Informix Product Family
Informix
Version 12.10

J/Foundation Developer's Guide

IBM

SC27-4528-00

Informix Product Family
Informix
Version 12.10

J/Foundation Developer's Guide

SC27-4528-00

Note

Before using this information and the product it supports, read the information in “Notices” on page B-1.

Contents

Introduction

This introduction provides an overview of the information in this publication and describes the conventions it uses.

About this publication

This publication describes how to write user-defined routines (UDRs) in the Java™ programming language for IBM® Informix® Dynamic Server with J/Foundation. The publication also describes the infrastructure that enables you to run Java applications in the database server. The publication describes the Java classes, methods, and interfaces that allow you to access databases from within IBM Informix Dynamic Server with J/Foundation, rather than from a client application.

Types of users

This publication is written for the following users:
- Database-application programmers
- DataBlade® module developers
- Java UDR developers
- Java server application developers

This publication 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 Development Kit
- Java Database Connectivity (JDBC), which is a Java application programming interface to SQL databases
- SQLJ: SQL Routines specification, which specifies the Java binding of SQL UDRs

Software compatibility

This publication assumes that you are using the following software:
- IBM Informix Dynamic Server with J/Foundation, Version 12.10

  J/Foundation includes Version 5.0 of the Java Runtime Environment (JRE) and uses it to execute your server-based Java routines. This specific version of the JRE ensures that the Java environment is known and reliable for this database server release.
- Java Development Kit (JDK), Version 5.0

  You need JDK to compile your Java programs.
- DataBlade Developers Kit (DBDK) for Java, Version 4.0 or greater

  You need DBDK only for DataBlade module development.

Assumptions about your locale

IBM Informix products can support many languages, cultures, and code sets. All the information related to character set, collation and representation of numeric data, currency, date, and time that is used by a language within a given territory and encoding is brought together in a single environment, called a Global Language Support (GLS) locale.

The IBM Informix OLE DB Provider follows the ISO string formats for date, time, and money, as defined by the Microsoft OLE DB standards. You can override that default by setting an Informix environment variable or registry entry, such as **DBDATE**.

If you use Simple Network Management Protocol (SNMP) in your Informix environment, note that the protocols (SNMPv1 and SNMPv2) recognize only English code sets. For more information, see the topic about GLS and SNMP in the *IBM Informix SNMP Subagent Guide*.

The examples in this publication are written with the assumption that you are using one of these locales: en_us.8859-1 (ISO 8859-1) on UNIX platforms or en_us.1252 (Microsoft 1252) in Windows environments. These locales support U.S. English format conventions for displaying and entering date, time, number, and currency values. They also support the ISO 8859-1 code set (on UNIX and Linux) or the Microsoft 1252 code set (on Windows), which includes the ASCII code set plus many 8-bit characters such as é, è, and ñ.

You can specify another locale if you plan to use characters from other locales in your data or your SQL identifiers, or if you want to conform to other collation rules for character data.

For instructions about how to specify locales, additional syntax, and other considerations related to GLS locales, see the *IBM Informix GLS User's Guide*.

Demonstration databases

The DB-Access utility, which is provided with your IBM 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 IBM Informix publications 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 *IBM Informix DB-Access User's Guide*. For descriptions of the databases and their contents, see the *IBM Informix Guide to SQL: Reference*.

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

Example code conventions

Examples of SQL code occur throughout this publication. Except as noted, the code is not specific to any single IBM 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 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. If you are using DB–Access, you must delimit multiple statements with semicolons.

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 documentation for your product.

Additional documentation

Documentation about this release of IBM Informix products is available in various formats.

You can access Informix technical information such as information centers, technotes, white papers, and IBM Redbooks® publications online at http://www.ibm.com/software/data/sw-library/.

Compliance with industry standards

IBM Informix products are compliant with various standards.

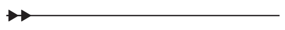
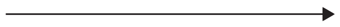
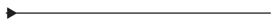
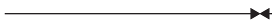
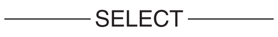
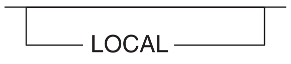
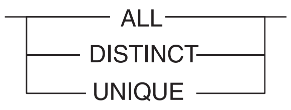
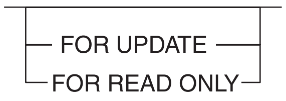
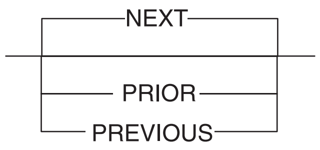
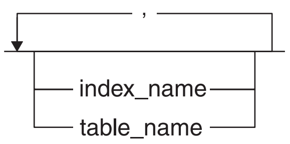
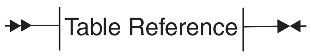
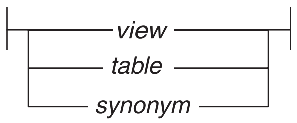
IBM Informix SQL-based products are fully compliant with SQL-92 Entry Level (published as ANSI X3.135-1992), which is identical to ISO 9075:1992. In addition, many features of IBM Informix database servers comply with the SQL-92 Intermediate and Full Level and X/Open SQL Common Applications Environment (CAE) standards.

The IBM Informix Geodetic DataBlade Module supports a subset of the data types from the *Spatial Data Transfer Standard (SDTS)—Federal Information Processing Standard 173*, as referenced by the document *Content Standard for Geospatial Metadata*, Federal Geographic Data Committee, June 8, 1994 (FGDC Metadata Standard).

Syntax diagrams

Syntax diagrams use special components to describe the syntax for statements and commands.

Table 1. Syntax Diagram Components

Component represented in PDF	Component represented in HTML	Meaning
	`>>----------------------`	Statement begins.
	`----------------------->`	Statement continues on next line.
	`>-----------------------`	Statement continues from previous line.
	`-----------------------><`	Statement ends.
	`--------SELECT----------`	Required item.
	`--+----------------+---` `'------LOCAL------'`	Optional item.
	`---+-----ALL-------+---` `+--DISTINCT-----+` `'---UNIQUE------'`	Required item with choice. Only one item must be present.
	`---+------------------+---` `+--FOR UPDATE-----+` `'--FOR READ ONLY--'`	Optional items with choice are shown below the main line, one of which you might specify.
	`.---NEXT---------.` `----+----------------+---` `+---PRIOR--------+` `'---PREVIOUS-----'`	The values below the main line are optional, one of which you might specify. If you do not specify an item, the value above the line is used by default.
	`.-------,-----------.` `V                   |` `---+----------------+---` `+---index_name---+` `'---table_name---'`	Optional items. Several items are allowed; a comma must precede each repetition.
	`>>-| Table Reference |-><`	Reference to a syntax segment.
Table Reference 	`Table Reference` `|--+-----view--------+--|` `+------table------+` `'----synonym------'`	Syntax segment.

How to read a command-line syntax diagram

Command-line syntax diagrams use similar elements to those of other syntax diagrams.

Some of the elements are listed in the table in Syntax Diagrams.

Creating a no-conversion job

```
►►──onpladm create job──job──┬─────────────────┬── -n── -d──device── -D──database────►
                             └─ -p──project──┘

►─ -t──table───────────────────────────────────────────────────────────────────────────►

   ┌───────────────────────────────────────────────────────────────────┐
   ▼                                                                   │
►──┴─┬───────────────────────────────────────────────────────────────┬─┴────────────►◄
     │                                                     (1)       │
     └─┬──────────────┬─┬──────────────┬─┤ Setting the Run Mode ├───┘
       └─ -S──server──┘ └─ -T──target──┘
```

Notes:

1 See page Z-1

This diagram has a segment named "Setting the Run Mode," which according to the diagram footnote is on page Z-1. If this was an actual cross-reference, you would find this segment on the first page of Appendix Z. Instead, this segment is shown in the following segment diagram. Notice that the diagram uses segment start and end components.

Setting the run mode:

```
                    ┌─l─┬─────┬─┐
                    │   └─c─┘ │
├── -f──┬─────┬─────┴─u───────┴───┬─────┬──┬─────┬──────────────────────────────────────┤
        ├─d──┤                    └─n─┘  └─N─┘
        ├─p──┤
        └─a──┘
```

To see how to construct a command correctly, start at the upper left of the main diagram. Follow the diagram to the right, including the elements that you want. The elements in this diagram are case-sensitive because they illustrate utility syntax. Other types of syntax, such as SQL, are not case-sensitive.

The Creating a No-Conversion Job diagram illustrates the following steps:

1. Type **onpladm create job** and then the name of the job.
2. Optionally, type **-p** and then the name of the project.
3. Type the following required elements:
   - **-n**
   - **-d** and the name of the device
   - **-D** and the name of the database
   - **-t** and the name of the table

4. Optionally, you can choose one or more of the following elements and repeat them an arbitrary number of times:
   - **-S** and the server name
   - **-T** and the target server name
   - The run mode. To set the run mode, follow the Setting the Run Mode segment diagram to type **-f**, optionally type **d**, **p**, or **a**, and then optionally type **l** or **u**.
5. Follow the diagram to the terminator.

Keywords and punctuation

Keywords are words reserved for statements and all commands except system-level commands.

When a keyword appears in a syntax diagram, it is shown in uppercase letters. When you use a keyword in a command, you can write it in uppercase or lowercase letters, but you must spell the keyword exactly as it appears in the syntax diagram.

You must also use any punctuation in your statements and commands exactly as shown in the syntax diagrams.

Identifiers and names

Variables serve as placeholders for identifiers and names in the syntax diagrams and examples.

You can replace a variable with an arbitrary name, identifier, or literal, depending on the context. Variables are also used to represent complex syntax elements that are expanded in additional syntax diagrams. When a variable appears in a syntax diagram, an example, or text, it is shown in *lowercase italic*.

The following syntax diagram uses variables to illustrate the general form of a simple SELECT statement.

```
►►──SELECT──column_name──FROM──table_name──────────────────────────────────────►◄
```

When you write a SELECT statement of this form, you replace the variables *column_name* and *table_name* with the name of a specific column and table.

How to provide documentation feedback

You are encouraged to send your comments about IBM Informix user documentation.

Use one of the following methods:
- Send email to docinf@us.ibm.com.
- In the Informix information center, which is available online at http://www.ibm.com/software/data/sw-library/, open the topic that you want to comment on. Click the feedback link at the bottom of the page, fill out the form, and submit your feedback.

- Add comments to topics directly in the information center and read comments that were added by other users. Share information about the product documentation, participate in discussions with other users, rate topics, and more!

Feedback from all methods is monitored by the team that maintains the user documentation. The feedback methods are reserved for reporting errors and omissions in the documentation. For immediate help with a technical problem, contact IBM Technical Support at http://www.ibm.com/planetwide/.

We appreciate your suggestions.

Chapter 1. Concepts

This section introduces the infrastructure for creating and executing user-defined routines (UDRs) and applications that you write in Java to run in the IBM Informix database server.

This section provides the following information:
- Basic characteristics of Java UDRs
- Basic architecture for executing Java UDRs in the database server
- Impact of Java UDRs on the database server system catalog tables

For general information about the purpose and the process of developing UDRs for the database server, see the *IBM Informix User-Defined Routines and Data Types Developer's Guide*. For information about how to access databases from Java UDRs, see the *IBM Informix JDBC Driver Programmer's Guide*.

Features of Java user-defined routines

The IBM Informix database server provides the infrastructure to support Java UDRs. The database server binds SQL UDR signatures to Java executable routines and provides mapping between SQL data values and Java objects so that the database server can pass parameters and retrieve returned results.

The IBM Informix database server also provides support for data type extensibility and sophisticated error handling.

Java virtual processors

Java UDRs execute on specialized virtual processors called Java virtual processors (JVPs). A Java Virtual Machine (JVM) is embedded in the code of each JVP.

The JVPs are responsible for executing all server-based Java UDRs and applications. Although the JVPs are used for Java-related computation, they have the same capabilities as a CPU VP, and they can process all types of SQL queries. This eliminates the need to ship 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 database server scheduler multiplexes IBM 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 server uses 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 make parallel calls to UDRs written in Java code. Because the JVMs embedded in different VPs do not share states, you cannot store global states by 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.

The consistency of multiple JVMs is not an issue on the Windows platform because all VPs are mapped to kernel threads instead of processes. Because 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 code.

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

The **syslangauth** table specifies who is allowed to use the language. For Java code, the default is the database administrator. For information about how to modify the usage privileges, see the GRANT statement in the *IBM Informix Guide to SQL: Syntax*.

The **sysprocedures** table gives information about both built-in routines and routines that you define.

For more information about these system catalog tables, see “Find information about user-defined routines” on page 4-16 and the *IBM Informix Guide to SQL: Reference*.

Chapter 2. Prepare for Java support

This section describes how to install and configure the database server to provide UDRs written in Java code.

To create and use UDRs written in Java code, you must install the following software:

- IBM Informix Dynamic Server with J/Foundation
- The Java Development Kit (JDK), Version 5.0

If you do not plan to develop Java UDRs, you do not need to install the JDK. J/Foundation includes a tested version of the Java Runtime Environment (JRE) to execute Java UDRs. You need to install the JDK only if you need to compile new Java source code. For more information about where to obtain the JDK, see the machine notes for your platform.

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

For more detailed information about the required software, see the release notes.

Install the JDBC Driver

J/Foundation includes the IBM Informix JDBC Driver. The IBM Informix JDBC Driver contains Java classes and shared-object files that allow you to write UDRs in Java code. The installation procedure installs these binary files in `$INFORMIXDIR/extend/krakatoa`.

For more information, see the machine notes file.

Configure Java support

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

Preparing to use Java code 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 need to include this path in several places as you prepare J/Foundation.

Create an sbspace

The database server stores Java JAR files as smart large objects in the system default sbspace. If you do not already have a default sbspace, you must create one.

For example, the following command creates an sbspace called **mysbspace**:

```
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 parameter in the `onconfig` file 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 JAR files that you plan to install.

Related reference:

The onspaces utility (Administrator's Reference)

"SBSPACENAME configuration parameter" on page 3-4

Creating the JVP properties file

A JVP properties file contains property settings that control various runtime behaviors of the Java virtual processor.

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 JVP properties file:

1. Copy the JVP 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 in "The com.informix.udr.UDRTraceable" on page 4-6.

Set configuration parameters

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

- JVPPROPFILE

- JVMTHREAD
- JVPCLASSPATH
- JVPLOGFILE

The following example shows sample settings for the Java-related configuration parameters on a UNIX Solaris system.

```
JVPLOGFILE    jvphome/jvp.log
JVPPROPFILE   jvphome/.jvpprops
#VPCLASS       jvp,num=1
JVMTHREAD     native
JVPCLASSPATH  jvphome/krakatoa.jar:jvphome/jdbc.jar
```

In this example, JVPCLASSPATH shows the default setting. To run in debug mode, add the **_g** suffix as shown in the following example:

```
JVPCLASSPATH  jvphome/krakatoa_g.jar:jvphome/jdbc_g.jar
```

For information about specific configuration parameter settings on your platform, see `$INFORMIXDIR/etc/onconfig.std`.

Related reference:

Chapter 3, “Configuration parameters,” on page 3-1

Set environment variables

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

The following sections describe the runtime environment variables that you can set.

JVM_MAX_HEAP_SIZE environment variable

Set the environment variable **`JVM_MAX_HEAP_SIZE`** to configure the heap size for the JVM. The default heap size is 16 MB. You can set this variable to the maximum heap size needed for the JVM, depending on the estimated requirements of the application.

JAR_TEMP_PATH environment variable

Set the **`JAR_TEMP_PATH`** environment variable to specify a local file system location where jar management procedures such as **install_jar** and **replace_jar** can store JAR files temporarily. This directory must have read and write permissions for the user who starts the database server. If the **`JAR_TEMP_PATH`** environment variable is not set, temporary copies of JAR files are stored in the `/tmp` directory of the local file system for the database server.

JAVA_COMPILER environment variable

To turn off just-in-time (JIT) compilation, set the **`JAVA_COMPILER`** environment variable to `NONE` or `none`. For more information about JIT compilation, see the Java documentation.

GLS support

When the database server starts a UDR, the routine runs in the locale that DB_LOCALE specifies. Consequently, the database server automatically converts parameters, return values, and output values between the DB_LOCALE code set and the Unicode code set so that Java code can use the values.

However, when a Java UDR creates a JDBC connection to the database server for access through SQL, you can set DB_LOCALE into the connection URL to control conversions and formatting between the Unicode code set and the code set of the database server locale. This setting of DB_LOCALE overrides any environment settings. In fact, DB_LOCALE does not need to be set in the environment. Similarly, you can also set **DBDATE**, **GL_DATE**, and **DBCENTURY** into the URL connection to control date conversion and formatting.

For example, when a UDR sends string or date data to the database server in an insert, the database server converts the data from Unicode to the locale that DB_LOCALE specifies, or it interprets dates and intervals by using your **DBDATE** or **GL_DATE** setting.

When the database server returns data to the Java UDR, the database server does the opposite conversion, so Java code sees only Unicode.

You can extend the Java Runtime Environment (JRE) that is distributed with server with your own CharsetProvider, and Charset classes. With the Java Runtime, you can extend the default list of character sets by creating a JAR file that contains your own CharsetProvider and Charset class implementations. To extend the JRE, copy the custom CharsetProvider JAR file into the `$INFORMIXDIR/extend/krakatoa/jre/lib/ext` directory. For more information about how to build a custom CharsetProvider JAR file, see that Java API documentation for CharsetProvider.

Important: This method must only be used for situations where the server supports a character set that does not have an equivalent (does not exist) within the JRE that is distributed with server.

NEWLOCALE and NEWCODESET connection properties

IBM Informix JDBC Driver uses the JDK globalization API to manipulate international data. The classes and methods in this API take a JDK locale or encoding as a parameter. Because the Informix DB_LOCALE and CLIENT_LOCALE properties specify the locale and code set based on Informix names, these Informix names are mapped to the JDK names. For example, the Informix name for the ASCII code set is 8859-1 and the JDK name for the ASCII code set is 8859_1. Informix JDBC Driver internally maps 8859-1 to 8859_1 and uses the appropriate JDK name in the JDK classes and methods.

Two new connection properties, NEWLOCALE and NEWCODESET, enable you to specify a locale or code set that is not yet mapped in the internal tables of the JDBC driver.

The NEWLOCALE and NEWCODESET properties have the following formats:

```
NEWLOCALE=<JDK locale>,<Ifx locale>:<JDK locale>,<Ifx locale>...
NEWCODESET=<JDK encoding>,<Ifx codeset name>,
<Ifx codeset number>:<JDK encoding>,<Ifx codeset name>,<Ifx codeset number>...
```

The following example shows a URL that uses these properties. (You must specify a valid URL on a single line.)

```
jdbc:informix-sqli://myhost:1533:informixserver=myserver;user=myname;
password=mypasswd;NEWLOCALE=en_us,en_us;NEWCODESET=8859_1,8859-1,819;
```

There is no limit to the number of locale or code-set mappings that you can specify. If you specify an incorrect number of parameters or values, you get a message that says, "Locale Not Supported" or "Encoding or Code Set Not Supported." If you set these properties in the URL or in an **IfmxDataSource** object, the new values in NEWLOCALE and NEWCODESET override the values in the JDBC internal tables. For example, if JDBC already maps 8859-1 to 8859_1, but you specify NEWCODESET=8888,8859-1,819, the new value, 8888, is used for the code-set conversion.

DBCENTURY environment variable

If a String represents a DATE or a DATETIME value that has less than a three-digit year value, the IBM Informix JDBC Driver uses the **DBCENTURY** environment variable to determine the correct four-digit year and performs a String-to-DATE or -DATETIME conversion.

The following table summarizes the affected methods and the conditions under which they are affected.

Table 2-1. Summary of affected methods and conditions

Method	Condition
IfxPreparedStatement.setString(String)	The target column is SQLDATE or SQLDTIME.
IfxPreparedStatement.setObject(String)	The target column is SQLDATE or SQLDTIME.
IfxPreparedStatement.IfxSetObject(String)	The target column is SQLDATE or SQLDTIME.
IfxResultSet.getDate()	The source column is a String type.
IfxResultSet.getTimestamp	The source column is a String type.
IfxResultSet.updateString(String)	The target column is SQLDATE or SQLDTIME.
IfxResultSet.updateObject(String)	The target column is SQLDATE or SQLDTIME.

The following example shows a URL that uses the **DBCENTURY** environment variable:

```
jdbc:informix-sqli://myhost:1533:informixserver=myserver;user=myname;password=
mypasswd;DBCENTURY=F;
```

You must specify a valid URL on a single line.

Related reference:

DBCENTURY environment variable (SQL Reference)

Chapter 3. Configuration parameters

This section documents the configuration parameters that you need to set to use UDRs written in Java code. Set these parameters in the database server configuration file (the `onconfig` file).

For a sample environment that configuration parameters establish, see the release notes file.

Related reference:

"Set configuration parameters" on page 2-2

JVPARGS configuration parameter

`onconfig.std` *value*
: None

takes effect
: When shared memory is initialized

The JVPARGS configuration parameter provides an easy way for you to set Java VM options.Use a semicolon to separate options. For example, if you want to change `Xms` and `Xmx` to `32m`, you can set those options with the JVPARGS parameter, as the following example shows:

```
JVPARGS -Xms32m;-Xmx32m
```

If you want to see `gc` information to determine whether you need to increase `ms` or `mx`, you can set JVPARGS, as the following example shows:

```
JVPARGS -verbose:gc
```

For more information about Java VM options, see your Java documentation.

JVPCLASSPATH configuration parameter

`onconfig.std` *value*
: `/usr/informix/extend/krakatoa/krakatoa.jar:`

  `/usr/informix/extend/krakatoa/jdbc.jar`

takes effect
: When shared memory is initialized

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

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

If you require the debug versions of the JAR files, use the following JVPCLASSPATH setting:

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

The total number of characters available for specifying configuration values in the `onconfig` file is 256. The database server imposes this limit.

To specify more than 256 characters for the value of the JVPCLASSPATH parameter, you can store the value in a file and specify the keyword `file:` on the parameter, followed by the file name. For example, if you set the path in a file called `classpath_fl` in the directory `/u/informix/iif2000/extend/java`, you can specify the JVPCLASSPATH parameter, as the following example shows:

```
JVPCLASSPATH file:/u/informix/iif2000/extend/java/classpath_fl
```

You must specify the complete value for JVPCLASSPATH on one line in the file, just as you would normally on the configuration parameter. Do not include the parameter name JVPCLASSPATH again. The database server considers the first carriage return in the line to be the terminating carriage return for the path name.

The JVPCLASSPATH parameter is required if the number of JVPs (set in VPCLASS JVP parameter) 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 configuration parameter

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 IBM Informix JDBC Driver are installed. To modify the default setting in the configuration file, replace `/usr/informix` with the path name 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 configuration parameter

onconfig.std *value*
`$INFORMIXDIR/extend/krakatoa/jre/`

takes effect
When shared memory is initialized

The JVPJAVAHOME configuration parameter specifies the directory where the JRE for the database server is installed. The database server includes a tested version of the JRE. The default location for the JRE is in `$INFORMIXDIR/extend/krakatoa/jre/`. To modify the default setting in the configuration file, replace `$INFORMIXDIR/extend/krakatoa/jre/` with the path name setting of **$INFORMIXDIR**, followed by `/extend/krakatoa/jre/`.

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

If you want to use a stand-alone JVM, without a JVP, install the JDK on your platform and use the JVM that is included.

JVPJAVALIB configuration parameter

`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 JVM libraries.

The value of this parameter is platform dependent. To find the proper value for JVPJAVALIB, see the machine and release notes.

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

JVPJAVAVM configuration parameter

`onconfig.std` *value*
: platform-specific value

separators
: colon (UNIX) or semicolon (Windows)

takes effect
: When shared memory is initialized

The JVPJAVAVM configuration parameter lists the JVM libraries that the database server loads. The names in this list exclude the **lib** prefix and `.so` or `.dll` suffix. Entries in the list are separated by colons or semicolons, depending on the operating system. This parameter is required if the number of JVPs (set in VPCLASS JVP) is greater than 0.

Depending on your application requirements, test and possibly increase the value for the **`JVM_MAX_HEAP_SIZE`** environment variable to configure the heap size for the new JVM.

The value of JVPJAVAVM is platform-dependent. To find the proper value for JVPJAVAVM, see `$INFORMIXDIR/etc/onconfig.std`.

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

If you use a nondebug JDBC driver, you can use the nondebug JDK libraries for better performance. Set JVPJAVAVM as follows: `server`

For Windows, use the following value for JVPJAVAVM if you are using a debug version of the JDBC driver: `jvm_g`

If you use a nondebug JDBC driver, you can use the nondebug JDK libraries for better performance. Set JVPJAVAVM as follows: `jvm`

JVPLOGFILE configuration parameter

`onconfig.std` *value*
: `/usr/informix/jvp.log`

range of values
: Any valid complete file name

takes effect
: When shared memory is initialized

The database server can generate Java trace outputs and stack dumps. The database server writes this output to the Java VP log file.

The JVPLOGFILE configuration parameter specifies the path to the Java VP log file. This parameter is optional.

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

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

If you do not specify a value for this parameter, the default value is derived from the `onconfig.std` file. If the JVPLOGFILE parameter is not present in the `onconfig` file, the default file location is `./jvp.log`, where '.' is the current directory of the user who runs **oninit**.

JVPPROPFILE configuration parameter

`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 configuration parameter

`onconfig.std` *value*
: blank

takes effect
: When shared memory is initialized

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

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

Tip: When you use UDRs written in Java code, create separate sbspaces for storing your smart large objects.

Related reference:

SBSPACENAME configuration parameter (Administrator's Reference)

"Create an sbspace" on page 2-2

CREATE TABLE statement (SQL Syntax)

VPCLASS configuration parameter

`onconfig.std` *value*
: set

range of values
: 0 and positive integers

takes effect
: When shared memory is initialized

The VPCLASS 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 to execute Java UDRs.

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, or if there is no VPCLASS parameter for the JVP class, execution of Java UDRs 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.

Related reference:

VPCLASS configuration parameter (Administrator's Reference)

Chapter 4. Create Java user-defined routines

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

This section provides the following information about UDRs written in Java code:
- What tasks a UDR can perform
- How to create a UDR

Java user-defined routines

The behaviors of installing and invoking UDRs written in Java code follow the SQLJ: SQL Routines specification. Every UDR written in Java code maps to an external Java static method whose class is in a Java Archive (JAR) file that was installed in a database. The SQL-to-Java data type mapping is done according to the IBM Informix JDBC Driver 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, see the *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

UDRs written in Java code can perform the following tasks.

Table 4-1. Tasks that can be performed by UDRs written in Java

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 PROCEDURE 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.)

Table 4-1. Tasks that can be performed by UDRs written in Java (continued)

Type of UDR	Purpose
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 code are supported by using some IBM 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
Negator function	A function that calculates the not operation for a particular operator or function

You cannot use UDRs written in Java code for any of the following features:
- Commutator functions
- Cost functions
- Operator-class functions
- Selectivity functions
- User-defined statistics functions

Limitations for Java UDRs

Java UDRs have the following limitations in IBM Informix:
- Return arrays for the **executeBatch()** method are not supported when using the direct connect method (**IfxDirectConnection**) for Java UDRs. Only the update count for the first statement executed in the batch is returned.
- BYTE OUT parameters for Java UDRs are not supported.
- A BYTE value cannot be retrieved from a BYTE column in Java UDRs.

Creating a Java user-defined routine

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

To create a Java UDR:
1. Write the UDR, which can use the IBM Informix JDBC Driver methods to interact with the database server.
2. If the UDR uses any user-defined data types (UDTs), for each UDT write a Java class that translates between the database server and Java representation of the type.

   This class implements the **SQLData** interface. For information about **SQLData**, see the Informix JDBC Driver 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 about how to develop a UDR, see the *IBM Informix User-Defined Routines and Data Types Developer's Guide*. The following sections briefly describe each of these steps in the development of a UDR.

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

Write a Java user-defined routine

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

- Java packages

  UDRs can use all the basic nongraphic 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 about these packages, see the JDK documentation.
- Java Database Connectivity (IBM Informix JDBC Driver) 1.0 API

  UDRs can use the Informix JDBC Driver 1.0 API to access the database. For more information, see "JDBC 1.0 API" on page 5-2.

  The `$INFORMIXDIR/extend/krakatoa/examples.tar` file of online examples includes a sample of Informix JDBC Driver in a UDR in `Informix JDBC Driver.java`.
- Informix JDBC Driver extensions

  UDRs can also use Informix extensions to Informix JDBC Driver 1.0 to access some Informix JDBC Driver 2.0 functionality. For more information, see Chapter 5, "The Informix JDBC Driver," on page 5-1.
- Informix extensions for UDRs written in Java code

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

The Informix `com.informix.udr` package provides extensions to SQLJ that allow applications to use the capabilities of Informix. Such extensions include logging, tracing, iterator support, and invocation-state management.

The com.informix.udr package

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

- The **com.informix.udr.UDRManager**

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

The following sections describe each of these extensions, which are specific to IBM Informix.

The 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 a 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 that represents 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.

The 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 the database.

The **getParamTypeName()** method returns the SQL data type names for the UDR arguments and **getReturnTypeName()** method returns the SQL data type names for the return value.

If you are using IBM Informix JDBC Driver 2.0, use the **getUDRs()** method of the **java.sql.DatabaseMetaData** class to obtain more information about a data type.

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 an Informix JDBC Driver 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 the 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.

The **getTraceable()** method returns a **UDRTraceable** interface for the UDRs to use.

Related reference:

"The com.informix.udr.UDRLog"

"The com.informix.udr.UDRTraceable" on page 4-6

The 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.

Related reference:

"The com.informix.udr.UDREnv" on page 4-4

"Generate log messages" on page 4-14

The 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 relationships, 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.

Table 4-2. Predefined trace levels for the traceLevel() method

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.

Related reference:

"The com.informix.udr.UDREnv" on page 4-4

"Generate log messages" on page 4-14

"Traceable events" on page 4-15

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 IBM Informix JDBC Driver specification.

For built-in SQL data types, the routine manager can use mappings to existing Informix JDBC Driver data types.

For any UDTs that your UDR uses, you must create mappings. You can use the following UDTs in UDRs written in Java code.

User-defined data type	SQL statement
Distinct data type	CREATE DISTINCT TYPE
Opaque data type	CREATE OPAQUE TYPE

Restriction: You cannot use row or collection data types in UDRs written in Java code.

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, see the Informix JDBC Driver 2.0 specification.
2. Bind this user-defined class to the user-defined SQL data type by using the **setUDTExtName** built-in procedure.

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

   - **sqlj.setUDTExtName()**

     This procedure defines the mapping between a UDT 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 code, `Circle.java`.

Registering Java user-defined routines

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.

Tip: Place your SQL statements for registering UDRs written in Java code in a deployment descriptor file.

The following sections describe the Java-specific syntax of the CREATE FUNCTION and CREATE PROCEDURE statements that affect UDR registration.

Related reference:

"Comply with SQLJ" on page 4-16

CREATE FUNCTION statement (SQL Syntax)

CREATE PROCEDURE statement (SQL Syntax)

Specify the JVP

To execute, a UDR written in Java code must run in a JVP. The JVP is a predefined virtual-processor class that contains a JVM to interpret Java byte codes.

Use the following syntax to specify that a UDR executes 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 code runs on a JVP by default. Therefore, the CLASS routine modifier is optional when you register a UDR written in Java code. To improve readability of your SQL statements, 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;
```

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.

Table 4-3. Routine modifiers for UDRs

Routine modifier	Type of UDR
CLASS	Accesses to the JVP
HANDLESNULLS	Handles SQL null values as arguments
ITERATORS	Iterator function
NEGATOR	Negator function
NOT VARIANT	Might return cached results
PARALLELIZABLE	Parallelizable UDR

Table 4-3. Routine modifiers for UDRs (continued)

Routine modifier	Type of 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 code:

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

Specify 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 code.

```
►►──EXTERNAL NAME──┤ Java external name ├────────────────────────────────────────►◄
```

Java external name:

```
├──'──┬─THISJAR──────────────────────────────────┬──:──┬─────────────────┬──class_name.method─────────►
      └─┬─────────────────────────────┬─jar_name─┘     └─package_name──.─┘
        └─┬───────────────┬─owner──.─┘
          └─database──.─┘

         ┌─,──────────────┐
         ▼                │
►──(─────┬──────────────┬─┴──)──'──────────────────────────────────────────────────┤
         └─java_datatype─┘
```

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.
method	Name of the static method of the UDR	Must be an existing method.
owner	Owner of the jar. If omitted, default is the current user.	Must be an existing user name.

Element	Purpose	Restrictions
package_name	Name of a package	Required if the UDR classes are in a package.

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.

Table 4-4. Mapping between SQL data values and Java types

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
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

A deployment descriptor

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

Important: The transaction handling of the current database controls the SQL statements that the deployment descriptor executes. Use a BEGIN WORK statement to begin a transaction before you execute 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.

Related reference:

sqlj.remove_jar (SQL Syntax)

sqlj.install_jar (SQL Syntax)

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 code 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).

Install 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 code, you must install the JAR file in the current database. When a JAR file is installed, the routine manager of the database server can load the appropriate Java class when the UDR is invoked.

To install a JAR file on IBM Informix, the JAR file must be have READ permissions for user **informix**.

If the IFX_EXTEND_ROLE configuration parameter is set to '`On`' or `1`, authorization to use the built-in routines that manipulate shared objects is available only to the Database Server Administrator, and to users to whom the DBSA has granted the EXTEND role. For IBM Informix 10.00.xC4 and later releases, IFX_EXTEND_ROLE is enabled by default.

For databases in which this security feature is not needed, see the description of IFX_EXTEND_ROLE in your *IBM Informix Administrator's Reference* for information about how the DBSA can disable this configuration parameter by resetting it.

To manage JAR files, use the EXECUTE PROCEDURE statement with the following SQLJ built-in procedures:

- **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(255), 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 lvarchar)**

  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, see the SQLJ: SQL Routines specification.

All SQLJ built-in procedures are 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 how to install JAR files, see the SQLJ: SQL Routines specification.

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

Related reference:

sqlj.remove_jar (SQL Syntax)

sqlj.install_jar (SQL Syntax)

Update JAR file names

The script `update_jars.sql` is provided to update the three-part names of installed JAR files when you rename the database to which the JAR file belongs. You must execute the `update_jars.sql` script in the database after you rename it. You need to execute the `update_jars.sql` script only if you rename a database that has one or more installed JAR files.

Execute the user-defined routine

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.

Related concepts:

Routine management (UDR and Data Type Guide)

Debugging and tracing

As with a UDR written in C, a UDR written in Java code might generate the SQL messages for UDR and DataBlade API errors when it executes. UDRs written in Java code adopt the IBM Informix JDBC Driver error-reporting mechanism as well.

The UDR throws an **SQLException** in case of an execution error such as a failed Informix JDBC Driver 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 run time. 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 for IBM support to perform debugging efforts. You should preserve it when possible.

Generate 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 code or the JVP.

  Use the following methods to write messages to the JVP log file from within a UDR:
  - **UDRLog.log()**
  - **UDRTraceable.tracePrint()**

By default, the JVP uses the following log file:

```
/usr/informix/jvp.log
```

where '.' is the current directory of the user who runs **oninit**.

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
```

Restriction: 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 throws an **SQLException**.

Related reference:

"The com.informix.udr.UDRLog" on page 4-5

"The com.informix.udr.UDRTraceable" on page 4-6

The administrative tool

The IBM Informix JDBC Driver includes a built-in iterative UDR that is a limited administrative tool, **informix.jvpcontrol()**. 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 start **informix.jvpcontrol()** with the following syntax:

```
EXECUTE FUNCTION informix.jvpcontrol (command lvarchar);
```

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

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

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 Java user-defined routine

To debug a UDR written in Java code, you can connect the Java debugger, **jdb**, to the embedded JVM for debugging. The agent password that **jdb** requires is 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.

Related reference:

"The com.informix.udr.UDRTraceable" on page 4-6

Find information about user-defined routines

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 code, the LANGUAGE clause must be as follows:

```
LANGUAGE JAVA
```

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 the Java language:

```
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 code. 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 code:

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

Related reference:

GRANT statement (SQL Syntax)

Comply with SQLJ

The syntax of Java UDRs that the IBM Informix database server supports 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	

Feature (SQLJ section #)	Function	Syntax	Definition and rules	Comments
Install, replace, or remove JAR files (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)[1]	Yes	Yes	Yes (required) No (optional)	No support of the optional create time jar validation and the Java main method.
Drop procedure, Drop function (5.2)	Yes	Yes	Yes	
Grant or revoke jar (5.3-5.4, optional)	No	No	No	
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	

Note:

1. For information about modifiers for Create Procedure and Create Function, see "Unsupported modifiers" and "Unsupported optional modifiers" on page 4-18.

Related reference:

"Registering Java user-defined routines" on page 4-8

Unsupported modifiers

Some modifiers for CREATE PROCEDURE and CREATE FUNCTION are not supported in this version of the database server.

IBM Informix UDRs do not support the following routine modifiers of the SQLJ specification.

Table 4-5. Unsupported modifiers

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
Nondeterministic	Use the Informix modifier VARIANT

Table 4-5. Unsupported modifiers (continued)

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

Unsupported optional modifiers

IBM 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 5. The Informix JDBC Driver

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

Generally, the IBM Informix server-side Informix JDBC Driver derives from the client-side driver so that the two drivers are essentially the same. Java UDRs require some differences, however, to use the Informix JDBC Driver from the server side. This section describes the public Informix JDBC Driver interfaces and Informix JDBC Driver subprotocols that the Informix JDBC Driver provides specifically for server-side Informix JDBC Driver applications, and restrictions that apply to server-side Informix JDBC Driver applications. For principal documentation of the IBM Informix JDBC Driver, see the *IBM Informix JDBC Driver Programmer's Guide*.

Public JDBC interfaces

IBM Informix JDBC Driver defines the **com.informix.jdbc.IfxConnection** and **com.informix.jdbc.IfxProtocol** public interfaces.

The client and server drivers for Informix JDBC Driver 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 code.

The com.informix.jdbc.IfxConnection

The **IfxConnection** interface is a subinterface of **java.sql.Connection** with methods specific to Informix 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.

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

A UDR written in Java code 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 Informix JDBC Driver management layer attempts to locate a driver that can connect to the database that the URL represents. To perform this task, the Informix JDBC Driver 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 Informix JDBC Driver subprotocol. The Informix implementation of UDRs written in Java code supports the **informix-direct** subprotocol in the database server.

For the **informix-direct** subprotocol, the Informix 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.

Host a Java application server with Solano-style connections

IBM Informix can host a JAVA application server, such as an EJB container or custom HTTP server, within the database server by using a Solano-style database server connection string. For example:

```
jdbc:informix-direct:/stores_demo:user=user;password=passwd;
```

If you use a Solano-style connection string, a Java UDR can run in the database server and communicate with both the database server and its own client connections (for example by listening on a port and communicating with web browsers). This can lead to performance benefits and greatly extends the functionality of Informix.

JDBC 1.0 API

The IBM Informix JDBC Driver 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.

Table 5-1. Java classes and interfaces

Classes	Interfaces
java.sql.DataTruncation	java.sql.CallableStatement

Table 5-1. Java classes and interfaces (continued)

Classes	Interfaces
java.sql.Date	java.sql.Connection
java.sql.DriverManager	java.sql.DatabaseMetaData
java.sql.DriverPropertyInfo	java.sql.Driver
java.sql.SQLException	java.sql.PreparedStatement
java.sql.SQLWarning	java.sql.ResultSet
java.sql.Time	java.sql.ResultSetMetaData
java.sql.Timestamp	java.sql.Statement
java.sql.Types	None

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

- **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.

JDBC 2.0

IBM Informix JDBC Driver 2.0 is a major leap from Informix JDBC Driver 1.0 in that it supports extensible data types and large objects.

The following extensions to Informix JDBC Driver 1.0 are provided to support user-defined data types (UDTs) 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()**

Support for opaque data types

Certain IBM Informix JDBC Driver 2.0 interfaces must 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 Informix JDBC Driver cannot look inside an opaque data type to determine the field lengths.

The Informix implementation of UDRs written in Java code provides the following extensions of the Informix JDBC Driver 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;
```

Related reference:

Chapter 6, "Opaque user-defined data types," on page 6-1

Interfaces updated for Java 2.0

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

- **com.informix.PreparedStatement2**

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

  This class includes the Informix JDBC Driver 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.

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:");
      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();
   }
}
```

After you create and install 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 “Specify the JVP” on page 4-8.

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

```
EXECUTE PROCEDURE showusers()
```

Chapter 6. Opaque user-defined data types

This section describes how to use opaque user-defined data types (UDTs). This section describes the default **SQLData** interface and how to override the default.

It provides the following information:
- The **SQLData** Interface
- SQL statements to create default I/O routines
- IBM Informix extensions to SQLInput and SQLOutput interfaces
- How to override the default I/O methods
- Required I/O function sets and related data types
- Limitations to Streams

Related reference:
"Support for opaque data types" on page 5-4

Using the SQLData interface

To implement a complete UDT in Java code, you must supply a set of data-formatting methods that convert to and from the various representations of the data type. These methods perform input and output operations for the data type such as converting text input to the internal structure that the database server uses.

All the database server I/O functions manipulate data formats that can be represented as Java streams. The streams encapsulate the data and implement methods needed to parse the source format or write the destination format.

To implement an opaque UDT and use the default data-translation I/O methods:
1. Supply the IBM Informix JDBC Driver **SQLData** interface: **readSQL()**, **writeSQL()**, and **getSQLTypeName()** methods.
2. Create the SQL routine and cast definitions for the I/O functions by calling **sqlj.registerJUDTfuncs(varchar(255))**, where the **varchar** argument is the SQL name of the type you are registering.

   For example, after creating the UDT **Record3** with the following statements:
   ```
   create opaque type Record3 (internallength = variable,
      alignment = 8, maxlen = 2048, cannothash );
   grant usage on type Record3 to public;
   execute procedure setUDTExtName("Record3",
      "informix.testclasses.jlm.udt.Record3");
   ```
   You could create the default casts and I/O functions with the following statement:
   ```
   execute procedure registerJUDTfuncs("Record3");
   ```

The **readSQL()** method converts a database type to a Java object and the **writeSQL()** method converts a Java object to the database type. The system supplies the appropriate stream type at run time.

You can back out default I/O functions and casts by calling **sqlj.unregisterJUDTfuncs(varchar(255))**, where the **varchar** argument is the SQL name of the type, as the following example shows:

```
execute procedure unregisterJUDTfuncs("Record3");
```

Default input/output routines

Because this interface uses Java, all the SQL I/O support functions are predefined when you register the UDT. You only need to supply the required **SQLData** implementation.

IBM Informix supplies extensions to the Stream arguments of **SQLData** methods to suit various uses. With these extensions, you can build I/O functions for a new Java UDT. All that you must do to implement any of the required function sets is select the Stream type.

Informix also supplies default Input and Output processing methods in Java code that are used to implement all UDT I/O operations. The database server contains these default I/O methods and executes them just like any other Java UDR. These methods use information in the SQL UDR definition to select the correct Streams and instantiate the right user-defined type objects at execution time.

"The circle class example" on page 6-4 illustrates the use of the **SQLData** interface.

SQL definitions for default I/O user-defined routines

After you register the Java UDT with the database server by using the SQL procedure **setUDTExtName()**, you can create SQL functions and casts for it, using either the default I/O wrapper methods or explicit methods in your Java UDT class.

For the default I/O wrapper methods, the **registerJUDTfuncs** function creates the SQL functions shown in the following example, where **SQLType** is the SQL UDT name, **JavaType** is the JUDT name, and **SQLBuffer** is the SQL *transport type* being converted, that is, SENDRECV:

```
-- Receive function

CREATE IMPLICIT CAST (SENDRECV as SQLUDT with
   IfxJavaSENDRECVInJavaUDT);
CREATE FUNCTION IfxJavaSENDRECVInJavaUDT (in SENDRECV)
   RETURNS SQLUDT
EXTERNAL NAME
'com.informix.jdbc.IfxDataPointer.IfxDataInput(java.lang.Object)'
LANGUAGE java;
GRANT EXECUTE ON FUNCTION IfxJavaSENDRECVInJavaUDT TO PUBLIC;

-- Send function

CREATE EXPLICIT CAST (SQLUDT as SENDRECV with
   IfxJavaSENDRECVOutJavaUDT);
CREATE FUNCTION IfxJavaSENDRECVOutJavaUDT(out SQLUDT) RETURNS
   SENDRECV
EXTERNAL NAME 'com.informix.jdbc.IfxDataPointer.IfxDataOutput(java.sql.SQLData)'
LANGUAGE java NOT VARIANT;
GRANT EXECUTE ON IfxJavaSENDRECVOutJavaUDT TO PUBLIC;
```

The default Input method cannot be declared not variant because it might need to perform SQL queries to instantiate the correct Java UDT class.

Informix extensions to SQLInput and SQLOutput

Some of the standard **SQLInput** and **SQLOutput** Stream methods need an additional length argument to read or write an opaque data type because the IBM Informix JDBC Driver cannot determine the field lengths for an opaque type.

Informix database server provides the **IfmxUDTSQLInput** and **IfmxUDTSQLOutput** extensions, which inherit from the standard Informix JDBC Driver 2.0 **SQLInput** and **SQLOutput** interfaces.

IfmxUDTSQLInput

The **IfmxUDTSQLInput** interface extends **SQLInput**, which contains the following public methods:

```
String readString()
boolean readBoolean()
byte readByte()
short readShort()
int readInt()
long readLong()
float readFloat()
double readDouble()
java.math.BigDecimal readBigDecimal()
byte[] readBytes()
java.sql.Date readDate()
java.sql.Time readTime()
java.sql.Timestamp readTimestamp()
java.io.Reader readCharacterStream()
java.io.InputStream readAsciiStream()
java.io.InputStream readBinaryStream()
Object readObject()
Ref readRef()
Blob readBlob()
Clob readClob()
Array readArray()
boolean wasNull()
```

The **IfmxUDTSQLInput** interface adds the following IBM Informix methods:

```
String readString(int maxlen)
byte[] readBytes(int maxlen)
Interval readInterval()
int available();
int length();
IfxUDTInfo getUDTInfo(int xid)
IfxUDTInfo getUDTInfo(String name, String owner)
```

All the **readXXX()** methods throw an **SQLException** when they detect parsing errors. Use the **readXXX()** methods to convert the buffer of the given Input stream into a Java object. When the Input stream is empty, each read method throws an **SQLException** with **e.getErrorcode** equal to -79772 or **IfxErrMsg.S_BADSQLDATA**. However, you can use the **length()** and **available()** methods to determine when the Input stream is exhausted while converting variable length UDTs to Java objects.

IfmxUDTSQLOutput

The **IfmxUDTSQLOutput** interface extends **SQLOutput**, which contains the following public methods:

```
void writeString(String x)
void writeBoolean(boolean x)
void writeByte(byte x)
void writeShort(short x)
void writeInt(int x)
void writeLong(long x)
void writeFloat(float x)
void writeDouble(double x)
void writeBigDecimal(java.math.BigDecimal x)
void writeBytes(byte[] x)
void writeDate(java.sql.Date x)
void writeTime(java.sql.Time x)
void writeTimestamp(java.sql.Timestamp x)
void writeCharacterStream(java.io.Reader x)
void writeAsciiStream(java.io.InputStream x)
void writeBinaryStream(java.io.InputStream x)
void writeObject(SQLData x)
void writeRef(Ref x)
void writeBlob(Blob x)
void writeClob(Clob x)
void writeStruct(Struct x)
void writeArray(Array x)
```

The **IfmxUDTSQLOutput** interface adds the following IBM Informix methods:

```
void writeString(String x, int length)
void writeBytes(byte[] b, int length)
void writeInterval(Interval intrvl)
int available()
int length()
IfxUDTInfo getUDTInfo(int xid)
IfxUDTInfo getUDTInfo(String name, String owner)
```

All the **writeXXX()** methods throw an exception when they encounter conversion errors. Use the Stream **write()** methods to convert a Java object into the given Output buffer. The **length()** method returns the number of bytes that remain in the buffer. The Informix JDBC Driver 2.0 class files describe the **SQLOutput** definition.

The circle class example

The **circle** class example implements a fixed-length opaque data type.

The **circle** data type includes X and Y coordinates (**xCoord** and **yCoord**), which represent the center of the circle and a radius value (**radius**). The **readSQL** method reads the input stream **SQLInput** to obtain the **xCoord**, **yCoord**, and **radius** values and saves the data type name from String **typename**. The **writeSQL** method writes the **xCoord**, **yCoord**, and **radius** values to the stream **SQLOutput**.

```
package informix.testclasses.jlm;

import java.sql.*;

public class circle implements SQLData
{
public int xCoord;
public int yCoord;
public int radius;
private String type;

   public String getSQLTypeName()
   {
      return type;
   }

   public void readSQL (SQLInput stream, String typeName)
      throws SQLException
   {
      xCoord = stream.readInt();
      yCoord = stream.readInt();
      radius = stream.readInt();

      type = typeName;
   }

   public void writeSQL (SQLOutput stream)
      throws SQLException
   {
      stream.writeInt(xCoord);
      stream.writeInt(yCoord);
      stream.writeInt(radius);
   }
}
```

The **SQLData** methods use I/O streams to translate between C and Java representations. The following C-language structure shows the C definition for the circle:

```
typedef struct
{
   int x;
   int y;
   int radius;
} circle;
```

Override the default I/O methods

If the default methods are not sufficient because, for example, you want to include parentheses and other delimiting characters in the text representation, you can explicitly override the defaults with definitions of your own, after you register the Java UDT.

I/O function sets and related types

The following table specifies the I/O functions that you must implement for the nondefault case, and their related data types.

Table 6-1. Nondefault I/O functions and types table

Function set	Data format	SQL buffer type	Java buffer type	Java stream implementation
Server UDR	UDT	Internal Representation	IfxDataPointer	IfmxSQLInStream IfmxSQLOutStream
Input Output	Text	LVARCHAR	String (String Buffer)	IfmxTextInStream IfmxTextOutStream
Send Receive	Client Binary	SENDRECV	IfxDataPointer	IfmxSRInStream IfmxSROutStream
Import Export	Text	IMPEXP	IfxDataPointer	IfmxIEInsStream IfmxIEOutStream
Binary Import Export	Client Binary	IMPEXBIN	IfxDataPointer	IfmxIEBinStream IfmxIEBOutStream

The columns in the preceding table represent the following:

- Function set

  Names the type of function in conformance with UDT specifications
- Data format

  A conceptual description of the format of the data in the SQL buffer that is being converted
- Buffer type

  Names the actual data types being read or written
  - SQLBuffer is the SQL (or database-server) type for this data.
  - JavaBuffer is the Java type to which the SQLBuffer is transformed before being passed to (or returned from) the I/O method.

It is an intermediate type that is contained in and manipulated by a Java Stream. It is also the argument type for input methods and the return type for output methods.

- Java Stream implementation

  Names the actual stream type that is passed to the **SQLData** interface when the default I/O functions are used. Each of the streams implements **IfmxUDTSQLInput** or **IfmxUDTSQLOutput**.

IfxDataPointer

The **IfxDataPointer** class encapsulates the IBM Informix C-language representation of a type and its corresponding data buffer. This is usually a database server buffer structure, with a few attributes extracted for easy access in Java code.

This class is used to transport the nontextual SQL data types to and from the I/O methods and is generally managed by an **IfmxUDTSQLInput** or **IfmxUDTSQLOutput** stream.

Methods in both streams might throw an SQLException with the **e.getErrorcode** equal to -79700 or **IfxErrMsg.S_MTHNSUPP**, if they are not implemented. These methods are generally not needed on the database server side but are useful in the client Informix JDBC Driver code.

For more documentation of these streams, see the *IBM Informix JDBC Driver Programmer's Guide*. For an example of using these streams, see "Usage example" on page 6-8.

Stream implementations

This section briefly describes the Java classes that implement the **IfmxUDTSQLInput** and **IfmxUDTSQLOutput** interfaces.

IfmxSQLInStream and IfmxSQLOutStream

These streams convert to and from the internal data representation that the database server uses.

IfmxTextInStream and IfmxTextOutStream

These streams convert to and from a textual data representation for Input and Output functions. does not support cross-locale Input and Output routines; all strings are assumed to be in US English.

These streams delimit each component of the composite type with a white space between record elements. The SQL type is an LVARCHAR that contains client text. The JavaBuffer type for Input is String, which contains the client text. The JavaBuffer type for Output is a StringBuffer. The **read()** and **write()** methods must convert between the client text representation and the relevant Java object.

IfmxSRInStream and IfmxSROutStream

These streams convert to and from the binary data representation of the client for send and receive functions. The SQL type is SENDRECV, which is an internal representation that contains binary data in the client format. The JavaBuffer type is **IfxDataPointer**. The **read()** and **write()** methods convert between the client representation and the relevant Java object.

IfmxIEInStream and IfmxIEOutStream

This stream converts to and from a canonical text representation for import and export functions. The **SQLBuffer** is an IMPEXP type that is an internal representation that contains canonical textual data. The JavaBuffer type is **IfxDataPointer**. The **read()** and **write()** methods convert between the text representation and the relevant Java objects. These streams inherit from the **IfmxTextInStream** and **IfmxTextOutStream** classes.

IfmxIEBInStream

This stream converts to and from a canonical binary representation for binary import and export functions. The **SQLBuffer** is an IMPEXPBIN type that is an internal representation that contains canonical binary data. The JavaBuffer type is **IfxDataPointer**. The **read()** and **write()** methods must convert between the binary representation and the relevant Java objects. These streams inherit from the **IfmxSRInStream** and **IfmxSRIOutStream** classes.

Class layout (for input)

The following figure describes the class layout for input. The class layout for output is similar; simply replace In with Out in the names.

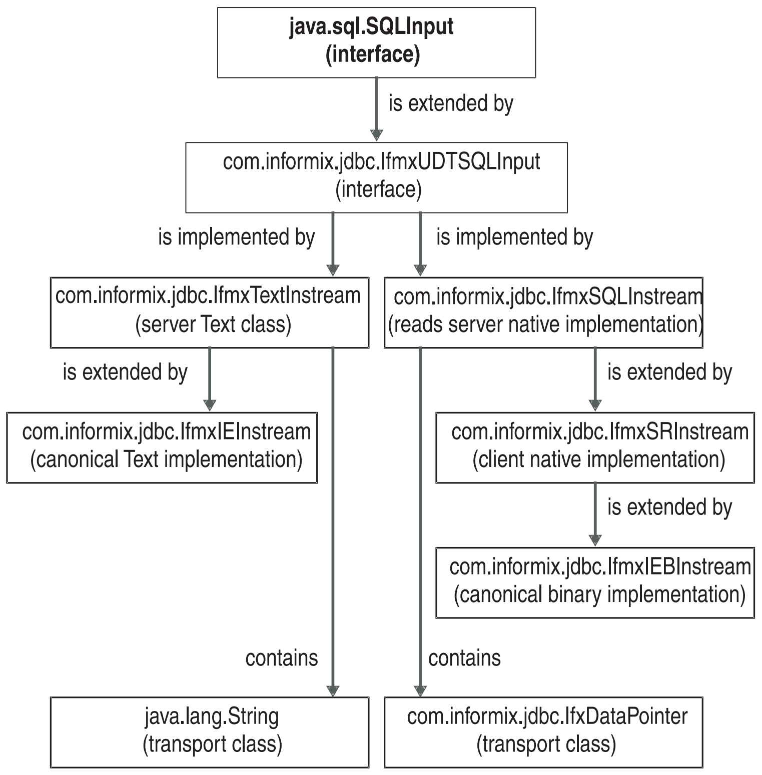

java.sql.SQLInput
(interface)
is extended by
com.informix.jdbc.IfmxUDTSQLInput
(interface)
is implemented by
is implemented by
com.informix.jdbc.IfmxTextInstream
(server Text class)
com.informix.jdbc.IfmxSQLInstream
(reads server native implementation)
is extended by
is extended by
com.informix.jdbc.IfmxIEInstream
(canonical Text implementation)
com.informix.jdbc.IfmxSRInstream
(client native implementation)
is extended by
com.informix.jdbc.IfmxIEBInstream
(canonical binary implementation)
contains
contains
java.lang.String
(transport class)
com.informix.jdbc.IfxDataPointer
(transport class)

Figure 6-1. .Input class layout

An example that overrides the default I/O methods

The following example illustrates a Java UDT class with nondefault definitions. **JavaType** is the new Java UDT, and **JavaBuffer** is the buffer type for the SQL data being converted, as "I/O function sets and related types" on page 6-5 shows. For a complete set of required and optional code, see "Usage example" on page 6-8.

```
public class JavaType implements SQLData
{
// Java data Object declarations for this Class....
// non-default Data Input function
   public static JavaType JavaTypeInput( JavaBuffer in )
   {
      JavaType x = new JavaType(); // make a new object
      // convert JavaBuffer fields to Java data objects in
      // this Class
      return( x );// return the new object
   }
   // non-default Data Output function
   public static JavaBuffer JavaTypeOutput( JavaType out )
   {
      JavaBuffer x = new JavaBuffer();
      // Do whatever it takes to translate object to output
      // buffer format
   return x;     // return the initialized buffer
   }
   // required SQLData implementation
   private String type;
   public String getSQLTypeName()
   {
      return type;
   }
   public void readSQL ( SQLInput instream, String typeName )
      throws SQLException
   {
      type = typeName;
      // cast up to Informix specific stream type
      IfmxUDTSQLInput in = (IfmxUDTSQLInput) instream;
      // read stream fields into Java data objects in this Class
      return;
   }

   public void writeSQL( SQLOutput outstream ) throws SQLException
   {
   // cast up to Informix specific stream type
      IfmxUDTSQLOutput out = (IfmxUDTSQLOutput) outstream;
   // write object to output stream
      return;
   }
}
```

For an example of the SQL definitions required to use the explicit methods in the preceding code, see "SQL definitions for a variable-length UDT example" on page 6-12.

Usage example

All Java UDT classes must implement the **readSQL()** and **writeSQL()** methods for the **SQLData** interface. The **readSQL()** method initializes a Java object by using data from the database server in a C-language format. The **writeSQL()** method converts a Java object back to the representation of the database server. The **readSQL()** and **writeSQL()** methods receive a Stream argument that encapsulates the conversion methods for each built-in type that the database server uses, for example, **int**, **float**, **decimal**.

For a fixed-length UDT, the **readSQL()** and **writeSQL()** methods know the order and number of fields they are to process. For a variable-length UDT, the programmer must rely on the **stream.available()** method and/or the **SQLException** to find the end of the data as this example shows.

Variable-length UDT including nondefault input and output methods:

```
/* Variable Length UDT example type: Record3
** Example of required and explicit method implementations.
**
**  The C language structure equivalent of this JUDT is:
**
** typedef struct
** {
**    mi_double_precision d;
**    mi_chara[4];
**    mi_integerb;
**    mi_realc;
**    mi_datee;
**    mi_smallintf;
**    mi_booleang[MAXBOOLS];
** } NewFixUDT;
**
**  Where the last boolean array can contain up to MAX values
**  but only valid values will be written to disk.
*/
// Put this in our test package,
//  could be anywhere but needs to match SQL definitons for UDRs.
package informix.testclasses.jlm.udt;
// get the usual suspect classes
import java.sql.*;
// get informix specific interfaces, etal.
import com.informix.jdbc.*;
// These are only needed for the non-default Input/Output
// functions, remove if you use defaults.
import informix.jvp.dbapplet.impl.IfmxTextInStream;
import informix.jvp.dbapplet.impl.IfmxTextOutStream;
/**************** Now here's our UDT *************/
public class Record3 implements SQLData
{
   // to turn debug print lines on and off
   private static boolean classDebug = true;

   // define storage for Java members of UDT
   private double d_double;
   private String a_char;
   private int b_int;
   private float c_float;
   private java.sql.Date e_date;
   private short f_smint;
   // could use a Vector for booleans, but would then need Boolean
   // objects ...so I've left it as an exercise for the reader...
   private static final int MAXBOOLS = 20;
   private boolean g_boolvals[] = new boolean[MAXBOOLS];
   private int numbools = 0;
   // dummy constructor just so we can log instantiation
   public Record3()
   {
      super();
      if( classDebug )
         System.out.println( "Record3() " + super.toString() + " created" );
   }
   // dummy finalizer just so we can log our own destruction
   protected void finalize()
   {
      super.finalize();
      if( classDebug )
         System.out.println( "Record3() " + super.toString() + " deleted" );
   }
/*********** REQUIRED SQLData implementation: ***********/
   // needed for SQLData interface
   private String type;
   public String getSQLTypeName()
   {
       return type;
   }
   // Called to convert an SQL buffer TYPE to JAVA class.
   //  note: we need to use SQLInput as the argument type or this
   // method signature won't resolve correctly.
   public void readSQL (SQLInput in, String typeName) throws
   SQLException
   {
      if( classDebug )
         System.out.println( "Record3.readSQL() entered" );
         // save the type name
      type = typeName;
      // cast the _real_ type of Stream for IFMX extensions.
      IfmxUDTSQLInput stream = (IfmxUDTSQLInput) in;
      // trap exceptions; don't really know how many bytes
      //  are in the input.
      try
      {
         d_double = stream.readDouble();
         a_char = stream.readString(4);
         b_int = stream.readInt();
         c_float = stream.readFloat();
         e_date = stream.readDate();
         f_smint = stream.readShort();
      // Read booleans until we get an exception:
      // converting a non-existant boolean will throw cookies.
      // but we can use available() to make sure there is more
      // to read...
         for( int count = 0; (stream.available() > 0) && (count
            < MAXBOOLS); ++count )
         {
            g_boolvals[count] = stream.readBoolean();
            ++numbools;
         }
      }
      catch (SQLException e)
      {
      // if we got something besides end of input rethrow,
      //  otherwise just assume we're done.
         if( e.getErrorCode() != IfxErrMsg.S_BADSQLDATA )
         {
            if( classDebug )
               System.out.println("Record3.readSQL() exception = " +
                                  e.toString());
            throw e;
         }
      }
   }

// Called to convert JAVA class to SQL buffer TYPE.
// note: we need to use SQLOutput as the argument type or this
// method signature won't resolve correctly.

   public void writeSQL( SQLOutput out ) throws SQLException
   {
      if( classDebug )
         System.out.println( "Record3.writeSQL() entered" );
   // cast up to _real_ type of Stream to use IFMX extensions.
      IfmxUDTSQLOutput stream = (IfmxUDTSQLOutput) out;
      stream.writeDouble(d_double);
      stream.writeString(a_char, 4);
      stream.writeInt(b_int);
      stream.writeFloat(c_float);
      stream.writeDate(e_date);
      stream.writeShort(f_smint);
      for( int i = 0; i < numbools; i++ )
         stream.writeBoolean(g_boolvals[i]);
```

```
   }
/*********** END SQLData implementation ***********/
/**** NON-DEFAULT implementation of Input and Output functions ****/
/* Remove all this if you only use the Defaults */
```

The following example illustrates the implementation of user-defined input and output functions that override the default I/O methods. If you use the default methods, you do not need to implement overriding methods like those that follow:

```
// Called as Input function to convert SQL lvarchar to JAVA class
public static Record3 fromString( String str )
{
   if( classDebug )
      System.out.println( "Record3.fromString(String) entered" );
   // Make a stream of the right kind.
   IfmxTextInStream stream = new IfmxTextInStream(str);
   // Make a new Java object of the right type.
   Record3 record = new Record3();
   // Just call readSQL ourselves.
   //  For a real implementation you would probably copy all the
   //   readXXX()'s and intersperse delimiting chars as needed...
   try
   {
      readSQL( stream, "Record3" );
   }
   catch (Exception e)
   {
      System.err.println(e.getMessage());
   }
   return record;
}

// Called as Output function; convert JAVA class to SQL lvarchar.
// note: could use toString() directly,
// except that the UDR method must be "static", and
// it needs to take a Record3 as an argument....

public static String makeString(Record3 x)
{
   if( classDebug )
      System.out.println( "Record3.makeString() entered" );
   return x.toString();
}

// Might as well implement the standard toString() as long as
//  we're doing non-defaults. If a different method name is
//  used here, Object.toString() will be called when the class
//  gets printed out in debug lines....

public String toString()
{
// Need to use a StringBuffer because we can't pass a
// reference to a String to be initialized.
// We could optimize by guessing at size of buffer, too.
// StringBuffer str = new StringBuffer();
// IfmxTextOutStream stream = new IfmxTextOutStream(str);
// Just call writeSQL.
// For a real implementation you would probably copy all the
// writeXXX()'s and intersperse delimiting chars as needed...
   try
   {
      writeSQL( stream );
   }
   catch (Exception e)
   {
      System.err.println(e.getMessage());
   // not sure if we need to clear out result string?
```

```
            str.setLength(0);
         }
         return str.toString();
      }
```

SQL definitions for a variable-length UDT example:
The SQL definitions for this example are:

```
-- VarLen UDT and support functions ----------------------------
create opaque type Record3 (internallength = variable,
   alignment = 8, maxlen = 2048, cannothash );
grant usage on type Record3 to public;
-- register JUDT implementation....
--  note package name needs to match class file package
execute procedure setUDTExtName("Record3",
   "informix.testclasses.jlm.udt.Record3");
-- Definitions for NON_DEFAULT Input/Output functions.
-- this overrides the defaults setup above
-- LVARCHAR INPUT
drop cast (Record3 as lvarchar);
create implicit cast (Record3 as lvarchar with record3_output);
create function record3_input (l lvarchar) returns Record3
   external name
'informix.testclasses.jlm.udt.Record3.fromString(java.lang.String)'
   language java not varient;
grant execute on function record3_input to public;
-- CHAR INPUT
drop cast (Record3 as char(100));
create implicit cast (Record3 as char(100) with record3_rout);
create function record3_rin (c char(100)) returns Record3
   external name
'informix.testclasses.jlm.udt.Record3.fromString(java.lang.String)'
   language java not varient;
grant execute on function record3_rin to public;

-- LVARCHAR OUTPUT
drop cast (lvarchar as Record3);
create explicit cast (lvarchar as Record3 with record3_input);
create function record3_output (c Record3) returns lvarchar
   external name
'informix.testclasses.jlm.udt.Record3.makeString(informix.testclasses.jlm.udt.Record3)'
   language java not varient;
grant execute on function record3_output to public;
-- CHAR OUTPUT
drop cast (char(100) as Record3);
create explicit cast (char(100) as Record3 with record3_rin);
create function record3_rout (c Record3) returns varchar(100) external name
'informix.testclasses.jlm.udt.Record3.makeString(informix.testclasses.jlm.udt.Record3)'
language java not varient;
grant execute on function record3_rout to public;

-- END definitions for NON_DEFAULT Input/Output functions.
-- end VarLen UDT and support functions --------------------------
-- Example Usage ---
create table rec3tab (record_col Record3);
insert into rec3tab values ('665.999 JAVA 398 197.236 1952-04-10 47 f t t');
insert into rec3tab values ('667.000 Jive 983 791.632 2002-04-11 42 f f f f f');
select * from rec3tab;
```

Limitations to streams

The following limitations apply to the I/O streams in :

- BLOBs and CLOBs are not supported.
- Text Input and Output across locales is not supported.
- Text Input and Output for intervals is not supported.

- Time stamps are only supported in their full format. Qualifiers are not supported.
- Byte arrays, **byte[ ]**, and Object/Stream I/O are not supported for either text or binary operations.

Appendix. Accessibility

IBM strives to provide products with usable access for everyone, regardless of age or ability.

Accessibility features for IBM Informix products

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use information technology products successfully.

Accessibility features

The following list includes the major accessibility features in IBM Informix products. These features support:

- Keyboard-only operation.
- Interfaces that are commonly used by screen readers.
- The attachment of alternative input and output devices.

Keyboard navigation

This product uses standard Microsoft Windows navigation keys.

Related accessibility information

IBM is committed to making our documentation accessible to persons with disabilities. Our publications are available in HTML format so that they can be accessed with assistive technology such as screen reader software.

IBM and accessibility

See the *IBM Accessibility Center* at http://www.ibm.com/able for more information about the IBM commitment to accessibility.

Dotted decimal syntax diagrams

The syntax diagrams in our publications are available in dotted decimal format, which is an accessible format that is available only if you are using a screen reader.

In dotted decimal format, each syntax element is written on a separate line. If two or more syntax elements are always present together (or always absent together), the elements can appear on the same line, because they can be considered as a single compound syntax element.

Each line starts with a dotted decimal number; for example, `3` or `3.1` or `3.1.1`. To hear these numbers correctly, make sure that your screen reader is set to read punctuation. All syntax elements that have the same dotted decimal number (for example, all syntax elements that have the number `3.1`) are mutually exclusive alternatives. If you hear the lines `3.1 USERID` and `3.1 SYSTEMID`, your syntax can include either `USERID` or `SYSTEMID`, but not both.

The dotted decimal numbering level denotes the level of nesting. For example, if a syntax element with dotted decimal number `3` is followed by a series of syntax elements with dotted decimal number `3.1`, all the syntax elements numbered `3.1` are subordinate to the syntax element numbered `3`.

Certain words and symbols are used next to the dotted decimal numbers to add information about the syntax elements. Occasionally, these words and symbols might occur at the beginning of the element itself. For ease of identification, if the word or symbol is a part of the syntax element, the word or symbol is preceded by the backslash (\) character. The `*` symbol can be used next to a dotted decimal number to indicate that the syntax element repeats. For example, syntax element `*FILE` with dotted decimal number 3 is read as `3 \* FILE`. Format `3* FILE` indicates that syntax element `FILE` repeats. Format `3* \* FILE` indicates that syntax element `* FILE` repeats.

Characters such as commas, which are used to separate a string of syntax elements, are shown in the syntax just before the items they separate. These characters can appear on the same line as each item, or on a separate line with the same dotted decimal number as the relevant items. The line can also show another symbol that provides information about the syntax elements. For example, the lines `5.1*`, `5.1 LASTRUN,` and `5.1 DELETE` mean that if you use more than one of the `LASTRUN` and `DELETE` syntax elements, the elements must be separated by a comma. If no separator is given, assume that you use a blank to separate each syntax element.

If a syntax element is preceded by the `%` symbol, that element is defined elsewhere. The string following the `%` symbol is the name of a syntax fragment rather than a literal. For example, the line `2.1 %OP1` refers to a separate syntax fragment `OP1`.

The following words and symbols are used next to the dotted decimal numbers:

? Specifies an optional syntax element. A dotted decimal number followed by the `?` symbol indicates that all the syntax elements with a corresponding dotted decimal number, and any subordinate syntax elements, are optional. If there is only one syntax element with a dotted decimal number, the `?` symbol is displayed on the same line as the syntax element (for example, `5? NOTIFY`). If there is more than one syntax element with a dotted decimal number, the `?` symbol is displayed on a line by itself, followed by the syntax elements that are optional. For example, if you hear the lines `5 ?`, `5 NOTIFY`, and `5 UPDATE`, you know that syntax elements `NOTIFY` and `UPDATE` are optional; that is, you can choose one or none of them. The `?` symbol is equivalent to a bypass line in a railroad diagram.

! Specifies a default syntax element. A dotted decimal number followed by the `!` symbol and a syntax element indicates that the syntax element is the default option for all syntax elements that share the same dotted decimal number. Only one of the syntax elements that share the same dotted decimal number can specify a `!` symbol. For example, if you hear the lines `2? FILE`, `2.1! (KEEP)`, and `2.1 (DELETE)`, you know that `(KEEP)` is the default option for the `FILE` keyword. In this example, if you include the `FILE` keyword but do not specify an option, default option `KEEP` is applied. A default option also applies to the next higher dotted decimal number. In this example, if the `FILE` keyword is omitted, default `FILE(KEEP)` is used. However, if you hear the lines `2? FILE`, `2.1`, `2.1.1! (KEEP)`, and `2.1.1 (DELETE)`, the default option KEEP only applies to the next higher dotted decimal number, `2.1` (which does not have an associated keyword), and does not apply to `2? FILE`. Nothing is used if the keyword `FILE` is omitted.

***** Specifies a syntax element that can be repeated zero or more times. A dotted decimal number followed by the `*` symbol indicates that this syntax element can be used zero or more times; that is, it is optional and can be repeated. For example, if you hear the line `5.1* data-area`, you know that you can include more than one data area or you can include none. If you hear the lines `3*`, `3 HOST`, and `3 STATE`, you know that you can include `HOST`, `STATE`, both together, or nothing.

Notes:

1. If a dotted decimal number has an asterisk (*) next to it and there is only one item with that dotted decimal number, you can repeat that same item more than once.
2. If a dotted decimal number has an asterisk next to it and several items have that dotted decimal number, you can use more than one item from the list, but you cannot use the items more than once each. In the previous example, you can write `HOST STATE`, but you cannot write `HOST HOST`.
3. The * symbol is equivalent to a loop-back line in a railroad syntax diagram.

+ Specifies a syntax element that must be included one or more times. A dotted decimal number followed by the + symbol indicates that this syntax element must be included one or more times. For example, if you hear the line `6.1+ data-area`, you must include at least one data area. If you hear the lines `2+`, `2 HOST`, and `2 STATE`, you know that you must include `HOST`, `STATE`, or both. As for the * symbol, you can repeat a particular item if it is the only item with that dotted decimal number. The + symbol, like the * symbol, is equivalent to a loop-back line in a railroad syntax diagram.

Notices

Trademarks

Index

Printed in USA

SC27-4528-00