Creating UDRs in Java

Version 9.2
September 1999
Part No. 000-6220

Published by Informix® Press

Informix Corporation
4100 Bohannon Drive
Menlo Park, CA 94025-1032

Documentation Team: Evelyn Eldridge, Mary Kraemer, Judith Sherwood

Table of Contents

Introduction

In This Introduction

This Introduction provides an overview of the information in this manual and describes the conventions it uses.

About This Manual

This manual describes how to use Informix J/Foundation to write user-defined routines in the Java programming language for Informix Dynamic Server 2000. It describes the library of classes and interfaces that allow programmers to create and execute user-defined routines that access Informix database servers.

Types of Users

This manual is written for the following users:

- Database-application programmers
- DataBlade module developers
- Developers of Java user-defined routines (UDRs)

This manual assumes that you have basic knowledge in the following areas:

- Your computer, your operating system, and the utilities that your operating system provides
- Object-relational databases or exposure to database concepts
- The Java language and the Java Developer's Kit

- Java Database Connectivity (JDBC) 2.0, which is a Java application programming interface to SQL databases
- SQLJ: SQL Routines specification, which specifies the Java binding of SQL user-defined routines

If you have limited experience with object-relational databases, SQL, or your operating system, refer to *Getting Started with Informix Dynamic Server 2000* for a list of supplementary titles.

Software Dependencies

This manual assumes that you are using the following software:

- Informix Dynamic Server 2000, Version 9.2 (delivered as part of Informix Internet Foundation 2000)
- The Java Development Kit (JDK)

  The Informix JDBC driver requires the JDK. However, the JDK is *not* bundled with the database server. You must obtain a JDK or Java Runtime Environment (JRE) release. Because embedding a Java Virtual Machine (VM) depends on the implementation of the particular VM, Informix must certify each new JDK release against the database server. You must obtain a JDK (or JRE) release that is compatible with this database server release. Refer to the release notes described in "Documentation Notes, Release Notes, Machine Notes" on page 12 to know which JDK (or JRE) to use.

In addition, the DataBlade Developer's Kit (DBDK) for Java facilitates DataBlade module development.

Assumptions About Your Locale

Informix products can support many languages, cultures, and code sets. All culture-specific information is brought together in a single environment, called a Global Language Support (GLS) locale.

This manual assumes that you use the U.S. 8859-1 English locale as the default locale. The default is **en_us.8859-1** (ISO 8859-1) on UNIX platforms or **en_us.1252** (Microsoft 1252) for Windows NT environments. This locale supports U.S. English format conventions for dates, times, and currency, and also supports the ISO 8859-1 or Microsoft 1252 code set, which includes the ASCII code set plus many 8-bit characters such as é, è, and ñ.

If you plan to use nondefault characters in your data or your SQL identifiers, or if you want to conform to the nondefault collation rules of character data, you need to specify the appropriate nondefault locale.

For instructions on how to specify a nondefault locale, additional syntax, and other considerations related to GLS locales, see the *Informix Guide to GLS Functionality*.

Demonstration Databases

The DB-Access utility, which is provided with your Informix database server products, includes one or more of the following demonstration databases:

- The **stores_demo** database illustrates a relational schema with information about a fictitious wholesale sporting-goods distributor. Many examples in Informix manuals are based on the **stores_demo** database.
- The **superstores_demo** database illustrates an object-relational schema. The **superstores_demo** database contains examples of extended data types, type and table inheritance, and user-defined routines.

For information about how to create and populate the demonstration databases, see the *DB-Access User's Manual*. For descriptions of the databases and their contents, see the *Informix Guide to SQL: Reference*.

The scripts that you use to install the demonstration databases reside in the **$INFORMIXDIR/bin** directory on UNIX platforms and in the **%INFORMIXDIR%\bin** directory in Windows environments.

New Features

For a comprehensive list of new features for your database server, see your release notes.

Documentation Conventions

This section describes the conventions that this manual uses. These conventions make it easier to gather information from this and other volumes in the documentation set.

The following conventions are discussed:

- Typographical conventions
- Icon conventions
- Syntax conventions
- Command-line conventions
- Sample-code conventions
- Screen-illustration conventions

Typographical Conventions

This manual uses the following conventions to introduce new terms, illustrate screen displays, describe command syntax, and so forth.

Convention	Meaning
KEYWORD	All primary elements in a programming language statement (keywords) appear in uppercase letters in a serif font.
italics ***italics*** *`italics`*	Within text, new terms and emphasized words appear in italics. Within syntax and code examples, variable values that you are to specify appear in italics.
boldface ***boldface***	Names of program entities (such as classes, events, and tables), environment variables, file and pathnames, and interface elements (such as icons, menu items, and buttons) appear in boldface.
`monospace` *`monospace`*	Information that the product displays and information that you enter appear in a monospace typeface.
KEYSTROKE	Keys that you are to press appear in uppercase letters in a sans serif font.
♦	This symbol indicates the end of one or more product- or platform-specific paragraphs.
→	This symbol indicates a menu item. For example, "Choose **Tools→Options**" means choose the **Options** item from the **Tools** menu.

Tip: *When you are instructed to "enter" characters or to "execute" a command, immediately press* RETURN *after the entry. When you are instructed to "type" the text or to "press" other keys, no* RETURN *is required.*

Icon Conventions

Comment icons identify three types of information, as the following table describes. This information always appears in italics.

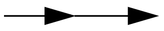
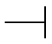
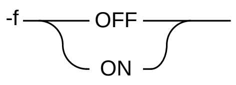
Icon	Label	Description
⚠	***Warning:***	Identifies paragraphs that contain vital instructions, cautions, or critical information
⇨	***Important:***	Identifies paragraphs that contain significant information about the feature or operation that is being described
💡	***Tip:***	Identifies paragraphs that offer additional details or shortcuts for the functionality that is being described

Command-Line Conventions

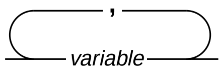
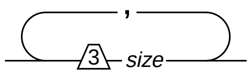
This section defines and illustrates the format of commands that are available in Informix products. These commands have their own conventions, which might include alternative forms of a command, required and optional parts of the command, and so forth.

Each diagram displays the sequences of required and optional elements that are valid in a command. A diagram begins at the upper-left corner with a command. It ends at the upper-right corner with a vertical line. Between these points, you can trace any path that does not stop or back up. Each path describes a valid form of the command. You must supply a value for words that are in italics.

You might encounter one or more of the following elements on a command-line path.

Element	Description
command	This required element is usually the product name or other short word that invokes the product or calls the compiler or preprocessor script for a compiled Informix product. It might appear alone or precede one or more options. You must spell a command exactly as shown and use lowercase letters.
variable	A word in italics represents a value that you must supply, such as a database, file, or program name. A table following the diagram explains the value.
-flag	A flag is usually an abbreviation for a function, menu, or option name, or for a compiler or preprocessor argument. You must enter a flag exactly as shown, including the preceding hyphen.
.ext	A filename extension, such as **.sql** or **.cob**, might follow a variable that represents a filename. Type this extension exactly as shown, immediately after the name of the file. The extension might be optional in certain products.
(. , ; + * - /)	Punctuation and mathematical notations are literal symbols that you must enter exactly as shown.
' '	Single quotes are literal symbols that you must enter as shown.
	Syntax within a pair of arrows indicates a subdiagram.
	The vertical line terminates the command.
	A branch below the main path indicates an optional path. (Any term on the main path is required, unless a branch can circumvent it.)

(1 of 2)

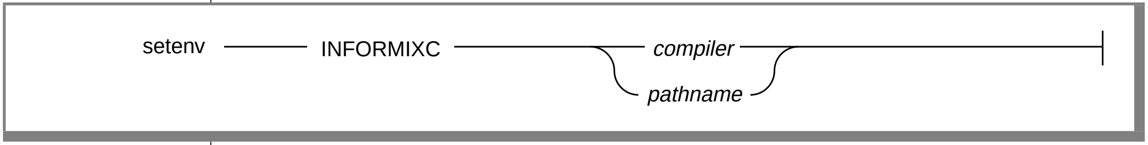
Element	Description
	A loop indicates a path that you can repeat. Punctuation along the top of the loop indicates the separator symbol for list items.
	A gate (3) on a path indicates that you can only use that path the indicated number of times, even if it is part of a larger loop. You can specify *size* no more than three times within this statement segment.

(2 of 2)

How to Read a Command-Line Diagram

Figure 1 shows a command-line diagram that uses some of the elements that are listed in the previous table.

Figure 1
Example of a Command-Line Diagram



To construct a command correctly, start at the top left with the command. Follow the diagram to the right, including the elements that you want. The elements in the diagram are case sensitive.

Figure 1 illustrates the following steps:

1. Type `setenv`.
2. Type `INFORMIXC`.
3. Supply either a compiler name or a pathname.

   After you choose *compiler* or *pathname*, you come to the terminator. Your command is complete.
4. Press RETURN to execute the command.

Sample-Code Conventions

Examples of SQL code occur throughout this manual. Except where noted, the code is not specific to any single Informix application development tool. If only SQL statements are listed in the example, they are not delimited by semicolons. For instance, you might see the code in the following example:

```
CONNECT TO stores_demo
...

DELETE FROM customer
    WHERE customer_num = 121
...

COMMIT WORK
DISCONNECT CURRENT
```

To use this SQL code for a specific product, you must apply the syntax rules for that product. For example, if you are using *DB-Access User's Manual*, you must delimit multiple statements with semicolons. If you are using an SQL API, you must use EXEC SQL at the start of each statement and a semicolon (or other appropriate delimiter) at the end of the statement.

Tip: *Ellipsis points in a code example indicate that more code would be added in a full application, but it is not necessary to show it to describe the concept being discussed.*

For detailed directions on using SQL statements for a particular application development tool or SQL API, see the manual for your product.

Additional Documentation

For additional information, you might want to refer to the following types of documentation:

- On-line manuals
- On-line examples
- Printed manuals
- On-line help

- Error message documentation
- Documentation notes, release notes, and machine notes
- Related reading

On-Line Manuals

An Answers OnLine CD that contains Informix manuals in electronic format is provided with your Informix products. You can install the documentation or access it directly from the CD. For information about how to install, read, and print on-line manuals, see the installation insert that accompanies Answers OnLine.

Informix on-line manuals are also available on the following Web site:

```
www.informix.com/answers
```

On-Line Examples

The file **$INFORMIXDIR/extend/krakatoa/examples.tar** contains examples of UDRs written in Java that you can study or use as models for your own UDRs.

The Informix by Example Web site includes examples of UDRs written in Java on the following web site:

```
http://examples.informix.com/topics/jsrvr
```

Printed Manuals

To order printed manuals, call 1-800-331-1763 or send email to moreinfo@informix.com. Please provide the following information when you place your order:

- The documentation that you need
- The quantity that you need
- Your name, address, and telephone number

WIN NT

On-Line Help

Informix provides on-line help with each graphical user interface (GUI) that displays information about those interfaces and the functions that they perform. Use the help facilities that each GUI provides to display the on-line help. ♦

Error Message Documentation

Informix software products provide ASCII files that contain all of the Informix error messages and their corrective actions.

UNIX

To read error messages and corrective actions on UNIX, use one of the following utilities.

Utility	Description
finderr	Displays error messages on line
rofferr	Formats error messages for printing

♦

WIN NT

To read error messages and corrective actions in Windows environments, use the **Informix Find Error** utility. To display this utility, choose **Start→Programs→Informix** from the Task Bar. ♦

Instructions for using the preceding utilities are available in Answers OnLine. Answers OnLine also provides a listing of error messages and corrective actions in HTML format.

Documentation Notes, Release Notes, Machine Notes

In addition to printed documentation, the following sections describe the on-line files that supplement the information in this manual. Please examine these files before you begin using your database server. They contain vital information about application and performance issues.

UNIX

On UNIX platforms, the following on-line files appear in the **$INFORMIXDIR/release/en_us/0333** directory.

On-Line File	Purpose
JAVADOC_9.2	The documentation-notes file for your version of this manual describes topics that are not covered in the manual or that were modified since publication.
SERVERS_9.2	The release-notes file describes feature differences from earlier versions of Informix products and how these differences might affect current products. This file also contains information about any known problems and their workarounds.
IDS_9.2	The machine-notes file describes any special actions that you must take to configure and use Informix products on your computer. Machine notes are named for the product described.

♦

WIN NT

The following items appear in the **Informix** folder. To display this folder, choose **Start→Programs→Informix** from the Task Bar.

Program Group Item	Description
Documentation notes	This item includes additions or corrections to manuals, along with information about features that might not be covered in the manuals or that have been modified since publication.
Release notes	This item describes feature differences from earlier versions of Informix products and how these differences might affect current products. This file also contains information about any known problems and their workarounds.

Machine notes do not apply to Windows environments. ♦

Related Reading

For a list of publications that provide an introduction to database servers and operating-system platforms, refer to your *Getting Started* manual.

In addition, the following items are essential readings:

- SQLJ: SQL Routines documentation on the following Web site:

  ```
  http://www.sqlj.org
  ```

  The Informix implementation of UDRs written in Java conforms with the SQLJ specification.
- JDBC specification, Versions 1.0 and 2.0

  Descriptions of the JDBC packages can be found on the Javasoft Web site:

  ```
  http://java.sun.com
  ```
- *Informix JDBC Driver Programmer's Guide*, Version 1.22

  This Informix document describes the classes that the Informix JDBC driver supports.

Compliance with Industry Standards

The SQLJ consortium defines various standards for Java in databases. The Informix implementation of UDRs written in Java conforms with the SQLJ: SQL Routines specification, which defines static Java stored procedures. Informix provides extensions that take full advantage of the extensibility capabilities of the database server.

A separate standard called Java Database Connectivity (JDBC) specifies how a Java application can access database management system (DBMS) resources.

- Version 1 of JDBC is a Java binding of ODBC and it does not address extended data types.
- Version 2 of JDBC adds support for distinct types, struct types, and other SQL 99 data types.

Informix support for UDRs written in Java follows JDBC 1.0 and part of JDBC 2.0 standards for accessing database resources.

Informix Welcomes Your Comments

Let us know what you like or dislike about our manuals. To help us with future versions of our manuals, we want to know about any corrections or clarifications that you would find useful. Include the following information:

- The name and version of the manual that you are using
- Any comments that you have about the manual
- Your name, address, and phone number

Send electronic mail to us at the following address:

doc@informix.com

The **doc** alias is reserved exclusively for reporting errors and omissions in our documentation.

We appreciate your suggestions.

Chapter

Concepts

1

In This Chapter

This chapter introduces the Informix infrastructure for creating and executing user-defined routines (UDRs) written in Java. UDRs written in Java are for:

- Database-application programmers
- DataBlade module developers who use Java to create objects that handle the new data types in their DataBlade modules
- Developers who create user-defined routines written in Java for inclusion in SQL programs

Features of UDRs Written in Java

Informix provides the infrastructure to support UDRs that are written in Java. To support these UDRs, the database server binds SQL UDR signatures to Java executables and provides mapping between SQL data values and Java objects so that it can pass parameters and retrieve returned results.

Informix also provides support for data type extensibility and sophisticated error handling.

Java Virtual Processors

UDRs written in Java execute on specialized virtual processors, called *Java virtual processors* (JVPs). A JVP embeds a Java virtual machine (JVM) in its code.

The JVPs are responsible for executing all UDRs written in Java. Although the JVPs are mainly used for Java-related computation, they have the same capabilities as a CPU VP and they can process all types of SQL queries. This *embedded VM architecture* avoids the cost of shipping Java-related queries back and forth between CPU VPs and JVPs.

Thread Scheduling

When the JVP starts the JVM, the entire database server component is thought of as running on one particular Java thread, called the *main thread*. The JVM controls the scheduling of Java threads and the Informix Dynamic Server scheduler multiplexes Informix threads on top of the Java main thread. In other words, the Informix thread package is stacked on top of the Java thread package.

Query Parallelization

While Java applications use threads for parallelism, the Informix database servers use threads for overlapping latency. That is, Informix threads run concurrently but not in parallel. To parallelize a query the database server must spread the work among multiple virtual processors.

Consequently, the database server must have multiple instances of JVPs to parallelize UDR calls. Because the JVM embedded in different VPs does not share states, you cannot store global states using Java class variables. All global states must be stored in the database to be consistent. The only guarantee from the database server is that any given UDR instance executes from start to finish on the same VP. The database server enforces a round-robin scheduling policy where the UDR instances are spread over the JVPs before they start executing.

WIN NT

The consistency of multiple JVMs is not an issue on the Windows NT platform because all VPs are mapped to kernel threads instead of processes. Since all VPs share the same process space, you do not need to start multiple instances of the JVM.♦

System Catalog Tables

The **sysroutinelangs**, **syslangauth**, and **sysprocedures** system catalog tables contain information about the UDRs written in Java.

The **sysroutinelangs** table lists the programming languages that you can use to write UDRs. The table gives the names of the language initialization function and the path for the language library.

The **syslangauth** table specifies who is allowed to use the language. For Java, the default is the database administrator. For information about modifying the use privileges, refer to the GRANT statement in *Informix Guide to SQL: Syntax*.

The **sysprocedures** table lists several built-in procedures that Informix provides.

For more information about these system catalog tables, refer to "Finding Information about UDRs" on page 4-26 and to the *Informix Guide to SQL: Reference*.

Chapter

Preparing to Support Java

2

In This Chapter

This chapter describes how to install and configure the database server to provide UDRs written in Java. To create and use UDRs written in Java, you must install the following software:

- Dynamic Server, Version 9.2
- Informix Internet Foundation 2000
- The Java Development Kit (JDK)

You might also want to install the DataBlade Developers Kit, Version 4.0 or greater, to facilitate development of UDRs in Java.

If you do not plan to develop UDRs in Java, you can install the Java Runtime Environment (JRE) instead of the JDK. The JRE is a subset of the JDK that allows you to execute existing Java binaries but not compile new source code.

This software is described in "Software Dependencies" on page 4. For more detailed information on the required software, refer to the release notes described in "Documentation Notes, Release Notes, Machine Notes" on page 13.

Installing the JDBC Driver

The Informix JDBC driver that is included with the database server contains Java classes and shared-object files that allow you to write user-defined routines (UDRs) in Java. The installation procedure installs these binaries in **$INFORMIXDIR/extend/krakatoa**.

For more information, refer to the machine-notes file described in "Documentation Notes, Release Notes, Machine Notes" on page 13.

Configuring to Support Java

The basic configuration procedure for an Informix database server is covered in the *Administrator's Guide for Informix Dynamic Server 2000*. Configuring the database server to support Java requires several additional steps. You might find it convenient to configure the database server without Java and then modify it to add Java support.

Preparing to use Java with the database server requires these additions to the basic configuration procedure:

- Create an sbspace to hold the Java jar files.
- Create the JVP properties file.
- Add (or modify) the Java configuration parameters in the ONCONFIG configuration file.
- Set environment variables.

$INFORMIXDIR/extend/krakatoa is your *jvphome*. You will need to include this path in several places as you prepare Java in the server.

Creating an sbspace

The database server stores Java jar files as smart large objects in the system default sbspace. You must create an sbspace. For example:

```
onspaces -c -S mysbspace -g 5 -p /dev/raw_dev1 -o 500 -s 20000
-m /dev/raw_dev2 500
```

After you create the sbspace, set the SBSPACENAME configuration to the name that you gave to the sbspace (**mysbspace** in the preceding example).

Jar files coexist in the system default sbspace with other smart large objects that you store in that space. When you choose the size for your default sbspace, you need to consider how much space those objects require, as well as the number and size of the jars you plan to install.

For information about the **onspaces** command, refer to the *Informix Administrator's Reference*.

Creating the JVP Properties File

A *JVP property file* contains property settings that control various runtime behaviors of the Informix JDBC driver. The JVPPROPFILE configuration parameter specifies the path to the properties file. When you initialize the database server, the JVP initializes the environment based on the settings in the JVP property file. The **.jvpprops.template** file in the **$INFORMIXDIR/extend/krakatoa** directory documents the properties that you can set.

To prepare the Java properties file

1. Copy the Java properties template file, ***jvphome*/.jvpprops.template** into ***jvphome*/.jvpprops** where *jvphome* is the directory **$INFORMIXDIR/extend/krakatoa**.
2. Edit **.jvpprops** to change the trace level or other properties if necessary.
3. Set the JVPPROPFILE configuration parameter to ***jvphome*/.jvpprops**.

A sample properties file might contain the following items:

```
JVP.trace.settings:JVP=2
JVP.trace.verbose:1
JVP.trace.timestampformat:HH:MM
JVP.splitLog:1000
JVP.monitor.port: 10000
```

The database server provides a fixed set of system trace events such as UDR sequence initialization, activation, and shutdown. You can also generate application-specific traces. For more information, see the description of the **UDRTraceable** class, "com.informix.udr.UDRTraceable" on page 4-11.

Setting Configuration Parameters

The ONCONFIG configuration file (**$INFORMIXDIR/etc/$ONCONFIG**) includes the following configuration parameters that affect Java:

- JDKVERSION
- JVPPROPFILE
- JVMTHREAD
- JVPCLASSPATH

- JVPHOME
- JVPJAVALIB
- JVPJAVAVM
- JVPLOGFILE
- VPCLASS

The following example shows sample settings for the Java-related configuration parameters on a UNIX Solaris system. In this example *jvphome* is **$INFORMIXDIR/extend/krakatoa**.

```
JVPHOME         jvphome
JVPLOGFILE      jvphome/jvp.log
JVPPROPFILE     jvphome/.jvpprops
JVPJAVAVM     java_g:net_g;zip_g:mmedia_g:jpeg_g:
                    sysresource_g:agent_g
VPCLASS         jvp,num=1
JDKVERSION      1.1
JVMTHREAD       native
JVPJAVALIB      /lib/sparc/native_threads
JVPCLASSPATH    jvphome/krakatoa_g.jar:jvphome/jdbc_g.jar
```

In this example, the JVPJAVAM and JVPCLASSPATH are set appropriately for debug mode. To run in nondebug mode, remove all the **_g** suffixes.

For more information about the configuration parameters, refer to Chapter 3, "Configuration Parameters." For information about specific settings of the configuration parameters on your platform, refer to the machine notes documented in "Documentation Notes, Release Notes, Machine Notes" on page 13 of the Introduction and to **$INFORMIXDIR/etc/onconfig.std**.

Setting Environment Variables

You do not need any extra environment variables to *execute* UDRs written in Java. However, you must include ***jvphome*/krakatoa.jar** in your **CLASSPATH** environment variable so that JDK can compile the Java source files that use Informix Java packages.

Chapter

Configuration Parameters

3

In This Chapter

This chapter documents the configuration parameters that you need to set to use UDRs written in Java. Set these parameters in the database server configuration file (the ONCONFIG file).

For a sample environment that configuration parameters establish, see the release notes described in "Documentation Notes, Release Notes, Machine Notes" on page 13.

JDKVERSION

onconfig.std *value*	1.1
range of values	For this release, the only valid value is 1.1.
takes effect	When shared memory is initialized

JDKVERSION is the major version of the JDK or JRE release. That is, the version number does not include *x* when the version is JDK 1.1.*x*.

This parameter is required if the number of JVPs (set in VPCLASS JVP) is greater than 0.

JVMTHREAD

onconfig.std *value*	native
range of values	green or native
takes effect	When shared memory is initialized

The JVMTHREAD configuration parameter specifies the thread package to use for the Java Virtual Machine (JVM). It is either *green* or *native*. A native thread uses platform-based kernel threads. The green thread package is a user-level thread package that the Javasoft implementation of JDK uses on platforms where the native-thread JDK is not yet mature.

The value of this parameter is platform dependent. To find the proper value for JVMTHREAD, refer to the machine and release notes described in "Documentation Notes, Release Notes, Machine Notes" on page 13 of the Introduction. The JVMTHREAD setting affects the JVPJAVAM and JVPJAVALIB settings.

This parameter is required if the number of JVPs (set in VPCLASS JVP) is greater than 0.

JVPCLASSPATH

onconfig.std *value*	**/usr/informix/extend/krakatoa/krakatoa_g.jar: /usr/informix/extend/krakatoa/jdbc_g.jar**
takes effect	When shared memory is initialized

The JVPCLASSPATH configuration parameter is the initial Java class path setting. You must modify the default setting in the configuration file by replacing **/usr/informix/extend/krakatoa** with *JVPHOME_path*, the pathname in your JVPHOME configuration parameter:

```
    JVPHOME_path/krakatoa_g.jar:JVPHOME_path/jdbc_g:jar
```

If you do not require debugging, you can use the nondebug JDK libraries for better performance. To use the nondebug version, use the following the JVPJAVAVM setting:

```
    java:net:zip:mmedia:jpeg:sysresource.
```

If you do not require the debug versions of the JAR files, use the following JVPCLASSPATH setting:

```
    JVPHOME_path/krakatoa_g.jar:JVPHOME_path/jdbc.jar
```

The JVPCLASSPATH parameter is required if the number of JVPs (set in VPCLASS JVP) is greater than 0.

Tip: *The JVP ignores the **CLASSPATH** environment variable. However, you must set the **CLASSPATH** environment variable so that you can compile your UDRs.*

JVPHOME

onconfig.std value	**/usr/informix/extend/krakatoa**
takes effect	When shared memory is initialized

The JVPHOME configuration parameter specifies the directory where the classes of the Informix JDBC driver are installed. To modify the default setting in the configuration file, replace **/usr/informix** with the pathname of your **$INFORMIXDIR**.

The JVPHOME value, *JVPHOME_path*, is used in several configuration parameters. If the JVPHOME location changes, you must change the configuration settings of all parameters that use the JVPHOME value.

This parameter is required if the number of JVPs (set in VPCLASS JVP) is greater than 0.

JVPJAVAHOME

onconfig.std *value*	**/usr/java**
takes effect	When shared memory is initialized

The JVPJAVAHOME configuration parameter specifies the directory where the JDK or JRE release is installed. Typically, JDK or JRE is installed in **/usr/java**. If you install JDK or JRE in another location, you must modify this parameter.

This parameter is required if the number of JVPs (set in VPCLASS JVP) is greater than 0.

JVPJAVALIB

onconfig.std *value*	*platform-specific value*
takes effect	When shared memory is initialized

The JVPJAVALIB configuration parameter specifies the path from **$JVPJAVAHOME** to the location of the Java VM libraries.

The value of this parameter is platform dependent. To find the proper value for JVPJAVALIB, refer to the machine and release notes described in "Documentation Notes, Release Notes, Machine Notes" on page 13 of the Introduction.

This parameter is required if the number of JVPs (set in VPCLASS JVP) is greater than 0.

JVPJAVAVM

***onconfig.std* value**	*platform-specific value*
separators	colon
takes effect	When shared memory is initialized

The JVPJAVAVM configuration parameter lists the JVM libraries that the database server should load. The names in this list exclude the **lib** prefix and **.so** or **.dll** suffix. Entries in the list are separated by colons.

This parameter is required if the number of JVPs (set in VPCLASS JVP) is greater than 0.

UNIX

For example, for UNIX Solaris, use the following value for JVPJAVAVM if you are using a debug version of the JDBC driver:

```
java_g:net_g:zip_g:mmedia_g:jpeg_g:sysresource_g:agent_g
```

If you use a nondebug JDBC driver, you can use the nondebug JDK libraries for better performance. Set JVPJAVAVM to the following:

```
java:net:zip:mmedia:jpeg:sysresource:agent
```

♦

The value of JVPJAVAVM is platform dependent. To find the proper value for JVPJAVAVM, refer to the machine and release notes described in “Documentation Notes, Release Notes, Machine Notes” on page 13 of the Introduction.

JVPLOGFILE

***onconfig.std** value*	**/usr/informix/jvp.lo**g
range of values	Any valid complete filename
takes effect	When shared memory is initialized

The JVPLOGFILE configuration parameter specifies the path to the Java VP log file. If this parameter is not specified, it defaults to *JVPHOME_path*/**jvp.log**.

The database server can generate Java trace outputs and stack dumps. The database server writes this output to the Java log file. If JVPLOGFILE is not set in the ONCONFIG file, the JVP uses the following log file:

```
JVPHOME/jvp.log
```

To change the location of the log file, change the JVPLOGFILE configuration parameter. For example, the following line sets the log file to **/u/sam/jvp.log**:

```
JVPLOGFILE /u/sam/jvp.log
```

This parameter is optional.

JVPPROPFILE

***onconfig.std** value*	**/usr/informix/extend/krakatoa/.jvpprops**
takes effect	When shared memory is initialized

The JVPPROPFILE configuration parameter specifies the path to the Java VP properties file, if any. Set this parameter as follows, where *JVPHOME_path* is the value in your JVPHOME configuration parameter:

```
JVPHOME_path/.jvpprops
```

This parameter is optional.

SBSPACENAME

***onconfig.std* value**	*blank*
takes effect	When shared memory is initialized
refer to	"Creating an sbspace" on page 2-4

The SBSPACENAME configuration parameter specifies the name of the system default sbspace. You must provide a smart large object where the database server can store the Java JAR files.

This parameter is not exclusively for Java. If your database tables include smart-large-object columns that do not explicitly specify a storage space, that data is stored in the sbspace specified by SBSPACENAME.

For information about specifying a storage space for smart large objects, refer to the CREATE TABLE statement in the *Informix Guide to SQL: Syntax*. For more information about SBSPACENAME, refer to the *Informix Administrator's Reference*.

Tip: *Informix suggests that when you use UDRs written in Java, you create separate sbspaces for storing your smart large objects.*

VPCLASS JVP

***onconfig.std* value**	*not set*
range of values	0 and positive integers
takes effect	When shared memory is initialized

The VPCLASS JVP configuration parameter specifies the number of virtual processors to initialize for a given virtual-processor class. The JVP option of VPCLASS specifies the number of Java virtual processors that the database server should start.

This parameter is required when you use the Informix JDBC driver.

Set this option as follows, where *number* is the number of Java virtual processors:

```
VPCLASS JVP,num=number
```

The default value of number is 1. If you set the number of JVPs to zero (0), execution of UDRs written in Java is disabled.

If you have not correctly installed and configured the software for Java in the server, the JVP fails to start when you start the database server. However, the database server itself continues to initialize normally. The main database log file contains a message that indicates the cause of the JVP failure.

For more information about the VPCLASS configuration parameter, refer to the *Informix Administrator's Reference*.

Chapter

Creating UDRs Written in Java

4

In This Chapter

A *user-defined routine* (UDR) is a routine that an SQL statement or another UDR can invoke. UDRs written in Java use the server-side implementation of the Informix JDBC driver to communicate with the database server.

This chapter provides the following information about UDRs written in Java:

- What tasks a UDR can perform
- How to create a UDR

UDRs Written in Java

The behaviors of installing and invoking UDRs written in Java follow the SQLJ: SQL Routines specification. Every UDR written in Java maps to an external Java static method whose class resides in a Java Archive (JAR) file that was installed in a database. The SQL-to-Java data type mapping is done according to the JDBC specification.

UDRs can be user-defined functions or user-defined procedures, which can return values or not, as follows:

- A *user-defined function* returns one or more values and therefore can be used in SQL expressions.

  For example, the following query returns the results of a UDR called **area()** as part of the query results:

  ```
  SELECT diameter, area(diameter) FROM shapes
  WHERE diameter > 6
  ```

- A *user-defined procedure* is a routine that optionally accepts a set of arguments and does not return any values.

  A procedure *cannot* be used in SQL expressions because it does not return a value. However, you can call it directly, as the following example shows:

  ```
  EXECUTE PROCEDURE myproc(1, 5)
  ```

  You can also call user-defined procedures within triggers.

For general information about UDRs, refer to *Extending Informix Dynamic Server 2000*.

UDRs written in Java can perform the following tasks.

Type of UDR	Purpose
End-user routine	A UDR that performs some common task for an end user.
User-defined aggregate	A UDR that calculates an aggregate value on a particular column or value.
Parallelizable UDR	A UDR that can run in parallel when executed within an SQL statement. (UDRs that open JDBC connections cannot run in parallel.)
Cast function	A UDR that converts or casts one data type to another.
Operator function	A UDR that implements some operator symbol (such as +, -, or /).
Iterator function	A user-defined function that returns more than one row of data. Iterator functions written in Java are supported using some Informix extensions.
Functional index	A UDR on which an index can be built.
Opaque data type support function	A user-defined function that tells the database server how to handle the data of an opaque data type. However, you cannot write **send** and **receive** support functions in this version of the database server.
Negator function	A function that calculates the *not* operation for a particular operator or function.

You *cannot* use UDRs written in Java for any of the following features:

- Commutator functions
- Cost functions
- Internal functions
- Operator-class functions
- Send, receive, importbin, or exportbin functions
- Selectivity functions
- User-defined statistics functions

Creating a UDR in Java

When you create a UDR in Java, you need to write and compile the source code and then install the finished code in the database server.

To create a UDR in Java

1. Write the UDR, which can use the JDBC methods to interact with the database server.
2. If the UDR uses any user-defined types, for each user-defined type write a Java class that translates between the server and Java representation of the type. This class should implement the **SQLData** interface. For information about **SQLData**, refer to the JDBC 2.0 specification.
3. Write the CREATE FUNCTION or CREATE PROCEDURE statement for registering the UDR.
4. Write the deployment descriptor, which contains the SQL statements for registering the UDR.
5. Prepare the manifest file.
6. Compile the Java source files and collect the compiled code into a JAR file.
7. Create a jar file that contains the classes, deployment descriptor, and manifest file.
8. Install the jar file that contains the UDR in the current database.

9. Execute the UDR.
10. Use tracing and the debugging features to work out any problems in the UDR.
11. Optimize performance of the UDR.

For general information on developing a UDR, refer to *Extending Informix Dynamic Server 2000*. This section briefly describes each of these steps in the development of a UDR.

Tip: *Informix recommends that you use the DataBlade Developers Kit (DBDK), Version 4.0 or later, to help write UDRs in Java. DBDK enforces standards that facilitate migration between different versions of the database server.*

Writing a UDR in Java

UDRs written in Java can use the following packages, interfaces, classes, and methods:

- Java packages

  UDRs can use all the basic nongraphical Java packages that are in the JDK. That is, UDRs can use **java.util.***, **java.io.***, **java.net.***, **java.rmi.***, and so on. UDRs cannot use **java.awt.***, **java.applet.*** and other user-interface packages. For more information on these packages, see the Java Development Kit documentation.

- Java Database Connectivity (JDBC) 1.0 API

  UDRs can use the JDBC 1.0 API to access the database. For more information, see "JDBC 1.0 API" on page 5-5.

  The **$INFORMIXDIR/extend/krakatoa/examples.tar** file of on-line examples includes a sample of JDBC in a UDR in **JDBC.java**.

- Informix JDBC Extensions

  UDRs can also use Informix extensions to JDBC 1.0 to access some JDBC 2.0 functionality. For more information, see Chapter 5, "The Informix JDBC Driver."

- Informix extensions for UDRs written in Java

  Certain Informix extensions are available to applications that need to exploit the capabilities of the database server. The Informix extensions reside in the **com.informix.udr** package.

The Informix **com.informix.udr** package data type provides extensions to SQLJ that allow applications to exploit the capabilities of Dynamic Server. Such extensions include logging, tracing, iterator support, and invocation-state management.

com.informix.udr Package

The **com.informix.udr** package contains the following public interfaces:

- **com.informix.udr.UDRManager**
- **com.informix.udr.UDREnv**
- **com.informix.udr.UDRLog**
- **com.informix.udr.UDRTraceable**

The following sections describe each of these Informix-specific extensions in more detail.

com.informix.udr.UDRManager

The **UDRManager** class provides a method for a UDR instance to obtain its **UDREnv** object. This class is defined as follows:

```
public class UDRManager
{
        static UDREnv getUDREnv();
}
```

The SQLJ: SQL Routines specification, which describes how to use static Java methods as database UDRs, does not provide a mechanism to save the user state across invocations. The **UDREnv** interface is an Informix-provided interface that maintains state information. You can use this state information, for example, to write iterator UDRs. The **UDREnv** object is maintained by the thread that manages the execution of the static method representing the UDR. Therefore, if the UDR forks its own threads, the **UDRManager.getUDREnv** method cannot be directly used by those secondary threads of the UDR. The UDR must explicitly pass the **UDREnv** object to the secondary threads that it creates.

com.informix.udr.UDREnv

The **UDREnv** interface consists of methods for accessing and manipulating the routine state of the UDR. It exposes a subset of the routine-state information in the MI_FPARAM structure (which holds routine-state information for C UDRs). It also contains some utilities related to the JVP, such as logging and tracing.

The online examples in **$INFORMIXDIR/extend/krakatoa/examples.tar** include an example of the **UDREnv** class in **Env.java**.

The **UDREnv** interface is defined as follows:

```
public interface UDREnv
{
    // Information about the UDR signature

    String getName();
    String[] getParamTypeName();
    String getReturnTypeName();

    // For maintaining state across UDR invocations

    void setUDRState (Object state);
    Object getUDRState();

    // For set/iterator processing

    public static final int UDR_SET_INIT = 1;
    public static final int UDR_SET_RETONE = 2;
    public static final int UDR_SET_END = 3;
    int getSetIterationState();
    void setSetIterationIsDone(boolean value);

    // Logging and Tracing

    UDRTraceable getTraceable();
    UDRLog getLog();
}
```

The **getName()** method returns the name of the UDR as it is registered in database.

The **getParamTypeName()** and **getReturnTypeName()** methods return the SQL data type names for the UDR arguments and the return value, respectively.

The **setUDRState()** method sets the user-state pointer for the UDR. It stores a given object in the context of the UDR instance. The object might contain states that are shared across UDR invocations (such as a JDBC connection handle or a **UDRLog** object). The **getUDRState()** method returns the object set by the latest call to **setUDRState()**.

The **getSetIterationState()** method retrieves the iterator status for an iterator function. (This method is analogous to the C-language accessor **mi_fp_request** for set iterators.) This method returns one of the following values.

Iterator-Status Constant	Meaning	Use
UDR_SET_INIT	This is *first* time that the iterator function is called.	Initialize the user state for the iterator function.
UDR_SET_RETONE	This is an actual iteration of the iterator function.	Return items of the active set, one per iteration.
UDR_SET_END	This is the last time that the iterator function is called.	Free any resources associated with the user state.

The **setSetIterationIsDone()** method sets the iterator-completion flag for an iterator function. Use the **setSetIterationIsDone()** method to tell the database server whether the current iterator function has reached its end condition. An *end condition* indicates that the generation of the active set is complete. The database server calls the iterator function with the UDR_SET_RETONE iterator-status value as long as the end condition has *not* been set.

The **getLog()** method returns a **UDRLog** interface for logging uses. For more information on the **UDRLog** interface, see "com.informix.udr.UDRLog" on page 4-10.

The **getTraceable()** method returns a **UDRTraceable** interface for the UDRs to use. For more information on the **UDRTraceable** interface, see "com.informix.udr.UDRTraceable" on page 4-11.

com.informix.udr.UDRLog

The **UDRLog** interface provides a simple logging facility for a UDR. The **UDRLog** interface is defined as follows:

```
public interface UDRLog
{
    void log(String msg);
}
```

The interface defines a single method, **log()**, which takes a String argument and appends it to the JVP log file, which the JVPLOGFILE configuration parameter specifies. For more information, see "Setting the JVP Log File" on page 4-24.

com.informix.udr.UDRTraceable

The **UDRTraceable** interface supports *zone-based* tracing. A trace zone is a conceptual code component. For example, you can put all UDRs in the same zone and all general-purpose Java applications in another. Each zone can have its own *trace level* that dictates the granularity of tracing. The zones form a hierarchy where subzones inherit the trace levels of their parents. You can define the zones, their hierarchical relationship, and trace levels with the following features:

- The settings in the JVP property file (which the JVPPROPFILE configuration parameter specifies)
- Calls to the **UDRTraceable** methods at program execution time

The **UDRTraceable** interface is defined as follows:

```
public interface UDRTraceable extends Traceable
{
    public static final int TRACE_OFF = 0;
    public static final int TRACE_MINIMAL = 1;
    public static final int TRACE_COARSE = 2;
    public static final int TRACE_MEDIUM = 3;
    public static final int TRACE_FINE = 4;
    public static final int TRACE_SUPERFINE = 5;

    int traceLevel(String zone);
    void traceSet(String zone, int level);
    void tracePrint(String zone, int level, String message);
}
```

The **traceLevel()** method returns the current trace-level setting for the given trace zone. The predefined trace levels are as follows.

Trace-Level Constant	Description
TRACE_OFF	No trace output is generated
TRACE_MINIMAL	Basic tracing
TRACE_COARSE	Coarse-grained tracing
TRACE_MEDIUM	Medium-grained tracing
TRACE_FINE	Fine-grained tracing
TRACE_SUPERFINE	For the trace sessions that require all possible details

The **traceSet()** method sets the specified trace zone to the specified trace level.

The **tracePrint()** method sends the specified message to the JVP log file if the trace zone has a trace level that is greater than or equal to the *level* parameter. The JVPLOGFILE configuration parameter specifies the JVP log file name. For more information, see "Setting the JVP Log File" on page 4-24.

Creating UDT-to-Java Mappings

The routine manager needs a mapping between SQL data values and Java objects to be able to pass parameters to and retrieve return results from a UDR. The SQL to Java data-type mapping is performed according to the JDBC specification. For built-in SQL data types, the routine manager can use mappings to existing JDBC data types.

For any user-defined SQL data types that your UDR uses, you must create mappings. You can use the following user-defined data types in UDRs written in Java.

User-Defined Data Type	SQL Statement
Distinct data type	CREATE DISTINCT TYPE
Opaque data type	CREATE OPAQUE TYPE

Warning: *You cannot use row or collection data types in UDRs written in Java.*

To create the mapping between a user-defined SQL data type and a Java object

1. Create a user-defined class that implements the **SQLData** interface. (For more information, refer to the JDBC 2.0 specification).
2. Bind this user-defined class to the user-defined SQL data type using the **setUDTExtName** built-in procedure.

   Because the SQL statements that create user-defined types do not currently provide a clause for specifying the external name of a user-defined data type, you must define this mapping. Use the following Informix built-in procedures with the EXECUTE PROCEDURE statement to define the mapping:

   - ❑ **sqlj.setUDTExtName()**

     This procedure defines the mapping between a user-defined data type and a Java data type.

   - ❑ **sqlj.unsetUDTExtName()**

     This procedure removes the SQL-to-Java mapping and removes any cached copy of the Java class from database server shared memory.

   For example:

   ```
   -- Creating or removing UDT-to-Java Mappings
   EXECUTE PROCEDURE sqlj.setUDTExtName('udt_name',
           'class_name.udtname’);
   EXECUTE PROCEDURE sqlj.unsetUDTExtName('udt_name');
   ```

The online examples in **$INFORMIXDIR/extend/krakatoa/examples.tar** include a sample implementation of a UDT written in Java, **Circle.java**.

Registering UDRs Written in Java

For a UDR to be invoked in an SQL statement, it must be registered in the current database. Use the CREATE FUNCTION and CREATE PROCEDURE statements to register UDRs. For details about SQLJ compliance, refer to "Complying with SQLJ" on page 4-27.

Tip: *Informix recommends that you place your SQL statements for registering UDRs written in Java in a deployment descriptor file.*

The following sections describe the Java-specific syntax of the CREATE FUNCTION and CREATE PROCEDURE statements that affects UDR registration. For information on the complete syntax of these SQL statements, see the *Informix Guide to SQL: Syntax*.

Specifying the JVP

To execute, a UDR written in Java must run in a Java virtual processor (JVP). The JVP is a predefined virtual-processor class that contains a Java Virtual Machine (JVM) to interpret Java byte codes. Use the following syntax to specify that a UDR should execute in the JVP class:

```
WITH (class='jvp')
```

By default, most UDRs run in the CPU VP, which does *not* contain a JVM. However, a UDR written in Java runs on a JVP by default. Therefore, the CLASS routine modifier is optional when you register a UDR written in Java. Informix recommends that, to improve readability of your SQL statements, you include the CLASS routine modifier when you register a UDR.

For example:

```
-- Specifying the JVP
CREATE PROCEDURE showusers()
    WITH (class='jvp')
    EXTERNAL NAME'thisjar:admin.showusers()'
    LANGUAGE java;
```

Using Routine Modifiers

The routine modifiers that you specify in the WITH clause of the CREATE FUNCTION or CREATE PROCEDURE statement tell the database server about attributes of the UDR. The database server supports the following routine modifiers for UDRs.

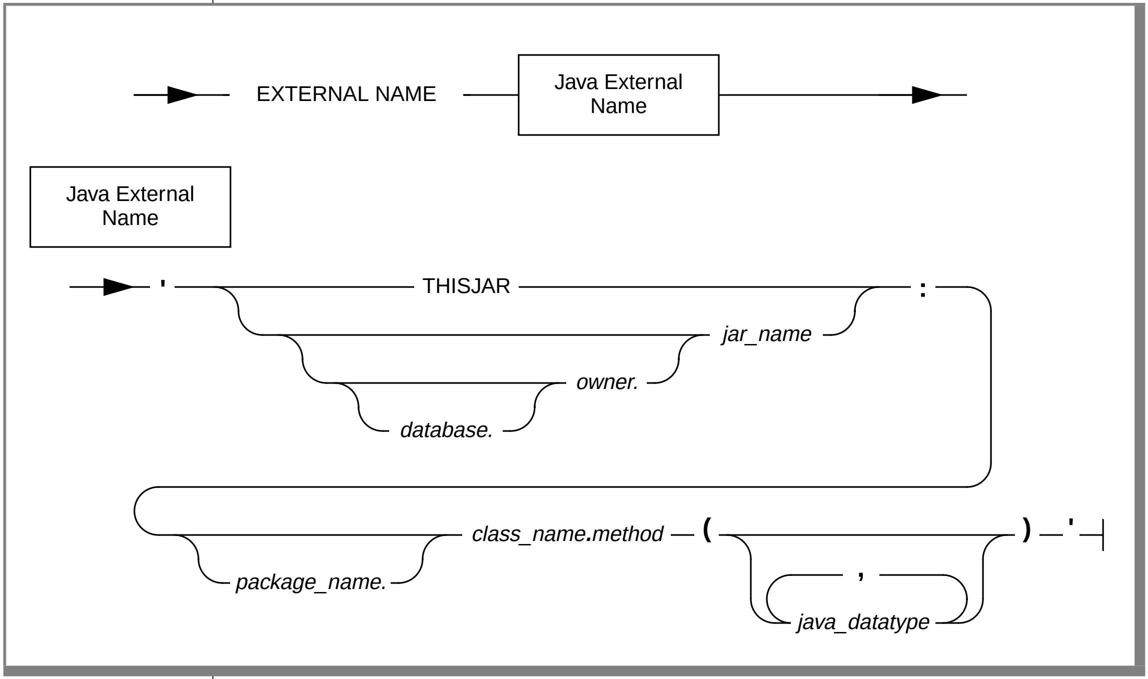
Routine Modifier	Type of UDR
CLASS	Accesses to the JVP
HANDLESNULLS	Handles SQL null values as arguments
ITERATOR	Iterator function
NEGATOR	Negator function
NOT VARIANT	Might return cached results
PARALLELIZABLE	Parallelizable UDR
VARIANT	Returns different results when invoked with the same arguments

The following routine modifiers are C-language specific and do not apply to UDRs in Java:

- COSTFUNC
- INTERNAL
- SELFUNC
- STACK
- PERCALL_COST
- SELCOST

Specifying the External Name

The following diagram details the external-name portion of the CREATE ROUTINE (or FUNCTION or PROCEDURE) statement for a UDR written in Java.



Element	Purpose	Restrictions
class_name	Class to which the UDR belongs	Must be an existing class
database	Database where the JAR exists If omitted, defaults to the current database.	Must be an existing database
jar_name	Jar identifier as specified in the **install_jar()** statement	Must be an existing JAR name
java_datatype	Name of a Java data type. The second column of the following table shows data types and class names that you can use for this variable.	Must be a Java data type

(1 of 2)

Element	Purpose	Restrictions
method	Name of the static method of the UDR	Must be an existing method
owner	Owner of the JAR If omitted, defaults to the current user.	Must be an existing user name
package_name	Name of a package	Required if the UDR classes are in a package

(2 of 2)

When used within a deployment descriptor, the THISJAR keyword automatically expands to the SQLJ-defined three-part JAR path.

The following table shows mapping between SQL data values and Java types. Use the values in the second column for the *java_datatype* variable.

SQL Data Type	Java Type
CHAR(1)	char
CHAR(1)	java.lang.Character
CHAR()	Java.lang.String
CHARACTER()	java.lang.String
CHARACTER VARYING()	java.lang.String
VARCHAR	java.lang.String
LVARCHAR	java.lang.String
SMALLINT	short
SMALLINT	java.lang.Short
INTEGER	int
INTEGER	java.lang.Integer
INT8	long
INT8	java.lang.Long
SMALLFLOAT	float
SMALLFLOAT	java.lang.Float

(1 of 2)

SQL Data Type	Java Type
REAL	float
REAL	java.lang.Float
FLOAT	double
FLOAT	java.lang.Double
DOUBLE PRECISION	double
DOUBLE PRECISION	java.lang.Double
DECIMAL	java.math.BigDecimal
MONEY	java.math.BigDecimal
NUMERIC	java.math.BigDecimal
BOOLEAN	boolean
BOOLEAN	java.lang.Boolean
DATE	java.sql.Date
DATETIME HOUR TO SECOND	java.sql.Time
DATETIME YEAR TO FRACTION	java.sql.Timestamp
INTERVAL	java.lang.String
BLOB	java.sql.Blob
CLOB	java.sql.Clob

(2 of 2)

Using a Deployment Descriptor

A *deployment descriptor* allows you to include the SQL statements for creating and dropping the UDRs in a JAR file. Both **sqlj.install_jar()** and **sqlj.remove_jar()** take a parameter that, when set appropriately, causes the procedure to search for deployment descriptor files in the JAR file.

You can include the following SQL statements in a deployment descriptor:

- CREATE FUNCTION
- CREATE PROCEDURE
- GRANT
- DROP FUNCTION
- DROP PROCEDURE

When you execute **sqlj.install_jar()** or **sqlj.remove_jar()**, the database server automatically performs the actions described by any deployment-descriptor files that exist in the JAR file.

Warning: *The transaction handling of the current database controls the SQL statements that the deployment descriptor executes. Informix recommends that you use a BEGIN WORK statement to begin a transaction before executing the* ***sqlj.install_jar()*** *or* ***sqlj.remove_jar()*** *procedure. In this way, a successful deployment can be committed, while a failed deployment can be rolled back.*

For example, you might prepare a file, **deploy.txt**, that includes the following statements:

```
SQLActions[] = {
"BEGIN INSTALL
    CREATE PROCEDURE showusers()
        WITH (class='jvp')
        EXTERNAL NAME'thisjar:admin.showusers()'
        LANGUAGE JAVA;
    GRANT EXECUTE ON PROCEDURE showusers() to informix;
END INSTALL",

"BEGIN REMOVE
    DROP PROCEDURE showusers();
END REMOVE"
}
```

For details on deployment-descriptor files, refer to the SQLJ: SQL Routines specification.

Using a Manifest File

The *manifest file* specifies the names of the deployment descriptor files that a JAR file contains. The **m** option of the **jar** command incorporates the manifest file into the default manifest of the JAR.

The following example shows the manifest file, **manifest.txt**, for a JAR with two deployment descriptors:

```
Name: deploy1.txt
SQLJDeploymentDescriptor: TRUE

Name: deploy2.txt
SQLJDeploymentDescriptor: TRUE
```

The following example shows the **jar** command that incorporates **manifest.txt** into a JAR file:

```
jar cvmf manifest.txt admin.jar deploy*.txt *.class
```

Compiling the Java Code

A UDR written in Java is implemented by a static method in a Java class.

To make the Java source code into an executable format

1. Compile the **java** files with the **javac** command to create class files.
2. Use the **jar** command to collect a set of class files into a JAR file.

   For example:

   ```
   # makefile for admin class
   JAR_NAME = admin.jar
   all:
       javac *.java
       jar cvmf manifest.txt $(JAR_NAME)
          deploy.txt *.class
       mv $(JAR_NAME) $(INFORMIXDIR)/jars
   cleanup:
       rm -f *.class $(INFORMIXDIR)/jars/$(JAR_NAME)
   ```

JAR files contain Java classes that in turn contain static methods corresponding to SQL UDRs. JAR files can also contain auxiliary classes and methods that are used by the UDRs (for example, to perform SQL-to-Java type mapping).

Installing a Jar File

JAR files contain the code for the UDRs. For an SQL statement to be able to include a UDR written in Java, you must install the JAR file in the current database. Once a JAR file is installed, the routine manager of the database server can load the appropriate Java class when the UDR is invoked.

Use the EXECUTE PROCEDURE statement with the following SQLJ built-in procedures to manage JAR files:

- **sqlj.install_jar(jar_url varchar(255), jar_id varchar(255), deploy_flag int)**

  Before a Java static method can be mapped to a UDR, the class file that defines the method must be installed in the database. The **install_jar()** procedure installs a Java jar file in the current database and assigns it a *jar identifier* (or *jar id*) for use in subsequent CREATE FUNCTION or CREATE PROCEDURE statements.

  For example:

  ```
  -- Installing a jar file
  EXECUTE PROCEDURE sqlj.install_jar
   ('file:$INFORMIXDIR/jars/admin.jar',
    'admin_jar', 1);
  ```

- **sqlj.replace_jar(jar_url varchar(255), jar_id varchar(255))**

  The replace_jar() procedure replaces a previously installed JAR file with a new version.

- **sqlj.remove_jar(jar_id varchar(2550, undeploy_flag int)**

  The remove_jar() procedure removes a previously installed JAR file from the current database.

- **sqlj.alter_java_path(jar_id varchar(255), path lvachar)**

  The alter_java_path() procedure specifies the *java-file search path* to use when the routine manager resolves related Java classes for the JAR file of a UDR.

For details about JAR-naming conventions, refer to the SQLJ: SQL Routines specification.

All SQLJ built-in procedures reside in the **sqlj** schema.

Both **sqlj.install_jar()** and **sqlj.remove_jar()** take a parameter that, when set appropriately, causes the procedure to execute the deployment descriptor files in the JAR file.

For more information about installing JAR files, refer to the SQLJ: SQL Routines section of the documentation on the following Web site:

```
http://www.sqlj.org/
```

The SQLJ: SQL Routines specification has detailed tutorials on writing, registering, installing, and calling routines written in Java.

Executing a UDR

After you register a UDR as an external routine in the database, the UDR can be invoked in SQL statements such as:

- in the select list of a SELECT statement.
- in the WHERE clause of a SELECT, UPDATE, or DELETE statement.
- with the EXECUTE PROCEDURE or EXECUTE FUNCTION statement.

The routine manager of the database server handles the execution of the UDR. For more information about the routine manager, see *Extending Informix Dynamic Server 2000*.

Tracing and Debugging

As with a UDR written in C, a UDR written in Java might generate the SQL messages for UDR and DataBlade API errors when it executes. UDRs written in Java adopt the JDBC error-reporting mechanism as well. The UDR throws an **SQLException** in case of an execution error such as a failed JDBC call. The routine manager detects such exceptions and translates it into a normal UDR error message.

In addition, the UDR can generate Java trace outputs and stack dumps at runtime. These additional Java messages are written to the *JVP log file*. The JVP log file is separate from the main database server log file, **online.log**. No JVP-specific messages appear in the database log. The JVP log file is intended to be the main destination for logging and tracing messages that are specific to the JVP and the UDR. This log is essential to support and debugging efforts. Informix encourages you to preserve it when possible.

Generating Log Messages

Log messages in the JVP log file can originate from any of the following sources:

- The JVP

  JVP messages report such conditions as:

  - JVP status (such as boot progress)
  - Warnings about missing or limited resources
  - Execution errors (such as being unable to locate a UDR)
  - Internal errors (such as unexpected exceptions)

  JVP log messages that report serious errors usually print a Java-method stack trace.

- The UDR

  Log messages from the UDR are messages that make sense only in the JVP and Java domain or that can complement the messages from SQL or the database server with annotations and references that are specific to Java or the JVP.

  You can use the following methods to write messages to the JVP log file from within a UDR:

  - **UDRLog.log()**
  - **UDRTraceable.tracePrint()**

Do *not* use the JVP log for error messages that need to be reported to the client application or to the main **online.log** file. Instead, the method should throw an **SQLException**.

Setting the JVP Log File

By default, the JVP uses the following log file:

```
JVPHOME/jvp.log
```

You can change this default log file with the JVPLOGFILE parameter in the ONCONFIG configuration file. Set this configuration parameter to the name of the log file that you want the JVP to use. For example, the following line sets the log file to **/usr/jvp.log**:

```
JVPLOGFILE /usr/jvp.log
```

Using the Administrative Tool

The Informix JDBC Driver includes a built-in iterative UDR that is a limited administrative tool:

```
informix.jvpcontrol (command lvarchar) returns lvarchar
```

The *command* can be one of the following forms, where *vpid* is the virtual processor ID:

- **threads *vpid***
- **memory *vpid***

The database server enables the **informix.jvpcontrol()** UDR when the JVPPROPFILE configuration parameter specifies a starting port number by using the **JVP.monitor.port** entry.

You can use the **onstat -g glo** command to list the **vpid** numbers.

The threads vpid Option

The **threads *vpid*** form lists the threads running on the Java VP whose ID is **vpid**. For example, if *command* is `threads 4`, the UDR might return the following output:

```
(expression) Thread[informix.jvp.dbapplet.impl.JVPControl#0,
9,informix.jvp.dbapplet.impl.JVPControl#0],UDR=JVPControlUDR
(java.lang.String), state = EXECUTE
(expression) Thread[JVP control monitor thread,10,main]
(expression) Thread[main,10,main]
(expression) Thread[SIGQUIT handler,0,system]
(expression) Thread[Finalizer thread,1,system]
5 row(s) retrieved.
```

The memory vpid Option

The **memory *vpid*** form lists memory use on the Java VP whose ID is **vpid**. For example, if *command* is `memory 4`, the UDR might return the following output:

```
(expression) Memory 16521840 bytes free, 16777208 bytes total
1 row(s) retrieved.
```

Debugging a UDR Written in Java

To debug a UDR written in Java, you can connect the Java debugger, **jdb**, to the embedded Java VM for debugging. The agent password required by **jdb** will be printed in the message log.

Traceable Events

The database server provides a fixed set of system trace events such as UDR sequence initialization, activation, and shutdown. You can also generate application-specific traces. For more information, refer to "com.informix.udr.UDRTraceable" on page 4-11.

Finding Information about UDRs

The system catalog tables contain information about UDRs. The LANGUAGE clause of the CREATE FUNCTION or CREATE PROCEDURE statement tells the database server in which language the UDR is written. For UDRs in Java, the LANGUAGE clause must be as follows:

```
LANGUAGE JAVA
```

The language specified (JAVA) is an entry in the **sysroutinelangs** system catalog table. The database server stores valid UDR languages in the **sysroutinelangs** table. The information includes an integer, the *language identifier,* in the **langid** column:

The following lines show the entry in the **sysroutinelangs** system catalog table for the Java language:

```
langid        3
langname      java
langinitfunc  udrlm_java_init
langpath      $INFORMIXDIR/extend/krakatoa/lmjava.so
langclass     jvp
```

The Java language has the same default privilege as the C language. The following entry in the **syslangauth** system catalog table specifies the privileges for Java:

```
grantor       informix
grantee       DBA
langid        3
langauth      u
```

By default, both user **informix** and the owner of the database are allowed to create UDRs in Java. If you attempt to execute the CREATE FUNCTION or CREATE PROCEDURE statement as some other user, the database server generates an error.

To allow other users to register UDRs in the database, user **informix** can grant the usage privilege on the Java language with the GRANT statement. The following GRANT statement allows any user who has Resource privileges on the database to register UDRs written in Java:

```
GRANT USAGE ON LANGUAGE JAVA TO public
```

For more information on the syntax of the GRANT statement, see the *Informix Guide to SQL: Syntax*.

Complying with SQLJ

The syntax of UDRs written in the Java supported by Informix usually follows the SQLJ specification. Where syntactic differences and missing features occur, the differences are mostly due to differences between Informix SQL and the SQL 3 standards. The following table summarizes the level of SQLJ compliance.

Feature (SQLJ Section #)	Function	Syntax	Definition and Rules	Comments
Jar names (3.1)	Yes	Yes	Yes	
Java path (3.2)	Yes	Yes	Yes	
Install, replace, or remove jars (4.1-4.3)	Yes	Yes	Yes (required) No (optional)	No support of the optional replacement jar validation rules.
Alter java path (4.4)	Yes	Yes	Yes	
Create procedure, Create function (5.1)	Yes	Yes	Yes (required) No (optional)	No support of the optional create time jar validation and the Java main method.
	For information about modifiers for Create Procedure and Create Function, refer to "Unsupported Modifiers" on page 4-28 and "Unsupported Optional Modifiers" on page 4-29			
Drop procedure, Drop function (5.2)	Yes	Yes	Yes	
Grant or revoke jar (5.3-5.4, optional)	No	No	No	

(1 of 2)

Feature (SQLJ Section #)	Function	Syntax	Definition and Rules	Comments
SQLJ function call (5.5)	Yes	Yes	Yes	
SQLJ procedure call (5.6)	Yes	Yes	Yes	
System properties and default connections	No	No	No	
Deployment-descriptor files (optional)	Yes	No	No	
Status codes, exception handling (7.1-7.2)	Yes	Yes	Yes	

(2 of 2)

Unsupported Modifiers

Some modifiers for CREATE PROCEDURE and CREATE FUNCTION are not supported in this version of the database server. Informix UDRs do not support the following routine modifiers of the SQLJ specification.

Modifier	How to handle the modifier
Read sql data	No Informix equivalent
Contains SQL	No Informix equivalent
Modifies SQL data	No Informix equivalent
No sql	No Informix equivalent
Return null on null input	Informix default for external routines
Call on null input	Use the Informix modifier HANDLESNULLS
Deterministic	Use the Informix modifier NOT VARIANT

(1 of 2)

Modifier	How to handle the modifier
Nondeterministic	Use the Informix modifier VARIANT
Returns Java data type in Java method signature	No Informix equivalent
In parameter	Informix default; no need to specify the modifier

(2 of 2)

Unsupported Optional Modifiers

Informix UDRs do not support the following optional routine modifiers of the SQLJ specification:

- Dynamic result sets
- Inout parameter
- Output parameters in callable statements

Chapter

The Informix JDBC Driver

5

In This Chapter

All UDRs written in Java can access the database server data through the JDBC application programming interface (API). This chapter briefly describes the Informix implementation of the JDBC API and the server-side Informix JDBC driver.

This chapter describes the public JDBC interfaces and JDBC subprotocols that server applications can use. For information about client-side applications, refer to the *Informix JDBC Driver Programmer's Guide*.

Public JDBC Interfaces

Java in the server defines the following public interfaces:

- **com.informix.jdbc.IfxConnection**
- **com.informix.jdbc.IfxProtocol**

The client and server JDBC drivers each have their own implementation of the preceding interfaces. The client driver provides access to databases from Java applications. The server driver provides database access from within the server through UDRs written in Java.

com.informix.jdbc.IfxConnection

The **IfxConnection** interface is a subinterface of **java.sql.Connection** with Informix-specific methods added. The **com.informix.jdbc.IfxDirectConnection** class implements the **com.informix.jdbc.IfxConnection** interface. This interface provides a connection to the current database server from within a UDR. The connection corresponds to a server-query context and is passed to the UDR by the SQLJ language manager. The transaction context of this connection is that of the query issuing the UDR call, and the call to create a UDR connection does not specify any database or user information.

com.informix.jdbc.IfxProtocol

The **IfxProtocol** interface represents the protocol and data exchange between the client application and an Informix database server. It sends and processes the messages and data flow between the client and database server. The **com.informix.jdbc.IfxDirectProtocol** class implements the **IfxProtocol** interface. It uses the Datablade API (DAPI) to access database resources.

The informix-direct Subprotocol

The JDBC **DriverManager** class provides services to connect to JDBC drivers. It assists in loading and initializing a requested JDBC driver. A UDR written in Java uses the **registerDriver()** method of **DriverManager** to register itself and to redirect user messages to the **DriverManager** logging facility.

A UDR written in Java or a Java client application that wants to connect to the database calls the **DriverManager.getConnection()** method to obtain a connection handle. This method takes a URL string as an argument. The JDBC management layer attempts to locate a driver that can connect to the database that the URL represents. To perform this task, the JDBC management layer asks each driver in turn if it can connect to the specified URL. Each driver examines the URL and determines if it supports the specified JDBC subprotocol. The Informix implementation of UDRs written in Java supports the **informix-direct** subprotocol in the database server.

For the **informix-direct** subprotocol, the JDBC driver loads and uses the following classes:

- The *connection class,* which you can specify with the ConnectionClass property. The connection class must implement **IfxConnection**.
- The *protocol class,* which you can specify with the ProtocolClass property. This protocol class must implement **IfxProtocol**.

These specifiers are optional in the URL string. If you do not specify ConnectionClass or ProtocolClass, the Informix JDBC driver can determine them from the subprotocol.

The following call opens a UDR connection with the class **IfxDirectConnection**. It uses the **IfxDirectProtocol** as the protocol for processing queries on the current database.

```
DriverManager.getconnection("jdbc:informix-direct:"+
"//ConnectionClass="com.informix.jdbc.IfxDirectConnection;"+
"//ProtocolClass=com.informix.jdbc.IfxDirectProtocol");
```

The UDR connection can only be opened by the thread that executes the UDR static method. In this way, the database server can ensure that the proper transaction context is used for the UDR.

JDBC 1.0 API

The JDBC 1.0 API consists of the following Java classes and interfaces that you can use to open connections to particular databases, execute SQL statements, and process the results.

Classes	Interfaces
java.sql.DataTruncation	java.sql.CallableStatement
java.sql.Date	java.sql.Connection
java.sql.DriverManager	java.sql.DatabaseMetaData
java.sql.DriverPropertyInfo	java.sql.Driver
java.sql.SQLException	java.sql.PreparedStatement

(1 of 2)

Classes	Interfaces
java.sql.SQLWarning	java.sql.ResultSet
java.sql.Time	java.sql.ResultSetMetaData
java.sql.Timestamp	java.sql.Statement
java.sql.Types	

(2 of 2)

The following JDBC 1.0 classes and interfaces are the most important for the development of UDRs in Java:

- **java.sql.DriverManager** handles loading of drivers and provides support for creating new database connections.
- **java.sql.Connection** represents a connection to a particular database.
- **java.sql.Statement** acts as a container for executing an SQL statement on a given connection.
- **java.sql.ResultSet** controls access to the row results of a given statement.
- **java.sql.PreparedStatement** handles execution of a pre-compiled SQL statement.
- **java.sql.CallableStatement** handles execution of a call to a database SPL routine.

For more documentation, refer to the Javasoft Web site at:

```
http://java.sun.com
```

Java 2.0

JDBC 2.0 is a major leap from JDBC 1.0 in that it supports extensible data types and large objects. Informix provides the following extensions to JDBC 1.0 to support user-defined types with JDK 1.1.x:

- **java.sql.Blob**
- **java.sql.Clob**

- **java.sql.SQLData**
- **java.sql.SQLInput**

  The following read / write methods are not supported for opaque types:

  - **readString()**

    Use the Informix extension **readString(len)**.
  - **readInterval()**
  - **readBytes()**

    Use the Informix extension **readBytes(len)**.
  - **readCharacterStream()**
  - **readAsciiStream()**
  - **readBinaryStream()**
  - **readObject()**
  - **readRef()**
  - **readArray()**
- **java.sql.SQLOutput**

  The following read / write methods are not supported for opaque types:

  - **writeString()**

    Use the Informix extension **writeString(len)**.
  - **writeInterval()**
  - **writeBytes()**

    Use the Informix extension **writeBytes(len)**.
  - **writeCharacterStream()**
  - **writeAsciiStream()**
  - **writeBinaryStream()**
  - **writeObject()**
  - **writeRef()**
  - **writeArray()**

Interfaces Updated for Java 2.0

The Informix implementation of UDRs written in Java also defines the following public interfaces:

- **com.informix.PreparedStatement2**

  This class includes the JDBC 2.0 methods **setBlob()** and **setClob()**.
- **com.informix.ResultSet2**

  This class includes the JDBC 2.0 methods **getBlob()** and **getClob()**.
- **com.informix.Types2**

  This class includes the type codes for the smart-large-object data types, BLOB and CLOB.

Support for Opaque Data Types

Certain JDBC 2.0 interfaces need to be extended to support opaque data types. Some of the methods need an additional length argument to read or write an opaque data type because the JDBC driver cannot look inside an opaque data type to determine the field lengths.

The Informix implementation of UDRs written in Java provides the following extensions of the JDBC user-defined-type (UDT) support:

- **java.sql.SQLUDTInput**
- **java.sql.SQLUDTOutput**

java.sql.SQLUDTInput

This class extends java.sql.SQLInput with the following methods:

```
public String readString(int maxlen) throws SQLException;
public byte[] readBytes(int maxlen) throws SQLException;
```

java.sql.SQLUDTOutput

This class extends **java.sql.SQLOutput** with the following methods:

```
public void writeString(String str, int maxlen) throws
SQLException;
public void writeBytes(byte[] b, int maxlen) throws
SQLException;
```

An Example That Shows Query Results

The following example implements a procedure called **showusers()**, which runs a query, retrieves all rows from the returned result, and prints the rows in the JVP log file:

```
import com.informix.udr.*;
import java.sql.*;

public class admin
{
    public static void showusers() throws SQLException
    {
        UDREnv env = UDRManager.getUDREnv();
        UDRLog log = env.getLog();
        String name = env.getName();

        Connection conn = DriverManager.getConnection
            ("jdbc:informix-direct:" +
                "//ConnectionClass=" +
                "com.informix.jdbc.IfxDirectConnection;" +
                "//ProtocolClass=" +
                "com.informix.jdbc.IfxDirectProtocol");
        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery
            ("SELECT * FROM Users");
        log.log("User information:");

        while ( rs.next() )
        {
            String UID = rs.getString(1);
            String Password = rs.getString(2);
            String Last = rs.getString(3);
            String First = rs.getString(4);

            // Write out the UDR name followed by the
            // columns values
            String line = name + " : " +
                UID + " " + Password + " " + Last + " " + First;
            log.log(line);
        }
        stmt.close();
        conn.close();
    }
}
```

Once you have created and installed the jar file that contains this Java method, the next task is to register the **showusers()** method as a UDR by giving it an SQL procedure signature. For the CREATE PROCEDURE statement that registers **showusers()**, see "Specifying the JVP" on page 4-14.

The syntax for invoking a UDR written in Java is no different from a standard UDR call, as follows:

```
EXECUTE PROCEDURE showusers()
```

Index