new copy, and vice versa.

Overrides:
: `clone` in class `java.lang.Object`

Returns:
: a copy of this SSL configuration

----

getKeyStore

```
public java.security.KeyStore getKeyStore(char[]password)
                               throws java.security.KeyStoreException
```

Returns a keystore of type "IbmISeriesKeyStore" using the given password. The keystore is intialized based on the configuration information currently stored in the object.

Parameters:
: `password` - used to initialize the keystore

Returns:
: KeyStore keystore initialized based on the configuration information currently stored in the object

Throws:
: `java.security.KeyStoreException` - if the keystore could not be created

----

getKeyStore

```
public java.security.KeyStore getKeyStore(java.lang.Stringtype,
                                          char[]password)
                                   throws java.security.KeyStoreException
```

Returns a keystore of the requested type using the given password. The keystore is initialized based on the configuration information currently stored in the object.

Parameters:
: `type` - type of keystore to be returned
: `password` - used to initialize the keystore

Returns:
: KeyStore keystore initialized based on the configuration information currently stored in the object

Throws:
: `java.security.KeyStoreException` - if the keystore could not be created

Examples: IBM Java Secure Sockets Extension

The JSSE examples show how a client and a server can use the native IBM i JSSE provider to create a context that enables secure communications.

Note: Both examples use the native IBM i JSSE provider, regardless of the properties specified by the java.security file.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

Example: SSL client using an SSLContext object

This example client program utilizes an SSLContext object, which it initializes to use the "MY_CLIENT_APP" application ID. This program will use the native IBM i implementation regardless of what is specified in the java.security file.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
////////////////////////////////////////////////////////////////////////////////////
//
// This example client program utilizes an SSLContext object, which it initializes
// to use the "MY_CLIENT_APP" application ID.
//
// The example uses the native JSSE provider, regardless of the
// properties specified by the java.security file.
//
// Command syntax:
//     java SslClient
//
////////////////////////////////////////////////////////////////////////////////////

import java.io.*;
import javax.net.ssl.*;
import java.security.*;
import com.ibm.i5os.jsse.SSLConfiguration;
/**
 * SSL Client Program.
```

```
 */
public class SslClient {

   /**
    * SslClient main method.
    *
    * @param args the command line arguments (not used)
    */
   public static void main(String args[]) {
       /*
        * Set up to catch any exceptions thrown.
        */
       try {
           /*
            * Initialize an SSLConfiguration object to specify an application
            * ID.  "MY_CLIENT_APP" must be registered and configured
            * correctly with the Digital Certificate Manager (DCM).
            */
           SSLConfiguration config = new SSLConfiguration();
           config.setApplicationId("MY_CLIENT_APP");
           /*
            * Get a KeyStore object from the SSLConfiguration object.
            */
           char[] password = "password".toCharArray();
           KeyStore ks = config.getKeyStore(password);
           /*
            * Allocate and initialize a KeyManagerFactory.
            */
           KeyManagerFactory kmf =
                             KeyManagerFactory.getInstance("IbmISeriesX509");
           kmf.init(ks, password);
           /*
            * Allocate and initialize a TrustManagerFactory.
            */
           TrustManagerFactory tmf =
                           TrustManagerFactory.getInstance("IbmISeriesX509");
           tmf.init(ks);
           /*
            * Allocate and initialize an SSLContext.
            */
           SSLContext c =
                      SSLContext.getInstance("SSL", "IBMi5OSJSSEProvider");
           c.init(kmf.getKeyManagers(), tmf.getTrustManagers(), null);
           /*
            * Get the an SSLSocketFactory from the SSLContext.
            */
           SSLSocketFactory sf = c.getSocketFactory();
           /*
            * Create an SSLSocket.
            *
            * Change the hard-coded IP address to the IP address or host name
            * of the server.
            */
           SSLSocket s = (SSLSocket) sf.createSocket("1.1.1.1", 13333);
           /*
            * Send a message to the server using the secure session.
            */
           String sent = "Test of java SSL write";
           OutputStream os = s.getOutputStream();
           os.write(sent.getBytes());
           /*
            * Write results to screen.
            */
           System.out.println("Wrote " + sent.length() + " bytes...");
           System.out.println(sent);
           /*
            * Receive a message from the server using the secure session.
            */
           InputStream is = s.getInputStream();
           byte[] buffer = new byte[1024];
           int bytesRead = is.read(buffer);
           if (bytesRead == -1)
               throw new IOException("Unexpected End-of-file Received");
           String received = new String(buffer, 0, bytesRead);
           /*
            * Write results to screen.
            */
           System.out.println("Read " + received.length() + " bytes...");
           System.out.println(received);
       } catch (Exception e) {
           System.out.println("Unexpected exception caught: " +
                              e.getMessage());
```

```
            e.printStackTrace();
        }
    }

}
```

Example: SSL server using an SSLContext object

The following server program utilizes an SSLContext object that it initializes with a previously created keystore file.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
///////////////////////////////////////////////////////////////////////////////////
//
// The following server program utilizes an SSLContext object that it
// initializes with a previously created keystore file.
//
// The keystore file has the following name and keystore password:
//   File name: /home/keystore.file
//   Password:  password
//
// The example program needs the keystore file in order to create an
// IbmISeriesKeyStore object. The KeyStore object must specify MY_SERVER_APP as
// the application identifier.
//
// To create the keystore file, you can use the following Qshell command:
//
//   java com.ibm.i5os.SSLConfiguration  -create  -keystore /home/keystore.file
//        -storepass password  -appid MY_SERVER_APP
//
// Command syntax:
//     java  JavaSslServer
//
// You can also create the keystore file by entering this command at an CL command prompt:
//
//  RUNJVA CLASS(com.ibm.i5os.SSLConfiguration) PARM('-create' '-keystore'
//      '/home/keystore.file' '-storepass' 'password' '-appid' 'MY_SERVER_APP')
//
//////////////////////////////////////////////////////////////////////////////////

import java.io.*;
import javax.net.ssl.*;
import java.security.*;
/**
* Java SSL Server Program using Application ID.
*/
public class JavaSslServer {

   /**
    * JavaSslServer main method.
    *
    * @param args the command line arguments (not used)
    */
   public static void main(String args[]) {
       /*
        * Set up to catch any exceptions thrown.
        */
       try {
           /*
            * Allocate and initialize a KeyStore object.
            */
           char[] password = "password".toCharArray();
           KeyStore ks = KeyStore.getInstance("IbmISeriesKeyStore");
           FileInputStream fis = new FileInputStream("/home/keystore.file");
           ks.load(fis, password);
           /*
            * Allocate and initialize a KeyManagerFactory.
            */
           KeyManagerFactory kmf =
                             KeyManagerFactory.getInstance("IbmISeriesX509");
           kmf.init(ks, password);
           /*
            * Allocate and initialize a TrustManagerFactory.
            */
           TrustManagerFactory tmf =
                           TrustManagerFactory.getInstance("IbmISeriesX509");
```

```
            tmf.init(ks);
            /*
             * Allocate and initialize an SSLContext.
             */
            SSLContext c =
                       SSLContext.getInstance("SSL", "IBMi5OSJSSEProvider");
            c.init(kmf.getKeyManagers(), tmf.getTrustManagers(), null);
            /*
             * Get the an SSLServerSocketFactory from the SSLContext.
             */
            SSLServerSocketFactory sf = c.getServerSocketFactory();
            /*
             * Create an SSLServerSocket.
             */
            SSLServerSocket ss =
                (SSLServerSocket) sf.createServerSocket(13333);
            /*
             * Perform an accept() to create an SSLSocket.
             */
            SSLSocket s = (SSLSocket) ss.accept();
            /*
             * Receive a message from the client using the secure session.
             */
            InputStream is = s.getInputStream();
            byte[] buffer = new byte[1024];
            int bytesRead = is.read(buffer);
            if (bytesRead == -1)
                throw new IOException("Unexpected End-of-file Received");
            String received = new String(buffer, 0, bytesRead);
            /*
             * Write results to screen.
             */
            System.out.println("Read " + received.length() + " bytes...");
            System.out.println(received);
            /*
             * Echo the message back to the client using the secure session.
             */
            OutputStream os = s.getOutputStream();
            os.write(received.getBytes());
            /*
             * Write results to screen.
             */
            System.out.println("Wrote " + received.length() + " bytes...");
            System.out.println(received);
        } catch (Exception e) {
            System.out.println("Unexpected exception caught: " +
                                   e.getMessage());
            e.printStackTrace();
        }
    }

}
```

Java Authentication and Authorization Service

The Java Authentication and Authorization Service (JAAS) is a standard extension to the Java 2 Platform, Standard Edition (J2SE). J2SE provides access controls that are based on where the code originated and who signed the code (code source-based access controls). It lacks, however, the ability to enforce additional access controls based on who runs the code. JAAS provides a framework that adds this support to the Java 2 security model.

The JAAS implementation on IBM i is compatible with the implementation of Oracle Corporation. This documentation covers the unique aspects of the IBM i implementation. We assume that you are familiar with the general documentation for the JAAS extensions. To make it easier for you to work with that and our IBM i information, we provide the following links.

Related concepts

Changes to adopted authority in IBM i 7.2
Support for adopted user profile authority through Java programs is not supported in IBM i 7.2. This topic describes how to determine if your applications are using adopted authority and how to modify your applications to accommodate this change.

Java security model

You can download Java applets from any system; thus, security mechanisms exist within the Java virtual machine to protect against malicious applets. The Java runtime system verifies the bytecodes as the Java virtual machine loads them. This ensures that they are valid bytecodes and that the code does not violate any of the restrictions that the Java virtual machine places on Java applets.

Java Cryptography Extension
The Java Cryptography Extension (JCE) provides a framework and implementations for encryption, key generation, and key agreement, as well as Message Authentication Code (MAC) algorithms. Support for encryption includes symmetric, asymmetric, block, and stream ciphers. The software also supports secure streams and sealed objects. JCE supplements the Java 2 platform, which already includes interfaces and implementations of message digests and digital signatures.

Java Secure Socket Extension
Java Secure Socket Extension (JSSE) is like a framework that abstracts the underlying mechanisms of both Secure Sockets Layer (SSL) and Transport Layer Security (TLS). By abstracting the complexity and peculiarities of the underlying protocols, JSSE enables programmers to use secure encrypted communications while at the same time minimizing possible security vulnerabilities. Java Secure Socket Extension (JSSE) uses both the SSL protocol and the TLS protocol to provide secure encrypted communications between your clients and servers.

IBM Java Generic Security Service (JGSS)
The Java Generic Security Service (JGSS) provides a generic interface for authentication and secure messaging. Under this interface you can plug a variety of security mechanisms based on secret-key, public-key, or other security technologies.

Related information

JAAS API Specification

JAAS LoginModule

Java Authentication and Authorization Service (JAAS) 1.0

The Java Authentication and Authorization Service (JAAS) is a standard extension to the Java 2 Software Development Kit. Currently, Java 2 provides codesource-based access controls (access controls based on *where* the code originated from and *who signed* the code). It lacks, however, the ability to additionally enforce access controls based on *who runs* the code. JAAS provides a framework that augments the Java 2 security model with such support.

Developer's Guide

- **AuthPermission**
- **PrivateCredentialPermission**

References

- Implementation
- "Hello World", JAAS style!
- Appendix A: JAAS Settings in the java.security Security Properties File
- Appendix B: Login Configuration File
- Appendix C: Authorization Policy File

Overview

The Java Authentication and Authorization Service (JAAS) is a standard extension to the Java 2 Software Development Kit, version 1.3. Currently, Java 2 provides codesource-based access controls (access controls based on *where* the code originated from and *who signed* the code). It lacks, however, the ability to additionally enforce access controls based on *who runs* the code. JAAS provides a framework that augments the Java 2 security model with such support.

Who Should Read This Document

This document is intended for experienced programmers wanting to create applications constrained by a codesource-based and Subject-based security model.

Related Documentation

This document assumes you have already read the following documentation:

- Java 2 Software Development Kit API Specification
- JAAS API Specification
- Security and the Java platform

A supplement to this guide is the LoginModule Developer's Guide that is supplied by Oracle Corporation.

Introduction

The JAAS infrastructure can be divided into two main components: an **authentication** component and an **authorization** component. The JAAS authentication component provides the ability to reliably and securely determine who is currently processing Java code, regardless of whether the code is running as an application, an applet, a bean, or a servlet. The JAAS authorization component supplements the existing Java 2 security framework by providing the means to restrict the processing Java code from performing sensitive tasks, depending on its codesource (as is done in Java 2) and depending on who was authenticated.

JAAS authentication is performed in a *pluggable* fashion. This permits Java applications to remain independent from underlying authentication technologies. Therefore new or updated authentication technologies can be plugged under an application without requiring modifications to the application itself. Applications enable the authentication process by instantiating a

```
LoginContext
```

object, which in turn references a

```
Configuration
```

to determine the authentication technology, or

```
LoginModule
```

, to be used in performing the authentication. Typical LoginModules may prompt for and verify a username and password. Others may read and verify a voice or fingerprint sample.

Once the user processing the code has been authenticated, the JAAS authorization component works in conjunction with the existing Java 2 access control model to protect access to sensitive resources. Unlike in Java 2, where access control decisions are based solely on code location and code signers (a

```
CodeSource
```

), in JAAS access control decisions are based both on the processing code's

```
CodeSource
```

, as well as on the user running the code, or the

```
Subject
```

. Note that the JAAS policy merely extends the Java 2 policy with the relevant Subject-based information. Therefore permissions recognized and understood in Java 2 (

```
java.io.FilePermission
```

and

```
java.net.SocketPermission
```

, for example) are also understood and recognized by JAAS. Furthermore, although the JAAS security policy is physically separate from the existing Java 2 security policy, the two policies, together, form one logical policy.

Core Classes

The JAAS core classes can be broken into 3 categories: Common, Authentication, and Authorization.

- Common Classes
  - Subject, Principals, Credentials
- Authentication Classes
  - LoginContext, LoginModule, CallbackHandler, Callback
- Authorization Classes
  - Policy, AuthPermission, PrivateCredentialPermission

Common Classes

Common classes are shared within both the JAAS authentication and authorization components.

The key JAAS class is

```
Subject
```

, which represents a grouping of related information for a single entity such as a person. It encompasses the entity's Principals, public credentials, and private credentials.

Note that JAAS uses the existing Java 2

```
java.security.Principal
```

interface to represent a Principal. Also note that JAAS does not introduce a separate credential interface or class. A credential, as defined by JAAS, may be any Object.

Subject

To authorize access to resources, applications first need to authenticate the source of the request. The JAAS framework defines the term, Subject, to represent the source of a request. A Subject may be any entity, such as a person or service. Once authenticated, a Subject is populated with associated identities, or Principals. A Subject may have many Principals. For example, a person may have a name Principal ("John Doe") and a SSN Principal ("123-45-6789") which distinguishes it from other Subjects.

A

```
Subject
```

may also own security-related attributes, which are referred to as credentials. Sensitive credentials that require special protection, such as private cryptographic keys, are stored within a private credential

```
Set
```

. Credentials intended to be shared, such as public key certificates or Kerberos tickets are stored within a public credential

```
Set
```

. Different permissions are required to access and modify the different credential Sets.

Subjects are created using these constructors:

```
    public Subject();

    public Subject(boolean readOnly, Set principals,
                   Set pubCredentials, Set privCredentials);
```

The first constructor creates a Subject with empty (non-null) Sets of Principals and credentials. The second constructor creates a Subject with the specified Sets of Principals and credentials. It also has a boolean argument which can create a read-only Subject (immutable Principal and credential Sets).

An alternative way to obtain a reference to an authenticated Subject without using these constructors will be shown in the LoginContext section.

If a Subject was not instantiated to be in a read-only state, it can be set to a read-only state by calling this method:

```
    public void setReadOnly();
```

An

```
AuthPermission("setReadOnly")
```

is required to invoke this method. Once in a read-only state, any attempt to add or remove Principals or credentials will result in an

```
IllegalStateException
```

being thrown.

This method may be called to test a Subject's read-only state:

```
    public boolean isReadOnly();
```

To retrieve the Principals associated with a Subject, two methods are available:

```
    public Set getPrincipals();
    public Set getPrincipals(Class c);
```

The first method returns all Principals contained in the Subject, while the second method only returns those Principals that are an instance of the specified Class c, or an instance of a subclass of Class c. An empty set will be returned if the Subject does not have any associated Principals.

To retrieve the public credentials associated with a Subject, these methods are available:

```
    public Set getPublicCredentials();
    public Set getPublicCredentials(Class c);
```

The observed behavior of these methods is identical to that for the

```
getPrincipals
```

method.

To access private credentials associated with a Subject, the following methods are available:

```
    public Set getPrivateCredentials();
    public Set getPrivateCredentials(Class c);
```

The observed behavior of these methods is identical to that for the

```
getPrincipals
```

and

```
getPublicCredentials
```

methods.

To modify or operate upon a Subject's Principal Set, public credential Set, or private credential Set, callers use the methods defined in the

```
java.util.Set
```

class. The following example demonstrates this:

```
    Subject subject;
    Principal principal;
    Object credential;

    // add a Principal and credential to the Subject
    subject.getPrincipals().add(principal);
    subject.getPublicCredentials().add(credential);
```

Note that an

```
AuthPermission("modifyPrincipals")
```

,

```
AuthPermission("modifyPublicCredentials")
```

, or

```
AuthPermission("modifyPrivateCredentials")
```

is required to modify the respective Sets. Also note that only the sets returned via the

```
getPrincipals
```

,

```
getPublicCredentials
```

, and

```
getPrivateCredentials
```

methods are backed by the Subject's respective internal sets. Therefore any modification to the returned set affects the internal sets as well. The sets returned via the

```
getPrincipals(Class c)
```

,

```
getPublicCredentials(Class c)
```

, and

```
getPrivateCredentials(Class c)
```

methods are not backed by the Subject's respective internal sets. A new set is created and returned for each method invocation. Modifications to these sets will not affect the Subject's internal sets. The following method returns the Subject associated with the specified

```
AccessControlContext
```

, or null if no Subject is associated with the specified

```
AccessControlContext
```

.

```
    public static Subject getSubject(final AccessControlContext acc);
```

An

```
AuthPermission("getSubject")
```

is required to call

```
Subject.getSubject
```

.

The Subject class also includes these methods inherited from

```
java.lang.Object
```

:

```
    public boolean equals(Object o);
    public String toString();
    public int hashCode();
```

The following static methods may be called to perform work as a particular Subject:

```
    public static Object doAs(final Subject subject,
                         final java.security.PrivilegedAction action);

    public static Object doAs(final Subject subject,
                         final java.security.PrivilegedExceptionAction action)
                         throws java.security.PrivilegedActionException;
```

Both methods first associate the specified ***subject*** with the current Thread's

```
AccessControlContext
```

, and then process the ***action***. This achieves the effect of having the ***action*** run as the ***subject***. The first method can throw runtime exceptions but normal processing has it returning an Object from the run() method of its action argument. The second method behaves similarly except that it can throw a checked exception from its

```
PrivilegedExceptionAction
```

run() method. An

```
AuthPermission("doAs")
```

is required to call the

```
doAs
```

methods.

Here are two examples utilizing the first

```
doAs
```

method. Assume that a

```
Subject
```

with a Principal of class

```
com.ibm.security.Principal
```

named "BOB" has been authenticated by a

```
LoginContext
```

"lc". Also, assume that a SecurityManager has been installed, and the following exists in the JAAS access control policy (see the Policy section for more details on the JAAS policy file):

```
   // Grant "BOB" permission to read the file "foo.txt"
   grant Principal com.ibm.security.Principal "BOB" {
      permission java.io.FilePermission "foo.txt", "read";
};
```

Subject.doAs Example 1

```
    class ExampleAction implements java.security.PrivilegedAction {
        public Object run() {
            java.io.File f = new java.io.File("foo.txt");

            // exists() invokes a security check
            if (f.exists()) {
                System.out.println("File foo.txt exists.");
            }
            return null;
        }
    }

    public class Example1 {
        public static void main(String[] args) {

            // Authenticate the subject, "BOB".
            // This process is described in the
            // LoginContext section.

            Subject bob;
            ...

            // perform "ExampleAction" as "BOB":
            Subject.doAs(bob, new ExampleAction());
        }
    }
```

During processing,

```
ExampleAction
```

will encounter a security check when it makes a call to,

```
f.exists()
```

. However, since

```
ExampleAction
```

is running as "BOB", and because the JAAS policy (above) grants the necessary

```
FilePermission
```

to "BOB", the

```
ExampleAction
```

will pass the security check.

Example 2 has the same scenario as Example 1.

Subject.doAs Example 2

```
    public class Example2 {
        // Example of using an anonymous action class.
```

```
        public static void main(String[] args) {
            // Authenticate the subject, "BOB".
            // This process is described in the
            // LoginContext section.

            Subject bob;
            ...

            // perform "ExampleAction" as "BOB":
            Subject.doAs(bob, new ExampleAction() {
                public Object run() {
                    java.io.File f = new java.io.File("foo.txt");
                    if (f.exists()) {
                        System.out.println("File foo.txt exists.");
                    }
                    return null;
                }
            });
        }
    }
```

Both examples throw a

```
SecurityException
```

if the example permission grant statement is altered incorrectly, such as adding an incorrect CodeBase or changing the Principal to "MOE". Removing the Principal field from the grant block and then moving it to a Java 2 policy file will not cause a

```
SecurityException
```

to be thrown because the permission is more general now (available to all Principals).

Since both examples perform the same function, there must be a reason to write code one way over the other. Example 1 may be easier to read for some programmers unfamiliar with anonymous classes. Also, the ***action*** class could be placed in a separate file with a unique CodeBase and then the permission grant could utilize this information. Example 2 is more compact and the ***action*** to be performed is easier to find since it is right there in the

```
doAs
```

call.

The following methods also perform work as a particular Subject. However, the

```
doAsPrivileged
```

methods will have security checks based on the supplied ***action*** and ***subject***. The supplied context will be tied to the specified ***subject*** and ***action***. A null context object will disregard the current

```
AccessControlContext
```

altogether.

```
    public static Object doAsPrivileged(final Subject subject,
                         final java.security.PrivilegedAction action,
                         final java.security.AccessControlContext acc);

    public static Object doAsPrivileged(final Subject subject,
                         final java.security.PrivilegedExceptionAction action,
                         final java.security.AccessControlContext acc)
                         throws java.security.PrivilegedActionException;
```

The

```
doAsPrivileged
```

methods behave similarly to the

```
doAs
```

methods: the ***subject*** is associated with the context ***acc***, an ***action*** is performed, and runtime exceptions or checked exceptions may be thrown. However, the

```
doAsPrivileged
```

methods first empties the existing Thread's

```
AccessControlContext
```

before associating the ***subject*** with the supplied context, and before invoking the ***action***. A null ***acc*** argument has the effect of causing access control decisions (invoked while the ***action*** processes) to be based solely upon the ***subject*** and ***action***. An

```
AuthPermission("doAsPrivileged")
```

is required when calling the

```
doAsPrivileged
```

methods.

Principals

As mentioned previously, Principals may be associated with a Subject. Principals represent Subject identities, and must implement the

```
java.security.Principal
```

and

```
java.io.Serializable
```

interfaces. The Subject section describes ways to update the Principals associated with a Subject.

Credentials

Public and private credential classes are not part of the core JAAS class library. Any java class, therefore, can represent a credential. However, developers may elect to have their credential classes implement two interfaces related to credentials: Refreshable and Destroyable.

Refreshable

This **interface** provides the capability for a credential to refresh itself. For example, a credential with a particular time-restricted lifespan may implement this interface to allow callers to refresh the time period for which it is valid. The interface has two abstract methods:

```
    boolean isCurrent();
```

Determines if the credential is current or valid.

```
    void refresh() throws RefreshFailedException;
```

Updates or extends the validity of the credential. This method implementation performs an

```
AuthPermission("refreshCredential")
```

security check to ensure the caller has permission to refresh the credential.

Destroyable

This **interface** provides the capability of destroying the contents within a credential. The interface has two abstract methods:

```
    boolean isDestroyed();
```

Determines if the credential has been destroyed.

```
    void destroy() throws DestroyFailedException;
```

Destroys and clears the information associated with this credential. Subsequent calls to certain methods on this credential will result in an

```
IllegalStateException
```

being thrown. This method implementation performs an

```
AuthPermission("destroyCredential")
```

security check to ensure the caller has permission to destroy the credential.

Authentication Classes

To authenticate a

```
Subject
```

, the following steps are performed:

1. An application instantiates a

   ```
   LoginContext
   ```

   .
2. The

   ```
   LoginContext
   ```

   consults a configuration to load all of the LoginModules configured for that application.
3. The application invokes the LoginContext's *login* method.
4. The *login* method invokes all of the loaded LoginModules. Each

   ```
   LoginModule
   ```

   attempts to authenticate the

```
Subject
```

. Upon success, LoginModules associate relevant Principals and credentials with the

```
Subject
```

.

5. The

```
LoginContext
```

returns the authentication status to the application.

6. If authentication succeeded, the application retrieves the authenticated

```
Subject
```

from the

```
LoginContext
```

.

LoginContext

The

```
LoginContext
```

class provides the basic methods used to authenticate Subjects, and provides a way to develop an application independent of the underlying authentication technology. The

```
LoginContext
```

consults a configuration

```
Configuration
```

to determine the authentication services, or LoginModules, configured for a particular application. Therefore, different LoginModules can be plugged in under an application without requiring any modifications to the application itself.

```
LoginContext
```

offers four constructors to choose from:

```
    public LoginContext(String name) throws LoginException;

    public LoginContext(String name, Subject subject) throws LoginException;

    public LoginContext(String name, CallbackHandler callbackHandler)
           throws LoginException

    public LoginContext(String name, Subject subject,
           CallbackHandler callbackHandler) throws LoginException
```

All of the constructors share a common parameter: ***name***. This argument is used by the

```
LoginContext
```

to index the login Configuration. Constructors that do not take a

```
Subject
```

as an input parameter instantiate a new

```
Subject
```

. Null inputs are disallowed for all constructors. Callers require an

```
AuthPermission("createLoginContext")
```

to instantiate a

```
LoginContext
```

.

Actual authentication occurs with a call to the following method:

```
    public void login() throws LoginException;
```

When *login* is invoked, all of the configured LoginModules' respective *login* methods are invoked to perform the authentication. If the authentication succeeded, the authenticated

```
Subject
```

(which may now hold Principals, public credentials, and private credentials) can be retrieved by using the following method:

```
    public Subject getSubject();
```

To logout a

```
Subject
```

and remove its authenticated Principals and credentials, the following method is provided:

```
    public void logout() throws LoginException;
```

The following snippet of code in an application will authenticate a Subject called "bob" after accessing a configuration file with a configuration entry named "moduleFoo":

```
    Subject bob = new Subject();
    LoginContext lc = new LoginContext("moduleFoo", bob);
    try {
        lc.login();
        System.out.println("authentication successful");
    } catch (LoginException le) {
        System.out.println("authentication unsuccessful"+le.printStackTrace());
    }
```

This snippet of code in an application will authenticate a "nameless" Subject and then use the getSubject method to retrieve it:

```
    LoginContext lc = new LoginContext("moduleFoo");
    try {
        lc.login();
        System.out.println("authentication successful");
    } catch (LoginException le) {
        System.out.println("authentication unsuccessful"+le.printStackTrace());
    }
    Subject subject = lc.getSubject();
```

If the authentication failed, then *getSubject* returns null. Also, there isn't an

```
AuthPermission("getSubject")
```

required to do this as is the case for

```
Subject.getSubject
```

LoginModule

The LoginModule **interface** gives developers the ability to implement different kinds of authentication technologies that can be plugged under an application. For example, one type of

```
LoginModule
```

may perform a username/password-based form of authentication.

The LoginModule Developer's Guide is a detailed document that gives developers step-by-step instructions for implementing LoginModules.

To instantiate a

```
LoginModule
```

, a

```
LoginContext
```

expects each

```
LoginModule
```

to provide a public constructor that takes no arguments. Then, to initialize a

```
LoginModule
```

with the relevant information, a

```
LoginContext
```

calls the LoginModule's

```
initialize
```

method. The provided *subject* is guaranteed to be non-null.

```
void initialize(Subject subject, CallbackHandler callbackHandler,
    Map sharedState, Map options);
```

This following method begins the authentication process:

```
boolean login() throws LoginException;
```

An example method implementation may prompt the user for a username and password, and then verify the information against the data stored in a naming service such as NIS or LDAP. Alternative implementations might interface smart cards and biometric devices, or may simply extract user information from the underlying operating system. This is considered ***phase 1*** of the JAAS authentication process.

The following method completes and finalizes the authentication process:

```
boolean commit() throws LoginException;
```

If ***phase 1*** of the authentication process was successful, then this method continues with ***phase 2***: associating Principals, public credentials, and private credentials with the Subject. If ***phase 1*** failed, then the *commit* method removes any previously stored authentication state, such as usernames and passwords.

The following method halts the authentication process if ***phase 1*** was unsuccessful:

```
boolean abort() throws LoginException;
```

Typical implementations of this method clean up previously stored authentication state, such as usernames or passwords. The following method logs out a Subject:

```
boolean logout() throws LoginException;
```

This method removes the Principals and credentials originally associated with the

```
Subject
```

during the

```
commit
```

operation. Credentials are destroyed upon removal.

CallbackHandler

In some cases a LoginModule must communicate with the user to obtain authentication information. LoginModules use a CallbackHandler for this purpose. Applications implement the CallbackHandler **interface** and pass it to the LoginContext, which forwards it directly to the underlying LoginModules. LoginModules use the CallbackHandler both to gather input from users (such as a password or smart card pin number) or to supply information to users (such as status information). By allowing the application to specify the CallbackHandler, underlying LoginModules can remain independent of the different ways applications interact with users. For example, the implementation of a CallbackHandler for a GUI application might display a Window to solicit input from a user. The implementation of a CallbackHandler for a non-GUI tool might prompt the user for input directly from the command line.

```
CallbackHandler
```

is an **interface** with one method to implement:

```
void handle(Callback[] callbacks)
throws java.io.IOException, UnsupportedCallbackException;
```

Callback

The javax.security.auth.callback package contains the Callback **interface** as well as several implementations. LoginModules may pass an array of Callbacks directly to the *handle* method of a CallbackHandler.

Consult the various Callback APIs for more information on their use.

Authorization Classes

Upon successful authentication of a

```
Subject
```

, fine-grained access controls can be placed upon that

```
Subject
```

by invoking the Subject.doAs or Subject.doAsPrivileged methods. The permissions granted to that

```
Subject
```

are configured in a JAAS

```
Policy
```

.

Policy

This is an **abstract** class for representing the system-wide JAAS access control. As a default, JAAS provides a file-based subclass implementation, PolicyFile. Each

```
Policy
```

subclass must implement the following methods:

```
public abstract java.security.PermissionCollection getPermissions
                (Subject subject,
                java.security.CodeSource cs);
public abstract void refresh();
```

The

```
getPermissions
```

method returns the permissions granted to the specified

```
Subject
```

and

```
CodeSource
```

. The

```
refresh
```

method updates the runtime

```
Policy
```

with any modifications made since the last time it was loaded from its permanent store (a file or database, for example). The

```
refresh
```

method requires an

```
AuthPermission("refreshPolicy")
```

.

The following method retrieves the current runtime

```
Policy
```

object, and is protected with a security check that requires the caller to have an

```
AuthPermission("getPolicy")
```

.

```
    public static Policy getPolicy();
```

The following example code demonstrates how a

```
Policy
```

object can be queried for the set of permissions granted to the specified

```
Subject
```

and

```
CodeSource
```

:

```
    policy = Policy.getPolicy();
    PermissionCollection perms = policy.getPermissions(subject, codeSource);
```

To set a new

```
Policy
```

object for the Java runtime, the

```
Policy.setPolicy
```

method may be used. This method requires the caller to have an

```
AuthPermission("setPolicy")
```

.

```
    public static void setPolicy(Policy policy);
```

Policy File Sample Entries:

These examples are relevant only for the default PolicyFile implementation.

Each entry in the

```
Policy
```

is represented as a ***grant*** entry. Each ***grant*** entry specifies a codebase/code-signers/Principals triplet, as well as the Permissions granted to that triplet. Specifically, the permissions will be granted to any code downloaded from the specified *codebase* and signed by the specified *code signers*, so long as the

```
Subject
```

running that code has all of the specified *Principals* in its

```
Principal
```

set. Refer to the Subject.doAs examples to see how a

```
Subject
```

becomes associated with running code.

```
    grant CodeBase ["URL"],
          Signedby ["signers"],
          Principal [Principal_Class] "Principal_Name",
          Principal ... {
        permission Permission_Class ["Target_Name"]
                                    [, "Permission_Actions"]
                                    [, signedBy "SignerName"];
    };

    // example grant entry
    grant CodeBase "http://griffin.ibm.com", Signedby "davis",
          Principal com.ibm.security.auth.NTUserPrincipal "kent" {
        permission java.io.FilePermission "c:/kent/files/*", "read, write";
    };
```

If no *Principal* information is specified in the JAAS

```
Policy
```

grant entry, a parsing exception will be thrown. However, grant entries that already exist in the regular Java 2 codesource-based policy file (and therefore have no *Principal* information) are still valid. In those cases, the *Principal* information is implied to be '*' (the grant entries applies to all Principals).

The CodeBase and Signedby components of the grant entry are optional in the JAAS

```
Policy
```

. If they are not present, then any codebase will match, and any signer (including unsigned code) will match.

In the example above, the ***grant*** entry specifies that code downloaded from "http://griffin.ibm.com", signed by "davis", and running as the NT user "kent", has one

```
Permission
```

. This

```
Permission
```

permits the processing code to read and write files in the directory "c:\kent\files".

Multiple Principals may be listed within one ***grant*** entry. The current

```
Subject
```

running the code must have all of the specified Principals in its

```
Principal
```

set to be granted the entry's Permissions.

```
    grant Principal com.ibm.security.auth.NTUserPrincipal "kent",
          Principal com.ibm.security.auth.NTSidGroupPrincipal "S-1-1-0" {
        permission java.io.FilePermission "c:/user/kent/", "read, write";
        permission java.net.SocketPermission "griffin.ibm.com", "connect";
    };
```

This entry grants any code running as both the NT user "kent" with the NT group identification number "S-1-1-0", permission to read and write files in "c:\user\kent", as well as permission to make socket connections to "griffin.ibm.com".

AuthPermission

This class encapsulates the basic permissions required for JAAS. An AuthPermission contains a name (also referred to as a "target name") but no actions list; you either have the named permission or you don't. In addition to inherited methods (from the

```
Permission
```

class), an

```
AuthPermission
```

has two public constructors:

```
    public AuthPermission(String name);
    public AuthPermission(String name, String actions);
```

The first constructor creates a new AuthPermission with the specified name. The second constructor also creates a new AuthPermission object with the specified name, but has an additional *actions* argument which is currently unused and are null. This constructor exists solely for the

```
Policy
```

object to instantiate new Permission objects. For most code, the first constructor is appropriate.

The AuthPermission object is used to guard access to the Policy, Subject, LoginContext, and Configuration objects. Refer to the AuthPermission Javadoc for the list of valid names that are supported.

PrivateCredentialPermission

This class protects access to a Subject's private credentials and provides one public constructor:

```
    public PrivateCredentialPermission(String name, String actions);
```

Refer to the PrivateCredentialPermission Javadoc for more detailed information on this class.

Implementation

Note: Appendix A contains a sample **java.security** file that includes the static properties mentioned here.

Because there exists default values for JAAS providers and policy files, users need not statically (in the java.security file) nor dynamically (command line ***-D*** option) list their values in order to implement JAAS. Also, the default configuration and policy file providers may be replaced by a user-developed provider. Therefore this section is an attempt to explain the JAAS default providers and policy files as well as the properties that enable alternative providers.

Read the Default Policy File API and Default Configuration File API for more information than is summarized here.

Authentication Provider

The authentication provider, or configuration class, is statically set with

```
login.configuration.provider=[class]
```

in the *java.security* file. This provider creates the

```
Configuration
```

object.

For example:

```
  login.configuration.provider=com.foo.Config
```

If the Security property

```
login.configuration.provider
```

is not found in java.security, then JAAS will set it to the default value:

```
com.ibm.security.auth.login.ConfigFile
```

.

If a security manager is set before the

```
Configuration
```

is created, then an

```
AuthPermission("getLoginConfiguration")
```

will be required to be granted.

There isn't a way to dynamically set the configuration provider on the command line.

Authentication Configuration File

The authentication configuration files may be statically set in *java.security* with

```
login.config.url.n=[URL]
```

, where *n* is a consecutively number integer starting with 1. The format is identical to the format for Java security policy files (policy.url.n=[URL]).

If the Security property

```
policy.allowSystemProperty
```

is set to "true" in java.security, then users can dynamically set policy files on the command line utilizing the ***-D*** option with this property:

```
java.security.auth.login.config
```

. The value may be a path or URL. For example (on NT):

```
  ... -Djava.security.auth.login.config=c:\config_policy\login.config ...
  or
  ... -Djava.security.auth.login.config=file:c:/config_policy/login.config ...
```

Note: using double equal signs (==) on the command line allows a user to override all other policy files found.

If no configuration files can be found statically or dynamically, JAAS will try to load the configuration file from this default location:

```
${user.home}\.java.login.config
```

where *${user.home}* is a system dependent location.

Authorization Provider

The authorization provider, or JAAS Policy class, is statically set with

```
auth.policy.provider=[class]
```

in the *java.security* file. This provider creates the JAAS Subject-based

```
Policy
```

object.

For example:

```
auth.policy.provider=com.foo.Policy
```

If the Security property

```
auth.policy.provider
```

is not found in java.security, then JAAS will set it to the default value:

```
com.ibm.security.auth.PolicyFile
```

.

If a security manager is set before the

```
Configuration
```

is created, then an

```
AuthPermission("getPolicy")
```

will be required to be granted.

There isn't a way to dynamically set the authorization provider on the command line.

Authorization Policy File

The authorization policy files may be statically set in *java.security* with

```
auth.policy.url.n=[URL]
```

, where *n* is a consecutively number integer starting with 1. The format is identical to the format for Java security policy files (policy.url.n=[URL]).

If the Security property

```
policy.allowSystemProperty
```

is set to "true" in java.security, then users can dynamically set policy files on the command line utilizing the ***-D*** option with this property:

```
java.security.auth.policy
```

. The value may be a path or URL. For example (on NT):

```
... -Djava.security.auth.policy=c:\auth_policy\java.auth.policy ...
or
... -Djava.security.auth.policy=file:c:/auth_policy/java.auth.policy ...
```

Note: using double equal signs (==) on the command line allows a user to override all other policy files found.

There is not a default location to load an authorization policy from.

"Hello World", JAAS style!

Put on your dark sunglasses and favorite fedora hat, then grab an alto sax ... it's time to get **JAAS**-y! Yes, another "Hello World!" program. In this section, a program will be made available to test your JAAS installation.

Installation It is assumed that JAAS has been installed. For example, the JAAS JAR files have been copied to your Development Kit's extensions directory.

Retrieve Files Download theHelloWorld.tar to your test directory. Expand it using "jar xvf HelloWorld.tar".

Verify the contents of your test directory.

source files:

- HWLoginModule.java
- HWPrincipal.java
- HelloWorld.java

class files

- The source files have been precompiled for you into the classes directory.

policy files

- jaas.config
- java2.policy
- jaas.policy

Compile Source Files The three source files, ***HWLoginModule.java***, ***HWPrincipal.java*** and ***HelloWorld.java***, are already compiled and therefore do not need to be compiled.

If any of the source files are modified, then change to the test directory that they were saved to and enter:

```
javac -d .\classes *.java
```

The classpath needs the classes directory (.\classes) added to it in order to compile the classes.

Note:

```
HWLoginModule
```

and

```
HWPrincipal
```

are in the

```
com.ibm.security
```

package and will be created in the appropriate directory during compilation (>test_dir<\classes\com\ibm\security).

Examine Policy Files The configuration file, ***jaas.config***, contains one entry:

```
helloWorld {
   com.ibm.security.HWLoginModule required debug=true;
};
```

Only one

```
LoginModule
```

is supplied with the test case. When processing the ***HelloWorld*** application, experiment by changing the

```
LoginModuleControlFlag
```

(required, requisite, sufficient, optional) and deleting the debug flag. If more LoginModules are available for testing, then feel free to alter this configuration and experiment with multiple LoginModules.

```
HWLoginModule
```

will be discussed shortly.

The Java 2 policy file, ***java2.policy***, contains one permission block:

```
grant {
   permission javax.security.auth.AuthPermission "createLoginContext";
   permission javax.security.auth.AuthPermission "modifyPrincipals";
   permission javax.security.auth.AuthPermission "doAsPrivileged";
};
```

The three permissions are required because the ***HelloWorld*** application (1) creates a LoginContext object, (2) modifies the Principals of the the authenticated

```
Subject
```

and (3) calls the doAsPrivileged method of the

```
Subject
```

class.

The JAAS policy file, ***jaas.policy***, also contains one permission block:

```
grant Principal com.ibm.security.HWPrincipal "bob" {
   permission java.util.PropertyPermission "java.home", "read";
   permission java.util.PropertyPermission "user.home", "read";
   permission java.io.FilePermission "foo.txt", "read";
};
```

The three permissions are initially granted to an

```
HWPrincipal
```

named ***bob***. The actual Principal added to the authenticated

```
Subject
```

is the username used during the login process (more later).

Here's the action code from ***HelloWorld*** with the three system calls (the reason for the required permissions) in **bold**:

```
Subject.doAsPrivileged(lc.getSubject(), new PrivilegedAction() {
    public Object run() {
        System.out.println("\nYour java.home property: "
                           +System.getProperty("java.home"));

        System.out.println("\nYour user.home property: "
                           +System.getProperty("user.home"));
```

```
        File f = new File("foo.txt");
        System.out.print("\nfoo.txt does ");
        if (!f.exists()) System.out.print("not ");
        System.out.println("exist in your current directory");

        System.out.println("\nOh, by the way ...");

        try {
            Thread.currentThread().sleep(2000);
        } catch (Exception e) {
            // ignore
        }
        System.out.println("\n\nHello World!\n");
        return null;
    }
}, null);
```

When running the ***HelloWorld*** program, use various usernames and alter ***jaas.policy*** accordingly. There is no need to alter ***java2.policy***. Also, create a file called ***foo.txt*** in the test directory to test the last system call.

Examine Source Files The LoginModule,

```
HWLoginModule
```

, authenticates any user who enters the correct password (case sensitive): **Go JAAS**.

The ***HelloWorld*** application permits users three attempts to do so. When **Go JAAS** is correctly entered, an

```
HWPrincipal
```

with a name equal the the username is added to the authenticated

```
Subject
```

.

The Principal class,

```
HWPrincipal
```

, represents a Principal based on the username entered. It is this name that is important when granting permissions to authenticated Subjects.

The main application,

```
HelloWorld
```

, first creates a

```
LoginContext
```

based on a configuration entry with the name **helloWorld**. The configuration file has already been discussed. Callbacks are used to retrieve user input. Look at the

```
MyCallbackHandler
```

class located in the ***HelloWorld.java*** file to see this process.

```
    LoginContext lc = null;
    try {
        lc = new LoginContext("helloWorld", new MyCallbackHandler());
```

```
    } catch (LoginException le) {
        le.printStackTrace();
        System.exit(-1);
    }
```

The user enters a username/password (up to three times) and if **Go JAAS** is entered as the password, then the Subject is authenticated (

```
HWLoginModule
```

adds a

```
HWPrincipal
```

to the Subject).

As mentioned previously, work is then performed as the authenticated Subject.

Run HelloWorld Test

To run the ***HelloWorld*** program, first change to the test directory. The configuration and policy files will need to be loaded. See Implementation for the correct properties to set either in *java.security* or on the command line. The latter method will be discussed here.

The following command has been broken up into several lines for clarity. Enter as one continuous command.

```
java -Djava.security.manager=
     -Djava.security.auth.login.config=.\jaas.config
     -Djava.security.policy=.\java2.policy
     -Djava.security.auth.policy=.\jaas.policy
     HelloWorld
```

Note: the use of ".\filename" for the policy files is necessary because each user's test directory canonical path will vary. If desired, substitute "." with the path to the test directory. For example, if the test directory is "c:\test\hello", then the first file is changed to:

```
     -Djava.security.auth.login.config=c:\test\hello\jaas.config
```

If the policy files are not found, a

```
SecurityException
```

will be thrown. Otherwise, information concerning your ***java.home*** and ***user.home*** properties will be displayed. Also, the existence of a file called ***foo.txt*** in your test directory will be checked. Finally, the ubiquitous "Hello World" message is displayed.

Having Fun With HelloWorld

Rerun ***HelloWorld*** as many times as you like. It has already been suggested to vary the username/passwords entered, change the configuration file entries, change the policy file permissions, and to even add (stack) additional LoginModules to the ***helloWorld*** configuration entry. You could add codebase fields to the policy files too.

Finally, try running the program without a SecurityManager to see how it works if you run into problems.

Appendix A: JAAS Settings in the java.security Security Properties File

Below is a copy of the

```
java.security
```

file that appears in every Java 2 installation. This file appears in the

```
lib/security
```

(

```
lib\security
```

on Windows) directory of the Java 2 runtime. Thus, if the Java 2 runtime is installed in a directory called

```
jdk1.3
```

, the file is

- ```
  jdk1.3/lib/security/java.security
  ```

  (Unix)
- ```
  jdk1.3\lib\security\java.security
  ```

  (Windows)

JAAS adds four new properties to

```
java.security
```

:

- Authentication Properties
  - ```
    login.configuration.provider
    ```
  - ```
    login.policy.url.n
    ```
- Authorization Properties
  - ```
    auth.policy.provider
    ```
  - ```
    auth.policy.url.n
    ```

The new JAAS properties are located at the end of this file:

```
#
# This is the "master security properties file".
#
# In this file, various security properties are set for use by
# java.security classes. This is where users can statically register
# Cryptography Package Providers ("providers" for short). The term
# "provider" refers to a package or set of packages that supply a
# concrete implementation of a subset of the cryptography aspects of
# the Java Security API. A provider may, for example, implement one or
# more digital signature algorithms or message digest algorithms.
#
# Each provider must implement a subclass of the Provider class.
# To register a provider in this master security properties file,
# specify the Provider subclass name and priority in the format
#
#    security.provider.n=className
#
# This declares a provider, and specifies its preference
```

```
# order n. The preference order is the order in which providers are
# searched for requested algorithms (when no specific provider is
# requested). The order is 1-based; 1 is the most preferred, followed
# by 2, and so on.
#
# className must specify the subclass of the Provider class whose
# constructor sets the values of various properties that are required
# for the Java Security API to look up the algorithms or other
# facilities implemented by the provider.
#
# There must be at least one provider specification in java.security.
# There is a default provider that comes standard with the JDK. It
# is called the "SUN" provider, and its Provider subclass
# named Sun appears in the sun.security.provider package. Thus, the
# "SUN" provider is registered via the following:
#
#    security.provider.1=sun.security.provider.Sun
#
# (The number 1 is used for the default provider.)
#
# Note: Statically registered Provider subclasses are instantiated
# when the system is initialized. Providers can be dynamically
# registered instead by calls to either the addProvider or
# insertProviderAt method in the Security class.

#
# List of providers and their preference orders (see above):
#
security.provider.1=sun.security.provider.Sun

#
# Class to instantiate as the system Policy. This is the name of the class
# that will be used as the Policy object.
#
policy.provider=sun.security.provider.PolicyFile

# The default is to have a single system-wide policy file,
# and a policy file in the user's home directory.
policy.url.1=file:${java.home}/lib/security/java.policy
policy.url.2=file:${user.home}/.java.policy

# whether or not we expand properties in the policy file
# if this is set to false, properties (${...}) will not be expanded in policy
# files.
policy.expandProperties=true

# whether or not we allow an extra policy to be passed on the command line
# with -Djava.security.policy=somefile. Comment out this line to disable
# this feature.
policy.allowSystemProperty=true

# whether or not we look into the IdentityScope for trusted Identities
# when encountering a 1.1 signed JAR file. If the identity is found
# and is trusted, we grant it AllPermission.
policy.ignoreIdentityScope=false

#
# Default keystore type.
#
keystore.type=jks

#
# Class to instantiate as the system scope:
#
system.scope=sun.security.provider.IdentityDatabase

##############################################################################
#
# Java Authentication and Authorization Service (JAAS)
# properties and policy files:
#

# Class to instantiate as the system Configuration for authentication.
# This is the name of the class that will be used as the Authentication
# Configuration object.
#
login.configuration.provider=com.ibm.security.auth.login.ConfigFile

# The default is to have a system-wide login configuration file found in
# the user's home directory.  For multiple files, the format is similar to
# that of CodeSource-base policy files above, that is policy.url.n
login.config.url.1=file:${user.home}/.java.login.config
```

```
# Class to instantiate as the system Principal-based Authorization Policy.
# This is the name of the class that will be used as the Authorization
# Policy object.
#
auth.policy.provider=com.ibm.security.auth.PolicyFile

# The default is to have a system-wide Principal-based policy file found in
# the user's home directory.  For multiple files, the format is similar to
# that of CodeSource-base policy files above, that is policy.url.n and
# auth.policy.url.n
auth.policy.url.1=file:${user.home}/.java.auth.policy
```

Appendix B: Login Configuration Files

A login configuration file contains one or more

```
LoginContext
```

application names which have the following form:

```
Application {
    LoginModule Flag ModuleOptions;
    > more LoginModule entries <
    LoginModule Flag ModuleOptions;
};
```

Login configuration files are located using the

```
login.config.url.n
```

security property found in the

```
java.security
```

file. For more information about this property and the location of the

```
java.security
```

file, see Appendix A.

The *Flag* value controls the overall behavior as authentication proceeds down the stack. The following represents a description of the valid values for *Flag* and their respective semantics:

1. **Required** The

   ```
   LoginModule
   ```

   is required to succeed. If it succeeds or fails, authentication still continues to proceed down the

   ```
   LoginModule
   ```

   list.
2. **Requisite** The

   ```
   LoginModule
   ```

   is required to succeed. If it succeeds, authentication continues down the

   ```
   LoginModule
   ```

list. If it fails, control immediately returns to the application (authentication does not proceed down the

```
LoginModule
```

list).

3. **Sufficient** The

```
LoginModule
```

is not required to succeed. If it does succeed, control immediately returns to the application (authentication does not proceed down the

```
LoginModule
```

list). If it fails, authentication continues down the

```
LoginModule
```

list.

4. **Optional** The

```
LoginModule
```

is not required to succeed. If it succeeds or fails, authentication still continues to proceed down the

```
LoginModule
```

list.

The overall authentication succeeds only if all *Required* and *Requisite* LoginModules succeed. If a *Sufficient*

```
LoginModule
```

is configured and succeeds, then only the *Required* and *Requisite* LoginModules prior to that *Sufficient*

```
LoginModule
```

need to have succeeded for the overall authentication to succeed. If no *Required* or *Requisite* LoginModules are configured for an application, then at least one *Sufficient* or *Optional*

```
LoginModule
```

must succeed.

Sample Configuration File:

```
/*  Sample Configuration File  */

Login1 {
   com.ibm.security.auth.module.SampleLoginModule required debug=true;
};

Login2 {
   com.ibm.security.auth.module.SampleLoginModule required;
   com.ibm.security.auth.module.NTLoginModule sufficient;
   ibm.loginModules.SmartCard requisite debug=true;
```

```
    ibm.loginModules.Kerberos optional debug=true;
};
```

Note: the Flags are not case sensitive. *REQUISITE = requisite = Requisite.*

Login1 only has one LoginModule which is an instance of the class

```
com.ibm.security.auth.module.SampleLoginModule
```

. Therefore, a

```
LoginContext
```

associated with **Login1** will have a successful authentication if and only if its lone module successfully authenticates. The *Required* flag is trivial in this example; flag values have a relevant effect on authentication when two or more modules are present.

Login2 is easier to explain with a table.

Login2 Authentication Status									
Sample Login Module	required	pass	pass	pass	pass	fail	fail	fail	fail
NT Login Module	sufficient	pass	fail	fail	fail	pass	fail	fail	fail
Smart Card	requisite	*	pass	pass	fail	*	pass	pass	fail
Kerberos	optional	*	pass	fail	*	*	pass	fail	*
Overall Authentication		pass	pass	pass	fail	fail	fail	fail	fail

* = trivial value due to control returning to the application because a previous *REQUISITE* module failed or a previous *SUFFICIENT* module succeeded.

Appendix C: Authorization Policy File

In case there weren't enough examples of Principal-based JAAS Policy grant blocks above, here are some more.

```
// SAMPLE JAAS POLICY FILE:  java.auth.policy

// The following permissions are granted to Principal 'Pooh' and all codesource:

grant Principal com.ibm.security.Principal "Pooh" {
   permission javax.security.auth.AuthPermission "setPolicy";
   permission java.util.PropertyPermission "java.home", "read";
   permission java.util.PropertyPermission "user.home", "read";
   permission java.io.FilePermission "c:/foo/jaas.txt", "read";
};

// The following permissions are granted to Principal 'Pooh' AND 'Eyeore'
// and CodeSource signedBy "DrSecure":

grant signedBy "DrSecure"
      Principal com.ibm.security.Principal "Pooh",
      Principal com.ibm.security.Principal "Eyeore" {
   permission javax.security.auth.AuthPermission "modifyPublicCredentials";
   permission javax.security.auth.AuthPermission "modifyPrivateCredentials";
   permission java.net.SocketPermission "us.ibm.com", "connect,accept,resolve";
   permission java.net.SocketPermission "griffin.ibm.com", "accept";
};
```

```
// The following permissions are granted to Principal 'Pooh' AND 'Eyeore' AND
// 'Piglet' and CodeSource from the c:\jaas directory signed by "kent" and "bruce":

grant codeBase "file:c:/jaas/*",
      signedBy "kent, bruce",
      Principal com.ibm.security.Principal "Pooh",
      Principal com.ibm.security.Principal "Eyeore",
      Principal com.ibm.security.Principal "Piglet"  {
   permission javax.security.auth.AuthPermission "getSubject";
   permission java.security.SecurityPermission "printIdentity";
   permission java.net.SocketPermission "guapo.ibm.com", "accept";
};
```

IBM Java Generic Security Service (JGSS)

The Java Generic Security Service (JGSS) provides a generic interface for authentication and secure messaging. Under this interface you can plug a variety of security mechanisms based on secret-key, public-key, or other security technologies.

By abstracting the complexity and peculiarities of the underlying security mechanisms to a standardized interface, JGSS provides the following benefits to the development of secure networking applications:

- You can develop the application to a single abstract interface
- You can use the application with different security mechanisms without any changes

JGSS defines the Java bindings for the Generic Security Service Application Programming Interface (GSS-API), which is a cryptographic API that has been standardized by the Internet Engineering Task Force (IETF) and adopted by the X/Open Group.

The IBM implementation of JGSS is called IBM JGSS. IBM JGSS is an implementation of the GSS-API framework that uses Kerberos V5 as the default underlying security system. It also features a Java Authentication and Authorization Service (JAAS) login module for creating and using Kerberos credentials. In addition, you can have JGSS perform JAAS authorization checks when you use those credentials.

IBM JGSS includes a native IBM i JGSS provider, a Java JGSS provider, and Java versions of the Kerberos credential management tools (kinit, ktab, and klist).

Note: The native IBM i JGSS provider uses the native IBM i Network Authentication Services (NAS) library. When you use the native provider, you must use the native IBM i Kerberos utilities. For more information, see JGSS providers.

Related concepts

Changes to adopted authority in IBM i 7.2
Support for adopted user profile authority through Java programs is not supported in IBM i 7.2. This topic describes how to determine if your applications are using adopted authority and how to modify your applications to accommodate this change.

Java security model
You can download Java applets from any system; thus, security mechanisms exist within the Java virtual machine to protect against malicious applets. The Java runtime system verifies the bytecodes as the Java virtual machine loads them. This ensures that they are valid bytecodes and that the code does not violate any of the restrictions that the Java virtual machine places on Java applets.

Java Cryptography Extension
The Java Cryptography Extension (JCE) provides a framework and implementations for encryption, key generation, and key agreement, as well as Message Authentication Code (MAC) algorithms. Support for encryption includes symmetric, asymmetric, block, and stream ciphers. The software also supports secure streams and sealed objects. JCE supplements the Java 2 platform, which already includes interfaces and implementations of message digests and digital signatures.

Java Secure Socket Extension
Java Secure Socket Extension (JSSE) is like a framework that abstracts the underlying mechanisms of both Secure Sockets Layer (SSL) and Transport Layer Security (TLS). By abstracting the complexity and peculiarities of the underlying protocols, JSSE enables programmers to use secure encrypted communications while at the same time minimizing possible security vulnerabilities. Java Secure

Socket Extension (JSSE) uses both the SSL protocol and the TLS protocol to provide secure encrypted communications between your clients and servers.

Java Authentication and Authorization Service
The Java Authentication and Authorization Service (JAAS) is a standard extension to the Java 2 Platform, Standard Edition (J2SE). J2SE provides access controls that are based on where the code originated and who signed the code (code source-based access controls). It lacks, however, the ability to enforce additional access controls based on who runs the code. JAAS provides a framework that adds this support to the Java 2 security model.

Java Security from Oracle Corporation.

Internet Engineering Task Force (IETF) RFC 2743 Generic Security Services Application Programming Interface Version 2, Update 1

IETF RFC 2853 Generic Security Service API Version 2: Java Bindings

JGSS concepts

JGSS operations consist of four distinct stages, as standardized by the Generic Security Service Application Programming Interface (GSS-API).

The stages are as follows:

1. Gathering of credentials for principals.
2. Creating and establishing a security context between the communicating peer principals.
3. Exchanging secure messages between the peers.
4. Cleaning up and releasing resources.

Additionally, JGSS leverages the Java Cryptographic Architecture to offer seamless pluggability of different security mechanisms.

Use the following links to read high-level descriptions of these important JGSS concepts.

Related concepts

Configuring your server to use IBM JGSS
How you configure your server to use JGSS depends on which version of the Java Platform, Standard Edition (J2SE) that you run on your system.

Running IBM JGSS applications
The IBM Java Generic Security Service (JGSS) API 1.0 shields secure applications from the complexities and peculiarities of the different underlying security mechanisms. JGSS uses features provided by Java Authentication and Authorization Service (JAAS) and IBM Java Cryptography Extension (JCE).

Developing IBM JGSS applications
Use JGSS to develop secure applications. Learn about generating transport tokens, creating JGSS objects, establishing context, and more.

JGSS debugging
When you are trying to identify JGSS problems, use the JGSS debugging capability to produce helpful categorized messages.

Samples: IBM Java Generic Security Service (JGSS)
The IBM Java Generic Security Service (JGSS) sample files include client and server programs, configuration files, policy files, and Javadoc reference information. Use the sample programs to test and verify your JGSS setup.

Related reference

IBM JGSS Javadoc reference information

The Javadoc reference information for IBM JGSS includes classes and methods in the org.ietf.jgss api package and the Java versions of some Kerberos credential management tools.

JGSS principals and credentials

The identity under which an application engages in JGSS secure communication with a peer is called a principal. A principal may be a real user or an unattended service. A principal acquires security mechanism-specific credentials as proof of identity under that mechanism.

For example, when using the Kerberos mechanism, a principal's credential is in the form of a ticket-granting ticket (TGT) issued by a Kerberos key distribution center (KDC). In a multi-mechanism environment, a GSS-API credential can contain multiple credential elements, each element representing an underlying mechanism credential.

The GSS-API standard does not prescribe how a principal acquires credentials, and GSS-API implementations typically do not provide a means for credential acquisition. A principal obtains credentials before using GSS-API; GSS-API merely queries the security mechanism for credentials on behalf of the principal.

IBM JGSS includes Java versions of Kerberos credential management tools com.ibm.security.krb5.internal.tools Class Kinit, com.ibm.security.krb5.internal.tools Class Ktab, and com.ibm.security.krb5.internal.tools Class Klist. Additionally, IBM JGSS enhances the standard GSS-API by providing an optional Kerberos login interface that uses JAAS. The pure Java JGSS provider supports the optional login interface; the native IBM i provider does not.

Related concepts

"Obtaining Kerberos credentials and creating secret keys" on page 328
The GSS-API does not define a way to get credentials. For this reason, the IBM JGSS Kerberos mechanism requires that the user obtain Kerberos credentials. This topic instructs you on how to obtain Kerberos credentials and create secret keys, and about using JAAS to perform Kerberos logins and authorization checks and review a list of JAAS permissions required by the Java virtual machine (JVM).

"JGSS providers" on page 325
IBM JGSS includes a native IBM i JGSS provider and a pure Java JGSS provider. The provider that you choose to use depends on the needs of your application.

com.ibm.security.krb5.internal.tools Class Klist
This class can execute as a command-line tool to list entries in credential cache and key tab.

```
java.lang.Object
|
+--com.ibm.security.krb5.internal.tools.Klist
```

public class **Klist**
extends java.lang.Object

This class can execute as a command-line tool to list entries in credential cache and key tab.

Constructor summary

`Klist()`

Method summary

static void	`main(java.lang.String[] args)` The main program that can be invoked at command line.

Methods inherited from class java.lang.Object

`equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor detail

Klist

```
public Klist()
```

Method detail

main

```
public static void main(java.lang.String[] args)
```

The main program that can be invoked at command line.

Usage: java com.ibm.security.krb5.tools.Klist [[-c] [-f] [-e] [-a]] [-k [-t] [-K]] [name]

Available options for credential caches:

- **-f** shows credentials flags
- **-e** shows the encryption type
- **-a** displays the address list

Available options for keytabs:

- **-t** shows keytab entry timestamps
- **-K** shows keytab entry DES keys

com.ibm.security.krb5.internal.tools Class Kinit
Kinit tool for obtaining Kerberos v5 tickets.

```
java.lang.Object
|
+--com.ibm.security.krb5.internal.tools.Kinit
```

public class **Kinit**
extends java.lang.Object

Kinit tool for obtaining Kerberos v5 tickets.

Constructor summary

`Kinit(java.lang.String[] args)`
Constructs a new Kinit object.

Method summary

`static void`	`main(java.lang.String[] args)` The main method is used to accept user command line input for ticket request.

Methods inherited from class java.lang.Object

```
equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait
```

Constructor detail

Kinit

```
public Kinit(java.lang.String[] args)
      throws java.io.IOException,
             RealmException,
             KrbException
```

Constructs a new Kinit object.

Parameters:
`args` - array of ticket request options. Available options are: -f, -F, -p, -P, -c, -k, principal, password.

Throws:

java.io.IOException - if an I/O error occurs.
RealmException - if the Realm could not be instantiated.
KrbException - if error occurs during Kerberos operation.

Method detail

main

`public static void main(java.lang.String[] args)`

The main method is used to accept user command line input for ticket request.

Usage: java com.ibm.security.krb5.tools.Kinit [-f] [-F] [-p] [-P] [-k] [-c cache name] [principal] [password]

- **-f** forwardable
- **-F** not forwardable
- **-p** proxiable
- **-P** not proxiable
- **-c** cache name (i.e., FILE:d:\temp\mykrb5cc)
- **-k** use keytab
- **-t** keytab file name
- principal the principal name (i.e., qwedf qwedf@IBM.COM)
- password the principal's Kerberos password

Use java com.ibm.security.krb5.tools.Kinit -help to bring up help menu.

We currently only support file-based credentials cache. By default, a cache file named krb5cc_{user.name} would be generated at {user.home} directory to store the ticket obtained from KDC. For instance, on Windows NT, it could be c:\winnt\profiles\qwedf\krb5cc_qwedf, in which qwedf is the {user.name}, and c:\winnt\profile\qwedf is the {user.home}. {user.home} is obtained by Kerberos from Java system property "user.home". If in some case {user.home} is null (which barely happens), the cache file would be stored in the current directory that the program is running from. {user.name} is operating system's login username. It could be different from user's principal name. One user could have multiple principal names, but the primary principal of the credentials cache could only be one, which means one cache file could only store tickets for one specific user principal. If the user switches the principal name at the next Kinit, the cache file generated for the new ticket would overwrite the old cache file by default. To avoid overwriting, you need to specify a different directory or different cache file name when you request a new ticket.

Cache file location

There are several ways to define user specific cache file name and location, they are listed as follows in the order that Kerberos searches for:

1. **-c** option. Use java com.ibm.security.krb5.tools.Kinit -c FILE:<user specific directory and file name>. "FILE:" is the prefix to identify the credentials cache type. The default is file-based type.
2. Set Java system property "KRB5CCNAME" by using -DKRB5CCNAME=FILE:<user specific directory and file name> during runtime.
3. Set environment variable "KRB5CCNAME" at command prompt before the runtime. Different operating system has different way to set environment variables. For example, Windows uses set KRB5CCNAME=FILE:<user specific directory and file name>, while UNIX uses export KRB5CCNAME=FILE:<user specific directory and file name>. Note that Kerberos relies on system specific command to retrieve environment variable. The command used on UNIX is "/usr/bin/env".

KRB5CCNAME is case sensitive and is all upper case.

If KRB5CCNAME is not set as described above, a default cache file is used. The default cache is located in the following order:

1. /tmp/krb5cc_<uid> on Unix platforms, where <uid> is the user id of the user running the Kinit JVM
2. <user.home>/krb5cc_<user.name>, where <user.home> and <user.name> are the Java user.home and user.name properties, respectively
3. <user.home>/krb5cc (if <user.name> cannot be obtained from the JVM)

KDC Communication Timeout

Kinit communicates with the Key Distribution Center (KDC) to acquire a ticket-granting ticket, that is, the credential. This communication can be set to timeout if the KDC does not respond within a certain period. The timeout period can be set (in milliseconds) in the Kerberos configuration file in the libdefaults stanza (to be applicable to all KDCs) or in individual KDC stanzas. The default timeout value is 30 seconds.

com.ibm.security.krb5.internal.tools Class Ktab
This class can execute as a command-line tool to help the user manage entries in the key table. Available functions include list/add/update/delete service key(s).

```
java.lang.Object
|
+--com.ibm.security.krb5.internal.tools.Ktab
```

public class **Ktab**
extends java.lang.Object

This class can execute as a command-line tool to help the user manage entires in the key table. Available functions include list/add/update/delete service key(s).

Constructor summary

`Ktab()`

Method summary

`static void`	`main(java.lang.String[] args)` The main program that can be invoked at command line.

Methods inherited from class java.lang.Object

`equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor detail

Ktab

`public Ktab()`

Method detail

main

`public static void main(java.lang.String[] args)`

The main program that can be invoked at command line.

Usage: java com.ibm.security.krb5.tools.Ktab <options>

Available options to Ktab:

- **-l** list the keytab name and entries
- **-a** <*principal name*>(<*password*>) add an entry to the keytab
- **-d** <*principal name*> delete an entry from the keytab
- **-k** <*keytab name*> specify keytab name and path with prefix FILE:
- **-help** display instructions

JGSS context establishment

Having acquired security credentials, the two communicating peers establish a security context using their credentials. Although the peers establish a single joint context, each peer maintains its own local copy of the context. Context establishment involves the initiating peer authenticating itself to the accepting peer. The initiator optionally may request mutual authentication, in which case the acceptor authenticates itself to the initiator.

When context establishment is complete, the established context embodies state information (such as shared cryptographic keys) that enable subsequent secure message exchange between the two peers.

JGSS message protection and exchange

Following context establishment, the two peers are ready to engage in secure message exchanges. The originator of the message calls on its local GSS-API implementation to encode the message, which ensures message integrity and, optionally, message confidentiality. The application then transports the resulting token to the peer.

The local GSS-API implementation of the peer uses information from the established context in the following ways:

- Verifies the integrity of the message
- Deciphers the message, if the message was encrypted

Resource cleanup and release

In order to free up resources, a JGSS application deletes a context that is no longer needed. Although a JGSS application can access a deleted context, any attempt to use it for message exchange results in an exception.

Security mechanisms

The GSS-API consists of an abstract framework over one or more underlying security mechanisms. How the framework interacts with the underlying security mechanisms is implementation specific.

Such implementations exist in two general categories:

- At one extreme, a monolithic implementation tightly binds the framework to a single mechanism. This kind of implementation precludes the use of other mechanisms or even different implementations of the same mechanism.
- At the other end of the spectrum, a highly modular implementation offers ease of use and flexibility. This kind of implementation offers the ability to seamlessly and easily plug different security mechanisms and their implementations into the framework.

IBM JGSS falls into the latter category. As a modular implementation, IBM JGSS leverages the provider framework defined by the Java Cryptographic Architecture (JCA) and treats any underlying mechanism as a (JCA) provider. A JGSS provider supplies a concrete implementation of a JGSS security mechanism. An application can instantiate and use multiple mechanisms.

It is possible for a provider to support multiple mechanisms, and JGSS makes it easy to use different security mechanisms. However, the GSS-API does not provide a means for two communicating peers to choose a mechanism when multiple mechanisms are available. One way to choose a mechanism is to start with the Simple And Protected GSS-API Negotiating Mechanism (SPNEGO), a pseudo-mechanism that negotiates an actual mechanism between the two peers. IBM JGSS does not include a SPNEGO mechanism.

For more information about SPNEGO, see Internet Engineering Task Force (IETF) RFC 2478 The Simple and Protected GSS-API Negotiation Mechanism

Configuring your server to use IBM JGSS

How you configure your server to use JGSS depends on which version of the Java Platform, Standard Edition (J2SE) that you run on your system.

Related concepts

JGSS concepts
JGSS operations consist of four distinct stages, as standardized by the Generic Security Service Application Programming Interface (GSS-API).

Running IBM JGSS applications
The IBM Java Generic Security Service (JGSS) API 1.0 shields secure applications from the complexities and peculiarities of the different underlying security mechanisms. JGSS uses features provided by Java Authentication and Authorization Service (JAAS) and IBM Java Cryptography Extension (JCE).

Developing IBM JGSS applications
Use JGSS to develop secure applications. Learn about generating transport tokens, creating JGSS objects, establishing context, and more.

JGSS debugging
When you are trying to identify JGSS problems, use the JGSS debugging capability to produce helpful categorized messages.

Samples: IBM Java Generic Security Service (JGSS)
The IBM Java Generic Security Service (JGSS) sample files include client and server programs, configuration files, policy files, and Javadoc reference information. Use the sample programs to test and verify your JGSS setup.

Related reference

IBM JGSS Javadoc reference information
The Javadoc reference information for IBM JGSS includes classes and methods in the org.ietf.jgss api package and the Java versions of some Kerberos credential management tools.

Configuring your IBM i to use JGSS

When you use the Java 2 Software Development Kit (J2SDK), version 7 or above on your server, JGSS is already configured. The default configuration uses the pure Java JGSS provider.

Changing JGSS providers

You can configure JGSS to use the native IBM i JGSS provider instead of the pure Java JGSS provider. Then, after you configure JGSS to use the native provider, you can easily switch between the two providers. For more information, see "JGSS providers" on page 325.

Security managers

If you are running your JGSS application with a Java security manager enabled, see Using a security manager.

JGSS providers

IBM JGSS includes a native IBM i JGSS provider and a pure Java JGSS provider. The provider that you choose to use depends on the needs of your application.

The pure Java JGSS provider offers the following features:

- Ensures the greatest level of portability for your application.
- Works with the optional JAAS Kerberos login interface.
- Compatible with the Java Kerberos credential management tools.

The native IBM i JGSS provider offers the following features:

- Uses the native IBM i Kerberos libraries.
- Compatible with Qshell Kerberos credential management tools.
- JGSS applications run faster.

Note: Both JGSS providers adhere to the GSS-API specification and so are compatible with each other. In other words, an application that uses the pure Java JGSS provider can interoperate with an application that uses the native IBM i JGSS provider.

Changing the JGSS provider

You can easily change the JGSS provider by using one of the following options:

- Edit the security provider list in `${java.home}/lib/security/java.security`

  Note: ${java.home} denotes the path to the location of the version of Java that you are using on your server. For example, if you are using Java SE 7 32bit, ${java.home} is /QOpenSys/QIBM/ProdData/JavaVM/jdk70/32bit.
- Specify the provider name in your JGSS application by using either GSSManager.addProviderAtFront() or GSSManager.addProviderAtEnd(). For more information, see the GSSManager javadoc.

Using a security manager

If you are running your JGSS application with a Java security manager enabled, you need to ensure that your application and JGSS have the necessary permissions.

JVM permissions

In addition to the access control checks performed by JGSS, the Java virtual machine (JVM) performs authorization checks when accessing a variety of resources, including files, Java properties, packages, and sockets.

The following list identifies the permissions required when you use the JAAS features of JGSS or use JGSS with a security manager:

- javax.security.auth.AuthPermission "modifyPrincipals"
- javax.security.auth.AuthPermission "modifyPrivateCredentials"
- javax.security.auth.AuthPermission "getSubject"
- javax.security.auth.PrivateCredentialPermission "javax.security.auth.kerberos.KerberosKey javax.security.auth.kerberos.KerberosPrincipal \"*\"", "read"
- javax.security.auth.PrivateCredentialPermission "javax.security.auth.kerberos.KerberosTicket javax.security.auth.kerberos.KerberosPrincipal \"*\"", "read"
- java.util.PropertyPermission "com.ibm.security.jgss.debug", "read"
- java.util.PropertyPermission "DEBUG", "read"
- java.util.PropertyPermission "java.home", "read"
- java.util.PropertyPermission "java.security.krb5.conf", "read"
- java.util.PropertyPermission "java.security.krb5.kdc", "read"
- java.util.PropertyPermission "java.security.krb5.realm", "read"
- java.util.PropertyPermission "javax.security.auth.useSubjectCredsOnly", "read"
- java.util.PropertyPermission "user.dir", "read"
- java.util.PropertyPermission "user.home", "read"
- java.lang.RuntimePermission "accessClassInPackage.sun.security.action"
- java.security.SecurityPermission "putProviderProperty.IBMJGSSProvider"

Related concepts

JAAS permission checks

IBM JGSS performs runtime permission checks at the time the JAAS-enabled program uses credentials and accesses services. You can disable this optional JAAS feature by setting the Java property javax.security.auth.useSubjectCredsOnly to false. Moreover, JGSS performs permission checks only when the application runs with a security manager.

JAAS permission checks

IBM JGSS performs runtime permission checks at the time the JAAS-enabled program uses credentials and accesses services. You can disable this optional JAAS feature by setting the Java property javax.security.auth.useSubjectCredsOnly to false. Moreover, JGSS performs permission checks only when the application runs with a security manager.

JGSS performs permission checks against the Java policy that is in effect for the current access control context. JGSS performs the following specific permission checks:

- javax.security.auth.kerberos.DelegationPermission
- javax.security.auth.kerberos.ServicePermission

DelegationPermission check

The DelegationPermission allows the security policy to control the use of the ticket forwarding and proxying features of Kerberos. Using these features, a client can allow a service to act on behalf of the client.

DelegationPermission takes two arguments, in the following order:

1. The subordinate principal, which is the name of the service principal that acts on behalf of, and under the authority of, the client.
2. The name of the service that the client wants to allow the subordinate principal to use.

Example: Using the DelegationPermission check

In the following example, superSecureServer is the subordinate principal and krbtgt/REALM.IBM.COM@REALM.IBM.COM is the service that we want to allow superSecureServer to use on behalf of the client. In this case, the service is the ticket-granting ticket for the client, which means that superSecureServer can get a ticket for any service on behalf of the client.

```
permission javax.security.auth.kerberos.DelegationPermission
     "\"superSecureServer/host.ibm.com@REALM.IBM.COM\"
     \"krbtgt/REALM.IBM.COM@REALM.IBM.COM\"";
```

In the previous example, DelegationPermission grants the client permission to get a new ticket-granting ticket from the Key Distribution Center (KDC) that only superSecureServer can use. After the client has sent the new ticket-granting ticket to superSecureServer, superSecureServer has the ability to act on behalf of the client.

The following example enables the client to get a new ticket that allows superSecureServer to access only the ftp service on behalf of the client:

```
permission javax.security.auth.kerberos.DelegationPermission
     "\"superSecureServer/host.ibm.com@REALM.IBM.COM\"
     \"ftp/ftp.ibm.com@REALM.IBM.COM\"";
```

ServicePermission check

ServicePermission checks restrict the use of credentials for context initiation and acceptance. A context initiator must have permission to initiate a context. Likewise, a context acceptor must have permission to accept a context.

Example: Using the ServicePermission check

The following example allows the client side to initiate a context with the ftp service by granting permission to the client:

```
permission javax.security.auth.kerberos.ServicePermission
     "ftp/host.ibm.com@REALM.IBM.COM", "initiate";
```

The following example allows the server side to access and use the secret key for the ftp service by granting permission to the server:

```
permission javax.security.auth.kerberos.ServicePermission
     "ftp/host.ibm.com@REALM.IBM.COM", "accept";
```

Related concepts

JVM permissions
In addition to the access control checks performed by JGSS, the Java virtual machine (JVM) performs authorization checks when accessing a variety of resources, including files, Java properties, packages, and sockets.

Related information

Oracle Corporation documentation

Running IBM JGSS applications

The IBM Java Generic Security Service (JGSS) API 1.0 shields secure applications from the complexities and peculiarities of the different underlying security mechanisms. JGSS uses features provided by Java Authentication and Authorization Service (JAAS) and IBM Java Cryptography Extension (JCE).

JGSS features include:

- Identity authentication
- Message integrity and confidentiality
- Optional JAAS Kerberos login interface and authorization checks

Related concepts

JGSS concepts
JGSS operations consist of four distinct stages, as standardized by the Generic Security Service Application Programming Interface (GSS-API).

Configuring your server to use IBM JGSS
How you configure your server to use JGSS depends on which version of the Java Platform, Standard Edition (J2SE) that you run on your system.

Developing IBM JGSS applications
Use JGSS to develop secure applications. Learn about generating transport tokens, creating JGSS objects, establishing context, and more.

JGSS debugging
When you are trying to identify JGSS problems, use the JGSS debugging capability to produce helpful categorized messages.

Samples: IBM Java Generic Security Service (JGSS)
The IBM Java Generic Security Service (JGSS) sample files include client and server programs, configuration files, policy files, and Javadoc reference information. Use the sample programs to test and verify your JGSS setup.

Related reference

IBM JGSS Javadoc reference information
The Javadoc reference information for IBM JGSS includes classes and methods in the org.ietf.jgss api package and the Java versions of some Kerberos credential management tools.

Obtaining Kerberos credentials and creating secret keys

The GSS-API does not define a way to get credentials. For this reason, the IBM JGSS Kerberos mechanism requires that the user obtain Kerberos credentials. This topic instructs you on how to obtain

Kerberos credentials and create secret keys, and about using JAAS to perform Kerberos logins and authorization checks and review a list of JAAS permissions required by the Java virtual machine (JVM).

You can obtain credentials by using one of the following methods:

- Kinit and Ktab tools
- Optional JAAS Kerberos login interface

Related concepts

The kinit and ktab tools
Your choice of a JGSS provider determines which tools that you use to obtain Kerberos credentials and secret keys.

JAAS Kerberos login interface
IBM JGSS features a Java Authentication and Authorizaiton Service (JAAS) Kerberos login interface. You can disable this feature by setting the Java property javax.security.auth.useSubjectCredsOnly to false.

Configuration and policy files
JGSS and JAAS depend on several configuration and policy files. You need to edit these files to conform to your environment and application. If you do not use JAAS with JGSS, you can safely ignore the JAAS configuration and policy files.

The kinit and ktab tools

Your choice of a JGSS provider determines which tools that you use to obtain Kerberos credentials and secret keys.

Using the pure Java JGSS provider

If you are using the pure Java JGSS provider, use the IBM JGSS kinit and ktab tools to obtain credentials and secret keys. The kinit and ktab tools use command-line interfaces and provide options similar to those offered by other versions.

- You can obtain Kerberos credentials by using the kinit tool. This tool contacts the Kerberos Distribution Center (KDC) and obtains a ticket-granting ticket (TGT). The TGT allows you to access other Kerberos-enabled services, including those that use the GSS-API.
- A server can obtain a secret key by using the ktab tool. JGSS stores the secret key in the key table file on the server. See the ktab Java documentation for more information.

Alternatively, your application can use the JAAS Login interface to obtain TGTs and secret keys.

Using the native IBM i JGSS provider

If you are using the native IBM i JGSS provider, use the Qshell kinit and klist utilities.

Related concepts

Obtaining Kerberos credentials and creating secret keys
The GSS-API does not define a way to get credentials. For this reason, the IBM JGSS Kerberos mechanism requires that the user obtain Kerberos credentials. This topic instructs you on how to obtain Kerberos credentials and create secret keys, and about using JAAS to perform Kerberos logins and authorization checks and review a list of JAAS permissions required by the Java virtual machine (JVM).

JAAS Kerberos login interface
IBM JGSS features a Java Authentication and Authorizaiton Service (JAAS) Kerberos login interface. You can disable this feature by setting the Java property javax.security.auth.useSubjectCredsOnly to false.

Configuration and policy files
JGSS and JAAS depend on several configuration and policy files. You need to edit these files to conform to your environment and application. If you do not use JAAS with JGSS, you can safely ignore the JAAS configuration and policy files.

"JAAS Kerberos login interface" on page 330

IBM JGSS features a Java Authentication and Authorizaiton Service (JAAS) Kerberos login interface. You can disable this feature by setting the Java property javax.security.auth.useSubjectCredsOnly to false.

Related reference

"com.ibm.security.krb5.internal.tools Class Kinit" on page 321
Kinit tool for obtaining Kerberos v5 tickets.

"com.ibm.security.krb5.internal.tools Class Ktab" on page 323
This class can execute as a command-line tool to help the user manage entries in the key table. Available functions include list/add/update/delete service key(s).

JAAS Kerberos login interface

IBM JGSS features a Java Authentication and Authorizaiton Service (JAAS) Kerberos login interface. You can disable this feature by setting the Java property javax.security.auth.useSubjectCredsOnly to false.

Note: Although the pure Java JGSS provider can use the login interface, the native IBM i JGSS provider cannot.

For more information about JAAS, see Java Authentication and Authorization Service.

JAAS and JVM permissions

If you are using a security manager, you need to ensure that your application and JGSS have the necessary JVM and JAAS permissions. For more information, see Using a security manager.

JAAS configuration file options

The login interface requires a JAAS configuration file that specifies com.ibm.security.auth.module.Krb5LoginModule as the login module to be used. The following table lists the options that Krb5LoginModule supports. Note that the options are not case-sensitive.

Option name	Value	Default	Explanation
principal	<string>	None; prompted for.	Kerberos principal name
credsType	initiator | acceptor | both	initiator	The JGSS credential type
forwardable	true|false	false	Whether to acquire a forwardable ticket-granting ticket (TGT)
proxiable	true|false	false	Whether to acquire a proxiable TGT
useCcache	<URL>	Don't use ccache	Retrieve TGT from the specified credential cache
useKeytab	<URL>	Don't use key table	Retrieve secret key from the specified key table
useDefaultCcache	true|false	Don't use default ccache	Retrieve TGT from default credential cache
useDefaultKeytab	true|false	Don't use default key table	Retrieve secret key from the specified key table

For a simple example of using Krb5LoginModule, see the Sample JAAS login configuration file.

Option incompatabilities

Some Krb5LoginModule options, excluding principal name, are incompatible with each other, meaning that you cannot specify them together. The following table represents compatible and incompatible login module options.

Indicators in the table describe the relationship between the two associated options:

- X = Incompatible
- N/A = Inapplicable combination
- Blank = Compatible

Krb5LoginModule option	credsType initiator	credsType acceptor	credsType both	forward	proxy	use Ccache	use Keytab	useDefault Ccache	useDefault Keytab
credsType=initiator		N/A	N/A				X		X
credsType=acceptor	N/A		N/A	X	X	X		X	
credsType=both	N/A	N/A							
forwardable		X				X	X	X	X
proxiable		X				X	X	X	X
useCcache		X		X	X		X	X	X
useKeytab	X			X	X	X		X	X
useDefaultCcache		X		X	X	X	X		X
useDefaultKeytab	X			X	X	X	X	X	

Principal name option

You can specify a principal name in combination with any other option. If you do not specify a principal name, the Krb5LoginModule may prompt the user for a principal name. Whether or not Krb5LoginModule prompts the user depends on the other options that you specify.

Service principal name format

You must use one of the following formats to specify a service principal name:

- <service_name> (for example, superSecureServer)
- <service_name>@<host> (for example, superSecureServer@myhost)

In the latter format, <host> is the hostname of the machine on which the service resides. You can (but do not have to) use a fully qualified hostname.

Note: JAAS recognizes certain characters as delimiters. When you use any of the following characters in a JAAS string (such as a principal name), enclose the character in quotes:

```
_  (underscore)
:  (colon)
/  (forward slash)
\  (back slash)
```

Prompting for the principal name and password

The options that you specify in the JAAS configuration file determine whether the Krb5LoginModule login is noninteractive or interactive.

- A noninteractive login does not prompt for any information whatsoever
- An interactive login prompts for principal name, password, or both

Noninteractive logins

The login proceeds noninteractively when you specify the credential type as initiator (`credsType=initiator`) and you perform one of the following actions:

- Specify the useCcache option
- Set the useDefaultCcache option to true

The login also proceeds noninteractively when you specify the credential type as acceptor or both (`credsType=acceptor` or `credsType=both`) and you perform one of the following actions:

- Specify the useKeytab option
- Set the useDefaultKeytab option to true

Interactive logins

Other configurations result in the login module prompting for a principal name and password so that it may obtain a TGT from a Kerberos KDC. The login module prompts for only a password when you specify the principal option.

Interactive logins require that the application specify com.ibm.security.auth.callback.Krb5CallbackHandler as the callback handler when creating the login context. The callback handler is responsible for prompting for input.

Credential type option

When you require the credential type to be both initiator and acceptor (`credsType=both`), Krb5LoginModule obtains both a TGT and a secret key. The login module uses the TGT to initiate contexts and the secret key to accept contexts. The JAAS configuration file must contain sufficient information to enable the login module to acquire the two types of credentials.

For credential types acceptor and both, the login module assumes a service principal.

Related concepts

Obtaining Kerberos credentials and creating secret keys
The GSS-API does not define a way to get credentials. For this reason, the IBM JGSS Kerberos mechanism requires that the user obtain Kerberos credentials. This topic instructs you on how to obtain Kerberos credentials and create secret keys, and about using JAAS to perform Kerberos logins and authorization checks and review a list of JAAS permissions required by the Java virtual machine (JVM).

The kinit and ktab tools
Your choice of a JGSS provider determines which tools that you use to obtain Kerberos credentials and secret keys.

Configuration and policy files
JGSS and JAAS depend on several configuration and policy files. You need to edit these files to conform to your environment and application. If you do not use JAAS with JGSS, you can safely ignore the JAAS configuration and policy files.

Configuration and policy files

JGSS and JAAS depend on several configuration and policy files. You need to edit these files to conform to your environment and application. If you do not use JAAS with JGSS, you can safely ignore the JAAS configuration and policy files.

Note: In the following instructions, ${java.home} denotes the path to the location of the version of Java that you are using on your server. For example, if you are using Java SE 7 32bit, ${java.home} is /QOpenSys/QIBM/ProdData/JavaVM/jdk70/32bit. Remember to replace ${java.home}in the property settings with the actual path to the Java home directory.

Kerberos configuration file

IBM JGSS requires a Kerberos configuration file. The default name and location of the Kerberos configuration file depends on the operating system being used. JGSS uses the following order to search for the default configuration file:

1. The file referenced by the Java property java.security.krb5.conf
2. ${java.home}/lib/security/krb5.conf
3. c:\winnt\krb5.ini on Microsoft Windows platforms
4. /etc/krb5/krb5.conf on Solaris platforms
5. /etc/krb5.conf on other Unix platforms

JAAS configuration file

The use of the JAAS login feature requires a JAAS configuration file. You can specify the JAAS configuration file by setting one of the following properties:

- The Java system property java.security.auth.login.config
- The security property login.config.url.<integer> in the ${java.home}/lib/security/java.security file

For more information, see the Oracle Java Authentication and Authorization Service (JAAS) Web site.

JAAS policy file

When using the default policy implementation, JGSS grants JAAS permissions to entities by recording the permissions in a policy file. You can specify the JAAS policy file by setting one of the following properties:

- The Java system property java.security.policy
- The security property policy.url.<integer> in the ${java.home}/lib/security/java.security file

If you are using Java SE, version 7 or a subsequent release, specifying a separate policy file for JAAS is optional. The default policy provider in Java SE, version 7 and above supports the policy file entries that JAAS requires.

For more information, see the Oracle Java Authentication and Authorization Service (JAAS) Web site.

Java master security properties file

A Java virtual machine (JVM) uses many important security properties that you set by editing the Java master security properties file. This file, named java.security, usually resides in the ${java.home}/lib/security directory on your server.

The following list describes several relevant security properties for using JGSS. Use the descriptions as a guide for editing the java.security file.

Note: When applicable, the descriptions include appropriate values required to run the JGSS samples.

security.provider.<integer>: The JGSS provider that you want to use. Also statically registers cryptographic provider classes. IBM JGSS uses cryptographic and other security services provided by the IBM JCE Provider. Specify the sun.security.provider.Sun and com.ibm.crypto.provider.IBMJCE packages exactly like the following example:

```
security.provider.1=sun.security.provider.Sun
security.provider.2=com.ibm.crypto.provider.IBMJCE
```

policy.provider: System policy handler class. For example:

```
policy.provider=sun.security.provider.PolicyFile
```

policy.url.<integer>: URLs of policy files. To use the sample policy file, include an entry such as:

```
policy.url.1=file:/home/user/jgss/config/java.policy
```

login.configuration.provider: JAAS login configuration handler class, for example:

```
login.configuration.provider=com.ibm.security.auth.login.ConfigFile
```

auth.policy.provider: JAAS principal-based access control policy handler class, for example:

```
auth.policy.provider=com.ibm.security.auth.PolicyFile
```

login.config.url.<integer>: URLs for JAAS login configuration files. To use the sample configuration file, include an entry similar to:

```
login.config.url.1=file:/home/user/jgss/config/jaas.conf
```

auth.policy.url.<integer>: URLs for JAAS policy files. You can include both principal-based and CodeSource-based constructs in the JAAS policy file. To use the sample policy file, include an entry such as:

```
auth.policy.url.1=file:/home/user/jgss/config/jaas.policy
```

Credentials cache and server key table

A user principal keeps its Kerberos credentials in a credentials cache. A service principal keeps its secret key in a key table. At runtime, IBM JGSS locates these caches in the following ways:

User credentials cache

JGSS uses the following order to locate the user credentials cache:

1. The file referenced by the Java property KRB5CCNAME
2. The file referenced by the environment variable KRB5CCNAME
3. /tmp/krb5cc_<uid> on Unix systems
4. ${user.home}/krb5cc_${user.name}
5. ${user.home}/krb5cc (if ${user.name} cannot be obtained)

Server key table

JGSS uses the following order to locate the server key table file:

1. The value of the Java property KRB5_KTNAME
2. default_keytab_name entry in the libdefaults stanza of the Kerberos configuration file
3. ${user.home}/krb5_keytab

Related concepts

Obtaining Kerberos credentials and creating secret keys
The GSS-API does not define a way to get credentials. For this reason, the IBM JGSS Kerberos mechanism requires that the user obtain Kerberos credentials. This topic instructs you on how to obtain Kerberos credentials and create secret keys, and about using JAAS to perform Kerberos logins and authorization checks and review a list of JAAS permissions required by the Java virtual machine (JVM).

The kinit and ktab tools
Your choice of a JGSS provider determines which tools that you use to obtain Kerberos credentials and secret keys.

JAAS Kerberos login interface
IBM JGSS features a Java Authentication and Authorizaiton Service (JAAS) Kerberos login interface. You can disable this feature by setting the Java property javax.security.auth.useSubjectCredsOnly to false.

Developing IBM JGSS applications

Use JGSS to develop secure applications. Learn about generating transport tokens, creating JGSS objects, establishing context, and more.

To develop JGSS applications, you need to be familiar with the high-level GSS-API specification and the Java bindings specification. IBM JGSS 1.0 is primarily based on and conforms to these specifications. See the following links for more information.

- RFC 2743: Generic Security Service Application Programming Interface Version 2, Update 1
- RFC 2853: Generic Security Service API Version 2: Java Bindings

Related concepts

JGSS concepts
JGSS operations consist of four distinct stages, as standardized by the Generic Security Service Application Programming Interface (GSS-API).

Configuring your server to use IBM JGSS

How you configure your server to use JGSS depends on which version of the Java Platform, Standard Edition (J2SE) that you run on your system.

Running IBM JGSS applications
The IBM Java Generic Security Service (JGSS) API 1.0 shields secure applications from the complexities and peculiarities of the different underlying security mechanisms. JGSS uses features provided by Java Authentication and Authorization Service (JAAS) and IBM Java Cryptography Extension (JCE).

JGSS debugging
When you are trying to identify JGSS problems, use the JGSS debugging capability to produce helpful categorized messages.

Samples: IBM Java Generic Security Service (JGSS)
The IBM Java Generic Security Service (JGSS) sample files include client and server programs, configuration files, policy files, and Javadoc reference information. Use the sample programs to test and verify your JGSS setup.

Related reference

IBM JGSS Javadoc reference information
The Javadoc reference information for IBM JGSS includes classes and methods in the org.ietf.jgss api package and the Java versions of some Kerberos credential management tools.

IBM JGSS application programming steps

There are multiple steps required to develop a JGSS application, including using transport tokens, creating the necessary JGSS objects, establishing and deleting context, and using per-message services.

Operations in a JGSS application follow the Generic Security Service Application Programming Interface (GSS-API) operational model. For information about concepts important to JGSS operations, see JGSS concepts.

JGSS transport tokens

Some of the important JGSS operations generate tokens in the form of Java byte arrays. It is the responsibility of the application to forward the tokens from one JGSS peer to the other. JGSS does not constrain in any way the protocol that the application uses for transporting tokens. Applications may transport JGSS tokens together with other application (that is, non-JGSS) data. However, JGSS operations accept and use only JGSS-specific tokens.

Sequence of operations in a JGSS application

JGSS operations require certain programming constructs that you must use in the order listed below. Each of the steps applies to both the initiator and the acceptor.

Note: The information includes snippets of example code that illustrate using the high-level JGSS APIs and assume that your application imports the org.ietf.jgss package. Although many of the high-level APIs are overloaded, the snippets show only the most commonly used forms of those methods. Of course, use the API mehods that best suit your needs.

Related reference

Using JAAS with your JGSS application
The IBM JGSS includes an optional JAAS login facility that allows the application to use JAAS to obtain credentials. After the JAAS login facility saves principal credentials and secret keys in the subject object of a JAAS login context, JGSS can retrieve the credentials from that subject.

Creating a GSSManager
The GSSManager abstract class serves as a factory for creating JGSS objects.

GSSManager abstract class creates the following:

- GSSName
- GSSCredential
- GSSContext

GSSManager also has methods for determining the supported security mechanisms and name types and for specifying JGSS providers. Use the GSSManager getInstance static method to create an instance of the default GSSManager:

```
GSSManager manager = GSSManager.getInstance();
```

Creating a GSSName
GSSName represents the identity of a GSS-API principal. A GSSName may contain many representations of the principal, one for each supported underlying mechanism. A GSSName that contains only one name representation is called a Mechanism Name (MN).

GSSManager has several overloaded methods for creating a GSSName from a string or a contiguous array of bytes. The methods interpret the string or byte array according to a specified name type. Typically, you use the GSSName byte-array methods to reconstitute an exported name. The exported name is usually a mechanism name of type GSSName.NT_EXPORT_NAME. Some of these methods allow you to specify a security mechanism with which to create the name.

Example: Using GSSName

The following basic code snippet shows how to use GSSName.

Note: Specify Kerberos service name strings as either <service> or <service@host> where <service> is the name of the service and <host> is the hostname of the machine on which the service runs. You can (but do not have to) use a fully qualified hostname. When you omit the @<host> portion of the string, GSSName uses the local hostname.

```
// Create GSSName for user foo.
GSSName fooName = manager.createName("foo", GSSName.NT_USER_NAME);

// Create a Kerberos V5 mechanism name for user foo.
Oid krb5Mech = new Oid("1.2.840.113554.1.2.2");
GSSName fooName = manager.createName("foo", GSSName.NT_USER_NAME, krb5Mech);

// Create a mechanism name from a non-mechanism name by using the GSSName
// canonicalize method.
GSSName fooName = manager.createName("foo", GSSName.NT_USER_NAME);
GSSName fooKrb5Name = fooName.canonicalize(krb5Mech);
```

Creating a GSSCredential
A GSSCredential contains all the cryptographic information necessary to create a context on behalf of a principal and can contain credential information for multiple mechanisms.

GSSManager has three credential creation methods. Two of the methods take for parameters a GSSName, the lifetime of the credential, one or more mechanisms from which to get credentials, and the credential usage type. The third method takes only a usage type and uses the default values for other parameters. Specifying a null mechanism also uses the default mechanism. Specifying a null array of mechanisms causes the method to return credentials for the default set of mechanisms.

Note: Because IBM JGSS supports only the Kerberos V5 mechanism, that is the default mechanism.

Your application can create only one of the three credentials types (*initiate, accept*, or *initiate and accept*) at a time.

- A context initiator creates *initiate* credentials
- An acceptor creates *accept* credentials
- An acceptor that also behaves as an initiator creates *initiate and accept* credentials.

Examples: Obtaining credentials

The following example obtains the default credentials for an initiator:

```
GSSCredentials fooCreds = manager.createCredentials(GSSCredential.INITIATE)
```

The following example obtains Kerberos V5 credentials for the initiator *foo* that have the default validity period:

```
GSSCredential fooCreds = manager.createCredential(fooName, GSSCredential.DEFAULT_LIFETIME,
                                                  krb5Mech,GSSCredential.INITIATE);
```

The following example obtains an all-default acceptor credential:

```
GSSCredential serverCreds = manager.createCredential(null, GSSCredential.DEFAULT_LIFETIME,
                                                     (Oid)null, GSSCredential.ACCEPT);
```

Creating GSSContext
IBM JGSS supports two methods provided by GSSManager for creating a context. These methods are a method used by the context initiator and a method used by the acceptor.

Note: GSSManager provides a third method for creating a context that involves recreating previously exported contexts. However, because IBM JGSS Kerberos V5 mechanism does not support the use of exported contexts, IBM JGSS does not support this method.

Your application cannot use an initiator context for context acceptance, nor can it use an acceptor context for context initiation. Both supported methods for creating a context require a credential as input. When the value of the credential is null, JGSS uses the default credential.

Examples: Using GSSContext

The following example creates a context with which the principal (foo) can initiate a context with the peer (superSecureServer) on the host (securityCentral). The example specifies the peer as superSecureServer@securityCentral. The created context is valid for the default period:

```
GSSName serverName = manager.createName("superSecureServer@securityCentral",
                                        GSSName.NT_HOSTBASED_SERVICE, krb5Mech);
GSSContext fooContext = manager.createContext(serverName, krb5Mech, fooCreds,
                                             GSSCredential.DEFAULT_LIFETIME);
```

The following example creates a context for superSecureServer in order to accept contexts initiated by any peer:

```
GSSContext serverAcceptorContext = manager.createContext(serverCreds);
```

Note that your application can create and simultaneously use both types of contexts.

Requesting optional JGSS security services
Your application can request any of several optional security services. IBM JGSS supports several services.

The supported optional services are:

- Delegation
- Mutual authentication
- Replay detection
- Out-of-sequence detection
- Available per-message confidentiality
- Available per-message integrity

To request an optional service, your application must explicitly request it by using the appropriate request method on the context. Only an initiator can request these optional services. The initiator must make the request before context establishment begins.

For more information about optional services, see Optional Service Support in Internet Engineering Task Force (IETF) RFC 2743 Generic Security Services Application Programming Interface Version 2, Update 1 .

Example: Requesting optional services

In the following example, a context (fooContext) makes requests to enable mutual authentication and delegation services:

```
fooContext.requestMutualAuth(true);
fooContext.requestCredDeleg(true);
```

Establishing JGSS context
The two communicating peers must establish a security context over which they can use per-message services.

The initiator calls initSecContext() on its context, which returns a token to the initiator application. The initiator application transports the context token to the acceptor application. The acceptor calls acceptSecContext() on its context, specifying the context token received from the initiator. Depending on the underlying mechanism and the optional services that the initiator selected, acceptSecContext() might produce a token that the acceptor application has to forward to the initiator application. The initiator application then uses the received token to call initSecContext() one more time.

An application can make multiple calls to GSSContext.initSecContext() and GSSContext.acceptSecContext(). An application can also exchange multiple tokens with a peer during context establishment. Hence, the typical method of establishing context uses a loop to call GSSContext.initSecContext() or GSSContext.acceptSecContext() until the applications establish context.

Example: Establishing context

The following example illustrates the initiator (foo) side of context establishment:

```
byte array[] inToken = null; // The input token is null for the first call
int inTokenLen = 0;

do {
    byte[] outToken = fooContext.initSecContext(inToken, 0, inTokenLen);

    if (outToken != null) {
        send(outToken); // transport token to acceptor
    }

    if( !fooContext.isEstablished()) {
        inToken = receive(); // receive token from acceptor
          inTokenLen = inToken.length;
    }
} while (!fooContext.isEstablished());
```

The following example illustrates the acceptor side of context establishment:

```
// The acceptor code for establishing context may be the following:
do {
    byte[] inToken = receive(); // receive token from initiator
    byte[] outToken =
        serverAcceptorContext.acceptSecContext(inToken, 0, inToken.length);

    if (outToken != null) {
        send(outToken); // transport token to initiator
    }
} while (!serverAcceptorContext.isEstablished());
```

Using JGSS per-message services
After establishing a security context, two communicating peers can exchange secure messages over the established context.

Either peer can originate a secure message, regardless of whether it served as initiator or acceptor when establishing context. To make the message secure, IBM JGSS computes a cryptographic message integrity code (MIC) over the message. Optionally, IBM JGSS can have the Kerberos V5 mechanism encrypt the message to help ensure privacy.

Sending messages

IBM JGSS provides two sets of methods for securing messages: wrap() and getMIC().

Using wrap()

The wrap method performs the following actions:

- Computes an MIC
- Encrypts the message (optional)
- Returns a token

The calling application uses the MessageProp class in conjunction with GSSContext to specify whether to apply encryption to the message.

The returned token contains both the MIC and text of the message. The text of the message is either ciphertext (for an encrypted message) or the original plaintext (for messages that are not encrypted).

Using getMIC()

The getMIC method performs the following actions but cannot encrypt the message:

- Computes an MIC
- Returns a token

The returned token contains only the computed MIC and does not include the original message. So, in addition to transporting the MIC token to the peer, the peer must somehow be made aware of the original message so that it can verify the MIC.

Example: Using per-message services to send a message

The following example shows how one peer (foo) can wrap a message for delivery to another peer (superSecureServer):

```
byte[] message = "Ready to roll!".getBytes();
MessageProp mprop = new MessageProp(true); // foo wants the message encrypted
byte[] wrappedMessage =
    fooContext.wrap(message, 0, message.length, mprop);
send(wrappedMessage); // transfer the wrapped message to superSecureServer

// This is how superSecureServer may obtain a MIC for delivery to foo:
byte[] message = "You bet!".getBytes();
MessageProp mprop = null;  // superSecureServer is content with
                           // the default quality of protection

byte[] mic =
    serverAcceptorContext.getMIC(message, 0, message.length, mprop);
send(mic);
// send the MIC to foo. foo also needs the original message to verify the MIC
```

Receiving messages

The receiver of a wrapped message uses unwrap() to decode the message. The unwrap method performs the following actions:

- Verifies the cryptographic MIC embedded in the message
- Returns the original message over which the sender computed the MIC

If the sender encrypted the message, unwrap() decrypts the message before verifying the MIC and then returns the original plaintext message. The receiver of an MIC token uses verifyMIC() to verify the MIC over a given a message.

The peer applications use their own protocol to deliver JGSS context and message tokens to each other. Peer applications also need to define a protocol for determining whether the token is an MIC or a wrapped message. For example, part of such a protocol may be as simple (and rigid) as that used by Simple Authentication and Security Layer (SASL) applications. The SASL protocol specifies that the context acceptor is always the first peer to send a per-message (wrapped) token following context establishment.

For more information, see Simple Authentication and Security Layer (SASL).

Example: Using per-message services to receive a message

The following examples shows how a peer (superSecureServer) unwraps the wrapped token that it received from another peer (foo):

```
MessageProp mprop = new MessageProp(false);

byte[] plaintextFromFoo =
    serverAcceptorContext.unwrap(wrappedTokenFromFoo, 0,
                                 wrappedTokenFromFoo.length, mprop);

// superSecureServer can now examine mprop to determine the message properties
// (such as whether the message was encrypted) applied by foo.

// foo verifies the MIC received from superSecureServer:

MessageProp mprop = new MessageProp(false);
fooContext.verifyMIC(micFromFoo, 0, micFromFoo.length, messageFromFoo, 0,
                     messageFromFoo.length, mprop);

// foo can now examine mprop to determine the message properties applied by

// superSecureServer. In particular, it can assert that the message was not

// encrypted since getMIC should never encrypt a message.
```

Deleting JGSS context
A peer deletes a context when the context is no longer needed. In JGSS operations, each peer unilaterally decides when to delete a context and does not need to inform its peer.

JGSS does not define a delete context token. To delete a context, the peer calls the dispose method of the GSSContext object to free up any resources used by the context. A disposed GSSContext object may still be accessible, unless the application sets the object to null. However, any attempt to use a disposed (but still accessible) context throws an exception.

Using JAAS with your JGSS application

The IBM JGSS includes an optional JAAS login facility that allows the application to use JAAS to obtain credentials. After the JAAS login facility saves principal credentials and secret keys in the subject object of a JAAS login context, JGSS can retrieve the credentials from that subject.

The default behavior of JGSS is to retrieve credentials and secret keys from the subject. You can disable this feature by setting the Java property javax.security.auth.useSubjectCredsOnly to false.

Note: Although the pure Java JGSS provider can use the login interface, the native IBM i JGSS provider cannot.

For more information about JAAS features, see "Obtaining Kerberos credentials and creating secret keys" on page 328.

To use the JAAS login facility, your application must follow the JAAS programming model in the following ways:

- Create a JAAS login context
- Operate within the confines of a JAAS Subject.doAs construct

The following code snippet illustrates the concept of operating within the confines of a JAAS Subject.doAs construction:

```
static class JGSSOperations implements PrivilegedExceptionAction {
    public JGSSOperations() {}
    public Object run () throws GSSException {
        // JGSS application code goes/runs here
    }
}

public static void main(String args[]) throws Exception {
    // Create a login context that will use the Kerberos
    // callback handler
    // com.ibm.security.auth.callback.Krb5CallbackHandler

    // There must be a JAAS configuration for "JGSSClient"
    LoginContext loginContext =
        new LoginContext("JGSSClient", new Krb5CallabackHandler());
```

```
            loginContext.login();

        // Run the entire JGSS application in JAAS privileged mode
        Subject.doAsPrivileged(loginContext.getSubject(),
                               new JGSSOperations(), null);
    }
```

Related concepts

IBM JGSS application programming steps
There are multiple steps required to develop a JGSS application, including using transport tokens, creating the necessary JGSS objects, establishing and deleting context, and using per-message services.

JGSS debugging

When you are trying to identify JGSS problems, use the JGSS debugging capability to produce helpful categorized messages.

You can turn on one or more categories by setting the appropriate values for the Java property com.ibm.security.jgss.debug. Activate multiple categories by using a comma to separate the category names.

Debugging categories include the following:

Category	Description
help	List debug catgories
all	Turn on debugging for all categories
off	Turn off debugging completely
app	Application debugging (default)
ctx	Context operations debugging
cred	Credentials (including name) operations
marsh	Marshaling of tokens
mic	MIC operations
prov	Provider operations
qop	QOP operations
unmarsh	Unmarshaling of tokens
unwrap	Unwrap operations
wrap	Wrap operations

JGSS Debug class

To debug your JGSS application programmatically, use the debug class in the IBM JGSS framework. Your application can use the debug class to turn on and off debug categories and display debugging information for the active categories.

The default debugging constructor reads the Java property com.ibm.security.jgss.debug to determine which categories to activate (turn on).

Example: Debugging for the application category

The following example shows how to request debug information for the application category:

```
    import com.ibm.security.jgss.debug;

    Debug debug = new Debug(); // Gets categories from Java property

    // Lots of work required to set up someBuffer. Test that the
```

```
    // category is on before setting up for debugging.

    if (debug.on(Debug.OPTS_CAT_APPLICATION)) {
        // Fill someBuffer with data.
        debug.out(Debug.OPTS_CAT_APPLICATION, someBuffer);
        // someBuffer may be a byte array or a String.
```

Related concepts

JGSS concepts
JGSS operations consist of four distinct stages, as standardized by the Generic Security Service Application Programming Interface (GSS-API).

Configuring your server to use IBM JGSS
How you configure your server to use JGSS depends on which version of the Java Platform, Standard Edition (J2SE) that you run on your system.

Running IBM JGSS applications
The IBM Java Generic Security Service (JGSS) API 1.0 shields secure applications from the complexities and peculiarities of the different underlying security mechanisms. JGSS uses features provided by Java Authentication and Authorization Service (JAAS) and IBM Java Cryptography Extension (JCE).

Developing IBM JGSS applications
Use JGSS to develop secure applications. Learn about generating transport tokens, creating JGSS objects, establishing context, and more.

Samples: IBM Java Generic Security Service (JGSS)
The IBM Java Generic Security Service (JGSS) sample files include client and server programs, configuration files, policy files, and Javadoc reference information. Use the sample programs to test and verify your JGSS setup.

Related reference

IBM JGSS Javadoc reference information
The Javadoc reference information for IBM JGSS includes classes and methods in the org.ietf.jgss api package and the Java versions of some Kerberos credential management tools.

Samples: IBM Java Generic Security Service (JGSS)

The IBM Java Generic Security Service (JGSS) sample files include client and server programs, configuration files, policy files, and Javadoc reference information. Use the sample programs to test and verify your JGSS setup.

You can view HTML versions of the samples or download the Javadoc information and source code for the sample programs. Downloading the samples enables you to view the Javadoc reference information, examine the code, edit the configuration and policy files, and compile and run the sample programs.

Description of the sample programs

The JGSS samples include four programs:

- non-JAAS server
- non-JAAS client
- JAAS-enabled server
- JAAS-enabled client

The JAAS-enabled versions are fully interoperable with their non-JAAS counterparts. So, you can run a JAAS-enabled client against a non-JAAS server and you can run a non-JAAS client against a JAAS-enabled server.

Note: When you run a sample, you can specify one or more optional Java properties, including the names of the configuration and policy files, JGSS debug options, and the security manager. You can also turn on and turn off the JAAS features.

You can run the samples in either a one-server or a two-server configuration. The one server configuration consists of a client communicating with a primary server. The two-server configuration consists of a primary and a secondary server, where the primary server acts as an initiator, or client, to the secondary server.

When using a two-server configuration, the client first initiates a context and exchanges secure messages with the primary server. Next, the client delegates its credentials to the primary server. Then, on behalf of the client, the primary server uses these credentials to initiate a context and exchange secure messages with the secondary server. You can also use a two-server configuration in which the primary server acts as a client on its own behalf. In this case, the primary server uses its own credentials to initiate a context and exchange messages with the secondary server.

You can simultaneously run any number of clients against the primary server. Although you can run a client directly against the secondary server, the secondary server cannot use delegated credentials or run as an initiator using its own credentials.

Related concepts

JGSS concepts
JGSS operations consist of four distinct stages, as standardized by the Generic Security Service Application Programming Interface (GSS-API).

Configuring your server to use IBM JGSS
How you configure your server to use JGSS depends on which version of the Java Platform, Standard Edition (J2SE) that you run on your system.

Running IBM JGSS applications
The IBM Java Generic Security Service (JGSS) API 1.0 shields secure applications from the complexities and peculiarities of the different underlying security mechanisms. JGSS uses features provided by Java Authentication and Authorization Service (JAAS) and IBM Java Cryptography Extension (JCE).

Developing IBM JGSS applications
Use JGSS to develop secure applications. Learn about generating transport tokens, creating JGSS objects, establishing context, and more.

JGSS debugging
When you are trying to identify JGSS problems, use the JGSS debugging capability to produce helpful categorized messages.

Related reference

IBM JGSS Javadoc reference information
The Javadoc reference information for IBM JGSS includes classes and methods in the org.ietf.jgss api package and the Java versions of some Kerberos credential management tools.

Viewing the IBM JGSS samples

The IBM Java Generic Security Service (JGSS) sample files include client and server programs, configuration files, policy files, and Javadoc reference information. Use the following links to view HTML versions of the JGSS samples.

Related concepts

"Samples: IBM Java Generic Security Service (JGSS)" on page 342
The IBM Java Generic Security Service (JGSS) sample files include client and server programs, configuration files, policy files, and Javadoc reference information. Use the sample programs to test and verify your JGSS setup.

Related tasks

Samples: Downloading and viewing Javadoc information for the IBM JGSS samples
To download and view the documentation for the IBM JGSS sample programs, complete the following steps.

Samples: Downloading and running the sample JGSS programs
This topic contains instructions for downloading and running the sample Javadoc information.

"Samples: Downloading and running the sample JGSS programs" on page 347

This topic contains instructions for downloading and running the sample Javadoc information.

Related reference

“Sample: IBM JGSS non-JAAS client program” on page 438
Use this JGSS sample client in conjunction with the JGSS sample server.

“Sample: IBM JGSS non-JAAS server program” on page 444
This example contains a JGSS sample server that is to be used in conjunction with a JGSS sample client.

“Sample: IBM JGSS JAAS-enabled client program” on page 454
This sample program performs a JAAS login and operates within the JAAS login context. It does not set the variable javax.security.auth.useSubjectCredsOnly, leaving the variable to default to "true" so that Java GSS acquires credentials from the JAAS Subject associated with login context created by the client.

“Sample: IBM JGSS JAAS-enabled server program” on page 455
This sample program performs a JAAS login and operates within the JAAS login context.

Sample: Kerberos configuration file

This topic contains the Kerberos configuration file for running the JGSS sample applications.

For more information about using the sample configuration file, see Downloading and running the IBM JGSS samples.

Note: By using the code examples, you agree to the terms of the “Code license and disclaimer information” on page 489.

```
# ---------------------------------------------------------------------------------
# Kerberos configuration file for running the JGSS sample applications.
# Modify the entries to suit your environment.
#----------------------------------------------------------------------------------

[libdefaults]
    default_keytab_name   = /QIBM/UserData/OS400/NetworkAuthentication/keytab/krb5.keytab
    default_realm         = REALM.IBM.COM
    default_tkt_enctypes  = des-cbc-crc
    default_tgs_enctypes  = des-cbc-crc
    default_checksum      = rsa-md5
    kdc_timesync          = 0
    kdc_default_options   = 0x40000010
    clockskew             = 300
    check_delegate        = 1
    ccache_type           = 3
    kdc_timeout           = 60000


[realms]
    REALM.IBM.COM = {
            kdc = kdc.ibm.com:88
    }

[domain_realm]
    .ibm.com = REALM.IBM.COM
```

Sample: JAAS login configuration file

This topic contains the JAAS login configuration for the JGSS samples.

For more information about using the sample configuration file, see Downloading and running the IBM JGSS samples.

Note: By using the code examples, you agree to the terms of the “Code license and disclaimer information” on page 489.

```
/**
 * ---------------------------------------------------------------------------------
 * JAAS Login Configuration for the JGSS samples.
 * ---------------------------------------------------------------------------------
 *
 * Code example disclaimer
 * IBM grants you a nonexclusive copyright license to use all programming code
 * examples from which you can generate similar function tailored to your own
 * specific needs.
 * All 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 non-infringement, merchantability and fitness
 * for a particular purpose are expressly disclaimed.
 *
 *
 * Supported options:
 *     principal=<string>
 *     credsType=initiator|acceptor|both (default=initiator)
 *     forwardable=true|false (default=false)
 *     proxiable=true|false (default=false)
 *     useCcache=<URL_string>
 *     useKeytab=<URL_string>
 *     useDefaultCcache=true|false (default=false)
 *     useDefaultKeytab=true|false (default=false)
 *     noAddress=true|false (default=false)
 *
 * Default realm (which is obtained from the Kerberos config file) is
 * used if the principal specified does not include a realm component.
 */

JAASClient {
    com.ibm.security.auth.module.Krb5LoginModule required
        useDefaultCcache=true;
};

JAASServer {
    com.ibm.security.auth.module.Krb5LoginModule required
        credsType=acceptor useDefaultKeytab=true
        principal=gss_service/myhost.ibm.com@REALM.IBM.COM;
};
```

Sample: JAAS policy file

This topic contains the JAAS policy file for running the JGSS sample applications.

For more information about using the sample policy file, see Downloading and running the IBM JGSS samples.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
// ---------------------------------------------------------------------------
// JAAS policy file for running the JGSS sample applications.
// Modify these permissions to suit your environment.
// Not recommended for use for any purpose other than that stated above.
// In particular, do not use this policy file or its
// contents to protect resources in a production environment.
//
// Code example disclaimer
// IBM grants you a nonexclusive copyright license to use all programming code
// examples from which you can generate similar function tailored to your own
// specific needs.
// All 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 non-infringement, merchantability and fitness
// for a particular purpose are expressly disclaimed.
//
// ---------------------------------------------------------------------------

//----------------------------------------------------------------------------
// Permissions for client only
//----------------------------------------------------------------------------

grant CodeBase "file:ibmjgsssample.jar",
    Principal javax.security.auth.kerberos.KerberosPrincipal
        "bob@REALM.IBM.COM"
{
    // foo needs to be able to initiate a context with the server
    permission javax.security.auth.kerberos.ServicePermission
        "gss_service/myhost.ibm.com@REALM.IBM.COM", "initiate";
```

```
    // So that foo can delegate his creds to the server
    permission javax.security.auth.kerberos.DelegationPermission
        "\"gss_service/myhost.ibm.com@REALM.IBM.COM\" \"krbtgt/REALM.IBM.COM@REALM.IBM.COM\"";
};

//-----------------------------------------------------------------------------
// Permissions for the server only
//-----------------------------------------------------------------------------

grant CodeBase "file:ibmjgsssample.jar",
    Principal javax.security.auth.kerberos.KerberosPrincipal
        "gss_service/myhost.ibm.com@REALM.IBM.COM"
{
    // Permission for the server to accept network connections on its host
    permission java.net.SocketPermission "myhost.ibm.com", "accept";

    // Permission for the server to accept JGSS contexts
    permission javax.security.auth.kerberos.ServicePermission
        "gss_service/myhost.ibm.com@REALM.IBM.COM", "accept";

    // The server acts as a client when communicating with the secondary (backup) server
    // This permission allows the server to initiate a context with the secondary server
    permission javax.security.auth.kerberos.ServicePermission
        "gss_service2/myhost.ibm.com@REALM.IBM.COM", "initiate";
};

//-----------------------------------------------------------------------------
// Permissions for the secondary server
//-----------------------------------------------------------------------------

grant CodeBase "file:ibmjgsssample.jar",
    Principal javax.security.auth.kerberos.KerberosPrincipal
        "gss_service2/myhost.ibm.com@REALM.IBM.COM"
{
    // Permission for the secondary server to accept network connections on its host
    permission java.net.SocketPermission "myhost.ibm.com", "accept";

    // Permission for the server to accept JGSS contexts
    permission javax.security.auth.kerberos.ServicePermission
        "gss_service2/myhost.ibm.com@REALM.IBM.COM", "accept";
};
```

Sample: Java policy file

This topic contains the Java policy file for running the JGSS sample applications on the server.

For more information about using the sample policy file, see Downloading and running the IBM JGSS samples.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
// -----------------------------------------------------------------
// Java policy file for running the JGSS sample applications on
// the server.
// Modify these permissions to suit your environment.
// Not recommended for use for any purpose other than that stated above.
// In particular, do not use this policy file or its
// contents to protect resources in a production environment.
//
// Code example disclaimer
// IBM grants you a nonexclusive copyright license to use all programming code
// examples from which you can generate similar function tailored to your own
// specific needs.
// All 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 non-infringement, merchantability and fitness
// for a particular purpose are expressly disclaimed.
//
//-----------------------------------------------------------------------

grant CodeBase "file:ibmjgsssample.jar" {

    permission javax.security.auth.AuthPermission "createLoginContext.JAASClient";
    permission javax.security.auth.AuthPermission "createLoginContext.JAASServer";
```

```
    permission javax.security.auth.AuthPermission "doAsPrivileged";

    // Permission to request a ticket from the KDC
    permission javax.security.auth.kerberos.ServicePermission
        "krbtgt/REALM.IBM.COM@REALM.IBM.COM", "initiate";

    // Permission to access sun.security.action classes
    permission java.lang.RuntimePermission "accessClassInPackage.sun.security.action";

    // A whole bunch of Java properties are accessed
    permission java.util.PropertyPermission "java.net.preferIPv4Stack", "read";
    permission java.util.PropertyPermission "java.version", "read";
    permission java.util.PropertyPermission "java.home", "read";
    permission java.util.PropertyPermission "user.home", "read";
    permission java.util.PropertyPermission "DEBUG", "read";
    permission java.util.PropertyPermission "com.ibm.security.jgss.debug", "read";
    permission java.util.PropertyPermission "java.security.krb5.kdc", "read";
    permission java.util.PropertyPermission "java.security.krb5.realm", "read";
    permission java.util.PropertyPermission "java.security.krb5.conf", "read";
    permission java.util.PropertyPermission "javax.security.auth.useSubjectCredsOnly",
    "read,write";

    // Permission to communicate with the Kerberos KDC host
    permission java.net.SocketPermission "kdc.ibm.com", "connect,accept,resolve";

    // I run the samples from my localhost
    permission java.net.SocketPermission "myhost.ibm.com", "accept,connect,resolve";
    permission java.net.SocketPermission "localhost", "listen,accept,connect,resolve";

    // Access to some possible Kerberos config locations
    // Modify the file paths as applicable to your environment
    permission java.io.FilePermission "${user.home}/krb5.ini", "read";
    permission java.io.FilePermission "${java.home}/lib/security/krb5.conf", "read";

    // Access to the Kerberos key table so we can get our server key.
    permission java.io.FilePermission
    "/QIBM/UserData/OS400/NetworkAuthentication/keytab/krb5.keytab", "read";

    // Access to the user's Kerberos credentials cache.
    permission java.io.FilePermission "${user.home}/krb5cc_${user.name}",
    "read";
};
```

Samples: Downloading and viewing Javadoc information for the IBM JGSS samples

To download and view the documentation for the IBM JGSS sample programs, complete the following steps.

1. Choose an existing directory (or create a new one) where you want to store the Javadoc information.
2. Download the Javadoc information (jgsssampledoc.zip) into the directory.
3. Extract the files from jgsssampledoc.zip into the directory.
4. Use your browser to access the index.htm file.

Related concepts

Viewing the IBM JGSS samples
The IBM Java Generic Security Service (JGSS) sample files include client and server programs, configuration files, policy files, and Javadoc reference information. Use the following links to view HTML versions of the JGSS samples.

Related tasks

Samples: Downloading and running the sample JGSS programs
This topic contains instructions for downloading and running the sample Javadoc information.

Samples: Downloading and running the sample JGSS programs

This topic contains instructions for downloading and running the sample Javadoc information.

Before modifying or running the samples, read the description of the sample programs in the Samples: IBM Java Generic Security Service (JGSS) article.

To run the sample programs, perform the following tasks:

1. Download the sample files to your server

2. Prepare to run the sample files
3. Run the sample programs

Related concepts

Viewing the IBM JGSS samples
The IBM Java Generic Security Service (JGSS) sample files include client and server programs, configuration files, policy files, and Javadoc reference information. Use the following links to view HTML versions of the JGSS samples.

"Samples: IBM Java Generic Security Service (JGSS)" on page 342
The IBM Java Generic Security Service (JGSS) sample files include client and server programs, configuration files, policy files, and Javadoc reference information. Use the sample programs to test and verify your JGSS setup.

Related tasks

Samples: Downloading and viewing Javadoc information for the IBM JGSS samples
To download and view the documentation for the IBM JGSS sample programs, complete the following steps.

"Samples: Downloading and running the sample JGSS programs" on page 347
This topic contains instructions for downloading and running the sample Javadoc information.

Samples: Downloading the IBM JGSS samples
This topic contains instructions for downloading the sample JGSS Javadoc information to your system.

Before modifying or running the samples, read the description of the sample programs.

To download the sample files and store them on your server, complete the following steps:

1. On your server, choose an existing directory (or create a new one) where you want to store the sample programs, configuration files, and policy files.
2. Download the sample programs (ibmjgsssample.zip).
3. Extract the files from ibmjgsssample.zip into the directory on the server.

Extracting the contents of ibmjgsssample.jar performs the following actions:

- Places ibmgjsssample.jar, which contains the sample .class files, into the selected directory
- Creates a subdirectory (named config) that contains the configuration and policy files
- Creates a subdirectory (named src) that contains the sample .java source files

Related information

You may want to read about related tasks or look at an example:

- "Samples: Preparing to run the JGSS sample programs" on page 348
- "Samples: Running the JGSS sample programs" on page 349
- "Example: Running the non-JAAS sample" on page 349

Samples: Preparing to run the JGSS sample programs
After you download the source code, you need to perform preparation before running the sample programs.

Before modifying or running the samples, see "Samples: IBM Java Generic Security Service (JGSS)" on page 342.

After you download the source code, you need to perform the following tasks before you can run the sample programs:

- Edit the configuration and policy files to suit your environment. For more information, refer to the comments in each configuration and policy file.
- Ensure that the java.security file contains the correct settings for your IBM i server. For more information, see "Configuration and policy files" on page 332.

- Place the modified Kerberos configuration file (krb5.conf) into the directory on your server that is appropriate for the version of the J2SDK that you are using:
  - For Version 1.7 of the J2SE: /QIBM/ProdData/Java400/jdk17/lib/security
  - For Version 1.8 of the J2SE: /QIBM/ProdData/Java400/jdk18/lib/security

Related tasks

"Samples: Downloading the IBM JGSS samples" on page 348
This topic contains instructions for downloading the sample JGSS Javadoc information to your system.

"Samples: Running the JGSS sample programs" on page 349
After you download and modify the source code, you can run one of the samples.

Related reference

"Example: Running the non-JAAS sample" on page 349
To run a sample, you need to download and modify the sample source code.

Samples: Running the JGSS sample programs
After you download and modify the source code, you can run one of the samples.

Before modifying or running the samples, read the description of the sample programs.

To run a sample, you must start the server program first. The server program must be running and ready to receive connections before you start the client program. The server is ready to receive connections when you see `listening on port <server_port>`. Make sure to remember or write down the <server_port >, which is the port number that you need to specify when you start the client.

Use the following command to start a sample program:

```
    java [-Dproperty1=value1 ... -DpropertyN=valueN] com.ibm.security.jgss.test.<program>
[options]
```

where

- [-DpropertyN=valueN] is one or more optional Java properties, including the names of the configuration and policy files, JGSS debug options, and the security manager. For more information, see the following example and Running JGSS applications.
- <program> is a required parameter that specifies the sample program that you want to run (either `Client`, `Server`, `JAASClient`, or `JAASServer`).
- [options] is an optional parameter for the sample program that you want to run. To display a list of supported options, use the following command:

  ```
      java com.ibm.security.jgss.test.<program> -?
  ```

Note: Turn off the JAAS features in a JGSS-enabled sample by setting the Java property javax.security.auth.useSubjectCredsOnly to false. Of course, the default value of the JAAS-enabled samples is to turn on JAAS, meaning that the property value is true. The non-JAAS client and server programs set the property to false, unless you have explicitly set the property value.

Related information

You may want to read about related tasks or look at an example:

- "Samples: Preparing to run the JGSS sample programs" on page 348
- "Samples: Downloading the IBM JGSS samples" on page 348
- "Example: Running the non-JAAS sample" on page 349

Example: Running the non-JAAS sample
To run a sample, you need to download and modify the sample source code.

For more information, see Downloading and running the sample programs.

Starting the primary server

Use the following command to start a non-JAAS server that listens on port 4444. The server runs as the principal (superSecureServer) and uses a secondary server (backupServer). The server also displays application and credential debugging information.

```
java -classpath ibmjgsssample.jar
     -Dcom.ibm.security.jgss.debug="app, cred"
     com.ibm.security.jgss.test.Server -p 4444
     -n superSecureServer -s backupServer
```

Successfully running this example displays the following message:

```
listening on port 4444
```

Starting the secondary server

Use the following command to start a non-JAAS secondary server that listens on port 3333 and runs as principal backupServer:

```
java -classpath ibmjgsssample.jar
     com.ibm.security.jgss.test.Server -p 3333
     -n backupServer
```

Starting the client

Use the following command (typed on a single line) to run JAAS-enabled client (myClient). The client communicates with the primary server on the host (securityCentral). The client runs with the default security manager enabled, uses the JAAS configuration and policy files and the Java policy file from the config directory. For more information about the config directory, see Downloading the IBM JGSS samples.

```
java -classpath ibmjgsssample.jar
     -Djava.security.manager
     -Djava.security.auth.login.config=config/jaas.conf
     -Djava.security.policy=config/java.policy
     -Djava.security.auth.policy=config/jaas.policy
     com.ibm.security.jgss.test.JAASClient -n myClient
     -s superSecureServer -h securityCentral:4444
```

IBM JGSS Javadoc reference information

The Javadoc reference information for IBM JGSS includes classes and methods in the org.ietf.jgss api package and the Java versions of some Kerberos credential management tools.

Although JGSS includes several publicly accessible packages (for example, com.ibm.security.jgss and com.ibm.security.jgss.spi), you should use only APIs from the standardized org.ietf.jgss package. Using only this package ensures that your application conforms to the GSS-API specifications and ensures optimum interoperability and portability.

- org.ietf.jgss
- “com.ibm.security.krb5.internal.tools Class Kinit” on page 321
- “com.ibm.security.krb5.internal.tools Class Ktab” on page 323
- “com.ibm.security.krb5.internal.tools Class Klist” on page 320

Related concepts

JGSS concepts
JGSS operations consist of four distinct stages, as standardized by the Generic Security Service Application Programming Interface (GSS-API).

Configuring your server to use IBM JGSS

How you configure your server to use JGSS depends on which version of the Java Platform, Standard Edition (J2SE) that you run on your system.

Running IBM JGSS applications
The IBM Java Generic Security Service (JGSS) API 1.0 shields secure applications from the complexities and peculiarities of the different underlying security mechanisms. JGSS uses features provided by Java Authentication and Authorization Service (JAAS) and IBM Java Cryptography Extension (JCE).

Developing IBM JGSS applications
Use JGSS to develop secure applications. Learn about generating transport tokens, creating JGSS objects, establishing context, and more.

JGSS debugging
When you are trying to identify JGSS problems, use the JGSS debugging capability to produce helpful categorized messages.

Samples: IBM Java Generic Security Service (JGSS)
The IBM Java Generic Security Service (JGSS) sample files include client and server programs, configuration files, policy files, and Javadoc reference information. Use the sample programs to test and verify your JGSS setup.

Tuning Java program performance

You should take several aspects of Java application performance into consideration when building a Java application.

Here are some actions you can take to achieve better performance:

- Improve performance of your Java code by using the Just-In-Time compiler or using a shared class cache.
- Carefully set your values for optimal garbage collection performance.
- Only use native methods to start system functions that are relatively long running and are not available directly in Java.
- Use Java exceptions in cases that are not the normal flow through your application.

Additional information on these and other performance considerations can be found at:

- IBM Center for Java Technology Developer Kit Diagnostic Guide
- IBM SDK for Java Troubleshooting

Any job session can start and end PEX. Normally, the data that is collected is system wide and pertains to all jobs on the system, including your Java programs. At times, it may be necessary to start and stop the performance collection from inside a Java application. This reduces the collection time and may reduce the large volume of data that is typically produced by a call or return trace. PEX cannot run from within a Java thread. To start and stop a collection, you need to write a native method that communicates to an independent job though a queue or shared memory. Then, the second job starts and stops the collection at the appropriate time.

The following list shows additional areas to consider that can affect Java performance:

Related concepts

"Java profiling performance tools" on page 353
System wide central processing unit (CPU) profiling calculates the relative amount of CPU time that is spent in each Java method and all system functions in use by your Java program.

Related information

Performance

Java garbage collection

Garbage collection is the process of freeing storage that is used by objects that are no longer referred to by a program. With garbage collection, programmers no longer have to write error prone code to explicitly "free" or "delete" their objects. This code frequently results in "memory leak" program errors. The garbage collector automatically detects an object or group of objects that the user program can no longer reach. It does this because there are no references to that object in any program structure. Once the object has been collected, you can allocate the space for other uses.

For additional information on IBM Technology for Java garbage collection refer to the following information:

- IBM Center for Java Technology Developer Kit Diagnostic Guide

Related concepts

Java Native Method Invocation performance considerations
IBM Technology for Java supports invocation of both ILE and PASE for i native methods. The invocation of an ILE native method is more expensive than the invocation of an PASE for i native method.

Java exception performance considerations
The IBM i exception architecture allows versatile interrupt and retry capabilities. It also allows mixed language interaction. Throwing Java exceptions on the IBM i platform may be more expensive than on other platforms. This should not affect overall application performance unless Java exceptions are routinely used in the normal application path.

Java profiling performance tools
System wide central processing unit (CPU) profiling calculates the relative amount of CPU time that is spent in each Java method and all system functions in use by your Java program.

Collecting Java performance data
This topic contains information about collecting and analyzing Java performance data .

"Collecting Java performance data" on page 354
This topic contains information about collecting and analyzing Java performance data .

Java Native Method Invocation performance considerations

IBM Technology for Java supports invocation of both ILE and PASE for i native methods. The invocation of an ILE native method is more expensive than the invocation of an PASE for i native method.

If you have a native method that will be invoked frequently, the native method should be written to run in PASE for i.

You should use native methods to start system functions that are relatively long running and are not available directly in Java.

Related concepts

Java garbage collection
Garbage collection is the process of freeing storage that is used by objects that are no longer referred to by a program. With garbage collection, programmers no longer have to write error prone code to explicitly "free" or "delete" their objects. This code frequently results in "memory leak" program errors. The garbage collector automatically detects an object or group of objects that the user program can no longer reach. It does this because there are no references to that object in any program structure. Once the object has been collected, you can allocate the space for other uses.

Java exception performance considerations
The IBM i exception architecture allows versatile interrupt and retry capabilities. It also allows mixed language interaction. Throwing Java exceptions on the IBM i platform may be more expensive than on other platforms. This should not affect overall application performance unless Java exceptions are routinely used in the normal application path.

Java profiling performance tools
System wide central processing unit (CPU) profiling calculates the relative amount of CPU time that is spent in each Java method and all system functions in use by your Java program.

Collecting Java performance data

This topic contains information about collecting and analyzing Java performance data .

Java exception performance considerations

The IBM i exception architecture allows versatile interrupt and retry capabilities. It also allows mixed language interaction. Throwing Java exceptions on the IBM i platform may be more expensive than on other platforms. This should not affect overall application performance unless Java exceptions are routinely used in the normal application path.

Related concepts

Java garbage collection
Garbage collection is the process of freeing storage that is used by objects that are no longer referred to by a program. With garbage collection, programmers no longer have to write error prone code to explicitly "free" or "delete" their objects. This code frequently results in "memory leak" program errors. The garbage collector automatically detects an object or group of objects that the user program can no longer reach. It does this because there are no references to that object in any program structure. Once the object has been collected, you can allocate the space for other uses.

Java Native Method Invocation performance considerations
IBM Technology for Java supports invocation of both ILE and PASE for i native methods. The invocation of an ILE native method is more expensive than the invocation of an PASE for i native method.

Java profiling performance tools
System wide central processing unit (CPU) profiling calculates the relative amount of CPU time that is spent in each Java method and all system functions in use by your Java program.

Collecting Java performance data
This topic contains information about collecting and analyzing Java performance data .

Java profiling performance tools

System wide central processing unit (CPU) profiling calculates the relative amount of CPU time that is spent in each Java method and all system functions in use by your Java program.

Use a Performance Explorer (PEX) definition that traces performance monitor counter overflow (*PMCO) run cycle events. Samples are typically specified in intervals of one millisecond. To collect a valid trace profile, you should run your Java application until it accumulates two to three minutes of CPU time. This should produce over 100,000 samples. The Print Performance Explorer Report (PRTPEXRPT) command produces a histogram of CPU time that is spent across the entire application. This includes every Java method and all system-level activity.

Note: CPU profiling does not show relative CPU usage for Java programs that are interpreted.

Related concepts

Java garbage collection
Garbage collection is the process of freeing storage that is used by objects that are no longer referred to by a program. With garbage collection, programmers no longer have to write error prone code to explicitly "free" or "delete" their objects. This code frequently results in "memory leak" program errors. The garbage collector automatically detects an object or group of objects that the user program can no longer reach. It does this because there are no references to that object in any program structure. Once the object has been collected, you can allocate the space for other uses.

Java Native Method Invocation performance considerations
IBM Technology for Java supports invocation of both ILE and PASE for i native methods. The invocation of an ILE native method is more expensive than the invocation of an PASE for i native method.

Java exception performance considerations
The IBM i exception architecture allows versatile interrupt and retry capabilities. It also allows mixed language interaction. Throwing Java exceptions on the IBM i platform may be more expensive than on other platforms. This should not affect overall application performance unless Java exceptions are routinely used in the normal application path.

Collecting Java performance data

This topic contains information about collecting and analyzing Java performance data .

Java Virtual Machine Tool Interface

The Java Virtual Machine Tool Interface (JVMTI) is an interface for analyzing the Java virtual machine (JVM).

JVM TI is the superseder of Java Virtual Machine Profiler Interface (JVMPI) and the Java Virtual Machine Debugger Interface (JVMDI). JVMTI contains all the functionality of both JVMDI and JVMPI, plus new functions. JVMTI was added as part of J2SE 5.0. In JDK 6, the JVMDI and JVMPI interfaces are no longer offered, and JVMTI is the only option available.

Note: IBM Technology for Java only supports JVMTI interfaces from PASE for i. As a result, ILE JVMTI agents will need to be ported to PASE for i.

Related information
Java Virtual Machine Tool Interface (JVMTI) by Oracle Corporation.

Collecting Java performance data

This topic contains information about collecting and analyzing Java performance data .

You can use the Work with JVM Jobs (WRKJVMJOB) CL command to collect performance data. You can access the information available from the WRKJVMJOB command from both the Work with Job (WRKJOB) screen as well as by issuing the WRKJVMJOB command.

The following information or functionality is available when using WRKJVMJOB:

- The arguments and options with which the JVM was started.
- Environment variables for both ILE and PASE for i.
- Java lock requests outstanding for the JVM job.
- Garbage collection information.
- Java system properties.
- The list of threads associated with the JVM.
- The partially completed job log for the JVM job.
- The ability to work with spooled input and output files for the JVM job.
- The ability to generate JVM (system, heap, Java) dumps from a panel option. These capabilities are also available from the Generate JVM Dump (GENJVMDMP) command.
- The ability to enable and disable verbose garbage collection from a panel option.

You can also use the IBM Monitoring and Diagnostic Tools for Java - Health Center (Health Center). Health Center enables you to assess the current status of a running Java application. Health Center gives clear and easy to understand information about performance, memory usage and management, optimization and profiling. Health Center interprets profiling data and provides recommendations to help with problem areas. For additional information about Health Center go to the Health Center Web site.

Related concepts
Java garbage collection
Garbage collection is the process of freeing storage that is used by objects that are no longer referred to by a program. With garbage collection, programmers no longer have to write error prone code to explicitly "free" or "delete" their objects. This code frequently results in "memory leak" program errors. The garbage collector automatically detects an object or group of objects that the user program can no longer reach. It does this because there are no references to that object in any program structure. Once the object has been collected, you can allocate the space for other uses.

Java Native Method Invocation performance considerations

IBM Technology for Java supports invocation of both ILE and PASE for i native methods. The invocation of an ILE native method is more expensive than the invocation of an PASE for i native method.

Java exception performance considerations
The IBM i exception architecture allows versatile interrupt and retry capabilities. It also allows mixed language interaction. Throwing Java exceptions on the IBM i platform may be more expensive than on other platforms. This should not affect overall application performance unless Java exceptions are routinely used in the normal application path.

Java profiling performance tools
System wide central processing unit (CPU) profiling calculates the relative amount of CPU time that is spent in each Java method and all system functions in use by your Java program.

"Java system properties" on page 14
Java system properties determine the environment in which you run your Java programs. They are similar to system values or environment variables in IBM i.

Related information

WRKJVMJOB CL command description

Java commands and tools

When using Java on IBM i, you can use either Java tools with the Qshell Interpreter or CL commands.

If you have prior Java programming experience, you may be more comfortable using the Qshell Interpreter Java tools, because they are similar to the tools that you would use with Oracle America, Inc. Java Development Kit. See the Qshell topic for information about using the Qshell environment.

If you are an IBM i programmer, you may want to use the CL commands for Java that are typical to the IBM i environment. Read on for more information about using CL commands and System i Navigator commands.

Related information

Qshell Interpreter

Java tools and utilities

The Qshell environment includes the Java development tools that are typically required for program development.

With a few exceptions, the Java tools support the syntax and options that are documented by Oracle Corporation.

All Java tools must all run by using the Qshell Interpreter. You can start the Qshell Interpreter by using the Start Qshell (STRQSH or QSH) command. When the Qshell Interpreter is running, a QSH Command Entry display appears. All output and messages from Java tools and programs that run under Qshell appear in this display. Any input to a Java program is also read from this display.

Related concepts

CL commands that are supported by Java
The CL environment contains CL commands for optimizing and managing Java programs.

"Native Abstract Windowing Toolkit" on page 234
Native Abstract Windowing Toolkit (NAWT) is a not really a toolkit but rather a term that has evolved to refer to the native IBM i support that provides Java applications and servlets with the capability to use

the Abstract Windowing Toolkit (AWT) graphics functions offered by the Java Platform, Standard Edition (J2SE).

Standard Java tools and utilities

Each Java Development Kit (JDK) version ships a version of the Java tools and utilities. In most cases, the Qshell version of the Java tools and utilities located in the `/usr/bin` directory invoke the proper version of the tool or utility based on what version of the JDK you are using.

The actual location of the Java tools and utilities is based on the `JAVA_HOME` environment variable, which determines the JDK that is used when running a Java application. The location of the Java tools and utilities is in one of two directories, either *<JAVA_HOME>*`/jre/bin` or *<JAVA_HOME>*`/bin`, where *<JAVA_HOME>* is the value of the `JAVA_HOME` environment variable. For example, if the `JAVA_HOME` environment variable is set to `/QOpenSys/QIBM/ProdData/JavaVM/jdk70/32bit`, indicating that IBM Technology for Java 7 32-bit is to be used, then the Java tools and utilities directories would be:

```
/QOpenSys/QIBM/ProdData/JavaVM/jdk70/32bit/bin
/QOpenSys/QIBM/ProdData/JavaVM/jdk70/32bit/jre/bin
```

Keep in mind these points when using the Java tools and utilities:

- If the `JAVA_HOME` environment variable is not set and you are using the version of the Java tool or utility that is in the `/usr/bin` directory, the default JDK is used. For more information about how JDKs are selected, see "Support for multiple Java Development Kits (JDKs)" on page 6.
- Only a subset of the Java tools and utilities located in the *<JAVA_HOME>*`/jre/bin` and *<JAVA_HOME>*`/bin` directories can be found in the `/usr/bin` directory.
- Not all the Java tools and utilities that are supported on IBM i are supported by all JDK versions. Search the *<JAVA_HOME>*`/jre/bin` and *<JAVA_HOME>*`/bin` directories to determine if a Java tool or utility is supported for the JDK version you are using.
- Extensive documentation on the Java tools and utilities can be found on the Oracle JDK Tools and Utilities Web site. For Java tools and utilities that are not documented on the Oracle Web site, run the Java tool or utility with the `-h` or `-help` option.

Basic tools and utilities

appletviewer (Java Applet Viewer)
Tests and runs applets outside a Web browser.

apt (Annotation Processing Tool)
Finds and executes annotation processors based on the annotations present in the set of specified source files being examined.

extcheck (Extcheck utility)
Detects version conflicts between a target JAR file and currently-installed extension JAR files.

jar (Java Archive Tool)
Combines multiple files into a single Java Archive (JAR) file.

java (Java Interpreter)
Runs Java classes. The Java Interpreter runs programs that are written in the Java programming language.

javac (Java Compiler)
Compiles programs that are written in the Java programming language into bytecode (compiled Java code).

javadoc (Java Documentation Generator)
Generates HTML pages of API documentation from Java source files.

javah (C Header and Stub File Generator)
Enables you to associate native methods with code written in the Java programming language.

javap (Class File Disassembler)
Disassembles compiled files and can print a representation of the bytecodes.

javaw (Java Interpreter)
Runs Java classes in the same way as the java command does, but does not use a console window.

jdb (Java Debugger)
Helps debug your Java programs. When invoked, a message is displayed indicating that the tool is not supported. For alternatives to this tool, see "Debugging Java programs on IBM i" on page 360.

Security tools and utilities

ikeyman (iKeyman GUI utility)
Allows you to manage keys, certificates, and certificate requests. For more information, see the iKeyman User's Guide. There is also a command-line version of this utility.

ikeycmd (iKeyman command-line utility)
Allows you to manage keys, certificates, and certificate requests from the command line. For more information, see the iKeyman User's Guide. There is also a GUI version of this utility.

jarsigner (JAR Signing and Verification Tool)
Generates signatures for JAR files and verifies the signatures of signed JAR files.

keytool (Key and Certificate Management Tool)
Manages a keystore (database) of private keys and their associated X.509 certificate chains that authenticate the corresponding public keys.

kinit
Obtains and caches Kerberos ticket-granting tickets. The Qshell version of this utility that is located in `/usr/bin` does not invoke the Java version of this utility. To determine which to use, see "The kinit and ktab tools" on page 329.

klist
Displays entries in the local credentials cache and key table. The Qshell version of this utility that is located in `/usr/bin` does not invoke the Java version of this utility. To determine which to use, see "The kinit and ktab tools" on page 329.

ktab
Manages the principal names and service keys stored in a local key table.

policytool (Policy File Creation and Management Tool)
Creates and modifies the external policy configuration files that define the security policy for your Java installation.

Internationalization tools and utilities

native2ascii (Native-To-ASCII Converter)
Converts a native encoding file to an ASCII file that contains characters encoded in either Latin-1 or Unicode, or both.

Remote method invocation (RMI) tools and utilities

rmic (Java Remote Method Invocation (RMI) Stub Converter)
Generates stubs, skeletons, and ties for remote objects. Includes RMI over Internet Inter-ORB Protocol (RMI-IIOP) support.

rmid (RMI activation system daemon)
Starts the activation system daemon so that objects can be registered and activated in a Java virtual machine (JVM).

rmiregistry (Java remote object registry)
Creates and starts a remote object registry on the specified port of the current host.

serialver (Serial Version Command)
Returns the serialVersionUID for one or more classes in a format that is suitable for copying into an evolving class.

Java IDL and RMI-IIOP tools and utilities

idlj (IDL to Java Compiler)
Generates Java bindings from a given Interface Definition Language (IDL) file.

orbd
Provides support for clients to transparently locate and invoke persistent objects on servers in the Common Object Request Broker Architecture (CORBA) environment.

tnameserv (CORBA transient naming service)
Starts the CORBA transient naming service.

Java deployment tools and utilities

pack200
Transforms a JAR file into a compressed `pack200` file using the Java `gzip` compressor.

unpack200
Transforms a packed file produced by `pack200` into a JAR file.

Java plug-in tools and utilities

HtmlConverter (Java Plug-in HTML Converter)
Converts an HTML page that contains applets to a format that can use the Java plug-in.

Java Web Start tools and utilities

javaws (Java Web Start)
Enables the deployment and automatic maintenance of Java applications. For more information, see Running Web Start.

Java troubleshooting, profiling, monitoring and management tools and utilities

jconsole (JConsole Monitoring and Management Tool)
Monitors local and remote JVMs using a GUI. The tool is JMX-compliant.

jdmpview (Cross-platform dump formatter)
Analyzes dumps. For more information, see the Diagnostics Guide.

jextract (Dump extractor)
Converts a system-produced dump into a common format that can be used by `jdmpview`. For more information, see `jdmpview`.

Java Web services tools and utilities

schemagen
Creates a schema file for each namespace referenced in your Java classes.

wsgen
Generates JAX-WS portable artifacts used in JAX-WS Web services.

wsimport
Generates JAX-WS portable artifacts from a Web Services Description Language (WSDL) file.

xjc
Compiles XML Schema files.

Related concepts
"Support for multiple Java Development Kits (JDKs)" on page 6

The IBM i platform supports multiple versions of the Java Development Kits (JDKs) and the Java 2 Platform, Standard Edition.

Related information

IBM Technology for Java Virtual Machine in IBM i5/OS

Oracle JDK Tools and Utilities

IBM Java tools and utilities

IBM provides additional tools in support of functions or features supported by IBM i. See the topics here for descriptions of the IBM Java tools.

Java hwkeytool

The `hwkeytool` application enables you to use the cryptography capabilities of the model 4764 Cryptographic Coprocessor with the Java Cryptography Extension (JCE) and Java Cryptography Architecture (JCA).

The `hwkeytool` application for hardware uses the same syntax and commands as the `keytool` application with the exception of two commands and the default keystore. The hardware keytool provides additional parameters to the `-genkey` and `delete` commands.

On the `-genkey` command, the following additional parameters are available:

-KeyLabel
: Allows you to set a specific label for the hardware key.

-hardwaretype
: Determine the type of key pair: Public key data set (PKDS) or RETAINED.

-hardwareusage
: Set the usage of the key pair being generated, either a signature-only key or a signature and key management key.

On the `delete` command, an additional parameter of **-hardwarekey** , which deletes the key pair from the keystore and from the hardware, is available.

The default keystore name is `.HWkeystore`. You can change this using the **-keystore** parameter.

4764 Cryptographic Coprocessor

Additional Java tools and utilities

IBM provides additional Java tools and utilities that are not part of the IBM i Java licensed product but can be used on IBM i servers.

- **IBM Support Assistant**

  A complimentary software offering which provides you with a workbench to help you with problem determination. With a focus on quickly finding key information, automating repetitive steps and arming you with a variety of serviceability tools, you'll be prepared for self-analysis and diagnosis of problems and faster time to resolution.

- **IBM Monitoring and Diagnostic Tools for Java**

  Provides tooling and documentation to assist in the understanding, monitoring, and problem diagnosis of applications and deployments running IBM Runtime Environments for Java.

CL commands that are supported by Java

The CL environment contains CL commands for optimizing and managing Java programs.

- Display Java Virtual Machine Jobs (DSPJVMJOB) command displays information about active JVM jobs to help you manage the application of program temporary fixes (PTFs). You can also find more details about DSPJVMJOB in "Applying program temporary fixes" on page 486.

- Generate JVM Dump (GENJVMDMP) command generates Java Virtual Machine (JVM) dumps upon request.
- Print JVM Job (PRTJVMJOB) command allows you to print Java Virtual Machines (JVMs) running in active jobs.
- JAVA command and Run Java (RUNJVA) command run IBM i Java programs.
- Work with JVM Jobs (WRKJVMJOB) displays information about jobs running in the IBM Technology for Java Virtual Machine.

Related concepts

Java tools and utilities
The Qshell environment includes the Java development tools that are typically required for program development.

Licensed Internal Code option parameter strings

Program and CL Command APIs

Debugging Java programs on IBM i

You have several options for debugging and troubleshooting Java programs that run on your system, including IBM i Debugger, the system interactive display, Java Debug Wire Protocol-enabled debuggers, and Heap Analysis Tools for Java.

The following information is not a comprehensive assessment of the possibilities, but does list several options.

One of the easiest ways to debug Java programs that run on your system is to use the IBM i Debugger. The IBM i Debugger provides a graphical user interface (GUI) that enables you to more easily use the debugging capabilities of your server. You can use the interactive display of your server to debug Java programs, although the IBM i Debugger provides a more easily usable GUI that enables you to perform the same functions.

Additionally, the IBM i Java virtual machine (JVM) supports the Java Debug Wire Protocol (JDWP), which is part of the Java Platform Debugger Architecture. JDWP-enabled debuggers allow you to perform remote debugging from clients that run different operating systems. (The IBM i Debugger also enables you to perform remote debugging in a similar way, although it does not use JDWP.) One such JDWP-enabled program is the Java debugger in the Eclipse project universal tool platform.

If the performance of your program degrades as it runs for a longer period of time, you may have inadvertently coded a memory leak. To help you debug your program and locate memory leaks, see "Finding memory leaks" on page 370.

IBM i Debugger

"Java Platform Debugger Architecture" on page 369
The Java Platform Debugger Architecture (JPDA) consists of the JVM Debug Interface/JVM Tool Interface, the Java Debug Wire Protocol, and the Java Debug Interface. All these parts of the JPDA enable any front end of a debugger that uses the JDWP to perform debugging operations. The debugger front end can either run remotely or run as a IBM i application.

Java development tool debug

Eclipse project Web site

Debugging Java programs using IBM i Debugger

The easiest way to debug Java programs that run on your system is to use the IBM i Debugger. The IBM i Debugger provides a graphical user interface that enables you to more easily use the debugging capabilities of your system.

For more information about using the IBM i Debugger to debug and test Java programs that run on your server, see IBM IBM i Debugger.

Related concepts

Java Platform Debugger Architecture

The Java Platform Debugger Architecture (JPDA) consists of the JVM Debug Interface/JVM Tool Interface, the Java Debug Wire Protocol, and the Java Debug Interface. All these parts of the JPDA enable any front end of a debugger that uses the JDWP to perform debugging operations. The debugger front end can either run remotely or run as a IBM i application.

Finding memory leaks

If the performance of your program degrades as it runs for a longer period of time, you may have erroneously coded a memory leak.

Using the Generate JVM Dump command

The Generate JVM Dump (GENJVMDMP) CL command can be used to generate Java, system, and heap dumps.

System debugging for IBM Technology for Java

These instructions present several options for debugging IBM Technology for Java JVMs.

Interactive debugging from the CL command line

The simplest way to start the system debugger is with the OPTION(*DEBUG) parameter of the JAVA CL command. For example:

```
> JAVA CLASS(Hello) OPTION(*DEBUG)
```

Enabling debug for the IBM Technology for Java JVM

In order to debug an IBM Technology for Java JVM job from another job, the JVM must be started with debug enabled. Java debug is managed by a debug agent. Starting this agent during JVM startup is the key to successfully debugging Java code. Once the JVM is successfully started with the debug agent, the JVM may be debugged using either the Start Service Job (STRSRVJOB) and Start Debug (STRDBG) commands or from the IBM i Debugger graphical user interface. The following sections describe various ways to start the debug agent. In each case the purpose of a parameter or environment variable is to indicate that the Java debug agent should be started. These descriptions start with the simplest situations and progress to more complicated ones.

Note:

- The debug agent does not suspend the JVM prior to entering the main method. For short-running programs or for debugging the main method, additional Java code may be needed in order to stop the JVM. One way to do this is with a timed wait-loop. Another way is to read from standard input.
- If debug is attempted on a JVM which does not have the debug agent enabled, a JVAB307 diagnostic message is sent to the job log of both the JVM job and the servicing job. The message text identifies the JVM job that does not have debug enabled. This message indicates that the JVM must be restarted in order to be successfully debugged. It is not possible to enable debug after the JVM is started.

Enabling Java debug from CL

To enable Java debug from CL, add the `AGTPGM(D9TI)` parameter to the JAVA CL command. For example:

```
> JAVA CLASS(Hello) AGTPGM(D9TI)
```

Enabling Java debug from Qshell or the PASE terminal

To enable Java debug from Qshell (QSH) or the PASE terminal (QP2TERM), add the `-debug` parameter in the `java` invocation. For example:

```
> java -debug Hello
```

Using the `-debug` parameter is a simplest way to start the debug agent. It is the same as adding the `-agentlib:d9ti` parameter. The following will also start the debug agent:

```
> java -agentlib:d9ti Hello
```

Enabling Java debug for a batch job JVM

If the batch job JVM is started with the Submit Job (SBMJOB) CL command, then the `AGTPGM(D9TI)` parameter may be added to the JAVA CL command. For example, the following will start the batch job JVM with a debug agent:

```
> SBMJOB CMD(JAVA CLASS(HELLO) AGTPGM(D9TI))
        CPYENVVAR(*YES) ALWMLTTHD(*YES)
```

If the batch job is started in some other way, then the JAVA_TOOL_OPTIONS environment variable may be used to start the debug agent. The JAVA_TOOL_OPTIONS environment variable is automatically queried by the JVM during startup. If it is set to *-agentlib:d9ti*, then the debug agent will be started for the JVM. For example:

```
> ADDENVVAR ENVVAR(JAVA_TOOL_OPTIONS) VALUE('-agentlib:d9ti')
```

If the batch job does not automatically inherit all environment variables, then the JAVA_TOOL_OPTIONS environment variable will need to be set system-wide. For example:

```
> ADDENVVAR ENVVAR(JAVA_TOOL_OPTIONS) VALUE('-agentlib:d9ti') LEVEL(*SYS)
```

Note: When you set the JAVA_TOOL_OPTIONS environment variable system-wide, all IBM Technology for Java JVMs started on the system are started with debug enabled. **This may result in a significant performance degradation**.

Enabling Java debug for a JVM created with the Java Invocation API

When using the JNI_CreateJavaVM API to create a JVM, or when calling a Java method from RPG using the EXTPROC(*JAVA) support, you can enable debug using any one of the following methods:

- Set the JAVA_TOOL_OPTIONS environment variable to *-agentlib:d9ti*.
- Add the `-debug` parameter to the options parameter list passed to the JNI_CreateJavaVM API.
- Add the `-agentlib:d9ti` parameter to the options parameter list passed to the JNI_CreateJavaVM API.
- Add "-debug" or "-agentlib:d9ti" to the QIBM_RPG_JAVA_OPTIONS environment variable if you are starting the JVM by calling a Java method from RPG.

Debugging must be done from another job.

In addition, in order to see the Java source code for the classes being debugged, the DEBUGSOURCEPATH environment variable may need to be set to the base directory location of the Java source code.

Starting the IBM Technology for Java JVM from the IBM i Debugger graphical user interface

In order to start an IBM Technology for Java JVM from the IBM i Debugger graphical user interface, the JAVA_HOME environment variable must be set when starting the JVM job. You can set this environment variable using the **Initialization Command** display when starting the JVM. This display is located in the Start Debug window of the IBM i Debugger interface.

For example, in order to start a 32-bit JDK 7.0 JVM, add the following to the **Initialization Command** display:

```
ADDENVVAR ENVVAR(JAVA_HOME) VALUE('/QOpenSys/QIBM/ProdData/JavaVM/jdk70/32bit')
```

Note: Watch points for local variables are not supported in the IBM Technology for Java JVM. The System Debug implementation for the IBM Technology for Java JVM uses JVMTI, which does not provide watch point capability for local variables.

Related information

IBM i Debugger

Debug operations

You can use the interactive display of your server to use the `*DEBUG` option to view the source code before running the program. Then, you can set breakpoints, or step over or into a program to analyze errors while the program is running.

Debugging Java programs using the *DEBUG option

To debug Java programs using the *DEBUG option, follow these steps:

1. Compile the Java program by using the DEBUG option, which is the -g option on the `javac` tool.
2. Insert the class file (.class) and source file (.java) in the same directory on your server.
3. Run the Java program by using the Run Java (RUNJVA) command on the IBM i command line. Specify OPTION(*DEBUG) on the Run Java (RUNJVA) command. For example: `RUNJVA CLASS(classname) OPTION(*DEBUG)`

   Only a class may be debugged. If a JAR file name is entered for the CLASS keyword, OPTION(*DEBUG) is not supported.
4. The Java program source is displayed.
5. Press F6 (Add/Clear breakpoint) to set breakpoints, or press F10 (Step) to step through the program.

Note:

- While using breakpoints and steps, check the logical flow of the Java program, then view and change variables, as necessary.
- Using OPTION(*DEBUG) on the RUNJVA command disables the Just-In-Time (JIT) compiler.
- If you are not authorized to use the Start Service Job (STRSRVJOB) command, OPTION(*DEBUG) is ignored.

Debugging Java programs from another display

When debugging a Java program by using the interactive display of your server, the program source displays whenever it encounters a breakpoint. This may interfere with the display output of the Java program. To avoid this, debug the Java program from another display. The output from the Java program displays where the Java command is running and the program source shows on the other display.

It is also possible to debug an already running Java program in this manner as long as it has been started with debug enabled.

Note: You can enable Java debug by adding the AGTPGM(D9TI) option to the JAVA/RUNJVA command in order to use the IBM i Debugger with the JVM. AGTPGM(D9TI) is not needed when using OPTION(*DEBUG).

To debug Java from another display, do the following:

1. The Java program must be held while you start setting up to debug.

   You can hold the Java program by making the program:

   - Wait for input from the keyboard.
   - Wait for a time interval.
   - Loop to test a variable, which requires that you set a value to eventually get the Java program out of the loop.

2. Once the Java program is held, go to another display to perform these steps:
   a. Enter the Work with Active Jobs (WRKACTJOB) command on the command line.
   b. Find the batch immediate (BCI) job where your Java program is running. Look under the Subsystem/Job listing for QJVACMDSRV. Look under the Current User listing for your User ID. Look under Type for BCI.
   c. Enter Option 5 to work with that job.
   d. At the top of the Work with Job display, the Number, User, and Job are displayed. Enter `STRSRVJOB Number/User/Job`
   e. Enter `STRDBG CLASS(classname)`. Classname is the name of the Java class that you want to debug. It can either be the class name that you specified on the Java command, or it can be another class.
   f. The source for that class appears in the Display Module Source display.
   g. Set breakpoints, by pressing F6 (Add/Clear breakpoint), whenever you would like to stop in that Java class. Press F14 to add other classes, programs, or service programs to debug.
   h. Press F12 (Resume) to continue running the program.
3. Stop holding your original Java program. When the breakpoints are hit, the Display Module Source display appears on the display where the Start Service Job (STRSRVJOB) command and the Start Debug (STRDBG) command were entered. When the Java program ends, a `Job being serviced ended` message appears.
4. Enter the End Debug (ENDDBG) command.
5. Enter the End Service Job (ENDSRVJOB) command.

When you debug a Java program, your Java program is actually running in the Java virtual machine in a batch immediate (BCI) job. Your source code displays in the interactive display, but the Java program is not running there. It is running in the other job, which is a serviced job. See the QIBM_CHILD_JOB_SNDINQMSG environment variable topic for more information about this variable that controls whether the BCI job waits before calling the Java virtual machine.

Related information

IBM i Debugger

Initial debugging displays for Java programs

As you debug your Java programs, follow these example displays for your programs. These displays show an example program, named Hellod.

- Enter `ADDENVVAR ENVVAR(CLASSPATH) VALUE ('/MYDIR')`.
- Enter this command: `RUNJVA CLASS(HELLOD) OPTION(*DEBUG)`. Insert the name of your Java program in place of HELLOD.
- Wait for the Display Module Source display to appear. This is the source for the HELLOD Java program.

```
+--------------------------------------------------------------------------------+
|                           Display Module Source                                |
|                                                                                |
|  Class file name:   HELLOD                                                     |
|       1  import java.lang.*;                                                   |
|       2                                                                        |
|       3  public class Hellod extends Object                                    |
|       4  {                                                                     |
|       5  int k;                                                                |
|       6  int l;                                                                |
|       7  int m;                                                                |
|       8  int n;                                                                |
|       9  int o;                                                                |
|      10  int p;                                                                |
|      11  String myString;                                                      |
|      12  Hellod myHellod;                                                      |
|      13  int myArray[];                                                        |
|      14                                                                        |
|      15  public Hellod()                                                       |
|                                                                    More...     |
```

```
|  Debug . . .                                                                  |
|                                                                               |
|  F3=End program   F6=Add/Clear breakpoint   F10=Step   F11=Display variable   |
|  F12=Resume       F17=Watch variable   F18=Work with watch   F24=More keys    |
|                                                                               |
+-------------------------------------------------------------------------------+
```

- Press F14 (Work with Module List).
- The Work with Module List display is shown. You can add other classes and programs to debug by entering option 1 (Add program). Display their source with option 5 (Display module source).

```
+-------------------------------------------------------------------------------+
|                                Work with Module List                          |
|                                                      System:   AS400          |
|  Type options, press enter.                                                   |
|    1=Add program   4=Remove program   5=Display module source                 |
|    8=Work with module breakpoints                                             |
|                                                                               |
|  Opt     Program/module     Library          Type                             |
|                             *LIBL            *SRVPGM                          |
|          HELLOD                              *CLASS      Selected             |
|                                                                               |
|                                                                               |
|                                                                               |
|                                                                               |
|                                                                               |
|                                                                               |
|                                                                               |
|                                                                               |
|                                                                    Bottom     |
|  Command                                                                      |
|  ===>                                                                         |
|  F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel                  |
|  F22=Display class file name                                                  |
|                                                                               |
+-------------------------------------------------------------------------------+
```

- When adding a class to debug, you may need to enter a package-qualified class name that is longer than the Program/module input field. To enter a longer name, follow these steps:
  1. Enter Option 1 (Add program).
  2. Leave the Program/module field blank.
  3. Leave the library field as *LIBL.
  4. Enter *CLASS for Type.
  5. Press Enter.
  6. A pop up window is displayed where you have more room to enter the package-qualified class file name. For example: `pkgname1.pkgname2.classname`

Setting breakpoints

You can control the running of a program with breakpoints. Breakpoints stop a running program at a specific statement.

To set breakpoints, perform the following steps:

1. Place the cursor on the line of code where you would like to set a breakpoint.
2. Press F6 (Add/Clear breakpoint) to set the breakpoint.
3. Press F12 (Resume) to run the program.

Note: Just before the line of code runs, where the breakpoint is set, the program source is displayed indicating that the breakpoint was hit.

```
+-------------------------------------------------------------------------------+
|                             Display Module Source                             |
|                                                                               |
|Current thread:   00000019     Stopped thread:   00000019                      |
|Class file name:   Hellod                                                      |
```

```
|35  public static void main(String[] args)                                   |
|36      {                                                                     |
|37    int i,j,h,B[],D[][];                                                    |
|38      Hellod A=new Hellod();                                                |
|39      A.myHellod = A;                                                       |
|40      Hellod C[];                                                           |
|41     C = new Hellod[5];                                                     |
|42      for (int counter=0; counter<2; counter++) {                           |
|43        C[counter] = new Hellod();                                          |
|44        C[counter].myHellod = C[counter];                                   |
|45      }                                                                     |
|46      C[2] = A;                                                             |
|47      C[0].myString = null;                                                 |
|48      C[0].myHellod = null;                                                 |
|                                                                            |
|49      A.method1();                                                          |
|Debug . . .                                                                 |
|                                                                            |
|F3=End program   F6=Add/Clear breakpoint   F10=Step   F11=Display variable  |
|F12=Resume   F17=Watch variable   F18=Work with watch   F24=More key        |
|Breakpoint added to line 41.                                                |
+----------------------------------------------------------------------------+
```

Once a breakpoint is encountered or a step is completed, you can use the TBREAK command to set a breakpoint that only applies to the current thread.

Stepping through Java programs

You can step through your program while debugging. You can either step over or step into other functions. Java programs and native methods can use the step function.

When the program source first displays, you can start stepping. The program stops before running the first statement. Press F10 (Step). Continue to press F10 (Step) to step through the program. Press F22 (Step into) to step into any function that your program calls. You can also start stepping anytime a breakpoint is hit. For information about setting breakpoints, see the Set breakpoints topic.

```
+----------------------------------------------------------------------------+
|                            Display Module Source                           |
|                                                                            |
|Current thread:   00000019     Stopped thread:   00000019                   |
|Class file name:   Hellod                                                   |
|35  public static void main(String[] args)                                   |
|36      {                                                                     |
|37    int i,j,h,B[],D[][];                                                    |
|38      Hellod A=new Hellod();                                                |
|39      A.myHellod = A;                                                       |
|40      Hellod C[];                                                           |
|41     C = new Hellod[5];                                                     |
|42      for (int counter=0; counter<2; counter++) {                           |
|43        C[counter] = new Hellod();                                          |
|44        C[counter].myHellod = C[counter];                                   |
|45      }                                                                     |
|46      C[2] = A;                                                             |
|47      C[0].myString = null;                                                 |
|48      C[0].myHellod = null;                                                 |
|49      A.method1();                                                          |
|Debug . . .                                                                 |
|                                                                            |
|F3=End program   F6=Add/Clear breakpoint   F10=Step   F11=Display variable  |
|F12=Resume   F17=Watch variable   F18=Work with watch   F24=More key        |
|Step completed at line 42 in thread 00000019                                |
+----------------------------------------------------------------------------+
```

To continue running the program, press F12 (Resume).

Evaluating variables in Java programs

There are two ways to evaluate a variable when a program stops running at a breakpoint or step.

- Option 1: Enter `EVAL VariableName` on the debug command line.
- Option 2: Put the cursor on the variable name in the displayed source code and press F11 (Display variable).

Note: You can also change the contents of a variable by using the EVAL command. For more information about the variations of the EVAL command, see WebSphere Development Studio: ILE C/C++ Programmer's Guide, SC09-2712 and online help information.

When looking at variables in a Java program, note the following:

- If you evaluate a variable that is an instance of a Java class, the first line of the display shows what kind of object it is. It also shows an identifier for the object. Following the first display line, the contents of each field in the object displays. If the variable is null, the first line of the display indicates that it is null. Asterisks show the contents of each field (of a null object).
- If you evaluate a variable that is a Java string object, the contents of that string displays. If the string is null, then null displays.
- You cannot change a variable that is a string or an object.
- If you evaluate a variable that is an array, 'ARR' displays followed by an identifier for that array. You can evaluate elements of the array by using a subscript of the variable name. If the array is null, then null displays.
- You cannot change a variable that is an array. You can change an element of an array if it is not an array of strings or objects.
- For variables that are arrays, you can specify `arrayname.length` to see how many elements are in the array.
- If you want to see the contents of a variable that is a field of a class, you can specify `classvariable.fieldname`.
- If you try to evaluate a variable before it has been initialized, one of two things can happen. Either a `Variable not available to display` message is shown, or the uninitialized contents of the variable are shown, which could be a strange value.

Debugging Java and native method programs

You can debug Java programs and native method programs at the same time. While you are debugging your source on the interactive display, you can debug a native method that is programmed in C, which is within a service program (*SRVPGM). The *SRVPGM must be compiled and created with debug data.

To use the interactive display of the server to debug Java programs and native method programs at the same time, complete the following steps:

1. Press F14 (Work with module list) when your Java program source is displayed to show the Work with Module List (WRKMODLST) display.
2. Select option 1 (Add program) to add your service program.
3. Select option 5 (Display module source) to display the *MODULE that you want to debug and the source.
4. Press F6 (Add/Clear breakpoint) to set breakpoints in the service program. For more information about setting breakpoints, see “Setting breakpoints” on page 365.
5. Press F12 (Resume) to run the program.

Note: When the breakpoint is hit in your service program, the program stops running, and the source for the service program displays.

Using the QIBM_CHILD_JOB_SNDINQMSG environment variable for debug

The QIBM_CHILD_JOB_SNDINQMSG environment variable is the variable that controls whether the batch immediate (BCI) job, where the Java virtual machine runs, waits before starting the Java virtual machine.

If you set the environment variable to 1 when the Run Java (RUNJVA) command runs, a message is sent to the user's message queue. The message is sent before the Java virtual machine starts in the BCI job. The message looks like this:

```
Spawned (child) process 023173/JOB/QJVACMDSRV is stopped (G C)
```

To view this message, enter SYSREQ and select option 4.

The BCI job waits until you enter a reply to this message. A reply of (G) starts the Java virtual machine.

You can set breakpoints in a *SRVPGM or *PGM, which the BCI job calls, before replying to the message.

Note: You cannot set breakpoints in a Java class, because at this point, the Java virtual machine has not been started.

Debugging Java classes loaded through a custom class loader

To use the interactive display of your server to debug a class loaded through a custom class loader, complete the following steps.

1. Set the DEBUGSOURCEPATH environment variable to the directory containing the source code, or in the case of a package-qualified class, the starting directory of the package names.

   For example, if the custom class loader loads classes located under the directory /MYDIR, perform the following:

   ```
   ADDENVVAR ENVVAR(DEBUGSOURCEPATH) VALUE('/MYDIR')
   ```

2. Add the class to the debug view from the Display Module Source screen.

   If the class has already been loaded into the Java virtual machine (JVM), just add the *CLASS as usual and display the source code to debug.

   For example, to view the source for pkg1/test14.class, enter the following:

   ```
   Opt     Program/module     Library        Type
   1       pkg1.test14_       *LIBL          *CLASS
   ```

   If the class has not been loaded into the JVM, perform the same steps to add the *CLASS as previously indicated. The **Java class file not available** message then displays. At this point, you may resume program processing. The JVM automatically stops when any method of the class matching the given name is entered. The source code for the class is displayed and can be debugged.

Debugging servlets

Debugging servlets is a special case of debugging classes loaded through a custom class loader. Servlets run in the Java runtime of an application server such as IBM WebSphere Application Server for IBM i or the IBM Integrated Web Application Server for i. You have several options to debug servlets.

You can debug servlets by following the instructions for classes loaded through a custom class loader.

You can also use the interactive display of your server to debug a servlet by completing the following steps:

1. Use the `javac -g` command in the Qshell Interpreter to compile your servlet.
2. Copy the source code (.java file) and compiled code (.class file) to a directory in your classpath.
3. Start the server.
4. Run the Start Service Job (STRSRVJOB) command on the job where the servlet runs.
5. Enter STRDBG CLASS(myServlet), where myServlet is the name of your servlet. The source should be displayed.
6. Set a breakpoint in the servlet and press F12.
7. Run your servlet. When the servlet hits the breakpoint, you can continue debugging.

Another way to debug Java programs and servlets that run on your system is to use the IBM i Debugger. The IBM i Debugger provides a graphical user interface that enables you to more easily use the debugging capabilities of your system.

Related information

IBM i Debugger

Java Platform Debugger Architecture

The Java Platform Debugger Architecture (JPDA) consists of the JVM Debug Interface/JVM Tool Interface, the Java Debug Wire Protocol, and the Java Debug Interface. All these parts of the JPDA enable any front end of a debugger that uses the JDWP to perform debugging operations. The debugger front end can either run remotely or run as a IBM i application.

Java Virtual Machine Tool Interface (JVMTI)

JVMTI supercedes Java Virtual Machine Debug Interface (JVMDI) and the Java Virtual Machine Profiler Interface (JVMPI). JVMTI contains all the functionality of both JVMDI and JVMPI, plus new functions. JVMTI was added as part of J2SE 5.0. In JDK 6, the JVMDI and JVMPI interfaces are no longer offered, and JVMTI is the only option available.

For more information about using JVMTI, see the JVMTI Reference page at the Oracle America, Inc. Web site.

Java Debug Wire Protocol

The Java Debug Wire Protocol (JDWP) is a defined communication protocol between a debugger process and the JVMDI/JVMTI. JDWP can be used from either a remote system or over a local socket. It is one layer removed from the JVMDI/JVMTI.

Start JDWP in QShell

To start JDWP and run the Java class SomeClass, enter the following command in QShell:

```
java -interpret -agentlib:jdwp=transport=dt_socket,
address=8000,server=y,suspend=n SomeClass
```

In this example, JDWP listens for connections from remote debuggers on TCP/IP port 8000, but you can use any port number you want; dt_socket is the name of the SRVPGM that handles the JDWP transport and does not change.

For additional options that you can use with -agentlib, see Sun VM Invocation Options by Oracle America, Inc.

Start JDWP from a CL command line

To start JDWP and run the Java class SomeClass, enter the following command:

```
JAVA CLASS(SomeClass) INTERPRET(*YES)
     PROP((os400.xrun.option 'jdwp:transport=dt_socket,address=8000,server=y,suspend=n'))
```

Java Debug Interface

Java Debug Interface (JDI) is a high-level Java language interface provided for tool development. JDI hides the complexity of JVMDI/JVMTI and JDWP behind some Java class definitions. JDI is included in the rt.jar file, so the front end of the debugger exists on any platform that has Java installed.

If you want to write debuggers for Java, you should use JDI because it is the simplest interface and your code is platform-independent.

For more information on JDPA, see Java Platform Debugger Architecture by Oracle America, Inc.

Related concepts

Debugging Java programs using IBM i Debugger

The easiest way to debug Java programs that run on your system is to use the IBM i Debugger. The IBM i Debugger provides a graphical user interface that enables you to more easily use the debugging capabilities of your system.

Finding memory leaks
If the performance of your program degrades as it runs for a longer period of time, you may have erroneously coded a memory leak.

Using the Generate JVM Dump command
The Generate JVM Dump (GENJVMDMP) CL command can be used to generate Java, system, and heap dumps.

Finding memory leaks

If the performance of your program degrades as it runs for a longer period of time, you may have erroneously coded a memory leak.

Memory leaks cause OutOfMemoryError exceptions or gradual loss of performance even if an OutOfMemory exception is never thrown. For more details on finding memory leaks see the Java Troubleshooting document in the IBM SDK for Java information center.

Related concepts

Debugging Java programs using IBM i Debugger
The easiest way to debug Java programs that run on your system is to use the IBM i Debugger. The IBM i Debugger provides a graphical user interface that enables you to more easily use the debugging capabilities of your system.

Java Platform Debugger Architecture
The Java Platform Debugger Architecture (JPDA) consists of the JVM Debug Interface/JVM Tool Interface, the Java Debug Wire Protocol, and the Java Debug Interface. All these parts of the JPDA enable any front end of a debugger that uses the JDWP to perform debugging operations. The debugger front end can either run remotely or run as a IBM i application.

Using the Generate JVM Dump command
The Generate JVM Dump (GENJVMDMP) CL command can be used to generate Java, system, and heap dumps.

Using the Generate JVM Dump command

The Generate JVM Dump (GENJVMDMP) CL command can be used to generate Java, system, and heap dumps.

The GENJVMDMP command generates Java Virtual Machine (JVM) dumps upon request. You can generate the following types of dumps:

***JAVA**
: Generates multiple files that contain diagnostic information for the JVM and the Java applications running within the JVM.

***SYSTEM**
: Generates a binary format raw memory image of the job that was running when the dump was initiated.

***HEAP**
: Generates a dump of all the heap space allocations which have not yet been freed.

Related concepts

Debugging Java programs using IBM i Debugger
The easiest way to debug Java programs that run on your system is to use the IBM i Debugger. The IBM i Debugger provides a graphical user interface that enables you to more easily use the debugging capabilities of your system.

Java Platform Debugger Architecture

The Java Platform Debugger Architecture (JPDA) consists of the JVM Debug Interface/JVM Tool Interface, the Java Debug Wire Protocol, and the Java Debug Interface. All these parts of the JPDA enable any front end of a debugger that uses the JDWP to perform debugging operations. The debugger front end can either run remotely or run as a IBM i application.

Finding memory leaks
If the performance of your program degrades as it runs for a longer period of time, you may have erroneously coded a memory leak.

Related information

Generate JVM Dump (GENJVMDMP) CL command

Java Code examples

The following is a list of Java code examples for the IBM i.

Internationalization

- “Example: Internationalization of dates using the java.util.DateFormat class” on page 373
- “Example: Internationalization of numeric display using the java.util.NumberFormat class” on page 373
- “Example: Internationalization of locale-specific data using the java.util.ResourceBundle class” on page 374

JDBC

- “Example: Access property” on page 375
- “Example: BLOB” on page 130
- “Example: CallableStatement interface for IBM Developer Kit for Java” on page 378
- “Example: Removing values from a table through another statement's cursor” on page 111
- “Example: CLOB” on page 134
- “Example: Creating a UDBDataSource and binding it with JNDI” on page 47
- “Example: Returning a list of tables using the DatabaseMetaData interface” on page 58
- “Example: Datalink” on page 138
- “Example: Distinct types” on page 139
- “Example: Embedding SQL Statements in your Java application” on page 175
- “Example: Ending a transaction” on page 80
- “Example: JDBC” on page 31
- “Example: Multiple connections that work on a transaction” on page 74
- “Example: ParameterMetaData” on page 89
- “Example: Changing values with a statement through another statement's cursor” on page 113
- “Example: ResultSet interface” on page 116
- “Example: ResultSet sensitivity” on page 104
- “Example: Sensitive and insensitive ResultSets” on page 103
- “Example: Setting up connection pooling with UDBDataSource and UDBConnectionPoolDataSource” on page 118
- “Example: SQLException” on page 62
- “Example: Suspending and resuming a transaction” on page 82
- “Example: Suspended ResultSets” on page 78
- “Example: Testing the performance of connection pooling” on page 119
- “Example: Testing the performance of two DataSources” on page 122
- “Example: Updating BLOBs” on page 132

- “Example: Updating CLOBs” on page 136
- “Example: Using a connection with multiple transactions” on page 76
- “Example: Using BLOBs” on page 133
- “Example: Using CLOBs” on page 137
- “Creating and populating a DB2CachedRowSet” on page 143
- “Example: Using JTA to handle a transaction” on page 72
- “Example: Using metadata ResultSets that have more than one column” on page 59
- “Example: Using native JDBC and IBM Toolbox for Java JDBC concurrently” on page 417
- “Example: Using PreparedStatement to obtain a ResultSet” on page 91
- “Example: Using the Statement object's executeUpdate method” on page 86

Java Authentication and Authorization Service

- “Examples: JAAS HelloWorld” on page 422
- “Example: JAAS SampleThreadSubjectLogin” on page 430

Java Generic Security Service

- “Sample: IBM JGSS non-JAAS client program” on page 438
- “Sample: IBM JGSS non-JAAS server program” on page 444
- “Sample: IBM JGSS JAAS-enabled client program” on page 454
- “Sample: IBM JGSS JAAS-enabled server program” on page 455

Java Secure Sockets Extension

- “Examples: IBM Java Secure Sockets Extension 1.4” on page 456

Java with other programming languages

- “Example: Calling a CL program with java.lang.Runtime.exec()” on page 217
- “Example: Calling a CL command with java.lang.Runtime.exec()” on page 218
- “Example: Calling another Java program with java.lang.Runtime.exec()” on page 216
- “Example: Calling Java from ILE C” on page 223
- “Example: Calling Java from RPG” on page 224
- “Example: Using input and output streams for interprocess communication” on page 222
- “Example: Java Invocation API” on page 213
- “Example: IBM PASE for i native method for Java” on page 207
- Sockets
- “Example: ILE native method for Java” on page 206

SQLJ

- “Example: Embedding SQL Statements in your Java application” on page 175

Secure sockets layer

- “Examples: Changing your Java code to use client socket factories” on page 261
- “Examples: Changing your Java code to use server socket factories” on page 259
- “Examples: Changing your Java client to use secure sockets layer” on page 264
- “Examples: Changing your Java server to use secure sockets layer” on page 262

INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OR CONDITIONS OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT, REGARDING THE PROGRAM OR TECHNICAL SUPPORT, IF ANY.

UNDER NO CIRCUMSTANCES IS IBM, ITS PROGRAM DEVELOPERS OR SUPPLIERS LIABLE FOR ANY OF THE FOLLOWING, EVEN IF INFORMED OF THEIR POSSIBILITY:

1. LOSS OF, OR DAMAGE TO, DATA;
2. DIRECT, SPECIAL, INCIDENTAL, OR INDIRECT DAMAGES, OR FOR ANY ECONOMIC CONSEQUENTIAL DAMAGES; OR
3. LOST PROFITS, BUSINESS, REVENUE, GOODWILL, OR ANTICIPATED SAVINGS.

SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF DIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, SO SOME OR ALL OF THE ABOVE LIMITATIONS OR EXCLUSIONS MAY NOT APPLY TO YOU.

Example: Internationalization of dates using the java.util.DateFormat class

This example shows how you can use locales to format dates.

Example 1: Demonstrates use of java.util.DateFormat class for internationalization of dates

Note: By using the code examples, you agree to the terms of the “Code license and disclaimer information” on page 489.

```
//************************
// File: DateExample.java
//************************

import java.text.*;
import java.util.*;
import java.util.Date;

public class DateExample {

   public static void main(String args[]) {

     // Get the Date
     Date now = new Date();

     // Get date formatters for default, German, and French locales
     DateFormat theDate = DateFormat.getDateInstance(DateFormat.LONG);
     DateFormat germanDate = DateFormat.getDateInstance(DateFormat.LONG, Locale.GERMANY);
     DateFormat frenchDate = DateFormat.getDateInstance(DateFormat.LONG, Locale.FRANCE);

     // Format and print the dates
     System.out.println("Date in the default locale: " + theDate.format(now));
     System.out.println("Date in the German locale : " + germanDate.format(now));
     System.out.println("Date in the French locale : " + frenchDate.format(now));
   }
}
```

“Examples: Creating an internationalized Java program” on page 26
If you need to customize a Java program for a specific region of the world, you can create an internationalized Java program with Java locales.

Example: Internationalization of numeric display using the java.util.NumberFormat class

This example shows how you can use locales to format numbers.

Example 1: Demonstrates use of java.util.NumberFormat class for internationalization of numeric output

Note: By using the code examples, you agree to the terms of the “Code license and disclaimer information” on page 489.

```
//**************************
// File: NumberExample.java
//**************************
```

```
import java.lang.*;
import java.text.*;
import java.util.*;

public class NumberExample {

    public static void main(String args[]) throws NumberFormatException {

      // The number to format
      double number = 12345.678;

      // Get formatters for default, Spanish, and Japanese locales
      NumberFormat defaultFormat = NumberFormat.getInstance();
      NumberFormat spanishFormat = NumberFormat.getInstance(new
Locale("es", "ES"));
      NumberFormat japaneseFormat = NumberFormat.getInstance(Locale.JAPAN);

      // Print out number in the default, Spanish, and Japanese formats
      // (Note: NumberFormat is not necessary for the default format)
      System.out.println("The number formatted for the default locale; " +
                         defaultFormat.format(number));
      System.out.println("The number formatted for the Spanish locale; " +
                         spanishFormat.format(number));
      System.out.println("The number formatted for the Japanese locale; " +
                         japaneseFormat.format(number));
    }
}
```

"Examples: Creating an internationalized Java program" on page 26
If you need to customize a Java program for a specific region of the world, you can create an internationalized Java program with Java locales.

Example: Internationalization of locale-specific data using the java.util.ResourceBundle class

This example shows how you can use locales with resource bundles to internationalize program strings.

These property files are required for the ResourceBundleExample program to work as intended:

Contents of RBExample.properties
Hello.text=Hello

Contents of RBExample_de.properties
Hello.text=Guten Tag

Contents of RBExample_fr_FR.properties
Hello.text=Bonjour

Example 1: Demonstrates use of java.util.ResourceBundle class for internationalization of locale-specific data

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
//*********************************
// File: ResourceBundleExample.java
//*********************************

import java.util.*;

public class ResourceBundleExample {
   public static void main(String args[]) throws MissingResourceException {

      String resourceName = "RBExample";
      ResourceBundle rb;

      // Default locale
      rb = ResourceBundle.getBundle(resourceName);
      System.out.println("Default : " + rb.getString("Hello" + ".text"));

      // Request a resource bundle with explicitly specified locale
      rb = ResourceBundle.getBundle(resourceName, Locale.GERMANY);
      System.out.println("German : " + rb.getString("Hello" + ".text"));
```

```
        // No property file for China in this example... use default
        rb = ResourceBundle.getBundle(resourceName, Locale.CHINA);
        System.out.println("Chinese : " + rb.getString("Hello" + ".text"));

        // Here is another way to do it...
        Locale.setDefault(Locale.FRANCE);
        rb = ResourceBundle.getBundle(resourceName);
        System.out.println("French : " + rb.getString("Hello" + ".text"));

        // No property file for China in this example... use default, which is now fr_FR.
        rb = ResourceBundle.getBundle(resourceName, Locale.CHINA);
        System.out.println("Chinese : " + rb.getString("Hello" + ".text"));
    }
}
```

"Examples: Creating an internationalized Java program" on page 26
If you need to customize a Java program for a specific region of the world, you can create an internationalized Java program with Java locales.

Example: Access property

This is an example of how to use the Java Access property.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
// This program assumes directory cujosql exists.
import java.sql.*;
import javax.sql.*;
import javax.naming.*;

public class AccessPropertyTest {
    public String url = "jdbc:db2:*local";
    public Connection connection = null;

    public static void main(java.lang.String[] args)
    throws Exception
    {
        AccessPropertyTest test = new AccessPropertyTest();

        test.setup();

        test.run();
        test.cleanup();
    }


/**
Set up the DataSource used in the testing.
**/
    public void setup()
    throws Exception
    {
        Class.forName("com.ibm.db2.jdbc.app.DB2Driver");

        connection = DriverManager.getConnection(url);
        Statement s = connection.createStatement();
        try {
            s.executeUpdate("DROP TABLE CUJOSQL.TEMP");
        } catch (SQLException e) { // Ignore it - it doesn't exist
        }

        try {
            String sql = "CREATE PROCEDURE CUJOSQL.TEMP "
                  + " LANGUAGE SQL SPECIFIC CUJOSQL.TEMP "
                  + " MYPROC: BEGIN"
                  + "   RETURN 11;"
                  + " END MYPROC";
            s.executeUpdate(sql);
        } catch (SQLException e) {
            // Ignore it - it exists.
        }
        s.executeUpdate("create table cujosql.temp (col1 char(10))");
        s.executeUpdate("insert into cujosql.temp values ('compare')");
        s.close();
    }
```

```
    public void resetConnection(String property)
    throws SQLException
    {
        if (connection != null)
            connection.close();

        connection = DriverManager.getConnection(url + ";access=" + property);
    }


    public boolean canQuery() {
        Statement s = null;
        try {
            s = connection.createStatement();
            ResultSet rs = s.executeQuery("SELECT * FROM cujosql.temp");
            if (rs == null)
                return false;

            rs.next();

            if (rs.getString(1).equals("compare   "))
                return true;

            return false;

        } catch (SQLException e) {
            // System.out.println("Exception: SQLState(" +
            //                    e.getSQLState() + ") " + e + " (" + e.getErrorCode() + ")");
            return false;
        } finally {
            if (s != null) {
                try {
                    s.close();
                } catch (Exception e) {
                    // Ignore it.
                }
            }
        }
    }


    public boolean canUpdate() {
        Statement s = null;
        try {
            s = connection.createStatement();
            int count = s.executeUpdate("INSERT INTO CUJOSQL.TEMP VALUES('x')");
            if (count != 1)
                return false;

            return true;

        } catch (SQLException e) {
            //System.out.println("Exception: SQLState(" +
            //                   e.getSQLState() + ") " + e + " (" + e.getErrorCode() + ")");
            return false;
        } finally {
            if (s != null) {
                try {
                    s.close();
                } catch (Exception e) {
                    // Ignore it.
                }
            }
        }
    }


    public boolean canCall() {
        CallableStatement s = null;
        try {
            s = connection.prepareCall("? = CALL CUJOSQL.TEMP()");
            s.registerOutParameter(1, Types.INTEGER);
            s.execute();
            if (s.getInt(1) != 11)
                return false;

            return true;

        } catch (SQLException e) {
            //System.out.println("Exception: SQLState(" +
            //                   e.getSQLState() + ") " + e + " (" + e.getErrorCode() + ")");
```

```
            return false;
        } finally {
            if (s != null) {
                try {
                    s.close();
                } catch (Exception e) {
                    // Ignore it.
                }
            }
        }
    }


    public void run()
    throws SQLException
    {
        System.out.println("Set the connection access property to read only");
        resetConnection("read only");

        System.out.println("Can run queries  -->" + canQuery());
        System.out.println("Can run updates  -->" + canUpdate());
        System.out.println("Can run sp calls -->" + canCall());

        System.out.println("Set the connection access property to read call");
        resetConnection("read call");

        System.out.println("Can run queries  -->" + canQuery());
        System.out.println("Can run updates  -->" + canUpdate());
        System.out.println("Can run sp calls -->" + canCall());

        System.out.println("Set the connection access property to all");
        resetConnection("all");

        System.out.println("Can run queries  -->" + canQuery());
        System.out.println("Can run updates  -->" + canUpdate());
        System.out.println("Can run sp calls -->" + canCall());

    }


    public void cleanup() {
        try {
            connection.close();
        } catch (Exception e) {
            // Ignore it.
        }
    }
}
```

Example: BLOB

This is an example of how a BLOB can be put into the database or retrieved from the database.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
/////////////////////////////////////////
// PutGetBlobs is an example application
// that shows how to work with the JDBC
// API to obtain and put BLOBs to and from
// database columns.
//
// The results of running this program
// are that there are two BLOB values
// in a new table. Both are identical
// and contain 500k of random byte
// data.
/////////////////////////////////////////
import java.sql.*;
import java.util.Random;

public class PutGetBlobs {
   public static void main(String[] args)
   throws SQLException
   {
       // Register the native JDBC driver.
       try {
          Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
      } catch (Exception e) {
```

```
            System.exit(1);  // Setup error.
        }

        // Establish a Connection and Statement with which to work.
        Connection c = DriverManager.getConnection("jdbc:db2:*local");
        Statement s = c.createStatement();

        // Clean up any previous run of this application.
        try {
            s.executeUpdate("DROP TABLE CUJOSQL.BLOBTABLE");
        } catch (SQLException e) {
            // Ignore it - assume the table did not exist.
        }

        // Create a table with a BLOB column. The default BLOB column
        // size is 1 MB.
        s.executeUpdate("CREATE TABLE CUJOSQL.BLOBTABLE (COL1 BLOB)");

        // Create a PreparedStatement object that allows you to put
        // a new Blob object into the database.
        PreparedStatement ps = c.prepareStatement("INSERT INTO CUJOSQL.BLOBTABLE VALUES(?)");

        // Create a big BLOB value...
        Random random = new Random ();
        byte [] inByteArray = new byte[500000];
        random.nextBytes (inByteArray);

        // Set the PreparedStatement parameter. Note: This is not
        // portable to all JDBC drivers. JDBC drivers do not have
        // support when using setBytes for BLOB columns. This is used to
        // allow you to generate new BLOBs. It also allows JDBC 1.0
        // drivers to work with columns containing BLOB data.
        ps.setBytes(1, inByteArray);

        // Process the statement, inserting the BLOB into the database.
        ps.executeUpdate();

        // Process a query and obtain the BLOB that was just inserted out
        // of the database as a Blob object.
        ResultSet rs = s.executeQuery("SELECT * FROM CUJOSQL.BLOBTABLE");
        rs.next();
        Blob blob = rs.getBlob(1);


        // Put that Blob back into the database through
        // the PreparedStatement.
        ps.setBlob(1, blob);
        ps.execute();

        c.close(); // Connection close also closes stmt and rs.
    }
}
```

Related reference

Example: Updating BLOBs
This is an example of how to update BLOBs in your Java applications.

Example: Using BLOBs
This is an example of how to use BLOBs in your Java applications.

Example: CallableStatement interface for IBM Developer Kit for Java

This is an example of how to use the CallableStatement interface.

Example: CallableStatement interface

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
// Connect to the server.
Connection c = DriverManager.getConnection("jdbc:db2://mySystem");

// Create the CallableStatement object.
// It precompiles the specified call to a stored procedure.
// The question marks indicate where input parameters must be set and
// where output parameters can be retrieved.
// The first two parameters are input parameters, and the third parameter is an output
```

```
parameter.
CallableStatement cs = c.prepareCall("CALL MYLIBRARY.ADD (?, ?, ?)");

// Set input parameters.
cs.setInt (1, 123);
cs.setInt (2, 234);

// Register the type of the output parameter.
cs.registerOutParameter (3, Types.INTEGER);

// Run the stored procedure.
cs.execute ();

// Get the value of the output parameter.
int sum = cs.getInt (3);

// Close the CallableStatement and the Connection.
cs.close();
c.close();
```

"CallableStatements" on page 93
The JDBC CallableStatement interface extends PreparedStatement and provides support for output and input/output parameters. The CallableStatement interface also has support for input parameters that is provided by the PreparedStatement interface.

Example: Removing values from a table through another statement's cursor

This Java example shows how to remove values from a table through another statement's cursor.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
import java.sql.*;

public class UsingPositionedDelete {
    public Connection connection = null;
    public static void main(java.lang.String[] args) {
        UsingPositionedDelete test = new UsingPositionedDelete();

        test.setup();
        test.displayTable();

        test.run();
        test.displayTable();

        test.cleanup();
    }



/**
Handle all the required setup work.
**/
    public void setup() {
        try {
            // Register the JDBC driver.
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");

            connection = DriverManager.getConnection("jdbc:db2:*local");

            Statement s = connection.createStatement();
            try {
                s.executeUpdate("DROP TABLE CUJOSQL.WHERECUREX");
            } catch (SQLException e) {
                // Ignore problems here.
            }

            s.executeUpdate("CREATE TABLE CUJOSQL.WHERECUREX ( " +
                            "COL_IND INT, COL_VALUE CHAR(20)) ");

            for (int i = 1; i <= 10; i++) {
                s.executeUpdate("INSERT INTO CUJOSQL.WHERECUREX VALUES(" + i + ", 'FIRST')");
            }

            s.close();

        } catch (Exception e) {
```

```
            System.out.println("Caught exception: " + e.getMessage());
            e.printStackTrace();
        }
    }


/**
In this section, all the code to perform the testing should
be added. If only one connection to the database is needed,
the global variable 'connection' can be used.
**/
    public void run() {
        try {
            Statement stmt1 = connection.createStatement();

            // Update each value using next().
            stmt1.setCursorName("CUJO");
            ResultSet rs = stmt1.executeQuery ("SELECT * FROM CUJOSQL.WHERECUREX " +
                                               "FOR UPDATE OF COL_VALUE");

            System.out.println("Cursor name is " + rs.getCursorName());

            PreparedStatement stmt2 = connection.prepareStatement
                                      ("DELETE FROM " + " CUJOSQL.WHERECUREX WHERE CURRENT OF "
+
                                      rs.getCursorName ());


            // Loop through the ResultSet and update every other entry.
            while (rs.next ()) {
                if (rs.next())
                    stmt2.execute ();
            }


            // Clean up the resources after they have been used.
            rs.close ();
            stmt2.close ();


        } catch (Exception e) {
            System.out.println("Caught exception: ");
            e.printStackTrace();
        }
    }



/**
In this section, put all clean-up work for testing.
**/
    public void cleanup() {
        try {
            // Close the global connection opened in setup().
            connection.close();

        } catch (Exception e) {
            System.out.println("Caught exception: ");
            e.printStackTrace();
        }
    }



/**
Display the contents of the table.
**/
    public void displayTable()
    {
        try {
            Statement s = connection.createStatement();
            ResultSet rs = s.executeQuery ("SELECT * FROM CUJOSQL.WHERECUREX");

            while (rs.next ()) {
                System.out.println("Index " + rs.getInt(1) + " value " + rs.getString(2));
            }

            rs.close ();
            s.close();
            System.out.println("----------------------------------------");
        } catch (Exception e) {
            System.out.println("Caught exception: ");
```

```
            e.printStackTrace();
        }
    }
}
```

Example: CLOB

This is an example of how a CLOB can be put into the database or retrieved from the database.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
/////////////////////////////////////////
// PutGetClobs is an example application
// that shows how to work with the JDBC
// API to obtain and put CLOBs to and from
// database columns.
//
// The results of running this program
// are that there are two CLOB values
// in a new table. Both are identical
// and contain about 500k of repeating
// text data.
/////////////////////////////////////////
import java.sql.*;

public class PutGetClobs {
   public static void main(String[] args)
   throws SQLException
   {
       // Register the native JDBC driver.
       try {
          Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
      } catch (Exception e) {
          System.exit(1);  // Setup error.
      }

      // Establish a Connection and Statement with which to work.
      Connection c = DriverManager.getConnection("jdbc:db2:*local");
      Statement s = c.createStatement();

      // Clean up any previous run of this application.
      try {
          s.executeUpdate("DROP TABLE CUJOSQL.CLOBTABLE");
      } catch (SQLException e) {
          // Ignore it - assume the table did not exist.
      }

      // Create a table with a CLOB column. The default CLOB column
      // size is 1 MB.
      s.executeUpdate("CREATE TABLE CUJOSQL.CLOBTABLE (COL1 CLOB)");

      // Create a PreparedStatement object that allow you to put
      // a new Clob object into the database.
      PreparedStatement ps = c.prepareStatement("INSERT INTO CUJOSQL.CLOBTABLE VALUES(?)");

      // Create a big CLOB value...
      StringBuffer buffer = new StringBuffer(500000);
      while (buffer.length() < 500000) {
          buffer.append("All work and no play makes Cujo a dull boy.");
      }
      String clobValue = buffer.toString();

      // Set the PreparedStatement parameter. This is not
      // portable to all JDBC drivers. JDBC drivers do not have
      // to support setBytes for CLOB columns. This is done to
      // allow you to generate new CLOBs. It also
      // allows JDBC 1.0 drivers a way to work with columns containing
      // Clob data.
      ps.setString(1, clobValue);

      // Process the statement, inserting the clob into the database.
      ps.executeUpdate();

      // Process a query and get the CLOB that was just inserted out of the
      // database as a Clob object.
      ResultSet rs = s.executeQuery("SELECT * FROM CUJOSQL.CLOBTABLE");
      rs.next();
      Clob clob = rs.getClob(1);
```

```
        // Put that Clob back into the database through
        // the PreparedStatement.
        ps.setClob(1, clob);
        ps.execute();

        c.close(); // Connection close also closes stmt and rs.
    }
}
```

Related reference

Example: Updating CLOBs
This is an example of how to update CLOBs in your Java applications.

Example: Using CLOBs
This is an example of how to use CLOBs in your Java applications.

Example: Returning a list of tables using the DatabaseMetaData interface

This example shows how to return a list of tables.

Note: By using the code examples, you agree to the terms of the “Code license and disclaimer information” on page 489.

```
// Connect to the server.
Connection c = DriverManager.getConnection("jdbc:db2:mySystem");

// Get the database meta data from the connection.
DatabaseMetaData dbMeta = c.getMetaData();

// Get a list of tables matching this criteria.
String catalog = "myCatalog";
String schema = "mySchema";
String table  = "myTable%"; // % indicates search pattern
String types[] = {"TABLE", "VIEW", "SYSTEM TABLE"}:
ResultSet rs = dbMeta.getTables(catalog, schema, table, types);

// ... iterate through the ResultSet to get the values.

// Close the connection.
c.close():
```

Related reference

Example: Using metadata ResultSets that have more than one column
This is an example of how to use metadata ResultSets that have more than one column.

Example: Datalink

This example application shows how to use the JDBC API to handle datalink database columns.

Note: By using the code examples, you agree to the terms of the “Code license and disclaimer information” on page 489.

```
/////////////////////////////////////////
// PutGetDatalinks is an example application
// that shows how to use the JDBC
// API to handle datalink database columns.
/////////////////////////////////////////
import java.sql.*;
import java.net.URL;
import java.net.MalformedURLException;

public class PutGetDatalinks {
    public static void main(String[] args)
    throws SQLException
    {
        // Register the native JDBC driver.
        try {
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
        } catch (Exception e) {
            System.exit(1);  // Setup error.
        }
```

```
        // Establish a Connection and Statement with which to work.
        Connection c = DriverManager.getConnection("jdbc:db2:*local");
        Statement s = c.createStatement();

        // Clean up any previous run of this application.
        try {
            s.executeUpdate("DROP TABLE CUJOSQL.DLTABLE");
        } catch (SQLException e) {
            // Ignore it - assume the table did not exist.
        }

        // Create a table with a datalink column.
        s.executeUpdate("CREATE TABLE CUJOSQL.DLTABLE (COL1 DATALINK)");

        // Create a PreparedStatement object that allows you to add
        // a new datalink into the database. Since conversing
        // to a datalink cannot be accomplished directly in the database, you
        // can code the SQL statement to perform the explicit conversion.
        PreparedStatement ps = c.prepareStatement("INSERT INTO CUJOSQL.DLTABLE
                                                   VALUES(DLVALUE( CAST(? AS VARCHAR(100))))");

        // Set the datalink. This URL points you to a topic about
        // the new features of JDBC 3.0.
        ps.setString (1, "http://www.ibm.com/developerworks/java/library/j-jdbcnew/index.html");

        // Process the statement, inserting the CLOB into the database.
        ps.executeUpdate();

        // Process a query and obtain the CLOB that was just inserted out of the
        // database as a Clob object.
        ResultSet rs = s.executeQuery("SELECT * FROM CUJOSQL.DLTABLE");
        rs.next();
        String datalink = rs.getString(1);


        // Put that datalink value into the database through
        // the PreparedStatement. Note: This function requires JDBC 3.0
        // support.
        /*
        try {
            URL url = new URL(datalink);
            ps.setURL(1, url);
            ps.execute();
        } catch (MalformedURLException mue) {
            // Handle this issue here.
        }

        rs = s.executeQuery("SELECT * FROM CUJOSQL.DLTABLE");
        rs.next();
        URL url = rs.getURL(1);
        System.out.println("URL value is " + url);
        */

        c.close(); // Connection close also closes stmt and rs.
    }
}
```

Example: Distinct types

This is an example of how to use distinct types.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
/////////////////////////////////////////
// This example program shows examples of
// various common tasks that can be done
// with distinct types.
/////////////////////////////////////////
import java.sql.*;

public class Distinct {
    public static void main(String[] args)
    throws SQLException
    {
        // Register the native JDBC driver.
        try {
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
```

```
        } catch (Exception e) {
            System.exit(1);  // Setup error.
        }

        Connection c = DriverManager.getConnection("jdbc:db2:*local");
        Statement s = c.createStatement();

        // Clean up any old runs.
        try {
            s.executeUpdate("DROP TABLE CUJOSQL.SERIALNOS");
        } catch (SQLException e) {
            // Ignore it and assume the table did not exist.
        }

        try {
            s.executeUpdate("DROP DISTINCT TYPE CUJOSQL.SSN");
        } catch (SQLException e) {
            // Ignore it and assume the table did not exist.
        }

        // Create the type, create the table, and insert a value.
        s.executeUpdate("CREATE DISTINCT TYPE CUJOSQL.SSN AS CHAR(9)");
        s.executeUpdate("CREATE TABLE CUJOSQL.SERIALNOS (COL1 CUJOSQL.SSN)");

        PreparedStatement ps = c.prepareStatement("INSERT INTO CUJOSQL.SERIALNOS VALUES(?)");
        ps.setString(1, "399924563");
        ps.executeUpdate();
        ps.close();

        // You can obtain details about the types available with new metadata in
        // JDBC 2.0
        DatabaseMetaData dmd = c.getMetaData();

        int types[] = new int[1];
        types[0] = java.sql.Types.DISTINCT;

        ResultSet rs = dmd.getUDTs(null, "CUJOSQL", "SSN", types);
        rs.next();
        System.out.println("Type name " + rs.getString(3) +
                           " has type " + rs.getString(4));


        // Access the data you have inserted.
        rs = s.executeQuery("SELECT COL1 FROM CUJOSQL.SERIALNOS");
        rs.next();
        System.out.println("The SSN is " + rs.getString(1));

        c.close(); // Connection close also closes stmt and rs.
    }
}
```

Related concepts

Writing code that uses BLOBs
There are a number of tasks that can be accomplished with database Binary Large Object (BLOB) columns through the Java Database Connectivity (JDBC) Application Programming Interface (API). The following topics briefly discuss these tasks and include examples on how to accomplish them.

Writing code that uses CLOBs
There are a number of tasks that can be performed with database CLOB and DBCLOB columns through the Java Database Connectivity (JDBC) Application Programming Interface (API). The following topics briefly discuss these tasks and include examples on how to accomplish them.

Writing code that uses Datalinks

How you work with Datalinks is dependent on what release you are working with. In JDBC 3.0, there is support to work directly with Datalink columns using the getURL and putURL methods.

Example: Embedding SQL Statements in your Java application

The following example SQLJ application, App.sqlj, uses static SQL to retrieve and update data from the EMPLOYEE table of the DB2 sample database.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
import java.sql.*;
import sqlj.runtime.*;
import sqlj.runtime.ref.*;

#sql iterator App_Cursor1 (String empno, String firstnme) ; // 1
#sql iterator App_Cursor2 (String) ;

class App
{

  /**********************
   **  Register Driver **
   **********************/

 static
 {
   try
   {
     Class.forName("com.ibm.db2.jdbc.app.DB2Driver").newInstance();
   }
   catch (Exception e)
   {
     e.printStackTrace();
   }
 }

  /********************
   **      Main      **
   ********************/

 public static void main(String argv[])
 {
   try
   {
     App_Cursor1 cursor1;
     App_Cursor2 cursor2;

     String str1 = null;
     String str2 = null;
     long   count1;

     // URL is jdbc:db2:dbname
     String url = "jdbc:db2:sample";

     DefaultContext ctx = DefaultContext.getDefaultContext();
     if (ctx == null)
     {
       try
       {
         // connect with default id/password
         Connection con = DriverManager.getConnection(url);
         con.setAutoCommit(false);
         ctx = new DefaultContext(con);
       }
       catch (SQLException e)
       {
         System.out.println("Error: could not get a default context");
         System.err.println(e) ;
         System.exit(1);
       }
       DefaultContext.setDefaultContext(ctx);
     }

     // retrieve data from the database
     System.out.println("Retrieve some data from the database.");
     #sql cursor1 = {SELECT empno, firstnme FROM employee}; // 2
```

```
      // display the result set
      // cursor1.next() returns false when there are no more rows
      System.out.println("Received results:");
      while (cursor1.next()) // 3
      {
        str1 = cursor1.empno(); // 4
        str2 = cursor1.firstnme();

        System.out.print (" empno= " + str1);
        System.out.print (" firstname= " + str2);
        System.out.println("");
      }
      cursor1.close(); // 9

      // retrieve number of employee from the database
      #sql { SELECT count(*) into :count1 FROM employee }; // 5
      if (1 == count1)
        System.out.println ("There is 1 row in employee table");
      else
        System.out.println ("There are " + count1
                             + " rows in employee table");

      // update the database
      System.out.println("Update the database.");
      #sql { UPDATE employee SET firstnme = 'SHILI' WHERE empno = '000010' };

      // retrieve the updated data from the database
      System.out.println("Retrieve the updated data from the database.");
      str1 = "000010";
      #sql cursor2 = {SELECT firstnme FROM employee WHERE empno = :str1}; // 6

      // display the result set
      // cursor2.next() returns false when there are no more rows
      System.out.println("Received results:");
      while (true)
      {
        #sql { FETCH :cursor2 INTO :str2 }; // 7
        if (cursor2.endFetch()) break; // 8

        System.out.print (" empno= " + str1);
        System.out.print (" firstname= " + str2);
        System.out.println("");
      }
      cursor2.close(); // 9

      // rollback the update
      System.out.println("Rollback the update.");
      #sql { ROLLBACK work };
      System.out.println("Rollback done.");
    }
    catch( Exception e )
    {
      e.printStackTrace();
    }
  }
}
```

[1]Declare iterators. This section declares two types of iterators:

- App_Cursor1: Declares column data types and names, and returns the values of the columns according to column name (Named binding to columns).
- App_Cursor2: Declares column data types, and returns the values of the columns by column position (Positional binding to columns).

[2]Initialize the iterator. The iterator object cursor1 is initialized using the result of a query. The query stores the result in cursor1.

[3]Advance the iterator to the next row. The cursor1.next() method returns a Boolean `false` if there are no more rows to retrieve.

[4]Move the data. The named accessor method empno() returns the value of the column named empno on the current row. The named accessor method firstnme() returns the value of the column named firstnme on the current row.

[5]SELECT data into a host variable. The SELECT statement passes the number of rows in the table into the host variable count1.

[6] Initialize the iterator. The iterator object cursor2 is initialized using the result of a query. The query stores the result in cursor2.

[7]Retrieve the data. The FETCH statement returns the current value of the first column declared in the ByPos cursor from the result table into the host variable str2.

[8]Check the success of a FETCH.INTO statement. The endFetch() method returns a Boolean true if the iterator is not positioned on a row, that is, if the last attempt to fetch a row failed. The endFetch() method returns false if the last attempt to fetch a row was successful. DB2 attempts to fetch a row when the next() method is called. A FETCH...INTO statement implicitly calls the next() method.

[9]Close the iterators. The close() method releases any resources held by the iterators. You should explicitly close iterators to ensure that system resources are released in a timely fashion.

Example: Ending a transaction

This is an example of ending a transaction in your application.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
import java.sql.*;
import javax.sql.*;
import java.util.*;
import javax.transaction.*;
import javax.transaction.xa.*;
import com.ibm.db2.jdbc.app.*;

public class JTATxEnd {

    public static void main(java.lang.String[] args) {
        JTATxEnd test = new JTATxEnd();

        test.setup();
        test.run();
    }


    /**
     * Handle the previous cleanup run so that this test can recommence.
     */
    public void setup() {

        Connection c = null;
        Statement s = null;
        try {
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
            c = DriverManager.getConnection("jdbc:db2:*local");
            s = c.createStatement();

            try {
                s.executeUpdate("DROP TABLE CUJOSQL.JTATABLE");
            } catch (SQLException e) {
                // Ignore... does not exist
            }

            s.executeUpdate("CREATE TABLE CUJOSQL.JTATABLE (COL1 CHAR (50))");
            s.executeUpdate("INSERT INTO CUJOSQL.JTATABLE VALUES('Fun with JTA')");
            s.executeUpdate("INSERT INTO CUJOSQL.JTATABLE VALUES('JTA is fun.)");

            s.close();
        } finally {
            if (c != null) {
                c.close();
            }
        }
    }


    /**
     * This test use JTA support to handle transactions.
     */
    public void run() {
        Connection c = null;

        try {
```

```
            Context ctx = new InitialContext();

            // Assume the data source is backed by a UDBXADataSource.
            UDBXADataSource ds = (UDBXADataSource) ctx.lookup("XADataSource");

            // From the DataSource, obtain an XAConnection object that
            // contains an XAResource and a Connection object.
            XAConnection  xaConn = ds.getXAConnection();
            XAResource    xaRes  = xaConn.getXAResource();
            Connection    c      = xaConn.getConnection();

            // For XA transactions, transaction identifier is required.
            // An implementation of the XID interface is not included
            // with the JDBC driver. See Transactions with JTA for a
            // description of this interface to build a class for it.
            Xid xid = new XidImpl();

            // The connection from the XAResource can be used as any other
            // JDBC connection.
            Statement stmt = c.createStatement();

            // The XA resource must be notified before starting any
            // transactional work.
            xaRes.start(xid, XAResource.TMNOFLAGS);

            // Create a ResultSet during JDBC processing and fetch a row.
            ResultSet rs = stmt.executeUpdate("SELECT * FROM CUJOSQL.JTATABLE");
            rs.next();

            // When the end method is called, all ResultSet cursors close.
            // Accessing the ResultSet after this point results in an
            // exception being thrown.
            xaRes.end(xid, XAResource.TMNOFLAGS);

            try {
                String value = rs.getString(1);
                System.out.println("Something failed if you receive this message.");
            } catch (SQLException e) {
                System.out.println("The expected exception was thrown.");
            }

            // Commit the transaction to ensure that all locks are
            // released.
            int rc = xaRes.prepare(xid);
            xaRes.commit(xid, false);


        } catch (Exception e) {
            System.out.println("Something has gone wrong.");
            e.printStackTrace();
        } finally {
            try {
                if (c != null)
                    c.close();
            } catch (SQLException e) {
                System.out.println("Note:  Cleaup exception.");
                e.printStackTrace();
            }
        }
    }
}
```

Related reference

Example: Using JTA to handle a transaction
This is an example of how to use the Java Transaction API (JTA) to handle a transaction in an application.

Example: Multiple connections that work on a transaction
This is an example of how to use multiple connections working on a single transaction.

Example: Using a connection with multiple transactions
This is an example of how to use a single connection with multiple transactions.

Example: Suspended ResultSets
This is an example of the how a Statement object is reprocessed under another transaction to perform work.

Example: Suspending and resuming a transaction

This is an example of a transaction that is suspended and then is resumed.

"JDBC distributed transactions" on page 70
Typically, transactions in Java Database Connectivity (JDBC) are local. This means that a single connection performs all the work of the transaction and that the connection can only work on one transaction at a time.

Example: JDBC

This is an example of how to use the BasicJDBC program. This program uses the native JDBC driver for the IBM Developer Kit for Java to build a simple table and process a query that displays the data in that table.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
//////////////////////////////////////////////////////////////////////////////////
//
// BasicJDBC example.  This program uses the native JDBC driver for the
// Developer Kit for Java to build a simple table and process a query
// that displays the data in that table.
//
// Command syntax:
//    BasicJDBC
//
//////////////////////////////////////////////////////////////////////////////////
//
// This source is an example of the native JDBC driver.
// IBM grants you a nonexclusive license to use this as an example
// from which you can generate similar function tailored to
// your own specific needs.
//
// This sample code is provided by IBM for illustrative purposes
// only. These examples have not been thoroughly tested under all
// conditions. IBM, therefore, cannot guarantee or imply
// reliability, serviceability, or function of these programs.
//
// All programs contained herein are provided to you "AS IS"
// without any warranties of any kind.  The implied warranties of
// merchantability and fitness for a particular purpose are
// expressly disclaimed.
//
// IBM Developer Kit for Java
// (C) Copyright IBM Corp. 2001
// All rights reserved.
// US Government Users Restricted Rights -
// Use, duplication, or disclosure restricted
// by GSA ADP Schedule Contract with IBM Corp.
//
//////////////////////////////////////////////////////////////////////////////////

// Include any Java classes that are to be used. In this application,
// many classes from the java.sql package are used and the
// java.util.Properties class is also used as part of obtaining
// a connection to the database.
import java.sql.*;
import java.util.Properties;

// Create a public class to encapsulate the program.
public class BasicJDBC {

    // The connection is a private variable of the object.
    private Connection connection = null;

    // Any class that is to be an 'entry point' for running
    // a program must have a main method. The main method
    // is where processing begins when the program is called.
    public static void main(java.lang.String[] args) {

        // Create an object of type BasicJDBC. This
        // is fundamental to object-oriented programming. Once
        // an object is created, call various methods on
        // that object to accomplish work.
        // In this case, calling the constructor for the object
        // creates a database connection that the other
        // methods use to do work against the database.
        BasicJDBC test = new BasicJDBC();
```

```
        // Call the rebuildTable method. This method ensures that
        // the table used in this program exists and looks
        // correct. The return value is a boolean for
        // whether or not rebuilding the table completed
        // successfully. If it did no, display a message
        // and exit the program.
        if (!test.rebuildTable()) {
            System.out.println("Failure occurred while setting up " +
                               " for running the test.");
            System.out.println("Test will not continue.");
            System.exit(0);
        }

        // The run query method is called next. This method
        // processes an SQL select statement against the table that
        // was created in the rebuildTable method. The output of
        // that query is output to standard out for you to view.
        test.runQuery();

        // Finally, the cleanup method is called. This method
        // ensures that the database connection that the object has
        // been hanging on to is closed.
        test.cleanup();
    }


    /**
    This is the constructor for the basic JDBC test. It creates a database
    connection that is stored in an instance variable to be used in later
    method calls.
    **/
    public BasicJDBC() {

        // One way to create a database connection is to pass a URL
        // and a java Properties object to the DriverManager. The following
        // code constructs a Properties object that has your user ID and
        // password. These pieces of information are used for connecting
        // to the database.
        Properties properties = new Properties ();
        properties.put("user", "cujo");
        properties.put("password", "newtiger");

        // Use a try/catch block to catch all exceptions that can come out of the
        // following code.
        try {
            // The DriverManager must be aware that there is a JDBC driver available
            // to handle a user connection request. The following line causes the
            // native JDBC driver to be loaded and registered with the DriverManager.
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");

            // Create the database Connection object that this program uses in all
            // the other method calls that are made. The following code specifies
            // that a connection is to be established to the local database and that
            // that connection should conform to the properties that were set up
            // previously (that is, it should use the user ID and password specified).
            connection = DriverManager.getConnection("jdbc:db2:*local", properties);

        } catch (Exception e) {
            // If any of the lines in the try/catch block fail, control transfers to
            // the following line of code. A robust application tries to handle the
            // problem or provide more details to you. In this program, the error
            // message from the exception is displayed and the application allows
            // the program to return.
            System.out.println("Caught exception: " + e.getMessage());
        }
    }


    /**
    Ensures that the qgpl.basicjdbc table looks you want it to at the start of
    the test.

    @returns boolean    Returns true if the table was rebuild successfully;
                        returns false if any failure occurred.
    **/
    public boolean rebuildTable() {
        // Wrap all the functionality in a try/catch block so an attempt is
        // made to handle any errors that may happen within this method.
        try {

            // Statement objects are used to process SQL statements against the
```

```
            // database. The Connection object is used to create a Statement
            // object.
            Statement s = connection.createStatement();

            try {
                // Build the test table from scratch. Process an update statement
                // that attempts to delete the table if it currently exists.
                s.executeUpdate("drop table qgpl.basicjdbc");
            } catch (SQLException e) {
                // Do not perform anything if an exception occurred. Assume
                // that the problem is that the table that was dropped does not
                // exist and that it can be created next.
            }

            // Use the statement object to create our table.
            s.executeUpdate("create table qgpl.basicjdbc(id int, name char(15))");

            // Use the statement object to populate our table with some data.
            s.executeUpdate("insert into qgpl.basicjdbc values(1, 'Frank Johnson')");
            s.executeUpdate("insert into qgpl.basicjdbc values(2, 'Neil Schwartz')");
            s.executeUpdate("insert into qgpl.basicjdbc values(3, 'Ben Rodman')");
            s.executeUpdate("insert into qgpl.basicjdbc values(4, 'Dan Gloore')");

            // Close the SQL statement to tell the database that it is no longer
            // needed.
            s.close();

            // If the entire method processed successfully, return true. At this point,
            // the table has been created or refreshed correctly.
            return true;

        } catch (SQLException sqle) {
            // If any of our SQL statements failed (other than the drop of the table
            // that was handled in the inner try/catch block), the error message is
            // displayed and false is returned to the caller, indicating that the table
            // may not be complete.
            System.out.println("Error in rebuildTable: " + sqle.getMessage());
            return false;
        }
    }



    /**
    Runs a query against the demonstration table and the results are displayed to
    standard out.
    **/
    public void runQuery() {
        // Wrap all the functionality in a try/catch block so an attempts is
        // made to handle any errors that might happen within this
        // method.
        try {
            // Create a Statement object.
            Statement s = connection.createStatement();

            // Use the statement object to run an SQL query. Queries return
            // ResultSet objects that are used to look at the data the query
            // provides.
            ResultSet rs = s.executeQuery("select * from qgpl.basicjdbc");

            // Display the top of our 'table' and initialize the counter for the
            // number of rows returned.
            System.out.println("--------------------");
            int i = 0;

            // The ResultSet next method is used to process the rows of a
            // ResultSet. The next method must be called once before the
            // first data is available for viewing. As long as next returns
            // true, there is another row of data that can be used.
            while (rs.next()) {

                // Obtain both columns in the table for each row and write a row to
                // our on-screen table with the data. Then, increment the count
                // of rows that have been processed.
                System.out.println("| " + rs.getInt(1) + " | " + rs.getString(2) + "|");
                i++;
            }

            // Place a border at the bottom on the table and display the number of rows
            // as output.
            System.out.println("--------------------");
            System.out.println("There were " + i + " rows returned.");
```

```
            System.out.println("Output is complete.");

        } catch (SQLException e) {
            // Display more information about any SQL exceptions that are
            // generated as output.
            System.out.println("SQLException exception: ");
            System.out.println("Message:....." + e.getMessage());
            System.out.println("SQLState:...." + e.getSQLState());
            System.out.println("Vendor Code:." + e.getErrorCode());
            e.printStackTrace();
        }
    }


    /**
    The following method ensures that any JDBC resources that are still
    allocated are freed.
    **/
    public void cleanup() {
        try {
            if (connection != null)
                connection.close();
        } catch (Exception e) {
            System.out.println("Caught exception: ");
            e.printStackTrace();
        }
    }
}
```

Example: Multiple connections that work on a transaction

This is an example of how to use multiple connections working on a single transaction.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
import java.sql.*;
import javax.sql.*;
import java.util.*;
import javax.transaction.*;
import javax.transaction.xa.*;
import com.ibm.db2.jdbc.app.*;
public class JTAMultiConn {
   public static void main(java.lang.String[] args) {
      JTAMultiConn test = new JTAMultiConn();
      test.setup();
      test.run();
   }
/**
* Handle the previous cleanup run so that this test can recommence.
*/
   public void setup() {
      Connection c = null;
      Statement s = null;
      try {
         Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
         c = DriverManager.getConnection("jdbc:db2:*local");
         s = c.createStatement();
         try {
            s.executeUpdate("DROP TABLE CUJOSQL.JTATABLE");
         }
         catch (SQLException e) {
         // Ignore... does not exist
         }
         s.executeUpdate("CREATE TABLE CUJOSQL.JTATABLE (COL1 CHAR
                         (50))");
               s.close();
      }
      finally {
         if (c != null) {
            c.close();
         }
      }
   }
/**
* This test uses JTA support to handle transactions.
*/
   public void run() {
      Connection c1 = null;
```

```
      Connection c2 = null;
      Connection c3 = null;
      try {
         Context ctx = new InitialContext();
         // Assume the data source is backed by a UDBXADataSource.
         UDBXADataSource ds = (UDBXADataSource)
                              ctx.lookup("XADataSource");
         // From the DataSource, obtain some XAConnection objects that
         // contain an XAResource and a Connection object.
         XAConnection xaConn1 = ds.getXAConnection();
         XAConnection xaConn2 = ds.getXAConnection();
         XAConnection xaConn3 = ds.getXAConnection();
         XAResource xaRes1 = xaConn1.getXAResource();
         XAResource xaRes2 = xaConn2.getXAResource();
         XAResource xaRes3 = xaConn3.getXAResource();
         c1 = xaConn1.getConnection();
         c2 = xaConn2.getConnection();
         c3 = xaConn3.getConnection();
         Statement stmt1 = c1.createStatement();
         Statement stmt2 = c2.createStatement();
         Statement stmt3 = c3.createStatement();
         // For XA transactions, a transaction identifier is required.
         // Support for creating XIDs is again left to the application
         // program.
         Xid xid = JDXATest.xidFactory();
         // Perform some transactional work under each of the three
         // connections that have been created.
         xaRes1.start(xid, XAResource.TMNOFLAGS);
         int count1 = stmt1.executeUpdate("INSERT INTO " + tableName + "VALUES('Value 1-A')");
         xaRes1.end(xid, XAResource.TMNOFLAGS);

         xaRes2.start(xid, XAResource.TMJOIN);
         int count2 = stmt2.executeUpdate("INSERT INTO " + tableName + "VALUES('Value 1-B')");
         xaRes2.end(xid, XAResource.TMNOFLAGS);

         xaRes3.start(xid, XAResource.TMJOIN);
         int count3 = stmt3.executeUpdate("INSERT INTO " + tableName + "VALUES('Value 1-C')");
         xaRes3.end(xid, XAResource.TMSUCCESS);
         // When completed, commit the transaction as a single unit.
         // A prepare() and commit() or 1 phase commit() is required for
         // each separate database (XAResource) that participated in the
         // transaction. Since the resources accessed (xaRes1, xaRes2, and xaRes3)
         // all refer to the same database, only one prepare or commit is required.
         int rc = xaRes.prepare(xid);
         xaRes.commit(xid, false);
      }
      catch (Exception e) {
         System.out.println("Something has gone wrong.");
         e.printStackTrace();
      }
      finally {
         try {
            if (c1 != null) {
               c1.close();
            }
         }
         catch (SQLException e) {
            System.out.println("Note: Cleaup exception " +
                               e.getMessage());
         }
         try {
            if (c2 != null) {
               c2.close();
            }
         }
         catch (SQLException e) {
            System.out.println("Note: Cleaup exception " +
                               e.getMessage());
         }
         try {
            if (c3 != null) {
               c3.close();
            }
         }
         catch (SQLException e) {
            System.out.println("Note: Cleaup exception " +
                               e.getMessage());
         }
      }
   }
}
```

Related reference

Example: Using JTA to handle a transaction
This is an example of how to use the Java Transaction API (JTA) to handle a transaction in an application.

Example: Using a connection with multiple transactions
This is an example of how to use a single connection with multiple transactions.

Example: Suspended ResultSets
This is an example of the how a Statement object is reprocessed under another transaction to perform work.

Example: Ending a transaction
This is an example of ending a transaction in your application.

Example: Suspending and resuming a transaction
This is an example of a transaction that is suspended and then is resumed.

Example: ParameterMetaData

This is an example of using the ParameterMetaData interface to retrieve information about parameters.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
//////////////////////////////////////////////////////////////////////////////////
//
// ParameterMetaData example. This program demonstrates
// the new support of JDBC 3.0 for learning information
// about parameters to a PreparedStatement.
//
// Command syntax:
//    java PMD
//
//////////////////////////////////////////////////////////////////////////////////
//
// This source is an example of the IBM Developer for Java JDBC driver.
// IBM grants you a nonexclusive license to use this as an example
// from which you can generate similar function tailored to
// your own specific needs.
//
// This sample code is provided by IBM for illustrative purposes
// only. These examples have not been thoroughly tested under all
// conditions. IBM, therefore, cannot guarantee or imply
// reliability, serviceability, or function of these programs.
//
// All programs contained herein are provided to you "AS IS"
// without any warranties of any kind.  The implied warranties of
// merchantability and fitness for a particular purpose are
// expressly disclaimed.
//
// IBM Developer Kit for Java
// (C) Copyright IBM Corp. 2001
// All rights reserved.
// US Government Users Restricted Rights -
// Use, duplication, or disclosure restricted
// by GSA ADP Schedule Contract with IBM Corp.
//
//////////////////////////////////////////////////////////////////////////////////

import java.sql.*;

public class PMD {

    // Program entry point.
    public static void main(java.lang.String[] args)
    throws Exception
    {
        // Obtain setup.
        Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
        Connection c = DriverManager.getConnection("jdbc:db2:*local");
        PreparedStatement ps = c.prepareStatement("INSERT INTO CUJOSQL.MYTABLE
VALUES(?, ?, ?)");
        ParameterMetaData pmd = ps.getParameterMetaData();

        for (int i = 1; i < pmd.getParameterCount(); i++) {
            System.out.println("Parameter number " + i);
```

```
            System.out.println("  Class name is " + pmd.getParameterClassName(i));
            // Note: Mode relates to input, output or inout
            System.out.println("  Mode is " + pmd.getParameterClassName(i));
            System.out.println("  Type is " + pmd.getParameterType(i));
            System.out.println("  Type name is " + pmd.getParameterTypeName(i));
            System.out.println("  Precision is " + pmd.getPrecision(i));
            System.out.println("  Scale is " + pmd.getScale(i));
            System.out.println("  Nullable? is " + pmd.isNullable(i));
            System.out.println("  Signed? is " + pmd.isSigned(i));
        }
    }
}
```

Example: Changing values with a statement through another statement's cursor

This Java example shows how to change values with a statement through another statement's cursor.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
import java.sql.*;

public class UsingPositionedUpdate {
    public Connection connection = null;
    public static void main(java.lang.String[] args) {

        UsingPositionedUpdate test = new UsingPositionedUpdate();

        test.setup();
        test.displayTable();

        test.run();
        test.displayTable();

        test.cleanup();
    }



/**
Handle all the required setup work.
**/
    public void setup() {
        try {
            // Register the JDBC driver.
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");

            connection = DriverManager.getConnection("jdbc:db2:*local");

            Statement s = connection.createStatement();
            try {
                s.executeUpdate("DROP TABLE CUJOSQL.WHERECUREX");
            } catch (SQLException e) {
                // Ignore problems here.
            }

            s.executeUpdate("CREATE TABLE CUJOSQL.WHERECUREX ( " +
                            "COL_IND INT, COL_VALUE CHAR(20)) ");

            for (int i = 1; i <= 10; i++) {
                s.executeUpdate("INSERT INTO CUJOSQL.WHERECUREX VALUES(" + i + ", 'FIRST')");
            }

            s.close();

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


/**
In this section, all the code to perform the testing should
be added. If only one connection to the database is required,
the global variable 'connection' can be used.
**/
```

```
    public void run() {
        try {
            Statement stmt1 = connection.createStatement();

            // Update each value using next().
            stmt1.setCursorName("CUJO");
            ResultSet rs = stmt1.executeQuery ("SELECT * FROM CUJOSQL.WHERECUREX " +
                                               "FOR UPDATE OF COL_VALUE");

            System.out.println("Cursor name is " + rs.getCursorName());

            PreparedStatement stmt2 = connection.prepareStatement ("UPDATE "
                                                                   + " CUJOSQL.WHERECUREX
                                                                       SET COL_VALUE =
'CHANGED'
                                                                       WHERE CURRENT OF "
                                                                   + rs.getCursorName ());


            // Loop through the ResultSet and update every other entry.
            while (rs.next ()) {
                if (rs.next())
                    stmt2.execute ();
            }


            // Clean up the resources after they have been used.
            rs.close ();
            stmt2.close ();


        } catch (Exception e) {
            System.out.println("Caught exception: ");
            e.printStackTrace();
        }
    }



/**
In this section, put all clean-up work for testing.
**/
    public void cleanup() {
        try {
            // Close the global connection opened in setup().
            connection.close();

        } catch (Exception e) {
            System.out.println("Caught exception: ");
            e.printStackTrace();
        }
    }



/**
Display the contents of the table.
**/
    public void displayTable()
    {
        try {
            Statement s = connection.createStatement();
            ResultSet rs = s.executeQuery ("SELECT * FROM CUJOSQL.WHERECUREX");

            while (rs.next ()) {
                System.out.println("Index " + rs.getInt(1) + " value " + rs.getString(2));
            }

            rs.close ();
            s.close();
            System.out.println("-----------------------------------------");
        } catch (Exception e) {
            System.out.println("Caught exception: ");
            e.printStackTrace();
        }
    }
}
```

Example: ResultSet interface

This is an example of how to use the ResultSet interface.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
import java.sql.*;

/**
ResultSetExample.java

This program demonstrates using a ResultSetMetaData and
a ResultSet to display all the data in a table even though
the program that gets the data does not know what the table
is going to look like (the user passes in the values for the
table and library).
**/
public class ResultSetExample {

    public static void main(java.lang.String[] args)
    {
        if (args.length != 2) {
            System.out.println("Usage:  java ResultSetExample <library> <table>");
            System.out.println(" where <library> is the library that contains <table>");
            System.exit(0);
        }

        Connection con = null;
        Statement s = null;
        ResultSet rs = null;
        ResultSetMetaData rsmd = null;

        try {
            // Get a database connection and prepare a statement.
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
            con = DriverManager.getConnection("jdbc:db2:*local");

            s = con.createStatement();

            rs = s.executeQuery("SELECT * FROM " + args[0] + "." + args[1]);
            rsmd = rs.getMetaData();

            int colCount = rsmd.getColumnCount();
            int rowCount = 0;
            while (rs.next()) {
                rowCount++;
                System.out.println("Data for row " + rowCount);
                for (int i = 1; i <= colCount; i++)
                    System.out.println("   Row " + i + ": " + rs.getString(i));
            }

        } catch (Exception e) {
            // Handle any errors.
            System.out.println("Oops... we have an error... ");
            e.printStackTrace();
        } finally {
            // Ensure we always clean up.  If the connection gets closed, the
            // statement under it closes as well.
            if (con != null) {
                try {
                    con.close();
                } catch (SQLException e) {
                    System.out.println("Critical error - cannot close connection object");
                }
            }
        }
    }
}
```

Related concepts

ResultSet characteristics
This topic discusses ResultSet characteristics such ResultSet types, concurrency, ability to close the ResultSet by committing the connection object, and specification of ResultSet characteristics.

Cursor movement

The IBM i Java Database Connectivity (JDBC) drivers support scrollable ResultSets. With a scrollable ResultSet, you can process rows of data in any order using a number of cursor-positioning methods.

Retrieving ResultSet data
The ResultSet object provides several methods for obtaining column data for a row. All are of the form get<*Type*>, where <*Type*> is a Java data type. Some examples of these methods include getInt, getLong, getString, getTimestamp, and getBlob. Nearly all of these methods take a single parameter that is either the column index within the ResultSet or the column name.

Creating ResultSets
To create a ResultSet object, you can use executeQuery methods, or other methods. This topic describes options for creating ResultSets.

Related tasks

Changing ResultSets
With the IBM i JDBC drivers, you can change ResultSets by performing several tasks.

Example: ResultSet sensitivity

The following example shows how a change can affect a where clause of an SQL statement based on the sensitivity of the ResultSet.

Some of the formatting in this example may be incorrect in order to fit this example on a printed page.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
import java.sql.*;

public class Sensitive2 {

    public Connection connection = null;

    public static void main(java.lang.String[] args) {
        Sensitive2 test = new Sensitive2();

        test.setup();
        test.run("sensitive");
        test.cleanup();

        test.setup();
        test.run("insensitive");
        test.cleanup();
    }


    public void setup() {

        try {
            System.out.println("Native JDBC used");
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
            connection = DriverManager.getConnection("jdbc:db2:*local");

            Statement s = connection.createStatement();
            try {
                s.executeUpdate("drop table cujosql.sensitive");
            } catch (SQLException e) {
                // Ignored.
            }

            s.executeUpdate("create table cujosql.sensitive(col1 int)");
            s.executeUpdate("insert into cujosql.sensitive values(1)");
            s.executeUpdate("insert into cujosql.sensitive values(2)");
            s.executeUpdate("insert into cujosql.sensitive values(3)");
            s.executeUpdate("insert into cujosql.sensitive values(4)");
            s.executeUpdate("insert into cujosql.sensitive values(5)");


            try {
                s.executeUpdate("drop table cujosql.sensitive2");
            } catch (SQLException e) {
                // Ignored.
            }
```

```
            s.executeUpdate("create table cujosql.sensitive2(col2 int)");
            s.executeUpdate("insert into cujosql.sensitive2 values(1)");
            s.executeUpdate("insert into cujosql.sensitive2 values(2)");
            s.executeUpdate("insert into cujosql.sensitive2 values(3)");
            s.executeUpdate("insert into cujosql.sensitive2 values(4)");
            s.executeUpdate("insert into cujosql.sensitive2 values(5)");

            s.close();

        } catch (Exception e) {
            System.out.println("Caught exception: " + e.getMessage());
            if (e instanceof SQLException) {
                SQLException another = ((SQLException) e).getNextException();
                System.out.println("Another:  " + another.getMessage());
            }
        }
    }


    public void run(String sensitivity) {
        try {

            Statement s = null;
            if (sensitivity.equalsIgnoreCase("insensitive")) {
                System.out.println("creating a TYPE_SCROLL_INSENSITIVE cursor");
                s = connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                    ResultSet.CONCUR_READ_ONLY);
            } else {
                System.out.println("creating a TYPE_SCROLL_SENSITIVE cursor");
                s = connection.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                    ResultSet.CONCUR_READ_ONLY);
            }

            ResultSet rs = s.executeQuery("select col1, col2 From cujosql.sensitive,
                                           cujosql.sensitive2 where col1 = col2");

            rs.next();
            System.out.println("value is " + rs.getInt(1));
            rs.next();
            System.out.println("value is " + rs.getInt(1));
            rs.next();
            System.out.println("value is " + rs.getInt(1));
            rs.next();
            System.out.println("value is " + rs.getInt(1));

            System.out.println("fetched the four rows...");

            // Another statement creates a value that does not fit the where clause.
            Statement s2 =
              connection.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
              ResultSet.CONCUR_UPDATEABLE);
            ResultSet rs2 = s2.executeQuery("select *
            from cujosql.sensitive where col1 = 5 FOR UPDATE");
            rs2.next();
            rs2.updateInt(1, -1);
            rs2.updateRow();
            s2.close();

            if (rs.next()) {
                System.out.println("There is still a row: " + rs.getInt(1));
            } else {
                System.out.println("No more rows.");
            }

        } catch (SQLException e) {
            System.out.println("SQLException exception: ");
            System.out.println("Message:....." + e.getMessage());
            System.out.println("SQLState:...." + e.getSQLState());
            System.out.println("Vendor Code:." + e.getErrorCode());
            System.out.println("----------------------------");
            e.printStackTrace();
        }
        catch (Exception ex) {
            System.out.println("An exception other
            than an SQLException was thrown: ");
            ex.printStackTrace();
        }
    }



    public void cleanup() {
```

```
        try {
            connection.close();
        } catch (Exception e) {
            System.out.println("Caught exception: ");
            e.printStackTrace();
        }
    }
}
```

Example: Sensitive and insensitive ResultSets

The following example shows the difference between sensitive and insensitive ResultSets when rows are inserted into a table.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
import java.sql.*;

public class Sensitive {

    public Connection connection = null;

    public static void main(java.lang.String[] args) {
        Sensitive test = new Sensitive();

        test.setup();
        test.run("sensitive");
        test.cleanup();

        test.setup();
        test.run("insensitive");
        test.cleanup();
    }


    public void setup() {

        try {
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
            connection = DriverManager.getConnection("jdbc:db2:*local");

            Statement s = connection.createStatement();
            try {
                s.executeUpdate("drop table cujosql.sensitive");
            } catch (SQLException e) {
                // Ignored.
            }

            s.executeUpdate("create table cujosql.sensitive(col1 int)");
            s.executeUpdate("insert into cujosql.sensitive values(1)");
            s.executeUpdate("insert into cujosql.sensitive values(2)");
            s.executeUpdate("insert into cujosql.sensitive values(3)");
            s.executeUpdate("insert into cujosql.sensitive values(4)");
            s.executeUpdate("insert into cujosql.sensitive values(5)");
            s.close();

        } catch (Exception e) {
            System.out.println("Caught exception: " + e.getMessage());
            if (e instanceof SQLException) {
                SQLException another = ((SQLException) e).getNextException();
                System.out.println("Another:  " + another.getMessage());
            }
        }
    }


    public void run(String sensitivity) {
        try {
            Statement s = null;
            if (sensitivity.equalsIgnoreCase("insensitive")) {
                System.out.println("creating a TYPE_SCROLL_INSENSITIVE cursor");
                s = connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                    ResultSet.CONCUR_READ_ONLY);
            } else {
                System.out.println("creating a TYPE_SCROLL_SENSITIVE cursor");
                s = connection.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                    ResultSet.CONCUR_READ_ONLY);
```

```
            }

            ResultSet rs = s.executeQuery("select * From cujosql.sensitive");

            // Fetch the five values that are there.
            rs.next();
            System.out.println("value is " + rs.getInt(1));
            rs.next();
            System.out.println("value is " + rs.getInt(1));
            rs.next();
            System.out.println("value is " + rs.getInt(1));
            rs.next();
            System.out.println("value is " + rs.getInt(1));
            rs.next();
            System.out.println("value is " + rs.getInt(1));
            System.out.println("fetched the five rows...");

            // Note:  If you fetch the last row, the ResultSet looks
            //        closed and subsequent new rows that are added
            //        are not be recognized.

            // Allow another statement to insert a new value.
            Statement s2 = connection.createStatement();
            s2.executeUpdate("insert into cujosql.sensitive values(6)");
            s2.close();

            // Whether a row is recognized is based on the sensitivity setting.
            if (rs.next()) {
                System.out.println("There is a row now: " + rs.getInt(1));
            } else {
                System.out.println("No more rows.");
            }


        } catch (SQLException e) {
            System.out.println("SQLException exception: ");
            System.out.println("Message:....." + e.getMessage());
            System.out.println("SQLState:...." + e.getSQLState());
            System.out.println("Vendor Code:." + e.getErrorCode());
            System.out.println("-------------------------------------");
            e.printStackTrace();
        }
        catch (Exception ex) {
            System.out.println("An exception other than an SQLException was thrown: ");
            ex.printStackTrace();
        }
    }



    public void cleanup() {
        try {
            connection.close();
        } catch (Exception e) {
            System.out.println("Caught exception: ");
            e.printStackTrace();
        }
    }
}
```

Example: Setting up connection pooling with UDBDataSource and UDBConnectionPoolDataSource

This is an example of how to use connection pooling with UDBDataSource and UDBConnectionPoolDataSource.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
import java.sql.*;
import javax.naming.*;
import com.ibm.db2.jdbc.app.UDBDataSource;
import com.ibm.db2.jdbc.app.UDBConnectionPoolDataSource;

public class ConnectionPoolingSetup
{
    public static void main(java.lang.String[] args)
```

```
    throws Exception
    {
        // Create a ConnectionPoolDataSource implementation
        UDBConnectionPoolDataSource cpds = new UDBConnectionPoolDataSource();
        cpds.setDescription("Connection Pooling DataSource object");

        // Establish a JNDI context and bind the connection pool data source
        Context ctx = new InitialContext();
        ctx.rebind("ConnectionSupport", cpds);

        // Create a standard data source that references it.
        UDBDataSource ds = new UDBDataSource();
        ds.setDescription("DataSource supporting pooling");
        ds.setDataSourceName("ConnectionSupport");
        ctx.rebind("PoolingDataSource", ds);
    }
}
```

Example: SQLException

This is an example of catching an SQLException and dumping all the information that it provides.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
import java.sql.*;

public class ExceptionExample {

    public static Connection connection = null;

    public static void main(java.lang.String[] args) {

        try {
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
            connection = DriverManager.getConnection("jdbc:db2:*local");

            Statement s = connection.createStatement();
            int count = s.executeUpdate("insert into cujofake.cujofake values(1, 2,3)");

            System.out.println("Did not expect that table to exist.");


        } catch (SQLException e) {
            System.out.println("SQLException exception: ");
            System.out.println("Message:....." + e.getMessage());
            System.out.println("SQLState:...." + e.getSQLState());
            System.out.println("Vendor Code:." + e.getErrorCode());
            System.out.println("-----------------------------------------------------");
            e.printStackTrace();
        } catch (Exception ex) {
            System.out.println("An exception other than an SQLException was thrown: ");
            ex.printStackTrace();
        } finally {
            try {
                if (connection != null) {
                    connection.close();
                }
            } catch (SQLException e) {
                System.out.println("Exception caught attempting to shutdown...");
            }
        }
    }
}
```

Example: Suspending and resuming a transaction

This is an example of a transaction that is suspended and then is resumed.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
import java.sql.*;
import javax.sql.*;
import java.util.*;
```

```
import javax.transaction.*;
import javax.transaction.xa.*;
import com.ibm.db2.jdbc.app.*;
import javax.naming.InitialContext;
import javax.naming.Context;

public class JTATxSuspend {

    public static void main(java.lang.String[] args) {
        JTATxSuspend test = new JTATxSuspend();

        test.setup();
        test.run();
    }


    /**
     * Handle the previous cleanup run so that this test can recommence.
     */
    public void setup() {

        Connection c = null;
        Statement s = null;
        try {
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
            c = DriverManager.getConnection("jdbc:db2:*local");
            s = c.createStatement();

            try {
                s.executeUpdate("DROP TABLE CUJOSQL.JTATABLE");
            } catch (SQLException e) {
                // Ignore... doesn't exist
            }

            s.executeUpdate("CREATE TABLE CUJOSQL.JTATABLE (COL1 CHAR (50))");
            s.executeUpdate("INSERT INTO CUJOSQL.JTATABLE VALUES('Fun with JTA')");
            s.executeUpdate("INSERT INTO CUJOSQL.JTATABLE VALUES('JTA is fun.)");

            s.close();
        } finally {
            if (c != null) {
                c.close();
            }
        }
    }


    /**
     * This test uses JTA support to handle transactions.
     */
    public void run() {
        Connection c = null;

        try {
            Context ctx = new InitialContext();

            // Assume the data source is backed by a UDBXADataSource.
            UDBXADataSource ds = (UDBXADataSource) ctx.lookup("XADataSource");

            // From the DataSource, obtain an XAConnection object that
            // contains an XAResource and a Connection object.
            XAConnection  xaConn = ds.getXAConnection();
            XAResource    xaRes  = xaConn.getXAResource();
            c      = xaConn.getConnection();

            // For XA transactions, a transaction identifier is required.
            // An implementation of the XID interface is not included with
            // the JDBC driver. See topic "Transactions with JTA" for a
            // description of this interface to build a class for it.
            Xid xid = new XidImpl();

            // The connection from the XAResource can be used as any other
            // JDBC connection.
            Statement stmt = c.createStatement();

            // The XA resource must be notified before starting any
            // transactional work.
            xaRes.start(xid, XAResource.TMNOFLAGS);

            // Create a ResultSet during JDBC processing and fetch a row.
            ResultSet rs = stmt.executeQuery("SELECT * FROM CUJOSQL.JTATABLE");
            rs.next();
```

```
            // The end method is called with the suspend option. The
            // ResultSets associated with the current transaction are 'on hold'.
            // They are neither gone nor accessible in this state.
            xaRes.end(xid, XAResource.TMSUSPEND);


            // Other work can be performed with the transaction.
            // As an example, you can create a statement and process a query.
            // This work and any other transactional work that the transaction may
            // perform is separate from the work done previously under the XID.
            Statement nonXAStmt = c.createStatement();
            ResultSet nonXARS = nonXAStmt.executeQuery("SELECT * FROM CUJOSQL.JTATABLE");
            while (nonXARS.next()) {
                // Process here...
            }
            nonXARS.close();
            nonXAStmt.close();


            // If an attempt is made to use any suspended transactions
            // resources, an exception results.
            try {
                rs.getString(1);
                System.out.println("Value of the first row is " + rs.getString(1));
            } catch (SQLException e) {
                System.out.println("This was an expected exception - " +
                                   "suspended ResultSet was used.");
            }


            // Resume the suspended transaction and complete the work on it.
            // The ResultSet is exactly as it was before the suspension.
            xaRes.start(newXid, XAResource.TMRESUME);
            rs.next();
            System.out.println("Value of the second row is " + rs.getString(1));


            // When the transaction has completed, end it
            // and commit any work under it.
            xaRes.end(xid, XAResource.TMNOFLAGS);
            int rc = xaRes.prepare(xid);
            xaRes.commit(xid, false);


        } catch (Exception e) {
            System.out.println("Something has gone wrong.");
            e.printStackTrace();
        } finally {
            try {
                if (c != null)
                    c.close();
            } catch (SQLException e) {
                System.out.println("Note:  Cleaup exception.");
                e.printStackTrace();
            }
        }
    }
}
```

Related reference

Example: Using JTA to handle a transaction
This is an example of how to use the Java Transaction API (JTA) to handle a transaction in an application.

Example: Multiple connections that work on a transaction
This is an example of how to use multiple connections working on a single transaction.

Example: Using a connection with multiple transactions
This is an example of how to use a single connection with multiple transactions.

Example: Suspended ResultSets
This is an example of the how a Statement object is reprocessed under another transaction to perform work.

Example: Ending a transaction

This is an example of ending a transaction in your application.

Example: Suspended ResultSets

This is an example of the how a Statement object is reprocessed under another transaction to perform work.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
import java.sql.*;
import javax.sql.*;
import java.util.*;
import javax.transaction.*;
import javax.transaction.xa.*;
import com.ibm.db2.jdbc.app.*;

public class JTATxEffect {

    public static void main(java.lang.String[] args) {
        JTATxEffect test = new JTATxEffect();

        test.setup();
        test.run();
    }


    /**
     * Handle the previous cleanup run so that this test can recommence.
     */
    public void setup() {

        Connection c = null;
        Statement s = null;
        try {
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
            c = DriverManager.getConnection("jdbc:db2:*local");
            s = c.createStatement();

            try {
                s.executeUpdate("DROP TABLE CUJOSQL.JTATABLE");
            } catch (SQLException e) {
                // Ignore... does not exist
            }

            s.executeUpdate("CREATE TABLE CUJOSQL.JTATABLE (COL1 CHAR (50))");
            s.executeUpdate("INSERT INTO CUJOSQL.JTATABLE VALUES('Fun with JTA')");
            s.executeUpdate("INSERT INTO CUJOSQL.JTATABLE VALUES('JTA is fun.)");

            s.close();
        } finally {
            if (c != null) {
                c.close();
            }
        }
    }


    /**
     * This test uses JTA support to handle transactions.
     */
    public void run() {
        Connection c = null;

        try {
            Context ctx = new InitialContext();

            // Assume the data source is backed by a UDBXADataSource.
            UDBXADataSource ds = (UDBXADataSource) ctx.lookup("XADataSource");

            // From the DataSource, obtain an XAConnection object that
            // contains an XAResource and a Connection object.
            XAConnection  xaConn = ds.getXAConnection();
            XAResource    xaRes  = xaConn.getXAResource();
            Connection    c      = xaConn.getConnection();

            // For XA transactions, a transaction identifier is required.
            // An implementation of the XID interface is not included with
```

```
            // the JDBC driver. See Transactions with JTA
            // for a description of this interface to build a
            // class for it.
            Xid xid = new XidImpl();

            // The connection from the XAResource can be used as any other
            // JDBC connection.
            Statement stmt = c.createStatement();

            // The XA resource must be notified before starting any
            // transactional work.
            xaRes.start(xid, XAResource.TMNOFLAGS);

            // Create a ResultSet during JDBC processing and fetch a row.
            ResultSet rs = stmt.executeUpdate("SELECT * FROM CUJOSQL.JTATABLE");
            rs.next();

            // The end method is called with the suspend option. The
            // ResultSets associated with the current transaction are 'on hold'.
            // They are neither gone nor accessible in this state.
            xaRes.end(xid, XAResource.TMSUSPEND);


            // In the meantime, other work can be done outside the transaction.
            // The ResultSets under the transaction can be closed if the
            // Statement object used to create them is reused.
            ResultSet nonXARS = stmt.executeQuery("SELECT * FROM CUJOSQL.JTATABLE");
            while (nonXARS.next()) {
                // Process here...
            }


            // Attempt to go back to the suspended transaction. The suspended
            // transaction's ResultSet has disappeared because the statement
            // has been processed again.
            xaRes.start(newXid, XAResource.TMRESUME);
            try {
                rs.next();
            } catch (SQLException ex) {
                System.out.println("This exception is expected. " +
                                   "The ResultSet closed due to another process.");
            }


            // When the transaction had completed, end it
            // and commit any work under it.
            xaRes.end(xid, XAResource.TMNOFLAGS);
            int rc = xaRes.prepare(xid);
            xaRes.commit(xid, false);


        } catch (Exception e) {
            System.out.println("Something has gone wrong.");
            e.printStackTrace();
        } finally {
            try {
                if (c != null)
                    c.close();
            } catch (SQLException e) {
                System.out.println("Note:  Cleaup exception.");
                e.printStackTrace();
            }
        }
    }
}
```

Related reference

Example: Using JTA to handle a transaction
This is an example of how to use the Java Transaction API (JTA) to handle a transaction in an application.

Example: Multiple connections that work on a transaction
This is an example of how to use multiple connections working on a single transaction.

Example: Using a connection with multiple transactions
This is an example of how to use a single connection with multiple transactions.

Example: Ending a transaction

This is an example of ending a transaction in your application.

Example: Suspending and resuming a transaction
This is an example of a transaction that is suspended and then is resumed.

Example: Testing the performance of connection pooling

This is an example of how to test the performance of the pooling example against the performance of the non-pooling example.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
import java.sql.*;
import javax.naming.*;
import java.util.*;
import javax.sql.*;

public class ConnectionPoolingTest
{
    public static void main(java.lang.String[] args)
    throws Exception
    {
        Context ctx = new InitialContext();
        // Do the work without a pool:
        DataSource ds = (DataSource) ctx.lookup("BaseDataSource");
        System.out.println("\nStart timing the non-pooling DataSource version...");

        long startTime = System.currentTimeMillis();
        for (int i = 0; i < 100; i++) {
            Connection c1 = ds.getConnection();
            c1.close();
        }
        long endTime = System.currentTimeMillis();
        System.out.println("Time spent: " + (endTime - startTime));

        // Do the work with pooling:
        ds = (DataSource) ctx.lookup("PoolingDataSource");
        System.out.println("\nStart timing the pooling version...");

        startTime = System.currentTimeMillis();
        for (int i = 0; i < 100; i++) {
            Connection c1 = ds.getConnection();
            c1.close();
        }
        endTime = System.currentTimeMillis();
        System.out.println("Time spent: " + (endTime - startTime));
    }
}
```

Example: Testing the performance of two DataSources

This is an example of testing one DataSource that uses connection pooling only and another DataSource that uses statement and connection pooling.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
import java.sql.*;
import javax.naming.*;
import java.util.*;
import javax.sql.*;
import com.ibm.db2.jdbc.app.UDBDataSource;
import com.ibm.db2.jdbc.app.UDBConnectionPoolDataSource;

public class StatementPoolingTest
{

    public static void main(java.lang.String[] args)
    throws Exception
    {
        Context ctx = new InitialContext();

        System.out.println("deploying statement pooling data source");
```

```
        deployStatementPoolDataSource();

        // Do the work with connection pooling only.
        DataSource ds = (DataSource) ctx.lookup("PoolingDataSource");
        System.out.println("\nStart timing the connection pooling only version...");

        long startTime = System.currentTimeMillis();
        for (int i = 0; i < 100; i++) {
            Connection c1 = ds.getConnection();
            PreparedStatement ps = c1.prepareStatement("select * from qsys2.sysprocs");
            ResultSet rs = ps.executeQuery();
            c1.close();
        }
        long endTime = System.currentTimeMillis();
        System.out.println("Time spent: " + (endTime - startTime));


        // Do the work with statement pooling added.
        ds = (DataSource) ctx.lookup("StatementPoolingDataSource");
        System.out.println("\nStart timing the statement pooling version...");

        startTime = System.currentTimeMillis();
        for (int i = 0; i < 100; i++) {
            Connection c1 = ds.getConnection();
            PreparedStatement ps = c1.prepareStatement("select * from qsys2.sysprocs");
            ResultSet rs = ps.executeQuery();
            c1.close();
        }
        endTime = System.currentTimeMillis();
        System.out.println("Time spent: " + (endTime - startTime));
    }



    private static void deployStatementPoolDataSource()
    throws Exception
    {
        // Create a ConnectionPoolDataSource implementation
        UDBConnectionPoolDataSource cpds = new UDBConnectionPoolDataSource();
        cpds.setDescription("Connection Pooling DataSource object with Statement pooling");
        cpds.setMaxStatements(10);

        // Establish a JNDI context and bind the connection pool data source
        Context ctx = new InitialContext();
        ctx.rebind("StatementSupport", cpds);


        // Create a standard datasource that references it.
        UDBDataSource ds = new UDBDataSource();
        ds.setDescription("DataSource supporting statement pooling");
        ds.setDataSourceName("StatementSupport");
        ctx.rebind("StatementPoolingDataSource", ds);

    }
}
```

Example: Updating BLOBs

This is an example of how to update BLOBs in your Java applications.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
/////////////////////////////////////////
// UpdateBlobs is an example application
// that shows some of the APIs providing
// support for changing Blob objects
// and reflecting those changes to the
// database.
//
// This program must be run after
// the PutGetBlobs program has completed.
/////////////////////////////////////////
import java.sql.*;

public class UpdateBlobs {
   public static void main(String[] args)
   throws SQLException
   {
```

```
        // Register the native JDBC driver.
        try {
           Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
       } catch (Exception e) {
           System.exit(1);  // Setup error.
       }

       Connection c = DriverManager.getConnection("jdbc:db2:*local");
       Statement s = c.createStatement();

       ResultSet rs = s.executeQuery("SELECT * FROM CUJOSQL.BLOBTABLE");

       rs.next();
       Blob blob1 = rs.getBlob(1);
       rs.next();
       Blob blob2 = rs.getBlob(1);


       // Truncate a BLOB.
       blob1.truncate((long) 150000);
       System.out.println("Blob1's new length is " + blob1.length());

       // Update part of the BLOB with a new byte array.
       // The following code obtains the bytes that are at
       // positions 4000-4500 and set them to positions 500-1000.

       // Obtain part of the BLOB as a byte array.
       byte[] bytes = blob1.getBytes(4000L, 4500);

       int bytesWritten = blob2.setBytes(500L, bytes);

       System.out.println("Bytes written is " + bytesWritten);

       // The bytes are now found at position 500 in blob2
       long startInBlob2 = blob2.position(bytes, 1);

       System.out.println("pattern found starting at position " + startInBlob2);

       c.close(); // Connection close also closes stmt and rs.
    }
}
```

Related reference

Example: BLOB
This is an example of how a BLOB can be put into the database or retrieved from the database.

Example: Using BLOBs
This is an example of how to use BLOBs in your Java applications.

Example: Updating CLOBs

This is an example of how to update CLOBs in your Java applications.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
/////////////////////////////////////////
// UpdateClobs is an example application
// that shows some of the APIs providing
// support for changing Clob objects
// and reflecting those changes to the
// database.
//
// This program must be run after
// the PutGetClobs program has completed.
/////////////////////////////////////////
import java.sql.*;

public class UpdateClobs {
   public static void main(String[] args)
   throws SQLException
   {
       // Register the native JDBC driver.
       try {
          Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
      } catch (Exception e) {
          System.exit(1);  // Setup error.
```

```
        }

        Connection c = DriverManager.getConnection("jdbc:db2:*local");
        Statement s = c.createStatement();

        ResultSet rs = s.executeQuery("SELECT * FROM CUJOSQL.CLOBTABLE");

        rs.next();
        Clob clob1 = rs.getClob(1);
        rs.next();
        Clob clob2 = rs.getClob(1);


        // Truncate a CLOB.
        clob1.truncate((long) 150000);
        System.out.println("Clob1's new length is " + clob1.length());

        // Update a portion of the CLOB with a new String value.
        String value = "Some new data for once";
        int charsWritten = clob2.setString(500L, value);

        System.out.println("Characters written is " + charsWritten);

        // The bytes can be found at position 500 in clob2
        long startInClob2 = clob2.position(value, 1);

        System.out.println("pattern found starting at position " + startInClob2);

        c.close(); // Connection close also closes stmt and rs.
    }
}
```

Related reference

Example: CLOB
This is an example of how a CLOB can be put into the database or retrieved from the database.

Example: Using CLOBs
This is an example of how to use CLOBs in your Java applications.

Example: Using a connection with multiple transactions

This is an example of how to use a single connection with multiple transactions.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
import java.sql.*;
import javax.sql.*;
import java.util.*;
import javax.transaction.*;
import javax.transaction.xa.*;
import com.ibm.db2.jdbc.app.*;

public class JTAMultiTx {

    public static void main(java.lang.String[] args) {
        JTAMultiTx test = new JTAMultiTx();

        test.setup();
        test.run();
    }


    /**
     * Handle the previous cleanup run so that this test can recommence.
     */
    public void setup() {

        Connection c = null;
        Statement s = null;
        try {
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
            c = DriverManager.getConnection("jdbc:db2:*local");
            s = c.createStatement();

            try {
                s.executeUpdate("DROP TABLE CUJOSQL.JTATABLE");
```

```
        } catch (SQLException e) {
            // Ignore... does not exist
        }

        s.executeUpdate("CREATE TABLE CUJOSQL.JTATABLE (COL1 CHAR (50))");

        s.close();
    } finally {
        if (c != null) {
            c.close();
        }
    }
}


/**
 * This test uses JTA support to handle transactions.
 */
public void run() {
    Connection c = null;

    try {
        Context ctx = new InitialContext();

        // Assume the data source is backed by a UDBXADataSource.
        UDBXADataSource ds = (UDBXADataSource) ctx.lookup("XADataSource");

        // From the DataSource, obtain an XAConnection object that
        // contains an XAResource and a Connection object.
        XAConnection  xaConn = ds.getXAConnection();
        XAResource    xaRes  = xaConn.getXAResource();
        Connection    c      = xaConn.getConnection();
        Statement     stmt   = c.createStatement();

        // For XA transactions, a transaction identifier is required.
        // This is not meant to imply that all the XIDs are the same.
        // Each XID must be unique to distinguish the various transactions
        // that occur.
        // Support for creating XIDs is again left to the application
        // program.
        Xid xid1 = JDXATest.xidFactory();
        Xid xid2 = JDXATest.xidFactory();
        Xid xid3 = JDXATest.xidFactory();

        // Do work under three transactions for this connection.
        xaRes.start(xid1, XAResource.TMNOFLAGS);
        int count1 = stmt.executeUpdate("INSERT INTO CUJOSQL.JTATABLE VALUES('Value 1-A')");
        xaRes.end(xid1, XAResource.TMNOFLAGS);

        xaRes.start(xid2, XAResource.TMNOFLAGS);
        int count2 = stmt.executeUpdate("INSERT INTO CUJOSQL.JTATABLE VALUES('Value 1-B')");
        xaRes.end(xid2, XAResource.TMNOFLAGS);

        xaRes.start(xid3, XAResource.TMNOFLAGS);
        int count3 = stmt.executeUpdate("INSERT INTO CUJOSQL.JTATABLE VALUES('Value 1-C')");
        xaRes.end(xid3, XAResource.TMNOFLAGS);

        // Prepare all the transactions
        int rc1 = xaRes.prepare(xid1);
        int rc2 = xaRes.prepare(xid2);
        int rc3 = xaRes.prepare(xid3);

        // Two of the transactions commit and one rolls back.
        // The attempt to insert the second value into the table is
        // not committed.
        xaRes.commit(xid1, false);
        xaRes.rollback(xid2);
        xaRes.commit(xid3, false);


    } catch (Exception e) {
        System.out.println("Something has gone wrong.");
        e.printStackTrace();
    } finally {
        try {
            if (c != null)
                c.close();
        } catch (SQLException e) {
            System.out.println("Note:  Cleaup exception.");
            e.printStackTrace();
        }
    }
```

```
    }
}
```

Related reference

Example: Using JTA to handle a transaction
This is an example of how to use the Java Transaction API (JTA) to handle a transaction in an application.

Example: Multiple connections that work on a transaction
This is an example of how to use multiple connections working on a single transaction.

Example: Suspended ResultSets
This is an example of the how a Statement object is reprocessed under another transaction to perform work.

Example: Ending a transaction
This is an example of ending a transaction in your application.

Example: Suspending and resuming a transaction
This is an example of a transaction that is suspended and then is resumed.

Example: Using BLOBs

This is an example of how to use BLOBs in your Java applications.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
/////////////////////////////////////////
// UseBlobs is an example application
// that shows some of the APIs associated
// with Blob objects.
//
// This program must be run after
// the PutGetBlobs program has completed.
/////////////////////////////////////////
import java.sql.*;

public class UseBlobs {
   public static void main(String[] args)
   throws SQLException
   {
       // Register the native JDBC driver.
       try {
          Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
      } catch (Exception e) {
          System.exit(1);  // Setup error.
      }

      Connection c = DriverManager.getConnection("jdbc:db2:*local");
      Statement s = c.createStatement();

      ResultSet rs = s.executeQuery("SELECT * FROM CUJOSQL.BLOBTABLE");

      rs.next();
      Blob blob1 = rs.getBlob(1);
      rs.next();
      Blob blob2 = rs.getBlob(1);


      // Determine the length of a LOB.
      long end = blob1.length();
      System.out.println("Blob1 length is " + blob1.length());

      // When working with LOBs, all indexing that is related to them
      // is 1-based, and is not 0-based like strings and arrays.
      long startingPoint = 450;
      long endingPoint = 500;

      // Obtain part of the BLOB as a byte array.
      byte[] outByteArray = blob1.getBytes(startingPoint, (int)endingPoint);

      // Find where a sub-BLOB or byte array is first found within a
      // BLOB. The setup for this program placed two identical copies of
      // a random BLOB into the database. Thus, the start position of the
      // byte array extracted from blob1 can be found in the starting
```

```
        // position in blob2. The exception would be if there were 50
        // identical random bytes in the LOBs previously.
        long startInBlob2 = blob2.position(outByteArray, 1);

        System.out.println("pattern found starting at position " + startInBlob2);

        c.close(); // Connection close closes stmt and rs too.
    }
}
```

Related reference

Example: BLOB
This is an example of how a BLOB can be put into the database or retrieved from the database.

Example: Updating BLOBs
This is an example of how to update BLOBs in your Java applications.

Example: Using CLOBs

This is an example of how to use CLOBs in your Java applications.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
/////////////////////////////////////////
// UpdateClobs is an example application
// that shows some of the APIs providing
// support for changing Clob objects
// and reflecting those changes to the
// database.
//
// This program must be run after
// the PutGetClobs program has completed.
/////////////////////////////////////////
import java.sql.*;

public class UseClobs {
    public static void main(String[] args)
    throws SQLException
    {
        // Register the native JDBC driver.
        try {
           Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
        } catch (Exception e) {
           System.exit(1);  // Setup error.
        }

        Connection c = DriverManager.getConnection("jdbc:db2:*local");
        Statement s = c.createStatement();

        ResultSet rs = s.executeQuery("SELECT * FROM CUJOSQL.CLOBTABLE");

        rs.next();
        Clob clob1 = rs.getClob(1);
        rs.next();
        Clob clob2 = rs.getClob(1);


        // Determine the length of a LOB.
        long end = clob1.length();
        System.out.println("Clob1 length is " + clob1.length());

        // When working with LOBs, all indexing that is related to them
        // is 1-based, and not 0-based like strings and arrays.
        long startingPoint = 450;
        long endingPoint = 50;

        // Obtain part of the CLOB as a byte array.
        String outString = clob1.getSubString(startingPoint, (int)endingPoint);
        System.out.println("Clob substring is " + outString);

        // Find where a sub-CLOB or string is first found within a
        // CLOB. The setup for this program placed two identical copies of
        // a repeating CLOB into the database. Thus, the start position of the
        // string extracted from clob1 can be found in the starting
        // position in clob2 if the search begins close to the position where
        // the string starts.
```

```
        long startInClob2 = clob2.position(outString, 440);

        System.out.println("pattern found starting at position " + startInClob2);

        c.close(); // Connection close also closes stmt and rs.
    }
}
```

Related reference

Example: CLOB
This is an example of how a CLOB can be put into the database or retrieved from the database.

Example: Updating CLOBs
This is an example of how to update CLOBs in your Java applications.

Example: Using JTA to handle a transaction

This is an example of how to use the Java Transaction API (JTA) to handle a transaction in an application.

Note: By using the code examples, you agree to the terms of the “Code license and disclaimer information” on page 489.

```
import java.sql.*;
import javax.sql.*;
import java.util.*;
import javax.transaction.*;
import javax.transaction.xa.*;
import com.ibm.db2.jdbc.app.*;

public class JTACommit {

    public static void main(java.lang.String[] args) {
        JTACommit test = new JTACommit();

        test.setup();
        test.run();
    }


    /**
     * Handle the previous cleanup run so that this test can recommence.
     */
    public void setup() {

        Connection c = null;
        Statement s = null;
        try {
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
            c = DriverManager.getConnection("jdbc:db2:*local");
            s = c.createStatement();

            try {
                s.executeUpdate("DROP TABLE CUJOSQL.JTATABLE");
            } catch (SQLException e) {
                // Ignore... does not exist
            }

            s.executeUpdate("CREATE TABLE CUJOSQL.JTATABLE (COL1 CHAR (50))");
            s.close();
        } finally {
            if (c != null) {
                c.close();
            }
        }
    }


    /**
     * This test uses JTA support to handle transactions.
     */
    public void run() {
        Connection c = null;

        try {
            Context ctx = new InitialContext();

            // Assume the data source is backed by a UDBXADataSource.
```

```
            UDBXADataSource ds = (UDBXADataSource) ctx.lookup("XADataSource");

            // From the DataSource, obtain an XAConnection object that
            // contains an XAResource and a Connection object.
            XAConnection  xaConn = ds.getXAConnection();
            XAResource    xaRes  = xaConn.getXAResource();
            Connection    c      = xaConn.getConnection();

            // For XA transactions, a transaction identifier is required.
            // An implementation of the XID interface is not included with the
            // JDBC driver. See Transactions with JTA for a description of
            // this interface to build a class for it.
            Xid xid = new XidImpl();

            // The connection from the XAResource can be used as any other
            // JDBC connection.
            Statement stmt = c.createStatement();

            // The XA resource must be notified before starting any
            // transactional work.
            xaRes.start(xid, XAResource.TMNOFLAGS);

            // Standard JDBC work is performed.
            int count =
              stmt.executeUpdate("INSERT INTO CUJOSQL.JTATABLE VALUES('JTA is pretty fun.')");

            // When the transaction work has completed, the XA resource must
            // again be notified.
            xaRes.end(xid, XAResource.TMSUCCESS);

            // The transaction represented by the transaction ID is prepared
            // to be committed.
            int rc = xaRes.prepare(xid);

            // The transaction is committed through the XAResource.
            // The JDBC Connection object is not used to commit
            // the transaction when using JTA.
            xaRes.commit(xid, false);


        } catch (Exception e) {
            System.out.println("Something has gone wrong.");
            e.printStackTrace();
        } finally {
            try {
                if (c != null)
                    c.close();
            } catch (SQLException e) {
                System.out.println("Note:  Cleaup exception.");
                e.printStackTrace();
            }
        }
    }
}
```

Related reference

Example: Multiple connections that work on a transaction
This is an example of how to use multiple connections working on a single transaction.

Example: Using a connection with multiple transactions
This is an example of how to use a single connection with multiple transactions.

Example: Suspended ResultSets
This is an example of the how a Statement object is reprocessed under another transaction to perform work.

Example: Ending a transaction
This is an example of ending a transaction in your application.

Example: Suspending and resuming a transaction

This is an example of a transaction that is suspended and then is resumed.

Example: Using metadata ResultSets that have more than one column

This is an example of how to use metadata ResultSets that have more than one column.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
//////////////////////////////////////////////////////////////////////////////////
//
// SafeGetUDTs example. This program demonstrates one way to deal with
// metadata ResultSets that have more columns in JDK 1.4 than they
// had in previous releases.
//
// Command syntax:
//    java SafeGetUDTs
//
//////////////////////////////////////////////////////////////////////////////////
//
// This source is an example of the IBM Developer for Java JDBC driver.
// IBM grants you a nonexclusive license to use this as an example
// from which you can generate similar function tailored to
// your own specific needs.
//
// This sample code is provided by IBM for illustrative purposes
// only. These examples have not been thoroughly tested under all
// conditions. IBM, therefore, cannot guarantee or imply
// reliability, serviceability, or function of these programs.
//
// All programs contained herein are provided to you "AS IS"
// without any warranties of any kind.  The implied warranties of
// merchantability and fitness for a particular purpose are
// expressly disclaimed.
//
// IBM Developer Kit for Java
// (C) Copyright IBM Corp. 2001
// All rights reserved.
// US Government Users Restricted Rights -
// Use, duplication, or disclosure restricted
// by GSA ADP Schedule Contract with IBM Corp.
//
//////////////////////////////////////////////////////////////////////////////////

import java.sql.*;

public class SafeGetUDTs {

    public static int jdbcLevel;

    // Note: Static block runs before main begins.
    // Therefore, there is access to jdbcLevel in
    // main.
    {
        try {
            Class.forName("java.sql.Blob");

            try {
                Class.forName("java.sql.ParameterMetaData");
                // Found a JDBC 3.0 interface.  Must support JDBC 3.0.
                jdbcLevel = 3;
            } catch (ClassNotFoundException ez) {
                // Could not find the JDBC 3.0 ParameterMetaData class.
                // Must be running under a JVM with only JDBC 2.0
                // support.
                jdbcLevel = 2;
            }

        } catch (ClassNotFoundException ex) {
            // Could not find the JDBC 2.0 Blob class.  Must be
            // running under a JVM with only JDBC 1.0 support.
            jdbcLevel = 1;
        }
    }

    // Program entry point.
    public static void main(java.lang.String[] args)
    {
        Connection c = null;
```

```
        try {
            // Get the driver registered.
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");

            c = DriverManager.getConnection("jdbc:db2:*local");
            DatabaseMetaData dmd = c.getMetaData();

            if (jdbcLevel == 1) {
                System.out.println("No support is provided for getUDTs.  Just return.");
                System.exit(1);
            }

            ResultSet rs = dmd.getUDTs(null, "CUJOSQL", "SSN%", null);
            while (rs.next()) {

                // Fetch all the columns that have been available since the
                // JDBC 2.0 release.
                System.out.println("TYPE_CAT is " + rs.getString("TYPE_CAT"));
                System.out.println("TYPE_SCHEM is " + rs.getString("TYPE_SCHEM"));
                System.out.println("TYPE_NAME is " + rs.getString("TYPE_NAME"));
                System.out.println("CLASS_NAME is " + rs.getString("CLASS_NAME"));
                System.out.println("DATA_TYPE is " + rs.getString("DATA_TYPE"));
                System.out.println("REMARKS is " + rs.getString("REMARKS"));

                // Fetch all the columns that were added in JDBC 3.0.
                if (jdbcLevel > 2) {
                    System.out.println("BASE_TYPE is " + rs.getString("BASE_TYPE"));
                }
            }
        } catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
        } finally {
            if (c != null) {
                try {
                    c.close();
                } catch (SQLException e) {
                    // Ignoring shutdown exception.
                }
            }
        }
    }
}
```

Related reference

Example: Returning a list of tables using the DatabaseMetaData interface
This example shows how to return a list of tables.

Example: Using native JDBC and IBM Toolbox for Java JDBC concurrently

This is an example of how to use the native JDBC connection and the IBM Toolbox for Java JDBC connection in a program.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
//////////////////////////////////////////////////////////////////////////////////
//
// GetConnections example.
//
// This program demonstrates being able to use both JDBC drivers at
// once in a program. Two Connection objects are created in this
// program. One is a native JDBC connection and one is a IBM Toolbox for Java
// JDBC connection.
//
// This technique is convenient because it allows you to use different
// JDBC drivers for different tasks concurrently. For example, the
// IBM Toolbox for Java JDBC driver is ideal for connecting to a remote IBM i
// server and the native JDBC driver is faster for local connections.
// You can use the strengths of each driver concurrently in your
// application by writing code similar to this example.
//
//////////////////////////////////////////////////////////////////////////////////
//
// This source is an example of the IBM Developer for Java JDBC driver.
// IBM grants you a nonexclusive license to use this as an example
// from which you can generate similar function tailored to
```

```
// your own specific needs.
//
// This sample code is provided by IBM for illustrative purposes
// only. These examples have not been thoroughly tested under all
// conditions. IBM, therefore, cannot guarantee or imply
// reliability, serviceability, or function of these programs.
//
// All programs contained herein are provided to you "AS IS"
// without any warranties of any kind.  The implied warranties of
// merchantability and fitness for a particular purpose are
// expressly disclaimed.
//
// IBM Developer Kit for Java
// (C) Copyright IBM Corp. 2001
// All rights reserved.
// US Government Users Restricted Rights -
// Use, duplication, or disclosure restricted
// by GSA ADP Schedule Contract with IBM Corp.
//
//////////////////////////////////////////////////////////////////////////////////
import java.sql.*;
import java.util.*;

public class GetConnections {

   public static void main(java.lang.String[] args)
   {
       // Verify input.
       if (args.length != 2) {
          System.out.println("Usage (CL command line):  java GetConnections PARM(<user>
<password>)");
          System.out.println(" where <user> is a valid IBM i user ID");
          System.out.println("   and <password> is the password for that user ID");
          System.exit(0);
       }

       // Register both drivers.
      try {
         Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
         Class.forName("com.ibm.as400.access.AS400JDBCDriver");
      } catch (ClassNotFoundException cnf) {
         System.out.println("ERROR: One of the JDBC drivers did not load.");
         System.exit(0);
      }

      try {
         // Obtain a connection with each driver.
         Connection conn1 = DriverManager.getConnection("jdbc:db2://localhost", args[0],
args[1]);
         Connection conn2 = DriverManager.getConnection("jdbc:as400://localhost", args[0],
args[1]);

         // Verify that they are different.
         if (conn1 instanceof com.ibm.db2.jdbc.app.DB2Connection)
            System.out.println("conn1 is running under the native JDBC driver.");
         else
            System.out.println("There is something wrong with conn1.");

         if (conn2 instanceof com.ibm.as400.access.AS400JDBCConnection)
            System.out.println("conn2 is running under the IBM Toolbox for Java JDBC driver.");
         else
            System.out.println("There is something wrong with conn2.");

         conn1.close();
         conn2.close();
      } catch (SQLException e) {
         System.out.println("ERROR: " + e.getMessage());
      }
   }
}
```

Example: Using PreparedStatement to obtain a ResultSet

This is an example of using a PreparedStatement object's executeQuery method to obtain a ResultSet.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
import java.sql.*;
import java.util.Properties;

public class PreparedStatementExample {

    public static void main(java.lang.String[] args)
    {
        // Load the following from a properties object.
        String DRIVER = "com.ibm.db2.jdbc.app.DB2Driver";
        String URL    = "jdbc:db2://*local";

        // Register the native JDBC driver. If the driver cannot
        // be registered, the test cannot continue.
        try {
            Class.forName(DRIVER);
        } catch (Exception e) {
            System.out.println("Driver failed to register.");
            System.out.println(e.getMessage());
            System.exit(1);
        }

        Connection c = null;
        Statement s = null;

        //    This program creates a table that is
        //    used by prepared statements later.
        try {
            // Create the connection properties.
            Properties properties = new Properties ();
            properties.put ("user", "userid");
            properties.put ("password", "password");

            // Connect to the local database.
            c = DriverManager.getConnection(URL, properties);

            // Create a Statement object.
            s = c.createStatement();
            // Delete the test table if it exists.  Note that
            // this example assumes throughout that the collection
            // MYLIBRARY exists on the system.
            try {
                s.executeUpdate("DROP TABLE MYLIBRARY.MYTABLE");
            } catch (SQLException e) {
                // Just continue... the table probably did not exist.
            }

            // Run an SQL statement that creates a table in the database.
            s.executeUpdate("CREATE TABLE MYLIBRARY.MYTABLE (NAME VARCHAR(20), ID INTEGER)");

        } catch (SQLException sqle) {
            System.out.println("Database processing has failed.");
            System.out.println("Reason: " + sqle.getMessage());
        } finally {
            // Close database resources
            try {
                if (s != null) {
                    s.close();
                }
            } catch (SQLException e) {
                System.out.println("Cleanup failed to close Statement.");
            }
        }


        //    This program then uses a prepared statement to insert many
        //    rows into the database.
        PreparedStatement ps = null;
        String[] nameArray = {"Rich", "Fred", "Mark", "Scott", "Jason",
            "John", "Jessica", "Blair", "Erica", "Barb"};
        try {
            // Create a PreparedStatement object that is used to insert data into the
            // table.
            ps = c.prepareStatement("INSERT INTO MYLIBRARY.MYTABLE (NAME, ID) VALUES (?, ?)");
```

```
            for (int i = 0; i < nameArray.length; i++) {
                ps.setString(1, nameArray[i]);      // Set the Name from our array.
                ps.setInt(2, i+1);                  // Set the ID.
                ps.executeUpdate();
            }

        } catch (SQLException sqle) {
            System.out.println("Database processing has failed.");
            System.out.println("Reason: " + sqle.getMessage());
        } finally {
            // Close database resources
            try {
                if (ps != null) {
                    ps.close();
                }
            } catch (SQLException e) {
                System.out.println("Cleanup failed to close Statement.");
            }
        }


        //    Use a prepared statement to query the database
        //    table that has been created and return data from it. In
        //    this example, the parameter used is arbitrarily set to
        //    5, meaning return all rows where the ID field is less than
        //    or equal to 5.
        try {
            ps = c.prepareStatement("SELECT * FROM MYLIBRARY.MYTABLE " +
                                    "WHERE ID <= ?");

            ps.setInt(1, 5);

            // Run an SQL query on the table.
            ResultSet rs = ps.executeQuery();
            // Display all the data in the table.
            while (rs.next()) {
                System.out.println("Employee " + rs.getString(1) + " has ID " + rs.getInt(2));
            }

        } catch (SQLException sqle) {
            System.out.println("Database processing has failed.");
            System.out.println("Reason: " + sqle.getMessage());
        } finally {
            // Close database resources
            try {
                if (ps != null) {
                    ps.close();
                }
            } catch (SQLException e) {
                System.out.println("Cleanup failed to close Statement.");
            }

            try {
                if (c != null) {
                    c.close();
                }
            } catch (SQLException e) {
                System.out.println("Cleanup failed to close Connection.");
            }

        }
    }
}
```

Related concepts

Creating and using PreparedStatements

The prepareStatement method is used to create new PreparedStatement objects. Unlike the createStatement method, the SQL statement must be supplied when the PreparedStatement object is created. At that time, the SQL statement is precompiled for use.

Processing PreparedStatements

Processing SQL statements with a PreparedStatement object is accomplished with the executeQuery, executeUpdate, and execute methods like Statement objects are processed. Unlike Statement versions, no parameters are passed on these methods because the SQL statement was already provided when the object was created. Because PreparedStatement extends Statement, applications can attempt to call versions of executeQuery, executeUpdate, and execute methods that take a SQL statement. Doing so results in an SQLException being thrown.

Example: Using the Statement object's executeUpdate method

This is an example of how to use the Statement object's executeUpdate method.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
import java.sql.*;
import java.util.Properties;

public class StatementExample {

    public static void main(java.lang.String[] args)
    {

        // Suggestion:  Load these from a properties object.
        String DRIVER = "com.ibm.db2.jdbc.app.DB2Driver";
        String URL    = "jdbc:db2://*local";

        // Register the native JDBC driver. If the driver cannot be
        // registered, the test cannot continue.
        try {
            Class.forName(DRIVER);
        } catch (Exception e) {
            System.out.println("Driver failed to register.");
            System.out.println(e.getMessage());
            System.exit(1);
        }

        Connection c = null;
        Statement s = null;

        try {
            // Create the connection properties.
            Properties properties = new Properties ();
            properties.put ("user", "userid");
            properties.put ("password", "password");

            // Connect to the local database.
            c = DriverManager.getConnection(URL, properties);

            // Create a Statement object.
            s = c.createStatement();
            // Delete the test table if it exists. Note: This
            // example assumes that the collection MYLIBRARY
            // exists on the system.
            try {
                s.executeUpdate("DROP TABLE MYLIBRARY.MYTABLE");
            } catch (SQLException e) {
                // Just continue... the table probably does not exist.
            }

            // Run an SQL statement that creates a table in the database.
            s.executeUpdate("CREATE TABLE MYLIBRARY.MYTABLE (NAME VARCHAR(20), ID INTEGER)");

            // Run some SQL statements that insert records into the table.
            s.executeUpdate("INSERT INTO MYLIBRARY.MYTABLE (NAME, ID) VALUES ('RICH', 123)");
            s.executeUpdate("INSERT INTO MYLIBRARY.MYTABLE (NAME, ID) VALUES ('FRED', 456)");
            s.executeUpdate("INSERT INTO MYLIBRARY.MYTABLE (NAME, ID) VALUES ('MARK', 789)");

            // Run an SQL query on the table.
            ResultSet rs = s.executeQuery("SELECT * FROM MYLIBRARY.MYTABLE");

            // Display all the data in the table.
            while (rs.next()) {
                System.out.println("Employee " + rs.getString(1) + " has ID " + rs.getInt(2));
            }

        } catch (SQLException sqle) {
            System.out.println("Database processing has failed.");
            System.out.println("Reason: " + sqle.getMessage());
        } finally {
            // Close database resources
            try {
                if (s != null) {
```

```
                    s.close();
                }
            } catch (SQLException e) {
                System.out.println("Cleanup failed to close Statement.");
            }
             }

            try {
                if (c != null) {
                    c.close();
                }
            } catch (SQLException e) {
                System.out.println("Cleanup failed to close Connection.");
            }
        }
    }
}
```

Examples: JAAS HelloWorld

These examples show you the three files that are needed to compile and run HelloWorld for JAAS.

HelloWorld.java

Here is the source for the file HelloWorld.java.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
/*
 * ===========================================================================
 * Licensed Materials - Property of IBM
 *
 * (C) Copyright IBM Corp. 2000 All Rights Reserved.
 *
 *  US Government Users Restricted Rights - Use, duplication or
 *  disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
 * ===========================================================================
 *
 * File: HelloWorld.java
 */

import java.io.*;
import java.util.*;
import java.security.Principal;
import java.security.PrivilegedAction;
import javax.security.auth.*;
import javax.security.auth.callback.*;
import javax.security.auth.login.*;
import javax.security.auth.spi.*;

/**
 * This SampleLogin application attempts to authenticate a user.
 *
 * If the user successfully authenticates itself,
 * the user name and number of Credentials is displayed.
 *
 * @version 1.1, 09/14/99
 */
public class HelloWorld {

    /**
     * Attempt to authenticate the user.
     */
    public static void main(String[] args) {
        // use the configured LoginModules for the "helloWorld" entry
        LoginContext lc = null;
        try {
            lc = new LoginContext("helloWorld", new MyCallbackHandler());
        } catch (LoginException le) {
            le.printStackTrace();
            System.exit(-1);
        }

        // the user has 3 attempts to authenticate successfully
        int i;
        for (i = 0; i < 3; i++) {
```

```
            try {

                // attempt authentication
                lc.login();

                // if we return with no exception, authentication succeeded
                break;

            } catch (AccountExpiredException aee) {

                System.out.println("Your account has expired");
                System.exit(-1);

            } catch (CredentialExpiredException cee) {

                System.out.println("Your credentials have expired.");
                System.exit(-1);

            } catch (FailedLoginException fle) {

                System.out.println("Authentication Failed");
                    try {
                        Thread.currentThread().sleep(3000);
                    } catch (Exception e) {
                        // ignore
                    }

            } catch (Exception e) {

                System.out.println("Unexpected Exception - unable to continue");
                e.printStackTrace();
                System.exit(-1);
            }
        }

        // did they fail three times?
        if (i == 3) {
            System.out.println("Sorry");
            System.exit(-1);
        }

        // Look at what Principals we have:
        Iterator principalIterator = lc.getSubject().getPrincipals().iterator();
        System.out.println("\n\nAuthenticated user has the following Principals:");
        while (principalIterator.hasNext()) {
            Principal p = (Principal)principalIterator.next();
            System.out.println("\t" + p.toString());
        }

        // Look at some Principal-based work:
        Subject.doAsPrivileged(lc.getSubject(), new PrivilegedAction() {
            public Object run() {
                System.out.println("\nYour java.home property: "
                                +System.getProperty("java.home"));

                System.out.println("\nYour user.home property: "
                                +System.getProperty("user.home"));

                File f = new File("foo.txt");
                System.out.print("\nfoo.txt does ");
                if (!f.exists()) System.out.print("not ");
                System.out.println("exist in your current directory");

                System.out.println("\nOh, by the way ...");

                try {
                    Thread.currentThread().sleep(2000);
                } catch (Exception e) {
                    // ignore
                }
                System.out.println("\n\nHello World!\n");
                return null;
            }
        }, null);
        System.exit(0);
    }
}


/**
 * The application must implement the CallbackHandler.
 *
```

```
 * This application is text-based.  Therefore it displays information
 * to the user using the OutputStreams System.out and System.err,
 * and gathers input from the user using the InputStream, System.in.
 */
class MyCallbackHandler implements CallbackHandler {

    /**
     * Invoke an array of Callbacks.
     *
     *
     * @param callbacks an array of Callback objects which contain
     *          the information requested by an underlying security
     *          service to be retrieved or displayed.
     *
     * @exception java.io.IOException if an input or output error occurs.
     *
     * @exception UnsupportedCallbackException if the implementation of this
     *          method does not support one or more of the Callbacks
     *          specified in the callbacks parameter.
     */
    public void handle(Callback[] callbacks)
    throws IOException, UnsupportedCallbackException {

    for (int i = 0; i < callbacks.length; i++) {
        if (callbacks[i] instanceof TextOutputCallback) {

        // display the message according to the specified type
        TextOutputCallback toc = (TextOutputCallback)callbacks[i];
        switch (toc.getMessageType()) {
        case TextOutputCallback.INFORMATION:
            System.out.println(toc.getMessage());
            break;
        case TextOutputCallback.ERROR:
            System.out.println("ERROR: " + toc.getMessage());
            break;
        case TextOutputCallback.WARNING:
            System.out.println("WARNING: " + toc.getMessage());
            break;
        default:
            throw new IOException("Unsupported message type: " +
                    toc.getMessageType());
        }

        } else if (callbacks[i] instanceof NameCallback) {

        // prompt the user for a user name
        NameCallback nc = (NameCallback)callbacks[i];

        // ignore the provided defaultName
        System.err.print(nc.getPrompt());
        System.err.flush();
        nc.setName((new BufferedReader
            (new InputStreamReader(System.in))).readLine());

        } else if (callbacks[i] instanceof PasswordCallback) {

        // prompt the user for sensitive information
        PasswordCallback pc = (PasswordCallback)callbacks[i];
        System.err.print(pc.getPrompt());
        System.err.flush();
        pc.setPassword(readPassword(System.in));

        } else {
        throw new UnsupportedCallbackException
            (callbacks[i], "Unrecognized Callback");
        }
    }
    }

    // Reads user password from given input stream.
    private char[] readPassword(InputStream in) throws IOException {

    char[] lineBuffer;
    char[] buf;
    int i;

    buf = lineBuffer = new char[128];

    int room = buf.length;
    int offset = 0;
    int c;
```

```
    loop:   while (true) {
        switch (c = in.read()) {
        case -1:
        case '\n':
        break loop;

        case '\r':
        int c2 = in.read();
        if ((c2 != '\n') && (c2 != -1)) {
            if (!(in instanceof PushbackInputStream)) {
            in = new PushbackInputStream(in);
            }
            ((PushbackInputStream)in).unread(c2);
        } else
            break loop;

        default:
        if (--room < 0) {
            buf = new char[offset + 128];
            room = buf.length - offset - 1;
            System.arraycopy(lineBuffer, 0, buf, 0, offset);
            Arrays.fill(lineBuffer, ' ');
            lineBuffer = buf;
        }
        buf[offset++] = (char) c;
        break;
        }
    }

    if (offset == 0) {
        return null;
    }

    char[] ret = new char[offset];
    System.arraycopy(buf, 0, ret, 0, offset);
    Arrays.fill(buf, ' ');

    return ret;
    }
}
```

HWLoginModule.java

Here is the source for HWLoginModule.java.

Note: Read the Code example disclaimer for important legal information.

```
/*
 * ===========================================================================
 * Licensed Materials - Property of IBM
 *
 * (C) Copyright IBM Corp. 2000 All Rights Reserved.
 *
 *  US Government Users Restricted Rights - Use, duplication or
 *  disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
 * ===========================================================================
 *
 * File: HWLoginModule.java
 */

package com.ibm.security;

import java.util.*;
import java.io.IOException;
import javax.security.auth.*;
import javax.security.auth.callback.*;
import javax.security.auth.login.*;
import javax.security.auth.spi.*;
import com.ibm.security.HWPrincipal;

/**
 * This LoginModule authenticates users with a password.
 *
 * This LoginModule only recognizes any user who enters
 *     the required password:   Go JAAS
 *
 * If the user successfully authenticates itself,
 * a HWPrincipal with the user name
 * is added to the Subject.
```

```
 *
 * This LoginModule recognizes the debug option.
 * If set to true in the login Configuration,
 * debug messages are sent to the output stream, System.out.
 *
 * @version 1.1, 09/10/99
 */
public class HWLoginModule implements LoginModule {

    // initial state
    private Subject subject;
    private CallbackHandler callbackHandler;
    private Map sharedState;
    private Map options;

    // configurable option
    private boolean debug = false;

    // the authentication status
    private boolean succeeded = false;
    private boolean commitSucceeded = false;

    // user name and password
    private String user name;
    private char[] password;

    private HWPrincipal userPrincipal;

    /**
     * Initialize this LoginModule.
     *
     * @param subject the Subject to be authenticated.
     *
     * @param callbackHandler a CallbackHandler for communicating
     *          with the end user (prompting for user names and
     *          passwords, for example).
     *
     * @param sharedState shared LoginModule state.
     *
     * @param options options specified in the login
     *          Configuration for this particular
     *          LoginModule.
     */
    public void initialize(Subject subject, CallbackHandler callbackHandler,
            Map sharedState, Map options) {

    this.subject = subject;
    this.callbackHandler = callbackHandler;
    this.sharedState = sharedState;
    this.options = options;

    // initialize any configured options
    debug = "true".equalsIgnoreCase((String)options.get("debug"));
    }

    /**
     * Authenticate the user by prompting for a user name and password.
     *
     *
     * @return true in all cases since this LoginModule
     *      should not be ignored.
     *
     * @exception FailedLoginException if the authentication fails.
     *
     * @exception LoginException if this LoginModule
     *      is unable to perform the authentication.
     */
    public boolean login() throws LoginException {

    // prompt for a user name and password
    if (callbackHandler == null)
        throw new LoginException("Error: no CallbackHandler available " +
            "to garner authentication information from the user");

    Callback[] callbacks = new Callback[2];
    callbacks[0] = new NameCallback("\n\nHWModule user name: ");
    callbacks[1] = new PasswordCallback("HWModule password: ", false);

    try {
        callbackHandler.handle(callbacks);
        user name = ((NameCallback)callbacks[0]).getName();
        char[] tmpPassword = ((PasswordCallback)callbacks[1]).getPassword();
```

```
        if (tmpPassword == null) {
        // treat a NULL password as an empty password
        tmpPassword = new char[0];
        }
        password = new char[tmpPassword.length];
        System.arraycopy(tmpPassword, 0,
            password, 0, tmpPassword.length);
        ((PasswordCallback)callbacks[1]).clearPassword();

    } catch (java.io.IOException ioe) {
        throw new LoginException(ioe.toString());
    } catch (UnsupportedCallbackException uce) {
        throw new LoginException("Error: " + uce.getCallback().toString() +
        " not available to garner authentication information " +
        "from the user");
    }

    // print debugging information
    if (debug) {
        System.out.println("\n\n\t[HWLoginModule] " +
                "user entered user name: " +
                user name);
        System.out.print("\t[HWLoginModule] " +
                "user entered password: ");
        for (int i = 0; i > password.length; i++)
        System.out.print(password[i]);
        System.out.println();
    }

    // verify the password
    if (password.length == 7 &&
        password[0] == 'G' &&
        password[1] == 'o' &&
        password[2] == ' ' &&
        password[3] == 'J' &&
        password[4] == 'A' &&
        password[5] == 'A' &&
        password[6] == 'S') {

        // authentication succeeded!!!
        if (debug)
        System.out.println("\n\t[HWLoginModule] " +
                "authentication succeeded");
        succeeded = true;
        return true;
    } else {

        // authentication failed -- clean out state
        if (debug)
        System.out.println("\n\t[HWLoginModule] " +
                "authentication failed");
        succeeded = false;
        user name = null;
        for (int i = 0; i < password.length; i++)
        password[i] = ' ';
        password = null;
        throw new FailedLoginException("Password Incorrect");
    }
    }

    /**
     * This method is called if the overall authentication of LoginContext
     * succeeded
     * (the relevant REQUIRED, REQUISITE, SUFFICIENT and OPTIONAL LoginModules
     * succeeded).
     *
     * If this LoginModule authentication attempt
     * succeeded (checked by retrieving the private state saved by the
     * login method), then this method associates a
     * SolarisPrincipal
     * with the Subject located in the
     * LoginModule.  If this LoginModule
     * authentication attempt failed, then this method removes
     * any state that was originally saved.
     *
     * @exception LoginException if the commit fails.
     *
     * @return true if the login and commit LoginModule
     *      attempts succeeded, or false otherwise.
     */
    public boolean commit() throws LoginException {
    if (succeeded == false) {
```

```
            return false;
        } else {
            // add a Principal (authenticated identity)
            // to the Subject

            // assume the user we authenticated is the HWPrincipal
            userPrincipal = new HWPrincipal(user name);
            final Subject s = subject;
            final HWPrincipal sp = userPrincipal;
            java.security.AccessController.doPrivileged
            (new java.security.PrivilegedAction() {
            public Object run() {
                if (!s.getPrincipals().contains(sp))
                s.getPrincipals().add(sp);
                return null;
            }
            });

            if (debug) {
            System.out.println("\t[HWLoginModule] " +
                    "added HWPrincipal to Subject");
            }

            // in any case, clean out state
            user name = null;
            for (int i = 0; i > password.length; i++)
            password[i] = ' ';
            password = null;

            commitSucceeded = true;
            return true;
        }
        }

        /**
         * This method is called if the overall authentication of LoginContext
         * failed.
         * (the relevant REQUIRED, REQUISITE, SUFFICIENT and OPTIONAL LoginModules
         * did not succeed).
         *
         * If this authentication attempt of LoginModule
         * succeeded (checked by retrieving the private state saved by the
         * login and commit methods),
         * then this method cleans up any state that was originally saved.
         *
         * @exception LoginException if the abort fails.
         *
         * @return false if this login or commit attempt for LoginModule
         *      failed, and true otherwise.
         */
        public boolean abort() throws LoginException {
        if (succeeded == false) {
            return false;
        } else if (succeeded == true && commitSucceeded == false) {
            // login succeeded but overall authentication failed
            succeeded = false;
            user name = null;
            if (password != null) {
            for (int i = 0; i > password.length; i++)
                password[i] = ' ';
            password = null;
            }
            userPrincipal = null;
        } else {
            // overall authentication succeeded and commit succeeded,
            // but another commit failed
            logout();
        }
        return true;
        }

        /**
         * Logout the user.
         *
         * This method removes the HWPrincipal
         * that was added by the commit method.
         *
         * @exception LoginException if the logout fails.
         *
         * @return true in all cases since this LoginModule
         *          should not be ignored.
         */
```

```
    public boolean logout() throws LoginException {

    final Subject s = subject;
    final HWPrincipal sp = userPrincipal;
    java.security.AccessController.doPrivileged
        (new java.security.PrivilegedAction() {
        public Object run() {
        s.getPrincipals().remove(sp);
        return null;
        }
    });

    succeeded = false;
    succeeded = commitSucceeded;
    user name = null;
    if (password != null) {
        for (int i = 0; i > password.length; i++)
        password[i] = ' ';
        password = null;
    }
    userPrincipal = null;
    return true;
    }
}
```

HWPrincipal.java

Here is the source for HWPrincipal.java.

Note: Read the Code example disclaimer for important legal information.

```
/*
 * ===========================================================================
 * Licensed Materials - Property of IBM
 *
 * (C) Copyright IBM Corp. 2000 All Rights Reserved.
 *
 *  US Government Users Restricted Rights - Use, duplication or
 *  disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
 * ===========================================================================
 *
 * File: HWPrincipal.java
 */

package com.ibm.security;

import java.security.Principal;

/**
 * This class implements the Principal interface
 * and represents a HelloWorld tester.
 *
 * @version 1.1, 09/10/99
 * @author D. Kent Soper
 */
public class HWPrincipal implements Principal, java.io.Serializable {

    private String name;

    /*
     * Create a HWPrincipal with the supplied name.
     */
    public HWPrincipal(String name) {
        if (name == null)
            throw new NullPointerException("illegal null input");

        this.name = name;
    }

    /*
     * Return the name for the HWPrincipal.
     */
    public String getName() {
        return name;
    }

    /*
     * Return a string representation of the HWPrincipal.
     */
```

```
    public String toString() {
        return("HWPrincipal:  " + name);
    }

    /*
     * Compares the specified Object with the HWPrincipal for equality.
     * Returns true if the given object is also a HWPrincipal and the
     * two HWPrincipals have the same user name.
     */
    public boolean equals(Object o) {
        if (o == null)
            return false;

        if (this == o)
            return true;

        if (!(o instanceof HWPrincipal))
            return false;
        HWPrincipal that = (HWPrincipal)o;

        if (this.getName().equals(that.getName()))
            return true;
        return false;
    }

    /*
     * Return a hash code for the HWPrincipal.
     */
    public int hashCode() {
        return name.hashCode();
    }
}
```

Example: JAAS SampleThreadSubjectLogin

This example shows you the implementation of the SampleThreadSubjectLogin class.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
//////////////////////////////////////////////////////////////////////
//
// File Name:   SampleThreadSubjectLogin.java
//
// Class:       SampleThreadSubjectLogin
//
/////////////////////////////////////////////////////////////////////
//
// CHANGE ACTIVITY:
//
//
// END CHANGE ACTIVITY
//
////////////////////////////////////////////////////////////////////

import com.ibm.security.auth.ThreadSubject;

import com.ibm.as400.access.*;

import java.io.*;

import java.util.*;

import java.security.Principal;

import javax.security.auth.*;

import javax.security.auth.callback.*;

import javax.security.auth.login.*;


/**
 * This SampleThreadSubjectLogin application authenticates a single
 * user, swaps the OS thread identity to the authenticated user,
 * and then writes "Hello World" into a privately authorized
 * file, thread.txt, in the user's test directory.
 *
 * The user is requested to enter the user id and password to
```

```
 * authenticate.
 *
 * If successful, the user name and number of Credentials
 * are displayed.
 *
 *

 Setup and run instructions:

 1) Create a new user, JAAS14, by invoking
 "CRTUSRPRF USRPRF(JAAS14) PASSWORD() TEXT('JAAS sample user id')"
 with *USER class authority.

 2) Allocate a dummy test file, "yourTestDir/thread.txt", and
 privately grant JAAS14 *RWX authority to it for write access.

 3) Copy SampleThreadSubjectLogin.java into your test directory.

 4) Change the current directory to your test directory and compile the
 java source code.

   Enter -

   strqsh

   cd 'yourTestDir'

   javac  -classpath /qibm/proddata/os400/java400/ext/jaas14.jar:
                /QIBM/ProdData/HTTP/Public/jt400/lib/jt400.jar:.
     -d ./classes
     *.java

 5) Copy threadLogin.config, threadJaas.policy, and threadJava2.policy
 into your test directory.

 6) If not already done, add the symbolic link to the extension
 directory for the jaas14.jar file.
 The extension class loader should normally load the JAR file.

 ADDLNK OBJ('/QIBM/ProdData/OS400/Java400/ext/jaas14.jar')
   NEWLNK('/QIBM/ProdData/Java400/jdk14/lib/ext/jaas14.jar')


 7) If not already done to run this sample, add the symbolic link to the extension
 directory for the jt400.jar and jt400ntv.jar files. This causes these
 files to be loaded by the extension class loader. The application class loader
 can also load these files by including them in the CLASSPATH.
 If these files are loaded from the class path directory,
 do not add the symbolic link to the extension directory.
 The jaas14.jar file requires these JAR files for the credential
 implementation classes which are part of the IBM Toolbox
 for Java Licensed Program Product.
 (See the IBM Toolbox for Java topic for documentation
 on the credential classes found in the left frame
 under Security Classes => Authentication. Select the link to the
 ProfileTokenCredential class. At the top select 'This Package' for the
 entire com/ibm/as400/security/auth Java package. Javadoc for the
 authentication classes can also be found by selecting 'Javadoc' =>
 'Access Classes' on the left frame. Select 'All Packages' at the top
 and look for the com.ibm.as400.security.* packages)


 ADDLNK OBJ('/QIBM/ProdData/HTTP/Public/jt400/lib/jt400.jar')
   NEWLNK('/QIBM/ProdData/Java400/jdk14/lib/ext/jt400.jar')

 ADDLNK OBJ('/QIBM/ProdData/OS400/jt400/lib/jt400Native.jar')
   NEWLNK('/QIBM/ProdData/Java400/jdk14/lib/ext/jt400Native.jar')

/////////////////////////////////////
 IMPORTANT NOTES -
/////////////////////////////////////

 When updating the Java2 policy files
 for a real application remember to grant the
 appropriate permissions to the actual locations of the IBM Toolbox
 for Java JAR files. Even though they are symbolically linked to
 the extension directories previously listed which are granted
 java.security.AllPermission in the
 ${java.home}/lib/security/java.policy file, authorization is based on
 the actual location of the JAR files.

 For example, to successfully use the credential classes
```

```
 in IBM Toolbox for Java, you would add the below to your application's
 Java2 policy file -

 grant  codeBase "file:/QIBM/ProdData/HTTP/Public/jt400/lib/jt400.jar"
 {
   permission javax.security.auth.AuthPermission "modifyThreadIdentity";
   permission java.lang.RuntimePermission "loadLibrary.*";
   permission java.lang.RuntimePermission "writeFileDescriptor";
   permission java.lang.RuntimePermission "readFileDescriptor";
 }

 You also need to add these permissions for the application's
 codeBase since the operations performed by the IBM Toolbox
 for Java JAR files do not run in privileged mode.

 This sample already grants these permissions to all java classes by
 omitting the codeBase parameter in the threadJava2.policy file.

 8) Make sure the Host Servers are started and running.
 The ProfileTokenCredential classes which reside in IBM Toolbox for Java,
 i.e. jt400.jar, are used as the credentials that are attached
 to the authenticated subject by the SampleThreadSubjectLogin.java
 program. The IBM Toolbox for Java credential classes require access
 to the Host Servers.

 9) Invoke SampleThreadSubjectLogin while signed on as a user that
 does not have access to 'yourTestDir/thread.txt'.

 10) Start the sample by entering the following CL commands  =>

 CHGCURDIR DIR('yourTestDir')

 JAVA CLASS(SampleThreadSubjectLogin)
  CLASSPATH('yourTestDir/classes')
  PROP(((java.security.manager)
       (java.security.auth.login.config
        'yourTestDir/threadLogin.config')
       (java.security.policy
        'yourTestDir/threadJava2.policy')
       (java.security.auth.policy
        'yourTestDir/threadJaas.policy'))

 Enter the user id and password when prompted from step 1.

 11) Check yourTestDir/thread.txt for the "Hello World" entry.

 *
 **/


public class SampleThreadSubjectLogin {
/**
 * Attempt to authenticate the user.
 *
 * @param args
 *      Input arguments for this application (ignored).
 *
 */
    public static void main(String[] args) {

    // use the configured LoginModules for the "AS400ToolboxApp" entry
    LoginContext lc = null;
    try {
        // if provided, the same subject is used for multiple login attempts
        lc = new LoginContext("AS400ToolboxApp",
                  new Subject(),
                  new SampleCBHandler());
    } catch (LoginException le) {
        le.printStackTrace();
        System.exit(-1);
    }

    // the user has 3 attempts to authenticate successfully
    int i;
    for (i = 0; i < 3; i++) {
        try {

        // attempt authentication
        lc.login();

        // if we return with no exception, authentication succeeded
        break;
```

```
            } catch (AccountExpiredException aee) {

            System.out.println("Your account has expired");
            System.exit(-1);

            } catch (CredentialExpiredException cee) {

            System.out.println("Your credentials have expired.");
            System.exit(-1);

            } catch (FailedLoginException fle) {

            System.out.println("Authentication Failed");
            try {
                Thread.currentThread().sleep(3000);
            } catch (Exception e) {
                // ignore
            }

            } catch (Exception e) {

            System.out.println("Unexpected Exception - unable to continue");
            e.printStackTrace();
            System.exit(-1);
            }
        }

        // did they fail three times?
        if (i == 3) {
            System.out.println("Sorry authentication failed");
            System.exit(-1);
        }


        // display authenticated principals & credentials
        System.out.println("Authentication Succeeded");

        System.out.println("Principals:");

        Iterator itr = lc.getSubject().getPrincipals().iterator();

        while (itr.hasNext())
            System.out.println(itr.next());

        itr = lc.getSubject().getPrivateCredentials().iterator();

        while (itr.hasNext())
            System.out.println(itr.next());

        itr = lc.getSubject().getPublicCredentials().iterator();

        while (itr.hasNext())
            System.out.println(itr.next());


            // let's do some Principal-based work:
        ThreadSubject.doAsPrivileged(lc.getSubject(), new java.security.PrivilegedAction() {
               public Object run() {
               System.out.println("\nYour java.home property: "
                      +System.getProperty("java.home"));
               System.out.println("\nYour user.home property: "
                          +System.getProperty("user.home"));
               File f = new File("thread.txt");
               System.out.print("\nthread.txt does ");
               if (!f.exists()) System.out.print("not ");
               System.out.println("exist in your current directory");

               try {
                   // write "Hello World number   x"  into thread.txt
                   PrintStream ps = new PrintStream(new FileOutputStream("thread.txt", true), true);

                   long flen = f.length();
                   ps.println("Hello World number    " +
                      Long.toString(flen/22) +
                      "\n");
                   ps.close();
               } catch (Exception e) {
                   e.printStackTrace();
               }

               System.out.println("\nOh, by the way, " + SampleThreadSubjectLogin.getCurrentUser());
```

```
            try {
                Thread.currentThread().sleep(2000);
            } catch (Exception e) {
             // ignore
                }
            System.out.println("\n\nHello World!\n");
            return null;
            }
        }, null);

        System.exit(0);

    }// end main()


// Returns the current OS identity for the main thread of the application.
// (This routine uses classes from IBM Toolbox for Java)
// Note - Applications running on a secondary thread cannot use this API to determine the
current user.
    static public String getCurrentUser() {

    try {
        AS400 localSys = new AS400("localhost", "*CURRENT", "*CURRENT");

        int ccsid = localSys.getCcsid();
        ProgramCall qusrjobi = new ProgramCall(localSys);
        ProgramParameter[] parms = new ProgramParameter[6];

        int rLength = 100;
        parms[0] = new ProgramParameter(rLength);
        parms[1] = new ProgramParameter(new AS400Bin4().toBytes(rLength));
        parms[2] = new ProgramParameter(new AS400Text(8, ccsid, localSys).toBytes("JOBI0600"));
        parms[3] = new ProgramParameter(new AS400Text(26,ccsid, localSys).toBytes("*"));
        parms[4] = new ProgramParameter(new AS400Text(16,ccsid, localSys).toBytes(""));
        parms[5] = new ProgramParameter(new AS400Bin4().toBytes(0));

        qusrjobi.setProgram(QSYSObjectPathName.toPath("QSYS", "QUSRJOBI", "PGM"), parms);
        AS400Text uidText = new AS400Text(10, ccsid, localSys);

    // Invoke the QUSRJOBI API
        qusrjobi.run();

        byte[] uidBytes = new byte[10];
        System.arraycopy((qusrjobi.getParameterList())[0].getOutputData(), 90, uidBytes, 0, 10);

        return ((String)(uidText.toObject(uidBytes))).trim();
    }


    catch (Exception e) {
        e.printStackTrace();
    }

    return "";
    }

} //end SampleThreadSubjectLogin class



/**
 * A CallbackHandler is passed to underlying security
 * services so that they may interact with the application
 * to retrieve specific authentication data,
 * such as user names and passwords, or to display certain
 * information, such as error and warning messages.
 *
 * CallbackHandlers are implemented in an application
 * and platform-dependent fashion. The implementation decides
 * how to retrieve and display information depending on the
 * Callbacks passed to it.
 *
 * This class provides a sample CallbackHandler. However, it is
 * not intended to fulfill the requirements of production applications.
 * As indicated, the CallbackHandler is ultimately considered to
 * be application-dependent, as individual applications have
 * unique error checking, data handling, and user
 * interface requirements.
 *
 * The following callbacks are handled:
 *
```

- *
- NameCallback *
- PasswordCallback *
- TextOutputCallback *

```
 *
 * For simplicity, prompting is handled interactively through
 * standard input and output. However, it is worth noting
 * that when standard input is provided by the console, this
 * approach allows passwords to be viewed as they are
 * typed. This should be avoided in production
 * applications.
 *
 * This CallbackHandler also allows a name and password
 * to be acquired through an alternative mechanism
 * and set directly on the handler to bypass the need for
 * user interaction on the respective Callbacks.
 *
 */
class SampleCBHandler implements CallbackHandler {
    private String name_ = null;
    private String password_ = null;
/**
 * Constructs a new SampleCBHandler.
 *
 */
public SampleCBHandler() {
    this(null, null);
}
/**
 * Constructs a new SampleCBHandler.
 *
 * A name and password can optionally be specified in
 * order to bypass the need to prompt for information
 * on the respective Callbacks.
 *
 * @param name
 *      The default value for name callbacks. A null
 *      value indicates that the user should be
 *      prompted for this information. A non-null value
 *      cannot be zero length or exceed 10 characters.
 *
 * @param password
 *      The default value for password callbacks. A null
 *      value indicates that the user should be
 *      prompted for this information. A non-null value
 *      cannot be zero length or exceed 10 characters.
 */
public SampleCBHandler(String name, String password) {
    if (name != null)
        if ((name.length()==0) || (name.length()>10))
            throw new IllegalArgumentException("name");
    name_ = name;

    if (password != null)
        if ((password.length()==0) || (password.length()>10))
            throw new IllegalArgumentException("password");
    password_ = password;
}
/**
 * Handle the given name callback.
 *
 * First check to see if a name has been passed in
 * on the constructor.  If so, assign it to the
 * callback and bypass the prompt.
 *
 * If a value has not been preset, attempt to prompt
 * for the name using standard input and output.
 *
 * @param c
 *      The NameCallback.
 *
 * @exception java.io.IOException
 *      If an input or output error occurs.
 *
 */
private void handleNameCallback(NameCallback c) throws IOException {
```

```
    // Check for cached value
    if (name_ != null) {
        c.setName(name_);
        return;
    }
    // No preset value; attempt stdin/out
    c.setName(
        stdIOReadName(c.getPrompt(), 10));
}
/**
 * Handle the given name callback.
 *
 * First check to see if a password has been passed
 * in on the constructor.  If so, assign it to the
 * callback and bypass the prompt.
 *
 * If a value has not been preset, attempt to prompt
 * for the password using standard input and output.
 *
 * @param c
 *      The PasswordCallback.
 *
 * @exception java.io.IOException
 *      If an input or output error occurs.
 *
 */
private void handlePasswordCallback(PasswordCallback c) throws IOException {
    // Check for cached value
    if (password_ != null) {
        c.setPassword(password_.toCharArray());
        return;
    }

    // No preset value; attempt stdin/out
    // Note - Not for production use.
    //        Password is not concealed by standard console I/O
    if (c.isEchoOn())
        c.setPassword(
            stdIOReadName(c.getPrompt(), 10).toCharArray());
    else
    {

        // Note - Password is not concealed by standard console I/O
        c.setPassword(stdIOReadName(c.getPrompt(), 10).toCharArray());

    }
}
/**
 * Handle the given text output callback.
 *
 * If the text is informational or a warning,
 * text is written to standard output. If the
 * callback defines an error message, text is
 * written to standard error.
 *
 * @param c
 *      The TextOutputCallback.
 *
 * @exception java.io.IOException
 *      If an input or output error occurs.
 *
 */
private void handleTextOutputCallback(TextOutputCallback c) throws IOException {
    if (c.getMessageType() == TextOutputCallback.ERROR)
        System.err.println(c.getMessage());
    else
        System.out.println(c.getMessage());
}
/**
 * Retrieve or display the information requested in the
 * provided Callbacks.
 *
 * The handle method implementation
 * checks the instance(s) of the Callback
 * object(s) passed in to retrieve or display the
 * requested information.
 *
 * @param callbacks
 *      An array of Callback objects provided
 *      by an underlying security service which contains
 *      the information requested to be retrieved or displayed.
 *
```

```
 * @exception java.io.IOException
 *      If an input or output error occurs.
 *
 * @exception UnsupportedCallbackException
 *      If the implementation of this method does not support
 *      one or more of the Callbacks specified in the
 *      callbacks parameter.
 *
 */
public void handle(Callback[] callbacks)
    throws IOException, UnsupportedCallbackException
{
    for (int i=0; i<callbacks.length; i++) {
        Callback c = callbacks[i];

        if (c instanceof NameCallback)
            handleNameCallback((NameCallback)c);
        else if (c instanceof PasswordCallback)
            handlePasswordCallback((PasswordCallback)c);
        else if (c instanceof TextOutputCallback)
            handleTextOutputCallback((TextOutputCallback)c);
        else
            throw new UnsupportedCallbackException
                (callbacks[i]);
    }
}
/**
 * Displays the given string using standard output,
 * followed by a space to separate from subsequent
 * input.
 *
 * @param prompt
 *      The text to display.
 *
 * @exception IOException
 *      If an input or output error occurs.
 *
 */
private void stdIOPrompt(String prompt) throws IOException {
    System.out.print(prompt + ' ');
    System.out.flush();
}
/**
 * Reads a String from standard input, stopped at
 * maxLength or by a newline.
 *
 * @param prompt
 *      The text to display to standard output immediately
 *      prior to reading the requested value.
 *
 * @param maxLength
 *      Maximum length of the String to return.
 *
 * @return
 *      The entered string. The value returned does
 *      not contain leading or trailing whitespace
 *      and is converted to uppercase.
 *
 * @exception IOException
 *      If an input or output error occurs.
 *
 */
private String stdIOReadName(String prompt, int maxLength) throws IOException {
    stdIOPrompt(prompt);
    String s =
        (new BufferedReader
            (new InputStreamReader(System.in))).readLine().trim();
    if (s.length() < maxLength)
        s = s.substring(0,maxLength);
    return s.toUpperCase();
}

}//end SampleCBHandler class
```

Sample: IBM JGSS non-JAAS client program

Use this JGSS sample client in conjunction with the JGSS sample server.

For more information about using the sample client program, see "Samples: Downloading and running the sample JGSS programs" on page 347.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
// IBM JGSS 1.0 Sample Client Program

package com.ibm.security.jgss.test;
import org.ietf.jgss.*;
import com.ibm.security.jgss.Debug;

import java.io.*;
import java.net.*;
import java.util.*;

/**
 * A JGSS sample client;
 * to be used in conjunction with the JGSS sample server.
 * The client first establishes a context with the server
 * and then sends wrapped message followed by a MIC to the server.
 * The MIC is calculated over the plain text that was wrapped.
 * The client requires to server to authenticate itself
 * (mutual authentication) during context establishment.
 * It also delegates its credentials to the server.
 *
 * It sets the JAVA variable
 * javax.security.auth.useSubjectCredsOnly to false
 * so that JGSS will not acquire credentials through JAAS.
 *
 * The client takes input parameters, and complements it
 * with information from the jgss.ini file; any required input not
 * supplied on the command line is taking from the jgss.ini file.
 *
 * Usage: Client [options]
 *
 * The -? option produces a help message including supported options.
 *
 * This sample client does not use JAAS.
 * The client can be run against the JAAS sample client and server.
 * See {@link JAASClient JAASClient} for a sample client that uses JAAS.
 */

class Client
{
    private Util testUtil      = null;
    private String myName      = null;
    private GSSName gssName    = null;
    private String serverName = null;
    private int servicePort    = 0;
    private GSSManager mgr     = GSSManager.getInstance();
    private GSSName service    = null;
    private GSSContext context        = null;
    private String program     = "Client";
    private String debugPrefix = "Client: ";
    private TCPComms tcp       = null;
    private String data               = null;
    private byte[] dataBytes   = null;
    private String serviceHostname= null;
    private GSSCredential gssCred = null;

    private static Debug debug        = new Debug();

    private static final String usageString =
          "\t[-?] [-d | -n name] [-s serverName]"
        + "\n\t[-h serverHost [:port]] [-p port] [-m msg]"
        + "\n"
        + "\n  -?\t\t\thelp; produces this message"
        + "\n  -n name\t\tthe client's principal name (without realm)"
        + "\n  -s serverName\t\tthe server's principal name (without realm)"
        + "\n  -h serverHost[:port]\tthe server's hostname"
        +         " (and optional port number)"
        + "\n  -p port\t\tthe port on which the server will be listening"
        + "\n  -m msg\t\tmessage to send to the server";
```

```
    // Caller must call initialize (may need to call processArgs first).
    public Client (String programName) throws Exception
    {
        testUtil = new Util();
        if (programName != null)
        {
            program = programName;
            debugPrefix = programName + ": ";
        }
    }

    // Caller must call initialize (may need to call processArgs first).
    Client (String programName, boolean useSubjectCredsOnly) throws Exception
    {
        this(programName);
        setUseSubjectCredsOnly(useSubjectCredsOnly);
    }

    public Client(GSSCredential myCred,
                  String serverNameWithoutRealm,
                  String serverHostname,
                  int serverPort,
                  String message)
        throws Exception
    {
        testUtil = new Util();

        if (myCred != null)
        {
            gssCred = myCred;
        }
        else
        {
            throw new GSSException(GSSException.NO_CRED, 0,
                                       "Null input credential");
        }

        init(serverNameWithoutRealm, serverHostname, serverPort, message);
    }

    void setUseSubjectCredsOnly(boolean useSubjectCredsOnly)
    {
        final String subjectOnly = useSubjectCredsOnly ? "true" : "false";
        final String property = "javax.security.auth.useSubjectCredsOnly";

        String temp = (String)java.security.AccessController.doPrivileged(
                        new sun.security.action.GetPropertyAction(property));

        if (temp == null)
        {
            debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
              + "setting useSubjectCredsOnly property to "
              + useSubjectCredsOnly);

            // Property not set. Set it to the specified value.

            java.security.AccessController.doPrivileged(
                 new java.security.PrivilegedAction() {
                   public Object run() {
                      System.setProperty(property, subjectOnly);
                      return null;
                   }
                 });
        }
        else
        {
            debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
              + "useSubjectCredsOnly property already set "
              + "in JVM  to " + temp);
        }
    }

    private void init(String myNameWithoutRealm,
                      String serverNameWithoutRealm,
                      String serverHostname,
                      int    serverPort,
                      String message) throws Exception
    {
        myName = myNameWithoutRealm;
        init(serverNameWithoutRealm, serverHostname, serverPort, message);
```

```
    }

    private void init(String serverNameWithoutRealm,
                      String serverHostname,
                      int    serverPort,
                      String message) throws Exception
    {
        // peer's name
        if (serverNameWithoutRealm != null)
        {
            this.serverName = serverNameWithoutRealm;
        }
        else
        {
            this.serverName = testUtil.getDefaultServicePrincipalWithoutRealm();
        }

        // peer's host
        if (serverHostname != null)
        {
            this.serviceHostname = serverHostname;
        }
        else
        {
            this.serviceHostname = testUtil.getDefaultServiceHostname();
        }

        // peer's port
        if (serverPort > 0)
        {
            this.servicePort = serverPort;
        }
        else
        {
            this.servicePort = testUtil.getDefaultServicePort();
        }

        // message for peer
        if (message != null)
        {
            this.data = message;
        }
        else
        {
            this.data = "The quick brown fox jumps over the lazy dog";
        }

        this.dataBytes = this.data.getBytes();

        tcp = new TCPComms(serviceHostname, servicePort);
    }


    void initialize() throws Exception
    {
         Oid krb5MechanismOid = new Oid("1.2.840.113554.1.2.2");

        if (gssCred == null)
        {
            if (myName != null)
            {
                debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
                                   + "creating GSSName USER_NAME for "
                                   + myName);

                gssName = mgr.createName(
                               myName,
                               GSSName.NT_USER_NAME,
                                krb5MechanismOid);

                debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
                                   + "Canonicalized GSSName=" + gssName);
            }
            else
                gssName = null; // for default credentials


            debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix + "creating"
               + ((gssName == null)? " default " : " ")
               + "credential");

            gssCred = mgr.createCredential(
```

```
                              gssName,
                              GSSCredential.DEFAULT_LIFETIME,
                              (Oid)null,
                              GSSCredential.INITIATE_ONLY);
        if (gssName == null)
        {
            gssName = gssCred.getName();

            myName  = gssName.toString();

            debug.out(Debug.OPTS_CAT_APPLICATION,
                debugPrefix + "default credential principal=" + myName);
        }
    }

    debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix + gssCred);

    debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
       + "creating canonicalized GSSName for serverName " + serverName);

    service = mgr.createName(serverName,
                             GSSName.NT_HOSTBASED_SERVICE,
                              krb5MechanismOid);

    debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
       + "Canonicalized server name = " + service);

    debug.out(Debug.OPTS_CAT_APPLICATION,
                        debugPrefix + "Raw data=" + data);
}


void establishContext(BitSet flags) throws Exception
{
    try {

      debug.out(Debug.OPTS_CAT_APPLICATION,
                        debugPrefix + "creating GSScontext");

      Oid defaultMech = null;
      context = mgr.createContext(service, defaultMech, gssCred,
                                  GSSContext.INDEFINITE_LIFETIME);

      if (flags != null)
      {
          if (flags.get(Util.CONTEXT_OPTS_MUTUAL))
          {
              debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
                   + "requesting mutualAuthn");

              context.requestMutualAuth(true);
          }

          if (flags.get(Util.CONTEXT_OPTS_INTEG))
          {
              debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
                   + "requesting integrity");

              context.requestInteg(true);
          }

          if (flags.get(Util.CONTEXT_OPTS_CONF))
          {
              context.requestConf(true);
              debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
                      + "requesting confidentiality");
          }

          if (flags.get(Util.CONTEXT_OPTS_DELEG))
          {
              context.requestCredDeleg(true);
              debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
                       + "requesting delegation");
          }

          if (flags.get(Util.CONTEXT_OPTS_REPLAY))
          {
              context.requestReplayDet(true);
              debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
                  + "requesting replay detection");
          }
```

```
            if (flags.get(Util.CONTEXT_OPTS_SEQ))
            {
                context.requestSequenceDet(true);
                debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
                         + "requesting out-of-sequence detection");
            }
            // Add more later!
        }

        byte[] response = null;
        byte[] request = null;
        int len = 0;
        boolean done = false;
        do {
            debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
                + "Calling initSecContext");

            request = context.initSecContext(response, 0, len);

            if (request != null)
            {
                debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
                  + "Sending initial context token");

                tcp.send(request);
            }
            done = context.isEstablished();

            if (!done)
            {
                debug.out(Debug.OPTS_CAT_APPLICATION,
                    debugPrefix + "Receiving response token");

                byte[] temp = tcp.receive();
                response = temp;
                len = response.length;
            }
        } while(!done);

        debug.out(Debug.OPTS_CAT_APPLICATION,
              debugPrefix + "context established with acceptor");

      } catch (Exception exc) {
          exc.printStackTrace();
          throw exc;
      }
  }

  void doMIC() throws Exception
  {
      debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix + "generating MIC");
      byte[] mic = context.getMIC(dataBytes, 0, dataBytes.length, null);

      if (mic != null)
      {
          debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix + "sending MIC");
          tcp.send(mic);
      }
      else
          debug.out(Debug.OPTS_CAT_APPLICATION,
                           debugPrefix + "getMIC Failed");
  }

  void doWrap() throws Exception
  {
      MessageProp mp = new MessageProp(true);
      mp.setPrivacy(context.getConfState());

      debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix + "wrapping message");

      byte[] wrapped = context.wrap(dataBytes, 0, dataBytes.length, mp);

      if (wrapped != null)
      {
          debug.out(Debug.OPTS_CAT_APPLICATION,
                  debugPrefix + "sending wrapped message");

          tcp.send(wrapped);
      }
      else
          debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix + "wrap Failed");
  }
```

```
    void printUsage()
    {
        System.out.println(program + usageString);
    }

    void processArgs(String[] args) throws Exception
    {
        String port            = null;
        String myName          = null;
        int servicePort               = 0;
        String serviceHostname = null;

        String sHost = null;
        String msg = null;

        GetOptions options = new GetOptions(args, "?h:p:m:n:s:");
        int ch = -1;
        while ((ch = options.getopt()) != options.optEOF)
        {
            switch(ch)
            {
                case '?':
                    printUsage();
                    System.exit(1);

                case 'h':
                    if (sHost == null)
                    {
                        sHost = options.optArgGet();
                        int p = sHost.indexOf(':');
                        if (p != -1)
                        {
                            String temp1 = sHost.substring(0, p);
                            if (port == null)
                                port = sHost.substring(p+1, sHost.length()).trim();
                            sHost = temp1;
                        }
                    }
                    continue;

                case 'p':
                    if (port == null)
                        port = options.optArgGet();
                    continue;

                case 'm':
                    if (msg == null)
                        msg = options.optArgGet();
                    continue;

                case 'n':
                    if (myName == null)
                        myName = options.optArgGet();
                    continue;

                case 's':
                    if (serverName == null)
                        serverName = options.optArgGet();
                    continue;
            }
        }

        if ((port != null) && (port.length() > 0))
        {
            int p = -1;
            try {
                p = Integer.parseInt(port);
            } catch (Exception exc) {
                System.out.println("Bad port input: "+port);
            }

            if (p != -1)
                servicePort = p;
        }

        if ((sHost != null) && (sHost.length() > 0)) {
                serviceHostname = sHost;
        }

        init(myName, serverName, serviceHostname, servicePort, msg);
    }
```

```
    void interactWithAcceptor(BitSet flags) throws Exception
    {
        establishContext(flags);
        doWrap();
        doMIC();
    }

    void interactWithAcceptor() throws Exception
    {
        BitSet flags = new BitSet();
        flags.set(Util.CONTEXT_OPTS_MUTUAL);
        flags.set(Util.CONTEXT_OPTS_CONF);
        flags.set(Util.CONTEXT_OPTS_INTEG);
        flags.set(Util.CONTEXT_OPTS_DELEG);
        interactWithAcceptor(flags);
    }

    void dispose() throws Exception
    {
        if (tcp != null)
        {
            tcp.close();
        }
    }

    public static void main(String args[]) throws Exception
    {
        System.out.println(debug.toString()); // XXXXXXX
        String programName = "Client";
        Client client = null;
        try {
            client = new Client(programName,
                                false); // don't use Subject creds.
            client.processArgs(args);
            client.initialize();
            client.interactWithAcceptor();
        } catch (Exception exc) {
            debug.out(Debug.OPTS_CAT_APPLICATION,
                        programName + " Exception: " + exc.toString());
            exc.printStackTrace();
            throw exc;
        } finally {
            try {
                if (client != null)
                    client.dispose();
            } catch (Exception exc) {}
        }

        debug.out(Debug.OPTS_CAT_APPLICATION, programName + ": done");
    }
}
```

Sample: IBM JGSS non-JAAS server program

This example contains a JGSS sample server that is to be used in conjunction with a JGSS sample client.

For more information about using the sample server program, see "Samples: Downloading and running the sample JGSS programs" on page 347.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
// IBM JGSS 1.0 Sample Server Program

package com.ibm.security.jgss.test;

import org.ietf.jgss.*;
import com.ibm.security.jgss.Debug;
import java.io.*;
import java.net.*;
import java.util.*;

/**
 * A JGSS sample server; to be used in conjunction with a JGSS sample client.
 *
 * It continuously listens for client connections,
```

```
 * spawning a thread to service an incoming connection.
 * It is capable of running multiple threads concurrently.
 * In other words, it can service multiple clients concurrently.
 *
 * Each thread first establishes a context with the client
 * and then waits for a wrapped message followed by a MIC.
 * It assumes that the client calculated the MIC over the plain
 * text wrapped by the client.
 *
 * If the client delegates its credential to the server, the delegated
 * credential is used to communicate with a secondary server.
 *
 * Also, the server can be started to act as a client as well as
 * a server (using the -b option). In this case, the first
 * thread spawned by the server uses the server principal's own credential
 * to communicate with the secondary server.
 *
 * The secondary server must have been started prior to the (primary) server
 * initiating contact with it (the scondary server).
 * In communicating with the secondary server, the primary server acts as
 * a JGSS initiator (i.e., client), establishing a context and engaging in
 * wrap and MIC per-message exchanges with the secondary server.
 *
 * The server takes input parameters, and complements it
 * with information from the jgss.ini file; any required input not
 * supplied on the command line is taking from the jgss.ini file.
 * Built-in defaults are used if there is no jgss.ini file or if a particular
 * variable is not specified in the ini file.
 *
 * Usage: Server [options]
 *
 * The -? option produces a help message including supported options.
 *
 * This sample server does not use JAAS.
 * It sets the JAVA variable
 * javax.security.auth.useSubjectCredsOnly to false
 * so that JGSS will not acquire credentials through JAAS.
 * The server can be run against the JAAS sample clients and servers.
 * See {@link JAASServer JAASServer} for a sample server that uses JAAS.
 */

class Server implements Runnable
{
    /*
     * NOTES:
     * This class, Server, is expected to be run in concurrent
     * multiple threads. The static variables consist of variables
     * set from command-line arguments and variables (such as
     * the server's own credentials, gssCred) that are set once during
     * during initialization. These variables do not change
     * once set and are shared between all running threads.
     *
     * The only static variable that is changed after being set initially
     * is the variable 'beenInitiator' which is set 'true'
     * by the first thread to run the server as initiator using
     * the server's own creds. This ensures the server is run as an initiator
     * once only. Querying and modifying 'beenInitiator' is synchronized
     * between the threads.
     *
     * The variable 'tcp' is non-static and is set per thread
     * to represent the socket on which the client being serviced
     * by the thread connected.
     */

    private static Util testUtil             = null;
    private static int myPort                = 0;
    private static Debug debug               = new Debug();
    private static String myName             = null;
    private static GSSCredential gssCred     = null;
    private static String serviceNameNoRealm = null;
    private static String serviceHost        = null;
    private static int    servicePort        = 0;
    private static String serviceMsg         = null;
    private static GSSManager mgr            = null;
    private static GSSName gssName           = null;
    private static String program            = "Server";
    private static boolean clientServer      = false;
    private static boolean primaryServer     = true;

    private static boolean beenInitiator     = false;

    private static final String usageString =
```

```
        "\t[-?] [-# number] [-d | -n name] [-p port]"
      + "\n\t[-s serverName] [-h serverHost [:port]] [-P serverPort] [- msg]"
      + "\n"
      + "\n  -?\t\t\thelp; produces this message"
      + "\n  -# number\t\tWhether primary or secondary server"
      +          " \n\t\t\t(1 = primary, 2 = secondary; default = first)"
      + "\n  -n name\t\tthe server's principal name (without realm)"
      + "\n  -p port\t\tthe port on which the server will be listening"
      + "\n  -s serverName\t\tsecondary server's principal name"
      +          " (without realm)"
      + "\n  -h serverHost[:port]\tsecondary server's hostname"
      +          " (and optional port number)"
      + "\n  -P port\t\tsecondary server's port number"
      + "\n  -m msg\t\tmessage to send to secondary server"
      + "\n  -b \t\trun as both client and server"
      +          " using the server's owns credentials";

   // Non-static variables are thread-specific
   // since each thread runs a separate instance of this class.

   private String debugPrefix = null;
   private TCPComms tcp       = null;

   static {
       try {
           testUtil = new Util();
       } catch (Exception exc) {
           exc.printStackTrace();
           System.exit(1);
       }
   }

   Server (Socket socket) throws Exception
   {
       debugPrefix = program + ": ";
       tcp = new TCPComms(socket);
   }

   Server (String program) throws Exception
   {
       debugPrefix = program + ": ";
       this.program = program;
   }

   Server (String program, boolean useSubjectCredsOnly) throws Exception
   {
       this(program);
       setUseSubjectCredsOnly(useSubjectCredsOnly);
   }

   void setUseSubjectCredsOnly(boolean useSubjectCredsOnly)
   {
       final String subjectOnly = useSubjectCredsOnly ? "true" : "false";
       final String property = "javax.security.auth.useSubjectCredsOnly";

       String temp = (String)java.security.AccessController.doPrivileged(
                       new sun.security.action.GetPropertyAction(property));

       if (temp == null)
       {
           debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
             + "setting useSubjectCredsOnly property to "
             + (useSubjectCredsOnly ? "true" : "false"));

           // Property not set. Set it to the specified value.

           java.security.AccessController.doPrivileged(
                new java.security.PrivilegedAction() {
                  public Object run() {
                     System.setProperty(property, subjectOnly);
                     return null;
                  }
                });
       }
       else
       {
           debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
             + "useSubjectCredsOnly property already set "
             + "in JVM  to " + temp);
       }
   }
```

```
    private void init(boolean  primary,
              String   myNameWithoutRealm,
              int      port,
              String   serverNameWithoutRealm,
              String   serverHostname,
              int      serverPort,
              String   message,
              boolean  clientServer)
        throws Exception
    {
        primaryServer = primary;
        this.clientServer = clientServer;

        myName = myNameWithoutRealm;

        // my port
        if (port > 0)
        {
            myPort = port;
        }
        else if (primary)
        {
            myPort = testUtil.getDefaultServicePort();
        }
        else
        {
            myPort = testUtil.getDefaultService2Port();
        }

        if (primary)
        {
            ///// peer's name
            if (serverNameWithoutRealm != null)
            {
                serviceNameNoRealm = serverNameWithoutRealm;
            }
            else
            {
                serviceNameNoRealm =
                      testUtil.getDefaultService2PrincipalWithoutRealm();
            }

            // peer's host
            if (serverHostname != null)
            {
                if (serverHostname.equalsIgnoreCase("localHost"))
                {
                    serverHostname = InetAddress.getLocalHost().getHostName();
                }

                serviceHost = serverHostname;
            }
            else
            {
                serviceHost = testUtil.getDefaultService2Hostname();
            }

            // peer's port
            if (serverPort > 0)
            {
                servicePort = serverPort;
            }
            else
            {
                servicePort = testUtil.getDefaultService2Port();
            }

            // message for peer
            if (message != null)
            {
                serviceMsg = message;
            }
            else
            {
                serviceMsg = "Hi there! I am a server."
                              + "But I can be a client, too";
            }
        }

        String temp = debugPrefix + "details"
                      + "\n\tPrimary:\t" + primary
                      + "\n\tName:\t\t" + myName
```

```
                  + "\n\tPort:\t\t" + myPort
                  + "\n\tClient+server:\t" + clientServer;
    if (primary)
    {
        temp += "\n\tOther Server:"
                  + "\n\t\tName:\t" + serviceNameNoRealm
                  + "\n\t\tHost:\t" + serviceHost
                  + "\n\t\tPort:\t" + servicePort
                  + "\n\t\tMsg:\t" + serviceMsg;
    }

    debug.out(Debug.OPTS_CAT_APPLICATION, temp);
}


void initialize() throws GSSException
{
   debug.out(Debug.OPTS_CAT_APPLICATION,
                     debugPrefix + "creating GSSManager");

    mgr = GSSManager.getInstance();

    int usage = clientServer ? GSSCredential.INITIATE_AND_ACCEPT
                             : GSSCredential.ACCEPT_ONLY;

    if (myName != null)
    {
        debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
                           + "creating GSSName for " + myName);

        gssName = mgr.createName(myName,
                                 GSSName.NT_HOSTBASED_SERVICE);

        Oid krb5MechanismOid = new Oid("1.2.840.113554.1.2.2");
        gssName.canonicalize(krb5MechanismOid);

        debug.out(Debug.OPTS_CAT_APPLICATION,
              debugPrefix + "Canonicalized GSSName=" + gssName);
    }
    else
        gssName = null;

    debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix + "creating"
           + ((gssName == null)? " default " : " ")
           + "credential");

    gssCred = mgr.createCredential(
                           gssName, GSSCredential.DEFAULT_LIFETIME,
                           (Oid)null, usage);
    if (gssName == null)
    {
        gssName = gssCred.getName();
        myName = gssName.toString();

        debug.out(Debug.OPTS_CAT_APPLICATION,
           debugPrefix + "default credential principal=" + myName);
    }
}



void processArgs(String[] args) throws Exception
{
    String port    = null;
    String name    = null;
    int iport      = 0;

    String sport   = null;
    int isport     = 0;
    String sname   = null;
    String shost   = null;
    String smessage        = null;

    boolean primary = true;
    String status  = null;

    boolean defaultPrinc = false;
    boolean clientServer = false;

    GetOptions options = new GetOptions(args, "?#:p:n:P:s:h:m:b");
    int ch = -1;
    while ((ch = options.getopt()) != options.optEOF)
```

```
        {
            switch(ch)
            {
                case '?':
                    printUsage();
                    System.exit(1);

                case '#':
                    if (status == null)
                        status = options.optArgGet();
                    continue;

                case 'p':
                    if (port == null)
                        port = options.optArgGet();
                    continue;

                case 'n':
                    if (name == null)
                        name = options.optArgGet();
                    continue;

                case 'b':
                    clientServer = true;
                    continue;

                ////// The other server

                case 'P':
                    if (sport == null)
                        sport = options.optArgGet();
                    continue;

                case 'm':
                    if (smessage == null)
                        smessage = options.optArgGet();
                    continue;

                case 's':
                    if (sname == null)
                        sname = options.optArgGet();
                    continue;

                case 'h':
                    if (shost == null)
                    {
                        shost = options.optArgGet();
                        int p = shost.indexOf(':');
                        if (p != -1)
                        {
                            String temp1 = shost.substring(0, p);
                            if (sport == null)
                                sport = shost.substring
                                               (p+1, shost.length()).trim();
                            shost = temp1;
                        }
                    }
                    continue;
            }
        }

        if (defaultPrinc && (name != null))
        {
            System.out.println(
              "ERROR: '-d' and '-n ' options are mutually exclusive");
            printUsage();
            System.exit(1);
        }

        if (status != null)
        {
            int p = -1;
            try {
                p = Integer.parseInt(status);
            } catch (Exception exc) {
                System.out.println( "Bad status input: "+status);
            }

            if (p != -1)
            {
                primary = (p == 1);
            }
```

```
        }

        if (port != null)
        {
            int p = -1;
            try {
                p = Integer.parseInt(port);
            } catch (Exception exc) {
                System.out.println( "Bad port input: "+port);
            }
            if (p != -1)
                iport = p;
        }

        if (sport != null)
        {
            int p = -1;
            try {
                p = Integer.parseInt(sport);
            } catch (Exception exc) {
                System.out.println( "Bad server port input: "+port);
            }
            if (p != -1)
                isport = p;
        }

        init(primary,  // first or second server
             name,     // my name
             iport,    // my port
             sname,    // other server's name
             shost,    // other server's hostname
             isport,   // other server's port
             smessage, // msg for other server
             clientServer); // whether to run as initiator with own creds
    }

    void processRequests() throws Exception
    {
        ServerSocket ssocket = null;
        Server server = null;
        try {
            ssocket = new ServerSocket(myPort);
            do {
               debug.out(Debug.OPTS_CAT_APPLICATION,
                     debugPrefix + "listening on port " + myPort + " ...");
                Socket csocket = ssocket.accept();

               debug.out(Debug.OPTS_CAT_APPLICATION,
                   debugPrefix + "incoming connection on " + csocket);

                server = new Server(csocket); // set client socket per thread
                Thread thread = new Thread(server);
               thread.start();
                if (!thread.isAlive())
                    server.dispose(); // close the client socket
           } while(true);
        } catch (Exception exc) {
           debug.out(Debug.OPTS_CAT_APPLICATION,
                debugPrefix + "*** ERROR processing requests ***");
            exc.printStackTrace();
        } finally {
            try {
                if (ssocket != null)
                   ssocket.close(); // close the server socket
                if (server != null)
                    server.dispose(); // close the client socket
            } catch (Exception exc) {}
        }
    }

    void dispose()
    {
        try {
            if (tcp != null)
            {
                tcp.close();
                tcp = null;
            }
        } catch (Exception exc) {}
    }

    boolean establishContext(GSSContext context) throws Exception
```

```
    {
        byte[] response        = null;
        byte[] request = null;

        debug.out(Debug.OPTS_CAT_APPLICATION,
                          debugPrefix + "establishing context");

        do {
            request = tcp.receive();
            if (request == null || request.length == 0)
            {
                debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
                    + "Received no data; perhaps client disconnected");

                return false;
            }

            debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix + "accepting");
            if ((response = context.acceptSecContext
                                 (request, 0, request.length)) != null)
            {
                 debug.out(Debug.OPTS_CAT_APPLICATION,
                           debugPrefix + "sending response");
                 tcp.send(response);
            }
        } while(!context.isEstablished());

        debug.out(Debug.OPTS_CAT_APPLICATION,
                    debugPrefix + "context established - " + context);

        return true;
    }

    byte[] unwrap(GSSContext context, byte[] msg) throws Exception
    {
        debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix + "unwrapping");

        MessageProp mp = new MessageProp(true);
        byte[] unwrappedMsg = context.unwrap(msg, 0, msg.length, mp);

        debug.out(Debug.OPTS_CAT_APPLICATION,
                             debugPrefix + "unwrapped msg is:");
        debug.out(Debug.OPTS_CAT_APPLICATION, unwrappedMsg);

        return unwrappedMsg;
    }

    void verifyMIC (GSSContext context, byte[] mic, byte[] raw) throws Exception
    {
        debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix + "verifying MIC");

        MessageProp mp = new MessageProp(true);
        context.verifyMIC(mic, 0, mic.length, raw, 0, raw.length, mp);

        debug.out(Debug.OPTS_CAT_APPLICATION,
                       debugPrefix + "successfully verified MIC");
    }

    void useDelegatedCred(GSSContext context) throws Exception
    {
        GSSCredential delCred = context.getDelegCred();
        if (delCred != null)
        {
            if (primaryServer)
            {
                debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix +
                        "Primary server received delegated cred; using it");
                runAsInitiator(delCred); // using delegated creds
            }
            else
            {
                debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix +
                       "Non-primary server received delegated cred; "
                            + "ignoring it");

            }
        }
        else
        {
             debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix +
                                      "ERROR: null delegated cred");
        }
```

```
    }

    public void run()
    {
        byte[] response               = null;
        byte[] request         = null;
        boolean unwrapped      = false;
        GSSContext context     = null;

        try {
          Thread currentThread = Thread.currentThread();
          String threadName    = currentThread.getName();

          debugPrefix = program + " " + threadName + ": ";

          debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
                               + "servicing client ...");

         debug.out(Debug.OPTS_CAT_APPLICATION,
              debugPrefix + "creating GSSContext");

          context = mgr.createContext(gssCred);

          // First establish context with the initiator.
          if (!establishContext(context))
              return;

          // Then process messages from the initiator.
          // We expect to receive a wrapped message followed by a MIC.
          // The MIC should have been calculated over the plain
          // text that we received wrapped.
          // Use delegated creds if any.
          // Then run as initiator using own creds if necessary; only
          // the first thread does this.

          do {
             debug.out(Debug.OPTS_CAT_APPLICATION,
                  debugPrefix + "receiving per-message request");

              request = tcp.receive();
              if (request == null || request.length == 0)
              {
                 debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
                     + "Received no data; perhaps client disconnected");

                 return;
              }

              // Expect wrapped message first.
              if (!unwrapped)
              {
                  response = unwrap(context, request);
                  unwrapped = true;
                  continue; // get next request
              }

              // Followed by a MIC.
              verifyMIC(context, request, response);

              // Impersonate the initiator if it delegated its creds to us.
              if (context.getCredDelegState())
                  useDelegatedCred(context);

              debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
                  + "clientServer=" + clientServer
                  + ", beenInitiator=" + beenInitiator);

              // If necessary, run as initiator using our own creds.
              if (clientServer)
                  runAsInitiatorOnce(currentThread);

              debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix + "done");
              return;

          } while(true);

        } catch (Exception exc) {
           debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix + "ERROR");
            exc.printStackTrace();

            // Squelch per-thread exceptions so we don't bring
            // the server down because of exceptions in
```

```
            // individual threads.
            return;
        } finally {
            if (context != null)
            {
                try {
                    context.dispose();
                } catch (Exception exc) {}
            }
        }
    }

    synchronized void runAsInitiatorOnce(Thread thread)
        throws InterruptedException
    {
        if (!beenInitiator)
        {
            // set flag true early to prevent subsequent threads
            // from attempting to runAsInitiator.
            beenInitiator = true;

            debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix +
                    "About to run as initiator with own creds ...");

            //thread.sleep(30*1000, 0);
            runAsInitiator();
        }
    }


    void runAsInitiator(GSSCredential cred)
    {
        Client client = null;
        try {
            client = new Client(cred,
                                serviceNameNoRealm,
                                serviceHost,
                                servicePort,
                                serviceMsg);

            client.initialize();

            BitSet flags = new BitSet();
            flags.set(Util.CONTEXT_OPTS_MUTUAL);
            flags.set(Util.CONTEXT_OPTS_CONF);
            flags.set(Util.CONTEXT_OPTS_INTEG);

            client.interactWithAcceptor(flags);

        } catch (Exception exc) {
               debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
                  + "Exception running as initiator");

            exc.printStackTrace();
        } finally {
            try {
                client.dispose();
            } catch (Exception exc) {}
        }
    }

    void runAsInitiator()
    {
        if (clientServer)
        {
           debug.out(Debug.OPTS_CAT_APPLICATION,
               debugPrefix + "running as initiator with own creds");

            runAsInitiator(gssCred); // use own creds;
        }
        else
        {
           debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
               + "Cannot run as initiator with own creds "
               + "\nbecause not running as both initiator and acceptor.");
        }
    }

    void printUsage()
    {
        System.out.println(program + usageString);
    }
```

```
    public static void main(String[] args) throws Exception
    {
        System.out.println(debug.toString()); // XXXXXXX
        String programName = "Server";
        try {
            Server server = new Server(programName,
                                       false); // don't use creds from Subject
            server.processArgs(args);
            server.initialize();
            server.processRequests();
        } catch (Exception exc) {
            debug.out(Debug.OPTS_CAT_APPLICATION, programName + ": EXCEPTION");
            exc.printStackTrace();
            throw exc;
        }
    }
}
```

Sample: IBM JGSS JAAS-enabled client program

This sample program performs a JAAS login and operates within the JAAS login context. It does not set the variable javax.security.auth.useSubjectCredsOnly, leaving the variable to default to "true" so that Java GSS acquires credentials from the JAAS Subject associated with login context created by the client.

For more information about using the sample client program, see "Samples: Downloading and running the sample JGSS programs" on page 347.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
// IBM Java GSS 1.0 sample JAAS-enabled client program

package com.ibm.security.jgss.test;
import com.ibm.security.jgss.Debug;
import com.ibm.security.auth.callback.Krb5CallbackHandler;
import javax.security.auth.Subject;
import javax.security.auth.login.LoginContext;
import java.security.PrivilegedExceptionAction;

/**
 * A Java GSS sample client that uses JAAS.
 *
 * It does a JAAS login and operates within the JAAS login context so created.
 *
 * It does not set the JAVA variable
 * javax.security.auth.useSubjectCredsOnly, leaving
 * the variable to default to true
 * so that Java GSS acquires credentials from the JAAS Subject
 * associated with login context (created by the client).
 *
 * The JAASClient is equivalent to its superclass {@link Client Client}
 * in all other respects, and it
 * can be run against the non-JAAS sample clients and servers.
 */

class JAASClient extends Client
{
    JAASClient(String programName) throws Exception
    {
        // Do not set useSubjectCredsOnly. Set only the program name.
        // useSubjectCredsOnly default to "true" if not set.
        super(programName);
    }

    static class JAASClientAction implements PrivilegedExceptionAction
    {
         private JAASClient client;

         public JAASClientAction(JAASClient client)
         {
             this.client = client;
         }

         public Object run () throws Exception
         {
             client.initialize();
```

```
            client.interactWithAcceptor();
            return null;
        }
    }

    public static void main(String args[]) throws Exception
    {
        String programName = "JAASClient";
        JAASClient client = null;
        Debug dbg = new Debug();

        System.out.println(dbg.toString()); // XXXXXXX

        try {
            client = new JAASClient(programName);//use Subject creds
            client.processArgs(args);

            LoginContext loginCtxt = new LoginContext("JAASClient",
                                     new Krb5CallbackHandler());

            loginCtxt.login();

            dbg.out(Debug.OPTS_CAT_APPLICATION,
                      programName + ": Kerberos login OK");

            Subject subject = loginCtxt.getSubject();

            PrivilegedExceptionAction jaasClientAction
                        = new JAASClientAction(client);

            Subject.doAsPrivileged(subject, jaasClientAction, null);

        } catch (Exception exc) {
            dbg.out(Debug.OPTS_CAT_APPLICATION,
                      programName + " Exception: " + exc.toString());
            exc.printStackTrace();
            throw exc;
        } finally {
            try {
                if (client != null)
                    client.dispose();
            } catch (Exception exc) {}
        }

        dbg.out(Debug.OPTS_CAT_APPLICATION,
                      programName + ": Done ...");
    }
}
```

Sample: IBM JGSS JAAS-enabled server program

This sample program performs a JAAS login and operates within the JAAS login context.

For more information about using the sample server program, see "Samples: Downloading and running the sample JGSS programs" on page 347.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
// IBM Java GSS 1.0 sample JAAS-enabled server program

package com.ibm.security.jgss.test;
import com.ibm.security.jgss.Debug;
import com.ibm.security.auth.callback.Krb5CallbackHandler;
import javax.security.auth.Subject;
import javax.security.auth.login.LoginContext;
import java.security.PrivilegedExceptionAction;

/**
 * A Java GSS sample server that uses JAAS.
 *
 * It does a JAAS login and operates within the JAAS login context so created.
 *
 * It does not set the JAVA variable
 * javax.security.auth.useSubjectCredsOnly, leaving
 * the variable to default to true
 * so that Java GSS acquires credentials from the JAAS Subject
```

```
 * associated with login context (created by the server).
 *
 * The JAASServer is equivalent to its superclass {@link Server Server}
 * in all other respects, and it
 * can be run against the non-JAAS sample clients and servers.
 */

class JAASServer extends Server
{
    JAASServer(String programName) throws Exception
    {
        super(programName);
    }

    static class JAASServerAction implements PrivilegedExceptionAction
    {
        private JAASServer server = null;

        JAASServerAction(JAASServer server)
        {
            this.server = server;
        }

        public Object run() throws Exception
        {
            server.initialize();
            server.processRequests();

            return null;
        }
    }

    public static void main(String[] args) throws Exception
    {
        String programName     = "JAASServer";
        Debug dbg              = new Debug();

        System.out.println(dbg.toString()); // XXXXXXX

        try {
            // Do not set useSubjectCredsOnly.
            // useSubjectCredsOnly defaults to "true" if not set.

            JAASServer server = new JAASServer(programName);

            server.processArgs(args);

            LoginContext loginCtxt = new LoginContext(programName,
                                               new Krb5CallbackHandler());

            dbg.out(Debug.OPTS_CAT_APPLICATION, programName + ": Login in ...");

            loginCtxt.login();

            dbg.out(Debug.OPTS_CAT_APPLICATION, programName +
                                                    ": Login successful");

            Subject subject = loginCtxt.getSubject();

            JAASServerAction serverAction = new JAASServerAction(server);

            Subject.doAsPrivileged(subject, serverAction, null);
        } catch (Exception exc) {
            dbg.out(Debug.OPTS_CAT_APPLICATION, programName +  " EXCEPTION");
            exc.printStackTrace();
            throw exc;
        }
    }
}
```

Examples: IBM Java Secure Sockets Extension 1.4

The JSSE examples show how a client and a server can use the native IBM i JSSE provider to create a context that enables secure communications.

Note: Both examples use the native IBM i JSSE provider, regardless of the properties specified by the java.security file.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

Example: Calling a CL program with java.lang.Runtime.exec()

This example shows how to run CL programs from within a Java program. In this example, the Java class CallCLPgm runs a CL program.

The CL program uses the Display JVM Jobs (DSPJVMJOB) CL command to display all of the jobs on the system that contain an active Java Virtual Machine. This example assumes that the CL program has been compiled and exists in a library that is called JAVSAMPLIB. The output from the CL program is in the QSYSPRT spooled file.

See "Example: Calling a CL command with java.lang.Runtime.exec()" on page 218 for an example of how to call a CL command from within a Java program.

Note: The JAVSAMPLIB is not created as part of the IBM Developer Kit licensed program (LP) number 5770-JV1 installation process. You must explicitly create the library.

Source code for CallCLPgm Java class

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
import java.io.*;

public class CallCLPgm
{
   public static void main(String[] args)
   {
      try
      {
         Process theProcess =
                 Runtime.getRuntime().exec("/QSYS.LIB/JAVSAMPLIB.LIB/DSPJVA.PGM");
      }
      catch(IOException e)
      {
         System.err.println("Error on exec() method");
         e.printStackTrace();
      }
   }
}
```

Source code for DSPJVA CL program

```
PGM
    DSPJVMJOB OUTPUT(*PRINT)
ENDPGM
```

Related reference

Example: Calling another Java program with java.lang.Runtime.exec()
This example shows how to call another Java program with `java.lang.Runtime.exec()`. This class calls the Hello program that is shipped as part of the IBM Developer Kit for Java. When the Hello class writes to System.out, this program gets a handle to the stream and can read from it.

Example: Calling a CL command with java.lang.Runtime.exec()
This example shows how to run a control language (CL) command from within a Java program.

Example: Calling a CL command with java.lang.Runtime.exec()

This example shows how to run a control language (CL) command from within a Java program.

In this example, the Java class runs a CL command. The CL command uses the Display JVM Jobs (DSPJVMJOB) CL command to display all of the jobs on the system that contain an active Java Virtual Machine. The output from the CL command is in the QSYSPRT spooled file.

CL commands that you pass into the `Runtime.getRuntime().exec()` function use the following format:

```
Runtime.getRuntime().exec("system CLCOMMAND");
```

where *CLCOMMAND* is the CL command you want to run.

Source code for Java class for calling a CL command

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
import java.io.*;

public class CallCLCom
{
   public static void main(String[] args)
   {
      try
      {
         Process theProcess =
            Runtime.getRuntime().exec("system DSPJVMJOB OUTPUT(*PRINT)");
      }
      catch(IOException e)
      {
         System.err.println("Error on exec() method");
         e.printStackTrace();
      }
   }
}
```

Related concepts

"Using java.lang.Runtime.exec()" on page 216
Use the `java.lang.Runtime.exec()` method to call programs or commands from within your Java program. Using `java.lang.Runtime.exec()` method creates one or more additional thread-enabled jobs. The additional jobs process the command string that you pass on the method.

"List of Java system properties" on page 15
Java system properties determine the environment in which the Java programs run. They are like system values or environment variables in IBM i.

Related reference

Example: Calling another Java program with java.lang.Runtime.exec()
This example shows how to call another Java program with `java.lang.Runtime.exec()`. This class calls the Hello program that is shipped as part of the IBM Developer Kit for Java. When the Hello class writes to System.out, this program gets a handle to the stream and can read from it.

Example: Calling a CL program with java.lang.Runtime.exec()
This example shows how to run CL programs from within a Java program. In this example, the Java class CallCLPgm runs a CL program.

Example: Calling another Java program with java.lang.Runtime.exec()

This example shows how to call another Java program with `java.lang.Runtime.exec()`. This class calls the Hello program that is shipped as part of the IBM Developer Kit for Java. When the Hello class writes to System.out, this program gets a handle to the stream and can read from it.

Source code for CallHelloPgm Java class

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
import java.io.*;

public class CallHelloPgm
{
   public static void main(String args[])
   {
```

```
        Process theProcess = null;
        BufferedReader inStream = null;

        System.out.println("CallHelloPgm.main() invoked");

        // call the Hello class
        try
        {
            theProcess = Runtime.getRuntime().exec("java QIBMHello");
        }
        catch(IOException e)
        {
           System.err.println("Error on exec() method");
           e.printStackTrace();
        }

        // read from the called program's standard output stream
        try
        {
           inStream = new BufferedReader(
                                new InputStreamReader( theProcess.getInputStream() ));
           System.out.println(inStream.readLine());
        }
        catch(IOException e)
        {
           System.err.println("Error on inStream.readLine()");
           e.printStackTrace();
        }
    }
}
```

Related reference

Example: Calling a CL program with java.lang.Runtime.exec()
This example shows how to run CL programs from within a Java program. In this example, the Java class CallCLPgm runs a CL program.

Example: Calling a CL command with java.lang.Runtime.exec()
This example shows how to run a control language (CL) command from within a Java program.

Example: Calling Java from ILE C

This is an example of an integrated language environment (ILE) C program that uses the `system()` function to call the Java Hello program.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
#include <stdlib.h>

 int main(void)
 {
   int result;

   /* The system function passes the given string
    * to the CL command processor for processing.
    */

   result = system("JAVA CLASS('QIBMHello')");
 }
```

Related concepts

Using sockets for interprocess communication
Sockets streams communicate between programs that are running in separate processes.

Using input and output streams for interprocess communication
Input and output streams communicate between programs that are running in separate processes.

Related reference

Example: Calling Java from RPG

This is an example of an RPG program that uses the QCMDEXC API to call the Java Hello program.

Example: Calling Java from RPG

This is an example of an RPG program that uses the QCMDEXC API to call the Java Hello program.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
D*              DEFINE  THE PARAMETERS FOR THE QCMDEXC API
D*
DCMDSTRING        S             25    INZ('JAVA CLASS(''QIBMHello'')')
DCMDLENGTH        S             15P 5 INZ(23)
D*              NOW THE CALL TO QCMDEXC WITH THE 'JAVA' CL COMMAND
C                   CALL      'QCMDEXC'
C                   PARM                    CMDSTRING
C                   PARM                    CMDLENGTH
C*        This next line displays 'DID IT' after you exit the
C*        Java Shell via F3 or F12.
C     'DID IT'      DSPLY
C*        Set On LR to exit the RPG program
C                   SETON                                        LR
C
```

Related concepts

Using sockets for interprocess communication
Sockets streams communicate between programs that are running in separate processes.

Using input and output streams for interprocess communication
Input and output streams communicate between programs that are running in separate processes.

Related reference

Example: Calling Java from ILE C
This is an example of an integrated language environment (ILE) C program that uses the `system()` function to call the Java Hello program.

Example: Using input and output streams for interprocess communication

This example shows how to call a C program from Java and use input and output streams for interprocess communication.

In this example, the C program writes a string to its standard output stream, and the Java program reads this string and displays it. This example assumes that a library, which is named JAVSAMPLIB, has been created and that the CSAMP1 program has been created in it.

Note: The JAVSAMPLIB is not created as part of the IBM Developer Kit licensed program (LP) number 5770-JV1 installation process. You must explicitly create it.

Source code for CallPgm Java class

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
import java.io.*;

public class CallPgm
{
   public static void main(String args[])
   {
      Process theProcess = null;
      BufferedReader inStream = null;

      System.out.println("CallPgm.main() invoked");

      // call the CSAMP1 program
      try
      {
          theProcess = Runtime.getRuntime().exec(
                       "/QSYS.LIB/JAVSAMPLIB.LIB/CSAMP1.PGM");
      }
```

```
        catch(IOException e)
        {
           System.err.println("Error on exec() method");
           e.printStackTrace();
        }

        // read from the called program's standard output stream
        try
        {
           inStream = new BufferedReader(new InputStreamReader
                               (theProcess.getInputStream()));
           System.out.println(inStream.readLine());
        }
        catch(IOException e)
        {
           System.err.println("Error on inStream.readLine()");
           e.printStackTrace();
        }
     }
  }
```

Source code for CSAMP1 C Program

Note: Read the Code example disclaimer for important legal information.

```
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char* args[])
{
     /* Convert the string to ASCII at compile time */
#pragma convert(819)
     printf("Program JAVSAMPLIB/CSAMP1 was invoked\n");
#pragma convert(0)
     /* Stdout may be buffered, so flush the buffer */

     fflush(stdout);
}
```

Example: Java Invocation API

This integrated language environment (ILE) C example follows the standard Invocation API paradigm.

It does the following:

- Creates a Java virtual machine by using `JNI_CreateJavaVM()`.
- Uses the Java virtual machine to find the class file that you want to run.
- Finds the methodID for the main method of the class.
- Calls the main method of the class.
- Reports errors if an exception occurs.

When you create the program, the QJVAJNI or QJVAJNI64 service program provides the `JNI_CreateJavaVM()` API function. `JNI_CreateJavaVM()` creates the Java virtual machine.

Note: QJVAJNI64 is a ervice program for teraspace/LLP64 native method and Invocation API support.

These service programs reside in the system binding directory and you do not need to explicitly identify them on a control language (CL) create command. For example, you would not explicitly identify the previously mentioned service programs when using the Create Program (CRTPGM) command or the Create Service Program (CRTSRVPGM) command.

One way to run this program is to use the following control language command:

```
     SBMJOB CMD(CALL PGM(YOURLIB/PGMNAME)) ALWMLTTHD(*YES)
```

Any job that creates a Java virtual machine must be multithread-capable. The output from the main program, as well as any output from the program, ends up in QPRINT spooled files. These spooled files are visible when you use the Work with Submitted Jobs (WRKSBMJOB) control language (CL) command and view the job that you started by using the Submit Job (SBMJOB) CL command.

Example: Using the Java Invocation API within ILE C

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
#define OS400_JVM_12
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#include <jni.h>

/* Specify the pragma that causes all literal strings in the
 * source code to be stored in ASCII (which, for the strings
 * used, is equivalent to UTF-8)
 */

#pragma convert(819)

/* Procedure: Oops
 *
 * Description: Helper routine that is called when a JNI function
 *              returns a zero value, indicating a serious error.
 *              This routine reports the exception to stderr and
 *              ends the JVM abruptly with a call to FatalError.
 *
 * Parameters:  env -- JNIEnv* to use for JNI calls
 *              msg -- char* pointing to error description in UTF-8
 *
 * Note:        Control does not return after the call to FatalError
 *              and it does not return from this procedure.
 */

void Oops(JNIEnv* env, char *msg) {
    if ((*env)->ExceptionOccurred(env)) {
        (*env)->ExceptionDescribe(env);
    }
    (*env)->FatalError(env, msg);
}

/* This is the program's "main" routine. */
int main (int argc, char *argv[])
{

    JavaVMInitArgs initArgs; /* Virtual Machine (VM) initialization structure, passed by
                              * reference to JNI_CreateJavaVM(). See jni.h for details
                              */
    JavaVM* myJVM;           /* JavaVM pointer set by call to JNI_CreateJavaVM */
    JNIEnv* myEnv;           /* JNIEnv pointer set by call to JNI_CreateJavaVM */
    char*   myClasspath;     /* Changeable classpath 'string' */
    jclass  myClass;         /* The class to call, 'NativeHello'. */
    jmethodID mainID;        /* The method ID of its 'main' routine. */
    jclass  stringClass;     /* Needed to create the String[] arg for main */
    jobjectArray args;       /* The String[] itself */
    JavaVMOption options[1]; /* Options array -- use options to set classpath */
    int     fd0, fd1, fd2;   /* file descriptors for IO */

    /* Open the file descriptors so that IO works. */
    fd0 = open("/dev/null", O_CREAT|O_TRUNC|O_RDWR,   S_IRUSR|S_IROTH);
    fd1 = open("/dev/null", O_CREAT|O_TRUNC|O_WRONLY, S_IWUSR|S_IWOTH);
    fd2 = open("/dev/null", O_CREAT|O_TRUNC|O_WRONLY, S_IWUSR|S_IWOTH);

    /*  Set the version field of the initialization arguments for JNI v1.4. */
    initArgs.version = 0x00010004;


    /* Now, you want to specify the directory for the class to run in the classpath.
     * with  Java2, classpath is passed in as an option.
     * Note: You must specify the directory name in UTF-8 format. So, you wrap
     *       blocks of code in #pragma convert statements.
     */
    options[0].optionString="-Djava.class.path=/CrtJvmExample";

    initArgs.options=options;  /* Pass in the classpath that has been set up. */
    initArgs.nOptions = 1;     /* Pass in classpath and version options */

    /*  Create the JVM -- a nonzero return code indicates there was
     *  an error. Drop back into EBCDIC and write a message to stderr
     *  before exiting the program.
     *  Note:  This will run the default JVM and JDK which is 32bit JDK 6.0.
```

```
     *  If you want to run a different JVM and JDK, set the JAVA_HOME environment
     *  variable to the home directory of the JVM you want to use
     *  (prior to the CreateJavaVM() call).
     */
    if (JNI_CreateJavaVM(&myJVM, (void **)&myEnv, (void *)&initArgs)) {
        #pragma convert(0)
            fprintf(stderr, "Failed to create the JVM\n");
        #pragma convert(819)
            exit(1);
    }

    /*  Use the newly created JVM to find the example class,
     *  called 'NativeHello'.
     */
    myClass = (*myEnv)->FindClass(myEnv, "NativeHello");
    if (! myClass) {
        Oops(myEnv, "Failed to find class 'NativeHello'");
    }

    /* Now, get the method identifier for the 'main' entry point
     * of the class.
     * Note: The signature of 'main' is always the same for any
     *       class called by the following java command:
     *           "main" , "([Ljava/lang/String;)V"
     */
    mainID = (*myEnv)->GetStaticMethodID(myEnv,myClass,"main",
                                     "([Ljava/lang/String;)V");
    if (! mainID) {
        Oops(myEnv, "Failed to find jmethodID of 'main'");
    }

    /* Get the jclass for String to create the array
     * of String to pass to 'main'.
     */
    stringClass = (*myEnv)->FindClass(myEnv, "java/lang/String");
    if (! stringClass) {
        Oops(myEnv, "Failed to find java/lang/String");
    }

    /*  Now, you need to create an empty array of strings,
     *  since main requires such an array as a parameter.
     */
    args = (*myEnv)->NewObjectArray(myEnv,0,stringClass,0);
    if (! args) {
        Oops(myEnv, "Failed to create args array");
    }

    /* Now, you have the methodID of main and the class, so you can
     * call the main method.
     */
    (*myEnv)->CallStaticVoidMethod(myEnv,myClass,mainID,args);

    /* Check for errors. */
    if ((*myEnv)->ExceptionOccurred(myEnv)) {
        (*myEnv)->ExceptionDescribe(myEnv);
    }

    /* Finally, destroy the JavaVM that you created. */
    (*myJVM)->DestroyJavaVM(myJVM);

    /* All done. */
    return 0;
}
```

For more information, see "Java Invocation API" on page 211.

Example: IBM PASE for i native method for Java

The PASE for i native method for Java example calls an instance of a native C method that then uses Java Native Interface (JNI) to call back into Java code. Rather than accessing the string directly from Java code, the example calls a native method that then calls back into Java through JNI to get the string value.

To see HTML versions of the example source files, use the following links:

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

- "Example: PaseExample1.java" on page 464

• "Example: PaseExample1.c" on page 465

Before you can run the PASE for i native method example, you must complete the tasks in the following topics:

1. "Example: Downloading the example source code to your AIX workstation" on page 465
2. "Example: Preparing the example source code" on page 466
3. "Example: Preparing your IBM i server to run the PASE for i native method for Java example" on page 467

Running the PASE for i native method for Java example

After you complete the previous tasks, you can run the example. Use either of the following commands to run the example program:

- From an IBM i command prompt:

```
JAVA CLASS(PaseExample1) CLASSPATH('/home/example')
```

- From a Qshell command prompt or PASE for i terminal session:

```
cd /home/example
java PaseExample1
```

Example: PaseExample1.java

This example program loads the native method library 'PaseExample1'. The source code for the native method is contained in PaseExample1.c. The printString method in this Java program uses a native method, getStringNative to retrieve the value of the String. The native method simply calls back into the getStringCallback method of this class.

Note: Read the Code example disclaimer for important legal information.

```
//////////////////////////////////////////////////////////////////////////////////
//
// This example program loads the native method library 'PaseExample1'.
// The source code for the native method is contained in PaseExample1.c
// The printString method in this Java program uses a native method,
// getStringNative to retrieve the value of the String. The native method
// simply calls back into the getStringCallback method of this class.
//
//////////////////////////////////////////////////////////////////////////////////



public class PaseExample1 {
    public static void main(String args[]) {
    PaseExample1 pe1 = new PaseExample1("String for PaseExample1");
    pe1.printString();
    }

    String str;

    PaseExample1(String s) {
    str = s;
    }


    //-----------------------------------------------------------------
    public void printString() {
    String result = getStringNative();
    System.out.println("Value of str is '" + result + "'");
    }

    // This calls getStringCallback through JNI.
    public native String getStringNative();

    // Called by getStringNative via JNI.
    public String getStringCallback() {
    return str;
```

```
    }

    //-----------------------------------------------------------------
    static {
    System.loadLibrary("PaseExample1");
    }

}
```

Example: PaseExample1.c

This native method implements the getStringNative method of class PaseExample1. It uses the JNI function CallObjectMethod to call back to the getStringCallback method of class PaseExample1.

Note: Read the Code example disclaimer for important legal information.

```
/*
 *

 * This native method implements the getStringNative method of class
 * PaseExample1. It uses the JNI function CallObjectMethod to call
 * back to the getStringCallback method of class PaseExample1.
 *
 * Compile this code in AIX or PASE for i to create module 'libPaseExample1.so'.
 *
 */

#include "PaseExample1.h"
#include <stdlib.h>

/*
 * Class:     PaseExample1
 * Method:    getStringNative
 * Signature: ()Ljava/lang/String;
 */
JNIEXPORT jstring JNICALL Java_PaseExample1_getStringNative(JNIEnv* env, jobject obj) {
   char* methodName = "getStringCallback";
   char* methodSig = "()Ljava/lang/String;";
   jclass clazz = (*env)->GetObjectClass(env, obj);
   jmethodID methodID = (*env)->GetMethodID(env, clazz, methodName, methodSig);
   return (*env)->CallObjectMethod(env, obj, methodID);
}
```

Example: Downloading the example source code to your AIX workstation

Before you can run the PASE for i native method for Java example, you need to download a compressed file that contains the source code. To download the compressed file to your AIX workstation, complete the following steps.

1. Create a temporary directory on your AIX workstation that you want to contain the source files.
2. Download the PASE for i example source code into the temporary directory.
3. Unzip the example files into the temporary directory.

For more information about the PASE for i native method for Java example, see the following topics:

"Example: IBM PASE for i native method for Java" on page 207
The PASE for i native method for Java example calls an instance of a native C method that then uses Java Native Interface (JNI) to call back into Java code. Rather than accessing the string directly from Java code, the example calls a native method that then calls back into Java through JNI to get the string value.

"Example: Preparing the example source code" on page 466
Before moving the PASE for i native method for Java example to your server, you need to compile the source code, create a C include file, and create a shared library object.

"Example: Preparing your IBM i server to run the PASE for i native method for Java example" on page 467

Before running the PASE for i native method for Java example, you need to prepare your server to run the example. Preparing the server requires copying the files to the server and adding the necessary environment variables to run the example.

Example: Preparing the example source code

Before moving the PASE for i native method for Java example to your server, you need to compile the source code, create a C include file, and create a shared library object.

The example includes the following C and Java source files:

- **PaseExample1.c:** C source code file that contains an implementation of getStringNative().
- **PaseExample1.java:** Java source code file that calls the native getStringNative method in the C program.

You use the compiled Java .class file to create a C include file, PaseExample1.h, which contains a function prototype for the getStringNative method contained in the C source code.

To prepare the example source code on your AIX workstation, complete the following steps:

1. Use the following command to compile the Java source code:

   ```
   javac PaseExample1.java
   ```

2. Use the following command to create a C include file that contains the native method prototypes:

   ```
   javah -jni PaseExample1
   ```

   The new C include file (PaseExample1.h) contains a function prototype for the getStringNative method. The example C source code (PaseExample1.c) already includes the information you would copy and modify from the C include file to use the getStringNative method. For more information about using JNI, see the Java Native Interface on the Oracle Web site.

3. Use the following command to compile the C source code and create a shared library object.

   ```
   xlc -G -I/usr/local/java/J1.5.0/include PaseExample1.c -o libPaseExample1.so
   ```

   The new shared library object file (libPaseExample1.so) contains the native method library "PaseExample1" that the example uses.

   Note: You may need to change the -I option to point to the directory that contains the correct Java native method include files (for example, jni.h) for your AIX system.

For more information about the PASE for i native method for Java example, see the following topics:

"Example: IBM PASE for i native method for Java" on page 207
The PASE for i native method for Java example calls an instance of a native C method that then uses Java Native Interface (JNI) to call back into Java code. Rather than accessing the string directly from Java code, the example calls a native method that then calls back into Java through JNI to get the string value.

"Example: Downloading the example source code to your AIX workstation" on page 465
Before you can run the PASE for i native method for Java example, you need to download a compressed file that contains the source code. To download the compressed file to your AIX workstation, complete the following steps.

"Example: Preparing your IBM i server to run the PASE for i native method for Java example" on page 467

Before running the PASE for i native method for Java example, you need to prepare your server to run the example. Preparing the server requires copying the files to the server and adding the necessary environment variables to run the example.

Example: Preparing your IBM i server to run the PASE for i native method for Java example

Before running the PASE for i native method for Java example, you need to prepare your server to run the example. Preparing the server requires copying the files to the server and adding the necessary environment variables to run the example.

To prepare your server, complete the following steps:

1. Create the following integrated file system directory on the server that you want to contain the example files. For example, use the following control language (CL) command to create the directory named /home/example:

```
mkdir '/home/example'
```

2. Copy the following files to the new directory:
   - PaseExample1.class
   - libPaseExample1.so
3. From an IBM i command prompt, use the following control language (CL) commands to add the necessary environment variables:

```
addenvvar PASE_THREAD_ATTACH 'Y'
addenvvar LIBPATH '/home/example'
```

Note: When using PASE native methods from an PASE for i terminal session, a 32-bit PASE for i environment is already started. In this case, set only PASE_THREAD_ATTACH to Y and. LIBPATH to the path for the PASE for i native method libraries.

For more information about the PASE for i native method for Java example, see the following topics:

"Example: IBM PASE for i native method for Java" on page 207
The PASE for i native method for Java example calls an instance of a native C method that then uses Java Native Interface (JNI) to call back into Java code. Rather than accessing the string directly from Java code, the example calls a native method that then calls back into Java through JNI to get the string value.

"Example: Downloading the example source code to your AIX workstation" on page 465
Before you can run the PASE for i native method for Java example, you need to download a compressed file that contains the source code. To download the compressed file to your AIX workstation, complete the following steps.

"Example: Preparing the example source code" on page 466
Before moving the PASE for i native method for Java example to your server, you need to compile the source code, create a C include file, and create a shared library object.

Example: ILE native method for Java

The integrated language environment (ILE) native method for Java example calls an instance of a native C method that then uses Java Native Interface (JNI) to call back into Java code to set the value of a Java string variable. The Java string variable is then written to standard out by the Java code.

To see HTML versions of the example source files, use the following links:

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

- "Example: NativeHello.java" on page 468
- "Example: NativeHello.c" on page 469

Before you can run the ILE native method example, you must complete the tasks in the following topics:

1. "Example: Preparing the ILE native method source code" on page 472
2. "Example: Creating the ILE native method program objects" on page 473

Running the ILE native method for Java example

After you complete the previous tasks, you can run the example. Use either of the following commands to run the example program:

- From an IBM i command prompt:

```
JAVA CLASS(NativeHello) CLASSPATH('/ileexample')
```

- From a Qshell command prompt:

```
cd /ileexample
java NativeHello
```

Related concepts

Teraspace storage model native methods for Java
The IBM i Java virtual machine (JVM) supports the use of teraspace storage model native methods. The teraspace storage model provides a large, process-local address environment for ILE programs. Using the teraspace storage model allows you to port native method code from other operating systems to IBM i with little or no source code changes.

Strings in ILE native methods
Many Java Native Interface (JNI) functions accept C language-style strings as parameters. For example, the `FindClass()` JNI function accepts a string parameter that specifies the fully-qualified name of a class file. If the class file is found, it is loaded by `FindClass()`, and a reference to it is returned to the caller of `FindClass()`.

Example: NativeHello.java

The Java source code is used in the integrated language environment (ILE) native method for Java example.

The `NativeHello.java` source code shows the Java code that is used to demonstrate ILE native methods for Java. The native method `setTheString()` is called by the Java code from within the `main()` method. The result of the call is that the function `Java_NativeHello_setTheString()`, defined in the C implementation code, gets control and uses Java Native Interface (JNI) to call back into the Java code to set the value of the Java string variable `theString`. Control is then returned to the Java code, and the string variable is written to stdout.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
public class NativeHello {

  //  Declare a field of type 'String' in the NativeHello object.
  //  This is an 'instance' field, so every NativeHello object
  //  contains one.
  public String theString;           // instance variable

  //  Declare the native method itself.  This native method
  //  creates a new string object, and places a reference to it
  //  into 'theString'
  public native void setTheString(); // native method to set string

  //  This 'static initializer' code is called before the class is
  //  first used.
  static {
     //  Attempt to load the native method library.  If you do not
     //  find it, write a message to 'out', and try a hardcoded path.
     //  If that fails, then exit.
     try {
        //  System.loadLibrary uses the java.library.path property or
        //  the LIBPATH environment variable.
        System.loadLibrary("NATHELLO");
```

```
      }
      catch (UnsatisfiedLinkError e1) {
         //  Did not find the service program.
         System.out.println("I did not find NATHELLO *SRVPGM.");
         System.out.println("(I will try a hardcoded path)");

         try {
            //  System.load takes the full integrated file system form path.
            System.load ("/qsys.lib/ileexample.lib/nathello.srvpgm");
         }
         catch (UnsatisfiedLinkError e2) {
            //  If you get to this point, then you are done!  Write the message
            //  and exit.
            System.out.println
               ("<sigh> I did not find NATHELLO *SRVPGM anywhere. Goodbye");
            System.exit(1);
         }
      }
   }

   //  Here is the 'main' code of this class. This is what runs when you
   //  enter 'java NativeHello' on the command line.
   public static void main(String argv[]){

      //  Allocate a new NativeHello object now.
      NativeHello nh = new NativeHello();

      //  Echo location.
      System.out.println("(Java) Instantiated NativeHello object");
      System.out.println("(Java) string field is '" + nh.theString + "'");
      System.out.println("(Java) Calling native method to set the string");

      //  Here is the call to the native method.
      nh.setTheString();

      //  Now, print the value after the call to double check.
      System.out.println("(Java) Returned from the native method");
      System.out.println("(Java) string field is '" + nh.theString + "'");
      System.out.println("(Java) All done...");
   }
}
```

Example: NativeHello.c

The C source code is used in the integrated language environment (ILE) native method for Java example.

The `NativeHello.c` source code shows the implementation of the native method in C. When the Java native method `setTheString()`, defined in the Java code, is called by the Java code the C function `Java_NativeHello_setTheString()` gets control and uses Java Native Interface (JNI) to call back into the Java code to set the value of the Java string variable `theString`. Control is then returned to the Java code, and the string variable is written to stdout out by the Java code.

This example shows how to link Java to native methods. However, it points out complications that arise from the fact that the IBM i is internally an extended binary-coded decimal interchange code (EBCDIC) machine. It also shows complications from the current lack of true internationalization elements in the JNI.

These reasons, although they are not new with the JNI, cause some unique IBM i server-specific differences in the C code that you write. You must remember that if you are writing to stdout or stderr or reading from stdin, your data is probably encoded in EBCDIC form.

In C code, you can easily convert most literal strings, those that contain 7-bit characters only, into the UTF-8 form that is required by the JNI. To do this, bracket the literal strings with the codepage conversion `pragma` statement. However, because you may write information directly to stdout or stderr from your C code, you might allow some literals to remain in EBCDIC.

Note: The `#pragma convert(0)` statements convert character data to EBCDIC. The `#pragma convert(819)` statements convert character data to American Standard Code for Information Interchange (ASCII). These statements convert character data in the C program at compile time.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
#include <stdlib.h>       /* malloc, free, and so forth */
#include <stdio.h>        /* fprintf(), and so forth */
#include <qtqiconv.H>     /* iconv() interface */
#include <string.h>       /* memset(), and so forth */
#include "NativeHello.h"  /* generated by 'javah-jni' */

/*  All literal strings are ISO-8859-1 Latin 1 code page
 * (and with 7-bit characters, they are also automatically UTF-8).
 */
#pragma convert(819)  /* handle all literal strings as ASCII */

/*  Report and clear a JNI exception.  */
static void HandleError(JNIEnv*);

/*  Print an UTF-8 string to stderr in the coded character
 *  set identifier (CCSID) of the current job.
 */
static void JobPrint(JNIEnv*, char*);

/*  Constants describing which direction to covert:  */
#define CONV_UTF2JOB 1
#define CONV_JOB2UTF 2

/*  Convert a string from the CCSID of the job to UTF-8, or vice-versa. */
int StringConvert(int direction, char *sourceStr, char *targetStr);

/*  Native method implementation of 'setTheString()'.  */

JNIEXPORT void JNICALL Java_NativeHello_setTheString
(JNIEnv *env, jobject javaThis)
{
    jclass thisClass;     /* class for 'this' object */
    jstring stringObject; /* new string, to be put in field in 'this' */
    jfieldID fid;         /* field ID required to update field in 'this' */
    jthrowable exception; /* exception, retrieved using ExceptionOccurred */

    /*  Write status to console.  */
    JobPrint(env, "( C ) In the native method\n");

    /* Build the new string object.  */
    if (! (stringObject = (*env)->NewStringUTF(env, "Hello, native world!")))
    {
      /*  For nearly every function in the JNI, a null return value indicates
       *  that there was an error, and that an exception had been placed where it
       *  could be retrieved by 'ExceptionOccurred()'. In this case, the error
       *  would typically be fatal, but for purposes of this example, go ahead
       *  and catch the error, and continue.
       */
       HandleError(env);
       return;
    }

    /* get the class of the 'this' object, required to get the fieldID  */
    if (! (thisClass = (*env)->GetObjectClass(env,javaThis)))
    {
      /*  A null class returned from GetObjectClass indicates that there
       *  was a problem. Instead of handling this problem, simply return and
       *  know that the return to Java automatically 'throws' the stored Java
       *  exception.
       */
       return;
    }

    /*  Get the fieldID to update.  */
    if (! (fid = (*env)->GetFieldID(env,
                                    thisClass,
                                    "theString",
                                    "Ljava/lang/String;")))
    {
      /*  A null fieldID returned from GetFieldID indicates that there
       *  was a problem.  Report the problem from here and clear it.
       *  Leave the string unchanged.
       */
        HandleError(env);
        return;
    }
```

```
    JobPrint(env, "( C ) Setting the field\n");

    /*  Make the actual update.
     *  Note: SetObjectField is an example of an interface that does
     *  not return a return value that can be tested.  In this case, it
     *  is necessary to call ExceptionOccurred() to see if there
     *  was a problem with storing the value
     */
    (*env)->SetObjectField(env, javaThis, fid, stringObject);

    /*  Check to see if the update was successful. If not, report the error.  */
    if ((*env)->ExceptionOccurred(env)) {

      /*  A non-null exception object came back from ExceptionOccurred,
       *  so there is a problem and you must report the error.
       */
       HandleError(env);
    }

    JobPrint(env, "( C ) Returning from the native method\n");
    return;
}

static void HandleError(JNIEnv *env)
{
    /*  A simple routine to report and handle an exception.  */
    JobPrint(env, "( C  ) Error occurred on JNI call: ");
    (*env)->ExceptionDescribe(env); /* write exception data to the console */
    (*env)->ExceptionClear(env);    /* clear the exception that was pending */
}

static void JobPrint(JNIEnv *env, char *str)
{
    char   *jobStr;
    char   buf[512];
    size_t len;

    len = strlen(str);

    /*  Only print non-empty string. */
    if (len) {
        jobStr = (len >= 512) ? malloc(len+1) : &buf;
        if (! StringConvert(CONV_UTF2JOB, str, jobStr))
            (*env)->FatalError
              (env,"ERROR in JobPrint: Unable to convert UTF2JOB");
        fprintf(stderr, jobStr);
        if (len >= 512) free(jobStr);
    }
}

int StringConvert(int direction, char *sourceStr, char *targetStr)
{
    QtqCode_T source, target;     /* parameters to instantiate iconv  */
    size_t    sStrLen, tStrLen;   /* local copies of string lengths   */
    iconv_t   ourConverter;       /* the actual conversion descriptor */
    int       iconvRC;            /* return code from the conversion  */
    size_t    originalLen;        /* original length of the sourceStr */

    /*  Make local copies of the input and output sizes that are initialized
     *  to the size of the input string. The iconv() requires the
     *  length parameters to be passed by address (that is as int*).
     */
    originalLen = sStrLen = tStrLen = strlen(sourceStr);

    /*  Initialize the parameters to the QtqIconvOpen() to zero.  */
    memset(&source,0x00,sizeof(source));
    memset(&target,0x00,sizeof(target));

    /*  Depending on direction parameter, set either SOURCE
     *  or TARGET CCSID to ISO 8859-1 Latin.
     */
    if (CONV_UTF2JOB == direction )
        source.CCSID = 819;
    else
        target.CCSID = 819;

    /*  Create the iconv_t converter object.  */
    ourConverter = QtqIconvOpen(&target,&source);

    /*  Make sure that you have a valid converter, otherwise return 0.  */
    if (-1 == ourConverter.return_value) return 0;
```

```
     /*  Perform the conversion.  */
     iconvRC = iconv(ourConverter,
                     (char**) &sourceStr, &sStrLen,
                     &targetStr, &tStrLen);

     /*  If the conversion failed, return a zero.  */
     if (0 != iconvRC ) return 0;

     /*  Close the conversion descriptor.  */
     iconv_close(ourConverter);

     /*  The targetStr returns pointing to the character just
      *  past the last converted character, so set the null there now.
      */
     *targetStr = '\0';

     /*  Return the number of characters that were processed. */
     return originalLen-tStrLen;
}
#pragma convert(0)
```

Example: Preparing the ILE native method source code

Before running the integrated language environment (ILE) native method for Java example, you need to prepare your server to run the example. Preparing the server requires creating the source files on the server.

To prepare your server, complete the following steps:

1. Create the following integrated file system directory on the server that you want to contain the example files. For example, use the following control language (CL) command to create the directory named `/ileexample`:

   ```
   mkdir '/ileexample'
   ```

2. Create two empty stream files named `NativeHello.java` and `NativeHello.c` in the directory created in step 1. For example, use the following CL commands to create the files:

   ```
   QSH CMD('touch -C 819 /ileexample/NativeHello.java')

   QSH CMD('touch -C 819 /ileexample/NativeHello.c')
   ```

3. Copy the Java source code shown in “Example: NativeHello.java” on page 468 to the file created in step 2 named `NativeHello.java`.
4. Copy the C source code shown in “Example: NativeHello.c” on page 469 to the file created in step 2 named `NativeHello.c`.
5. Create a library named `ileexample` that will contain the service program containing the C implementation code for the Java native method. For example, use the following CL command to create the library:

   ```
   crtlib ileexample
   ```

“Example: ILE native method for Java” on page 206
The integrated language environment (ILE) native method for Java example calls an instance of a native C method that then uses Java Native Interface (JNI) to call back into Java code to set the value of a Java string variable. The Java string variable is then written to standard out by the Java code.

“Example: NativeHello.java” on page 468
The Java source code is used in the integrated language environment (ILE) native method for Java example.

“Example: NativeHello.c” on page 469
The C source code is used in the integrated language environment (ILE) native method for Java example.

“Example: Creating the ILE native method program objects” on page 473

Before you can run the integrated language environment (ILE) native method example on your server, you need to compile the source code, create a C include file, and create a ILE service program.

Example: Creating the ILE native method program objects

Before you can run the integrated language environment (ILE) native method example on your server, you need to compile the source code, create a C include file, and create a ILE service program.

You use the compiled Java `.class` file to create a C include file, `NativeHello.h`, which contains a function prototype for the `setTheString()` method contained in the C source code.

To prepare the example source code on your server, complete the following steps:

Note: The following steps should be preformed only after performing the steps listed in "Example: Preparing the ILE native method source code" on page 472.

1. From the command line, change the current working directory to `/ileexample` by using the following command:

   ```
   cd '/ileexample'
   ```

2. Compile the Java source file. For example, use the following command to compile the Java source code from the command line:

   ```
   QSH CMD('javac NativeHello.java')
   ```

3. Use the following command to create a C include file that contains the native method prototypes:

   ```
   QSH CMD('javah -jni NativeHello')
   ```

   The new C include file (`NativeHello.h`) contains a function prototype for the `setTheString()` method. The example C source code `NativeHello.c` already includes the include file.

4. Use the following commands to compile the C source code and create a service program.

   ```
       CRTCMOD MODULE(ILEEXAMPLE/NATHELLO) SRCSTMF(NativeHello.c) TERASPACE(*YES)

       CRTSRVPGM SRVPGM(ILEEXAMPLE/NATHELLO) MODULE(ILEEXAMPLE/NATHELLO)
   EXPORT(*ALL)
   ```

   The new service program (`NATHELLO`) contains the native method `setTheString()` that the example uses.

"Example: ILE native method for Java" on page 206
The integrated language environment (ILE) native method for Java example calls an instance of a native C method that then uses Java Native Interface (JNI) to call back into Java code to set the value of a Java string variable. The Java string variable is then written to standard out by the Java code.

"Example: NativeHello.java" on page 468
The Java source code is used in the integrated language environment (ILE) native method for Java example.

"Example: NativeHello.c" on page 469
The C source code is used in the integrated language environment (ILE) native method for Java example.

"Example: Preparing the ILE native method source code" on page 472

Before running the integrated language environment (ILE) native method for Java example, you need to prepare your server to run the example. Preparing the server requires creating the source files on the server.

Example: Using sockets for interprocess communication

This example uses sockets to communicate between a Java program and a C program.

You should start the C program first, which listens on a socket. Once the Java program connects to the socket, the C program sends it a string by using that socket connection. The string that is sent from the C program is an American Standard Code for Information Interchange (ASCII) string in codepage 819.

The Java program should be started using this command, `java TalkToC xxxxx nnnn` on the Qshell Interpreter command line or on another Java platform. Or, enter `JAVA TALKTOC PARM(xxxxx nnnn)` on the IBM i command line to start the Java program. xxxxx is the domain name or Internet Protocol (IP) address of the system on which the C program is running. nnnn is the port number of the socket that the C program is using. You should also use this port number as the first parameter on the call to the C program.

Source code for TalkToC client Java class

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
import java.net.*;
import java.io.*;

class TalkToC
{
   private String host = null;
   private int port = -999;
   private Socket socket = null;
   private BufferedReader inStream = null;

   public static void main(String[] args)
   {
      TalkToC caller = new TalkToC();
      caller.host = args[0];
      caller.port = new Integer(args[1]).intValue();
      caller.setUp();
      caller.converse();
      caller.cleanUp();
   }

   public void setUp()
   {
      System.out.println("TalkToC.setUp() invoked");

      try
      {
         socket = new Socket(host, port);
         inStream = new BufferedReader(new InputStreamReader(
                                       socket.getInputStream()));
      }
      catch(UnknownHostException e)
      {
         System.err.println("Cannot find host called: " + host);
         e.printStackTrace();
         System.exit(-1);
      }
      catch(IOException e)
      {
         System.err.println("Could not establish connection for " + host);
         e.printStackTrace();
         System.exit(-1);
      }
   }

   public void converse()
   {
      System.out.println("TalkToC.converse() invoked");

      if (socket != null && inStream != null)
      {
         try
         {
```

```
            System.out.println(inStream.readLine());
         }
         catch(IOException e)
         {
            System.err.println("Conversation error with host " + host);
            e.printStackTrace();
         }
      }
   }

   public void cleanUp()
   {
      try
      {
         if (inStream != null)
            inStream.close();
         if (socket != null)
            socket.close();
      }
      catch(IOException e)
      {
         System.err.println("Error in cleanup");
         e.printStackTrace();
         System.exit(-1);
      }
   }
}
```

SockServ.C starts by passing in a parameter for the port number. For example, `CALL SockServ '2001'`.

Source code for SockServ.C server program

Note: Read the Code example disclaimer for important legal information.

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <unistd.h>
#include <sys/time.h>

void main(int argc, char* argv[])
{
   int    portNum = atoi(argv[1]);
   int    server;
   int    client;
   int    address_len;
   int    sendrc;
   int    bndrc;
   char*  greeting;
   struct sockaddr_in  local_Address;
   address_len = sizeof(local_Address);

   memset(&local_Address,0x00,sizeof(local_Address));
   local_Address.sin_family = AF_INET;
   local_Address.sin_port = htons(portNum);
   local_Address.sin_addr.s_addr = htonl(INADDR_ANY);

   #pragma convert (819)
   greeting = "This is a message from the C socket server.";
   #pragma convert (0)

   /*  allocate socket    */
   if((server = socket(AF_INET, SOCK_STREAM, 0))<0)
   {
      printf("failure on socket allocation\n");
      perror(NULL);
      exit(-1);
   }

   /* do bind   */
   if((bndrc=bind(server,(struct sockaddr*)&local_Address, address_len))<0)
   {
     printf("Bind failed\n");
     perror(NULL);
     exit(-1);
```

```
    }

    /* invoke listen   */
    listen(server, 1);

    /* wait for client request */
    if((client = accept(server,(struct sockaddr*)NULL, 0))<0)
    {
      printf("accept failed\n");
      perror(NULL);
      exit(-1);
    }

    /* send greeting to client    */
    if((sendrc = send(client, greeting, strlen(greeting),0))<0)
    {
       printf("Send failed\n");
       perror(NULL);
       exit(-1);
    }

    close(client);
    close(server);
}
```

Example: Embedding SQL Statements in your Java application

The following example SQLJ application, App.sqlj, uses static SQL to retrieve and update data from the EMPLOYEE table of the DB2 sample database.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
import java.sql.*;
import sqlj.runtime.*;
import sqlj.runtime.ref.*;

#sql iterator App_Cursor1 (String empno, String firstnme) ; // 1
#sql iterator App_Cursor2 (String) ;

class App
{

  /**********************
   **  Register Driver **
   **********************/

 static
 {
   try
   {
     Class.forName("com.ibm.db2.jdbc.app.DB2Driver").newInstance();
   }
   catch (Exception e)
   {
     e.printStackTrace();
   }
 }

  /********************
   **      Main      **
   ********************/

 public static void main(String argv[])
 {
   try
   {
     App_Cursor1 cursor1;
     App_Cursor2 cursor2;

     String str1 = null;
     String str2 = null;
     long   count1;

     // URL is jdbc:db2:dbname
     String url = "jdbc:db2:sample";

     DefaultContext ctx = DefaultContext.getDefaultContext();
```

```
     if (ctx == null)
     {
       try
       {
         // connect with default id/password
         Connection con = DriverManager.getConnection(url);
         con.setAutoCommit(false);
         ctx = new DefaultContext(con);
       }
       catch (SQLException e)
       {
         System.out.println("Error: could not get a default context");
         System.err.println(e) ;
         System.exit(1);
       }
       DefaultContext.setDefaultContext(ctx);
     }

     // retrieve data from the database
     System.out.println("Retrieve some data from the database.");
     #sql cursor1 = {SELECT empno, firstnme FROM employee}; // 2

     // display the result set
     // cursor1.next() returns false when there are no more rows
     System.out.println("Received results:");
     while (cursor1.next()) // 3
     {
       str1 = cursor1.empno(); // 4
       str2 = cursor1.firstnme();

       System.out.print (" empno= " + str1);
       System.out.print (" firstname= " + str2);
       System.out.println("");
     }
     cursor1.close(); // 9

     // retrieve number of employee from the database
     #sql { SELECT count(*) into :count1 FROM employee }; // 5
     if (1 == count1)
       System.out.println ("There is 1 row in employee table");
     else
       System.out.println ("There are " + count1
                           + " rows in employee table");

     // update the database
     System.out.println("Update the database.");
     #sql { UPDATE employee SET firstnme = 'SHILI' WHERE empno = '000010' };

     // retrieve the updated data from the database
     System.out.println("Retrieve the updated data from the database.");
     str1 = "000010";
     #sql cursor2 = {SELECT firstnme FROM employee WHERE empno = :str1}; // 6

     // display the result set
     // cursor2.next() returns false when there are no more rows
     System.out.println("Received results:");
     while (true)
     {
       #sql { FETCH :cursor2 INTO :str2 }; // 7
       if (cursor2.endFetch()) break; // 8

       System.out.print (" empno= " + str1);
       System.out.print (" firstname= " + str2);
       System.out.println("");
     }
     cursor2.close(); // 9

     // rollback the update
     System.out.println("Rollback the update.");
     #sql { ROLLBACK work };
     System.out.println("Rollback done.");
   }
   catch( Exception e )
   {
     e.printStackTrace();
   }
 }
}
```

[1]Declare iterators. This section declares two types of iterators:

- App_Cursor1: Declares column data types and names, and returns the values of the columns according to column name (Named binding to columns).
- App_Cursor2: Declares column data types, and returns the values of the columns by column position (Positional binding to columns).

[2]Initialize the iterator. The iterator object cursor1 is initialized using the result of a query. The query stores the result in cursor1.

[3]Advance the iterator to the next row. The cursor1.next() method returns a Boolean `false` if there are no more rows to retrieve.

[4]Move the data. The named accessor method empno() returns the value of the column named empno on the current row. The named accessor method firstnme() returns the value of the column named firstnme on the current row.

[5]SELECT data into a host variable. The SELECT statement passes the number of rows in the table into the host variable count1.

[6] Initialize the iterator. The iterator object cursor2 is initialized using the result of a query. The query stores the result in cursor2.

[7]Retrieve the data. The FETCH statement returns the current value of the first column declared in the ByPos cursor from the result table into the host variable str2.

[8]Check the success of a FETCH.INTO statement. The endFetch() method returns a Boolean true if the iterator is not positioned on a row, that is, if the last attempt to fetch a row failed. The endFetch() method returns false if the last attempt to fetch a row was successful. DB2 attempts to fetch a row when the next() method is called. A FETCH...INTO statement implicitly calls the next() method.

[9]Close the iterators. The close() method releases any resources held by the iterators. You should explicitly close iterators to ensure that system resources are released in a timely fashion.

Examples: Changing your Java code to use client socket factories

These examples show you how to change a simple socket class, named simpleSocketClient, so that it uses socket factories to create all of the sockets. The first example shows you the simpleSocketClient class without socket factories. The second example shows you the simpleSocketClient class with socket factories. In the second example, simpleSocketClient is renamed to factorySocketClient.

Example 1: Socket client program without socket factories

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
/* Simple Socket Client Program */

import java.net.*;
import java.io.*;

public class simpleSocketClient {
  public static void main (String args[]) throws IOException {

    int serverPort = 3000;

    if (args.length < 1) {
      System.out.println("java simpleSocketClient serverHost serverPort");
      System.out.println("serverPort defaults to 3000 if not specified.");
      return;
    }
    if (args.length == 2)
      serverPort = new Integer(args[1]).intValue();

    System.out.println("Connecting to host " + args[0] + " at port " +
                       serverPort);

    // Create the socket and connect to the server.
    Socket  s = new Socket(args[0], serverPort);
    .
    .
    .
```

```
    // The rest of the program continues on from here.
```

Example 2: Simple socket client program with socket factories

```
/* Simple Socket Factory Client Program */

// Notice that javax.net.* is imported to pick up the SocketFactory class.
import javax.net.*;
import java.net.*;
import java.io.*;

public class factorySocketClient {
  public static void main (String args[]) throws IOException {

    int serverPort = 3000;

    if (args.length < 1) {
      System.out.println("java factorySocketClient serverHost serverPort");
      System.out.println("serverPort defaults to 3000 if not specified.");
      return;
    }
    if (args.length == 2)
      serverPort = new Integer(args[1]).intValue();

    System.out.println("Connecting to host " + args[0] + " at port " +
                       serverPort);

    // Change the original simpleSocketClient program to create a
    // SocketFactory and then use the socket factory to create sockets.

    SocketFactory socketFactory = SocketFactory.getDefault();

    // Now the factory creates the socket.  This is the last change
    // to the original simpleSocketClient program.

    Socket  s = socketFactory.createSocket(args[0], serverPort);
    .
    .
    .

    // The rest of the program continues on from here.
```

Related reference

Examples: Changing your Java code to use server socket factories
These examples show you how to change a simple socket class, named simpleSocketServer, so that it uses socket factories to create all of the sockets. The first example shows you the simpleSocketServer class without socket factories. The second example shows you the simpleSocketServer class with socket factories. In the second example, simpleSocketServer is renamed to factorySocketServer.

Examples: Changing your Java code to use server socket factories

These examples show you how to change a simple socket class, named simpleSocketServer, so that it uses socket factories to create all of the sockets. The first example shows you the simpleSocketServer class without socket factories. The second example shows you the simpleSocketServer class with socket factories. In the second example, simpleSocketServer is renamed to factorySocketServer.

Example 1: Socket server program without socket factories

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
/* File simpleSocketServer.java*/

import java.net.*;
import java.io.*;

public class simpleSocketServer {
  public static void main (String args[]) throws IOException {

    int serverPort = 3000;

    if (args.length < 1) {
      System.out.println("java simpleSocketServer serverPort");
```

```
      System.out.println("Defaulting to port 3000 since serverPort not specified.");
    }
    else
        serverPort = new Integer(args[0]).intValue();

    System.out.println("Establishing server socket at port " + serverPort);

    ServerSocket    serverSocket =
      new ServerSocket(serverPort);

    // a real server would handle more than just one client like this...

    Socket s = serverSocket.accept();
    BufferedInputStream is = new BufferedInputStream(s.getInputStream());
    BufferedOutputStream os = new BufferedOutputStream(s.getOutputStream());

    // This server just echoes back what you send it...

    byte buffer[] = new byte[4096];

    int bytesRead;

    // read until "eof" returned
    while ((bytesRead = is.read(buffer)) > 0) {
      os.write(buffer, 0, bytesRead); // write it back
      os.flush();    // flush the output buffer
    }


    s.close();
    serverSocket.close();
  }       // end main()

}       // end class definition
```

Example 2: Simple socket server program with socket factories

```
/* File factorySocketServer.java */

// need to import javax.net to pick up the ServerSocketFactory class
import javax.net.*;
import java.net.*;
import java.io.*;

public class factorySocketServer {
  public static void main (String args[]) throws IOException {

    int serverPort = 3000;

    if (args.length < 1) {
      System.out.println("java simpleSocketServer serverPort");
      System.out.println("Defaulting to port 3000 since serverPort not specified.");
    }
    else
        serverPort = new Integer(args[0]).intValue();

    System.out.println("Establishing server socket at port " + serverPort);

    // Change this to create an SSLServerSocketFactory instead of a ServerSocketFactory.
    ServerSocketFactory serverSocketFactory =
      SSLServerSocketFactory.getDefault();
    // Now have the factory create the server socket.  This is the last
    // change from the original program.
    ServerSocket    serverSocket =
      serverSocketFactory.createServerSocket(serverPort);

    // a real server would handle more than just one client like this...

    Socket s = serverSocket.accept();
    BufferedInputStream is = new BufferedInputStream(s.getInputStream());
    BufferedOutputStream os = new BufferedOutputStream(s.getOutputStream());

    // This server just echoes back what you send it...

    byte buffer[] = new byte[4096];

    int bytesRead;

    while ((bytesRead = is.read(buffer)) > 0) {
      os.write(buffer, 0, bytesRead);
```

```
      os.flush();
    }


    s.close();
    serverSocket.close();
  }

}
```

Related reference

Examples: Changing your Java code to use client socket factories
These examples show you how to change a simple socket class, named simpleSocketClient, so that it uses socket factories to create all of the sockets. The first example shows you the simpleSocketClient class without socket factories. The second example shows you the simpleSocketClient class with socket factories. In the second example, simpleSocketClient is renamed to factorySocketClient.

Examples: Changing your Java client to use secure sockets layer

These examples show you how to change one class, named factorySocketClient, to use secure sockets layer (SSL). The first example shows you the factorySocketClient class not using SSL. The second example shows you the same class, renamed factorySSLSocketClient, using SSL.

Example 1: Simple factorySocketClient class without SSL support

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
/* Simple Socket Factory Client Program */

import javax.net.*;
import java.net.*;
import java.io.*;

public class factorySocketClient {
  public static void main (String args[]) throws IOException {

    int serverPort = 3000;

    if (args.length < 1) {
      System.out.println("java factorySocketClient serverHost serverPort");
      System.out.println("serverPort defaults to 3000 if not specified.");
      return;
    }
    if (args.length == 2)
      serverPort = new Integer(args[1]).intValue();

    System.out.println("Connecting to host " + args[0] + " at port " +
                       serverPort);

    SocketFactory socketFactory = SocketFactory.getDefault();

    Socket  s = socketFactory.createSocket(args[0], serverPort);
    .
    .
    .

    // The rest of the program continues on from here.
```

Example 2: Simple factorySocketClient class with SSL support

```
// Notice that we import javax.net.ssl.* to pick up SSL support
import javax.net.ssl.*;
import javax.net.*;
import java.net.*;
import java.io.*;

public class factorySSLSocketClient {
  public static void main (String args[]) throws IOException {

    int serverPort = 3000;

    if (args.length < 1) {
      System.out.println("java factorySSLSocketClient serverHost serverPort");
```

```
      System.out.println("serverPort defaults to 3000 if not specified.");
      return;
    }
    if (args.length == 2)
      serverPort = new Integer(args[1]).intValue();

    System.out.println("Connecting to host " + args[0] + " at port " +
                       serverPort);

    // Change this to create an SSLSocketFactory instead of a SocketFactory.
    SocketFactory socketFactory = SSLSocketFactory.getDefault();

    // We do not need to change anything else.
    // That's the beauty of using factories!
    Socket  s = socketFactory.createSocket(args[0], serverPort);
    .
    .
    .

    // The rest of the program continues on from here.
```

Related reference

Examples: Changing your Java server to use secure sockets layer
These examples show you how to change one class, named factorySocketServer, to use secure sockets layer (SSL).

Examples: Changing your Java server to use secure sockets layer

These examples show you how to change one class, named factorySocketServer, to use secure sockets layer (SSL).

The first example shows you the factorySocketServer class not using SSL. The second example shows you the same class, renamed factorySSLSocketServer, using SSL.

Example 1: Simple factorySocketServer class without SSL support

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 489.

```
/* File factorySocketServer.java  */
// need to import javax.net to pick up the ServerSocketFactory class
import javax.net.*;
import java.net.*;
import java.io.*;

public class factorySocketServer {
  public static void main (String args[]) throws IOException {

    int serverPort = 3000;

    if (args.length < 1) {
      System.out.println("java simpleSocketServer serverPort");
      System.out.println("Defaulting to port 3000 since serverPort not specified.");
    }
    else
        serverPort = new Integer(args[0]).intValue();

    System.out.println("Establishing server socket at port " + serverPort);

    // Change this to create an SSLServerSocketFactory instead of a ServerSocketFactory.
    ServerSocketFactory serverSocketFactory =
      SSLServerSocketFactory.getDefault();
    // Now have the factory create the server socket.  This is the last
    // change from the original program.
    ServerSocket    serverSocket =
      serverSocketFactory.createServerSocket(serverPort);

    // a real server would handle more than just one client like this...

    Socket s = serverSocket.accept();
    BufferedInputStream is = new BufferedInputStream(s.getInputStream());
    BufferedOutputStream os = new BufferedOutputStream(s.getOutputStream());

    // This server just echoes back what you send it.

    byte buffer[] = new byte[4096];
```

```
    int bytesRead;

    while ((bytesRead = is.read(buffer)) > 0) {
      os.write(buffer, 0, bytesRead);
      os.flush();
    }

    s.close();
    serverSocket.close();
  }
}
```

Example 2: Simple factorySocketServer class with SSL support

```
/* File factorySocketServer.java */

// need to import javax.net to pick up the ServerSocketFactory class
import javax.net.*;
import java.net.*;
import java.io.*;

public class factorySocketServer {
  public static void main (String args[]) throws IOException {

    int serverPort = 3000;

    if (args.length < 1) {
      System.out.println("java simpleSocketServer serverPort");
      System.out.println("Defaulting to port 3000 since serverPort not specified.");
    }
    else
        serverPort = new Integer(args[0]).intValue();

    System.out.println("Establishing server socket at port " + serverPort);

    // Change the original simpleSocketServer to use a
    // ServerSocketFactory to create server sockets.
    ServerSocketFactory serverSocketFactory =
      ServerSocketFactory.getDefault();
    // Now have the factory create the server socket.  This is the last
    // change from the original program.
    ServerSocket    serverSocket =
      serverSocketFactory.createServerSocket(serverPort);

    // a real server would handle more than just one client like this...

    Socket s = serverSocket.accept();
    BufferedInputStream is = new BufferedInputStream(s.getInputStream());
    BufferedOutputStream os = new BufferedOutputStream(s.getOutputStream());

    // This server just echoes back what you send it.

    byte buffer[] = new byte[4096];

    int bytesRead;

    while ((bytesRead = is.read(buffer)) > 0) {
      os.write(buffer, 0, bytesRead);
      os.flush();
    }

    s.close();
    serverSocket.close();
  }
}
```

Related reference

Examples: Changing your Java client to use secure sockets layer

These examples show you how to change one class, named factorySocketClient, to use secure sockets layer (SSL). The first example shows you the factorySocketClient class not using SSL. The second example shows you the same class, renamed factorySSLSocketClient, using SSL.

Troubleshooting Java programs

This topic shows you how to find job logs and collect data for Java program analysis. This topic also provides information about program temporary fixes (PTFs) and getting support for IBM Developer Kit for Java.

If the performance of your program degrades as it runs for a longer period of time, you may have erroneously coded a memory leak. You can use the JavaWatcher, a component of IBM iDoctor, to help you debug your program and locate any memory leaks. For more information, see Heap Analysis Tools for Java.

Limitations

This list identifies any known limitations, restrictions, or unique behaviors in Java on the IBM i server.

- The java.net backlog parameter on the IBM i may behave differently than on other platforms. For example:
  - Listen backlogs 0, 1
    - Listen(0) means to allow one pending connection; it does not disable a socket.
    - Listen(1) means to allow one pending connection, and means the same as Listen(0).
  - Listen backlogs > 1
    - This allows many pending requests to remain on the listen queue. If a new connection request arrives and the queue is at the limit, then it deletes one of the pending requests.
- You can only use the Java virtual machine, regardless of the JDK version you are using, in multi-thread capable (that is, thread-safe) environments. The IBM i platform is thread-safe, but some file systems are not. For a list of file systems that are not thread-safe, see the Integrated File System topic.

Related concepts

Applying program temporary fixes
Some of the Java topics has information that assumes you have loaded the latest levels of the IBM Developer Kit for Java on your IBM i server. To get to the latest levels of Java on your server, load the latest Java Program Temporary Fix (PTF) group.

Getting support for Java on IBM i
Support services for Java on IBM i are provided under the usual terms and conditions for IBM i software products. Support services include program services, voice support, and consulting services.

Related tasks

Finding job logs for Java problem analysis
Use the job log from the job that ran the Java command, and the batch immediate (BCI) job log where the Java program ran, to analyze causes of a Java failure. They both may contain important error information.

Collecting data for Java problem analysis
To collect data for an authorized program analysis report (APAR), follow these steps.

Finding job logs for Java problem analysis

Use the job log from the job that ran the Java command, and the batch immediate (BCI) job log where the Java program ran, to analyze causes of a Java failure. They both may contain important error information.

There are two ways to find the job log for the BCI job. You can find the name of the BCI job that is logged in the job log of the job that ran the Java command. Then, use that job name to find the job log for the BCI job.

You can also find the job log for the BCI job by following these steps:

1. Enter the Work with Submitted Jobs (WRKSBMJOB) command on the IBM i command line.
2. Go to the bottom of the list.
3. Look for the last job in the list, called QJVACMDSRV.
4. Enter option 8 (Work with Spooled Files) for that job.
5. A file called QPJOBLOG displays.
6. Press F11 to see view 2 of the spooled files.
7. Verify that the date and time match the date and time when the failure occurred.

   If the date and time do not match the date and time when you signed off, continue looking through the list of submitted jobs. Try to find a QJVACMDSRV job log with a date and time that matches when you signed off.

If you are unable to find a job log for the BCI job, one may not have been produced. This happens if you set the ENDSEP value for the QDFTJOBD job description too high or the LOG value for the QDFTJOBD job description specifies *NOLIST. Check these values, and change them so that a job log is produced for the BCI job.

To produce a job log for the job that ran the Run Java (RUNJVA) command, perform the following steps:

1. Enter SIGNOFF *LIST.
2. Then, sign back on.
3. Enter the Work with Spooled Files (WRKSPLF) command on the IBM i command line.
4. Go to the bottom of the list.
5. Find a file named QPJOBLOG.
6. Press F11.
7. Verify that the date and time match the date and time when you entered the signoff command.

   If the date and time do not match the date and time when you signed off, continue looking through the list of submitted jobs. Try to find a QJVACMDSRV job log with a date and time that matches when you signed off.

Related concepts

Limitations
This list identifies any known limitations, restrictions, or unique behaviors in Java on the IBM i server.

Applying program temporary fixes
Some of the Java topics has information that assumes you have loaded the latest levels of the IBM Developer Kit for Java on your IBM i server. To get to the latest levels of Java on your server, load the latest Java Program Temporary Fix (PTF) group.

Getting support for Java on IBM i
Support services for Java on IBM i are provided under the usual terms and conditions for IBM i software products. Support services include program services, voice support, and consulting services.

Related tasks

Collecting data for Java problem analysis
To collect data for an authorized program analysis report (APAR), follow these steps.

Collecting data for Java problem analysis

To collect data for an authorized program analysis report (APAR), follow these steps.

1. Include a complete description of the problem.
2. Save the Java class file that caused the problem while running.
3. You can use the SAV command to save objects from the integrated file system. You may need to save other class files that this program must run. You may also want to save and send in an entire directory for IBM to use when trying to reproduce the problem, if necessary. This is an example of how to save an entire directory.

Example: Save a directory

Note: Read the Code example disclaimer for important legal information.

```
SAV DEV('/QSYS.LIB/TAP01.DEVD') OBJ(('/mydir'))
```

If possible, save the source files for any Java classes that are involved in the problem. This is helpful to IBM when reproducing and analyzing the problem.

4. Save any service programs that contain native methods that are required to run the program.
5. Save any data files that are required to run the Java program.
6. Add a complete description of how to reproduce the problem.

   This includes:

   - The value of the CLASSPATH environment variable.
   - A description of the Java command that was run.
   - A description of how to respond to any input that is required by the program.
7. Include any vertical licensed internal code (VLIC) logs that have occurred near the time of failure, particularly those with a major code of 4700 or 4300.
8. Add the job log from both the interactive job and the BCI job where the Java virtual machine was running.
9. Add information requested in the IBM Center for Java Technology Developer Diagnostic Guide.

Related concepts

Limitations
This list identifies any known limitations, restrictions, or unique behaviors in Java on the IBM i server.

Applying program temporary fixes
Some of the Java topics has information that assumes you have loaded the latest levels of the IBM Developer Kit for Java on your IBM i server. To get to the latest levels of Java on your server, load the latest Java Program Temporary Fix (PTF) group.

Getting support for Java on IBM i
Support services for Java on IBM i are provided under the usual terms and conditions for IBM i software products. Support services include program services, voice support, and consulting services.

Related tasks

Finding job logs for Java problem analysis
Use the job log from the job that ran the Java command, and the batch immediate (BCI) job log where the Java program ran, to analyze causes of a Java failure. They both may contain important error information.

Applying program temporary fixes

Some of the Java topics has information that assumes you have loaded the latest levels of the IBM Developer Kit for Java on your IBM i server. To get to the latest levels of Java on your server, load the latest Java Program Temporary Fix (PTF) group.

Java PTFs and PTFs that impact Java are periodically packaged as part of a PTF group. A PTF group consists of a list of PTFs defined for the purpose of managing those PTFs as one entity. A new level of the Java PTF group may be released several times within a calender year. It is recommended that you install the latest PTFs to upgrade to the latest level of the IBM Developer Kit for Java. See the Preventive Service Planning - PSP Web page for the Java PTF group number and latest group level that is available.

Note: A Java PTF that can be applied immediately may not affect a Java Virtual Machine (JVM) that is running within a job. However, in some cases, applying the PTF will cause unpredictable results. You can use the Display Java Virtual Machine Jobs (DSPJVMJOB) CL command to manage your JVM jobs and apply PTFs while the system is active. The DSPJVMJOB command shows which jobs have JVMs running in them. The information can then be used to end the jobs containing active JVMs before applying PTFs, instead of waiting for an initial program load (IPL) for the PTFs to be applied.

Related concepts

Limitations
This list identifies any known limitations, restrictions, or unique behaviors in Java on the IBM i server.

Getting support for Java on IBM i
Support services for Java on IBM i are provided under the usual terms and conditions for IBM i software products. Support services include program services, voice support, and consulting services.

Related tasks

Finding job logs for Java problem analysis
Use the job log from the job that ran the Java command, and the batch immediate (BCI) job log where the Java program ran, to analyze causes of a Java failure. They both may contain important error information.

Collecting data for Java problem analysis
To collect data for an authorized program analysis report (APAR), follow these steps.

Related information

Maintain and manage IBM i and related software

Use software fixes

Getting support for Java on IBM i

Support services for Java on IBM i are provided under the usual terms and conditions for IBM i software products. Support services include program services, voice support, and consulting services.

Use the online information that is provided at IBM i home page under the topic "Support" for more information. Use IBM Support Services for licensed program 5770-JV1. Or, contact your local IBM representative.

You may, at IBM direction, be required to obtain a more current level of the licensed program to receive Continued Program Services. For more information, see Support for multiple Java Development Kits (JDKs).

Defect resolution is supported under program services or voice support. Resolving application programming or debugging issues are supported under consulting services.

The Java application programming interface (API) calls are supported under consulting services, unless:

1. It is clearly a Java API defect as demonstrated by re-creation in a relatively simple program.
2. It is a question that asks for documentation clarification.
3. It is a question about the location of samples or documentation.

All programming assistance is supported under consulting services. This includes the program samples that are provided in the product. Additional samples may be available on the Internet at IBM i home page on an unsupported basis.

The IBM Developer Kit for Java LP provides information about solving problems.

Related concepts

Limitations
This list identifies any known limitations, restrictions, or unique behaviors in Java on the IBM i server.

Applying program temporary fixes
Some of the Java topics has information that assumes you have loaded the latest levels of the IBM Developer Kit for Java on your IBM i server. To get to the latest levels of Java on your server, load the latest Java Program Temporary Fix (PTF) group.

Related tasks

Finding job logs for Java problem analysis
Use the job log from the job that ran the Java command, and the batch immediate (BCI) job log where the Java program ran, to analyze causes of a Java failure. They both may contain important error information.

Collecting data for Java problem analysis

To collect data for an authorized program analysis report (APAR), follow these steps.

Related information

Listed here are sources that relate to Java on the IBM i server.

Web sites

- IBM i Technology Updates for Java

  Provide additional details about Java support on IBM i.

- The Source for Java Developers (http://www.oracle.com/technetwork/java/index.html)

  Visit Oracle for information about the various uses for Java, including new technologies.

- IBM developerWorks Java technology zone

  Offers information, education, and tool to help you use Java, IBM products, and other technologies to create business solutions.

- IBM alphaWorks® Java

  Includes information about new Java technologies, including downloads and links to development resources.

Javadoc

Javadoc reference information for Java classes can be found at Java 2 Platform, Standard Edition API Specification by Oracle.

See the following reference information that relates to Java on the IBM i server.

Java Naming and Directory Interface

The Java Naming and Directory Interface (JNDI) is part of the JavaSoft platform application programming interface (API). With JNDI, you can connect seamlessly to multiple naming and directory services. You can build powerful and portable directory-enabled Java applications by using this interface.

JavaSoft developed the JNDI specification with leading industry partners, including IBM, SunSoft, Novell, Netscape, and Hewlett-Packard Co.

Note: The IBM i Java Runtime Environment (JRE) and the versions of the Java 2 Platform, Standard Edition (J2SE) offered by the IBM Developer Kit for Java include the Sun LDAP provider. Because IBM i Java support includes the Sun LDAP provider, that support no longer includes the ibmjndi.jar file. The ibmjndi.jar file offered an IBM-developed LDAP service provider for older versions of the J2SDK.

Java Naming and Directory interface by Oracle.

JavaMail

The JavaMail API provides a set of abstract classes that models an electronic (e-mail) system. The API provides general mail functions for reading and sending mail, and requires service providers to implement the protocols.

Service providers implement specific protocols. For example, Simple Mail Transfer Protocol (SMTP) is a transport protocol for sending e-mail. Post Office Protocol 3 (POP3) is the standard protocol for receiving e-mail. Internet Message Access Protocol (IMAP) is an alternative protocol to POP3.

In addition to service providers, JavaMail requires the JavaBeans Activation Framework (JAF) to handle mail content that is not plain text. This includes Multipurpose Internet Mail Extensions (MIME), Uniform Resource Locator (URL) pages, and file attachments.

All the JavaMail components are shipped as part of SS1 (product ID 5770-SS1) Option 3. These components include the following:

- **mail.jar** This JAR file contains JavaMail APIs, the SMTP service provider, the POP3 service provider, and the IMAP service provider.
- **activation.jar** This JAR file contains the JavaBeans Activation Framework.

JavaMail

Java Print Service

The Java Print Service (JPS) API allows printing on all Java platforms. Java 1.4 and subsequent versions provide a framework in which Java runtime environments and third parties can provide stream generator plugins for producing various formats for printing, such as PDF, Postscript, and Advanced Function Presentation (AFP). These plugins create the output formats from bi-dimensional (2D) graphic calls.

A IBM i print service represents a printer device that is configured with the IBM i command Create Device Description (Printer) (CRTDEVPRT). Specify the publishing information parameters when you create a printer device. This increases the number of print service attributes supported by the IBM i print services.

If a printer supports Simple Network Management Protocol (SNMP), configure the printer on the server. Specify *IBMSNMPDRV for the value of the system driver program parameter in the CRTDEVPRT command. The print services use SNMP to retrieve specific information (printer service attributes) about a configured printer.

The Doc Flavors supported by the IBM i include *AFPDS, *SCS, *USERASCII - (PCL), *USERASCII - (Postscript), and *USERASCII - (PDF). Specify the Doc Flavors that the printer supports in the Data Streams Supported parameter within the Publishing Information of the CRTDEVPRT command.

When an application uses a print service to print a job (document) on the IBM i server, the print service places the document into a spooled file on an output queue with the same name as the printer device (also the same name as specified in the PrinterName attribute). Start a printer writer with the command STRPRTWTR before the documents print on the printer device.

In addition to the attributes defined by the Java Print Service specification, the IBM i print services support the following attributes for all Doc Flavors:

- PrinterFile (specifies a printer file, name and library, to be used when creating the spooled file)
- SaveSpooledFile (indicates whether to save the spooled file)
- UserData (a 10 character string of user defined data)
- JobHold (indicates whether to hold the spooled file)
- SourceDrawer (indicates the source drawer to use for the output media)

How to enable JPS

To enable the Java Print Service, ensure the following JAR files are added to the classpath:

- `/QIBM/ProdData/OS400/jt400/lib/jt400Native.jar`
- `/QIBM/ProdData/OS400/Java400/ext/ibmjps.jar`

Related information
Java Print Service by Oracle.

Code license and disclaimer information

IBM grants you a nonexclusive copyright license to use all programming code examples from which you can generate similar function tailored to your own specific needs.

SUBJECT TO ANY STATUTORY WARRANTIES WHICH CANNOT BE EXCLUDED, IBM, ITS PROGRAM DEVELOPERS AND SUPPLIERS MAKE NO WARRANTIES OR CONDITIONS EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OR CONDITIONS OF MERCHANTABILITY,

FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT, REGARDING THE PROGRAM OR TECHNICAL SUPPORT, IF ANY.

UNDER NO CIRCUMSTANCES IS IBM, ITS PROGRAM DEVELOPERS OR SUPPLIERS LIABLE FOR ANY OF THE FOLLOWING, EVEN IF INFORMED OF THEIR POSSIBILITY:

1. LOSS OF, OR DAMAGE TO, DATA;
2. DIRECT, SPECIAL, INCIDENTAL, OR INDIRECT DAMAGES, OR FOR ANY ECONOMIC CONSEQUENTIAL DAMAGES; OR
3. LOST PROFITS, BUSINESS, REVENUE, GOODWILL, OR ANTICIPATED SAVINGS.

SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF DIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, SO SOME OR ALL OF THE ABOVE LIMITATIONS OR EXCLUSIONS MAY NOT APPLY TO YOU.

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Software Interoperability Coordinator, Department YBWA
3605 Highway 52 N
Rochester, MN 55901
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs.

© Copyright IBM Corp. _enter the year or years_.

Programming interface information

This IBM Developer Kit for Java publication documents intended Programming Interfaces that allow the customer to write programs to obtain the services of IBM i.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml.

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Linux® is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Java and all Java-based trademarks and logos are trademarks of Oracle, Inc. in the United States, other countries, or both.

Other product and service names might be trademarks of IBM or other companies.

Terms and conditions

Permissions for the use of these publications is granted subject to the following terms and conditions.

Personal Use: You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative works of these publications, or any portion thereof, without the express consent of IBM.

Commercial Use: You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

Product Number: 5770-JV1