HCL Informix 14.10 - JDBC Driver Programmer's Guide

Contents

Chapter 1. Informix® JDBC Driver Guide

These topics describe how to install, load, and use HCL Informix® JDBC Driver to connect to the Informix® database from within a Java™ application or applet.

These topics describe the Informix® extensions to JDBC in a task-oriented format; it does not include every method and parameter in the interface. For the complete reference, including all methods and parameters, see the online Javadoc™, which appears in the `doc/javadoc` directory where you installed HCL Informix® JDBC Driver.

You can also use Informix® JDBC Driver for writing user-defined routines that are executed in the server.

These topics are written for Java™ programmers who use the JDBC API to connect to Informix® databases with the Informix® JDBC Driver. To use these topics, you should know how to program in Java™ and, in particular, understand the classes and methods of the JDBC API.

For information about software compatibility, see the HCL Informix® JDBC Driver release notes.

Getting started

These topics provide an overview of HCL Informix® JDBC Driver and the JDBC API.

What is JDBC?

Java™ database connectivity (JDBC) is the JavaSoft specification of a standard application programming interface (API) that allows Java™ programs to access database management systems. The JDBC API consists of a set of interfaces and classes written in the Java™ programming language.

Using these standard interfaces and classes, programmers can write applications that connect to databases, send queries written in structured query language (SQL), and process the results.

Since JDBC is a standard specification, one Java™ program that uses the JDBC API can connect to any database management system (DBMS), as long as a driver exists for that particular DBMS.

What is a JDBC driver?

The JDBC API defines the Java™ interfaces and classes that programmers use to connect to databases and send queries. A JDBC driver implements these interfaces and classes for a particular DBMS vendor.

A Java™ program that uses the JDBC API loads the specified driver for a particular DBMS before it actually connects to a database. The JDBC **DriverManager** class then sends all JDBC API calls to the loaded driver.

There are four types of JDBC drivers:

JDBC-ODBC bridge plus ODBC driver, also called Type 1 driver

Translates JDBC API calls into Microsoft™ ODBC calls that are then passed to the ODBC driver

The ODBC binary code must be loaded on every client computer that uses this type of driver.

ODBC is an acronym for Open Database Connectivity.

Native-API, partly Java™ driver, also called Type 2 driver

Converts JDBC API calls into DBMS-specific client API calls

Like the bridge driver, this type of driver requires that some binary code is loaded on each client computer.

JDBC-Net, pure-Java driver, also called Type 3 driver

Sends JDBC API calls to a middle-tier server that translates the calls into the DBMS-specific network protocol

The translated calls are then sent to a particular DBMS.

Native-protocol, pure-Java driver, also called Type 4 driver

Converts JDBC API calls directly into the DBMS-specific network protocol without a middle tier

This driver allows the client applications to connect directly to the database server.

HCL Informix® JDBC Driver

HCL Informix® JDBC Driver is a native-protocol, pure-Java driver (Type 4) that supports the JDBC specification.

For information about JDBC specification compliance, go to Java™ software development kit.

When you use Informix® JDBC Driver in a Java™ program to interact with the Informix® database, your session connects directly to the database or database server, without a middle tier.

You can use the JDBC driver for Java™ applications that access the Informix® database server. The installation includes `ifxjdbc.jar` driver and several support `.jar` files.

Javadoc™ pages describe the Informix® extension classes, interfaces, and methods in detail.

- UNIX™: Javadoc™ pages are in `$JDBCLOCATION/doc/javadoc`, where `$JDBCLOCATION` refers to the directory where you installed Informix® JDBC Driver.
- Windows™: Javadoc™ pages are located in `%JDBCLOCATION%\doc\javadoc`, where `%JDBCLOCATION%` refers to the directory where you installed Informix® JDBC Driver.

Important:

If you want to run Java™ user-defined routines (UDRs) in the database server, you must use the server-side version of the driver that is installed with the optional J/Foundation component of the database server. The server-side driver includes a `jdbc.jar` file, which is derived from the `ifxjdbc.jar` file. The *HCL® J/Foundation Developer's Guide*

describes the interfaces and subprotocols that the HCL Informix® JDBC Driver provides specifically for server-side JDBC applications, as well as restrictions that apply to server-side JDBC applications.

Classes implemented in Informix® JDBC Driver

To support **DataSource** objects, connection pooling, and distributed transactions, HCL Informix® JDBC Driver provides classes that implement interfaces and classes for compliance with the Java™ Database Connectivity (JDBC) 4.0 specification.

Informix® classes that implement Java™ interfaces

The following table lists the Java™ interfaces and classes and the HCL Informix® classes that implement them.

JDBC interface or class	Informix® class
java.io.Serializable	com.informix.jdbcx.IfxCoreDataSource
java.sql.Connection	com.informix.jdbc.IfmxConnection
javax.sql.ConnectionEventListener	com.informix.jdbcx.IfxConnectionEventListener
javax.sql.ConnectionPoolDataSource	com.informix.jdbcx.IfxConnectionPoolDataSource
javax.sql.DataSource	com.informix.jdbcx.IfxDataSource
javax.sql.PooledConnection	com.informix.jdbcx.IfxPooledConnection
javax.sql.XADataSource	com.informix.jdbcx.IfxXADataSource
java.sql.ParameterMetaData	com.informix.jdbc.IfxParameterMetaData

HCL Informix® JDBC Driver, Version 3.0, and later implements the updateXXX() methods defined in the **ResultSet** interface by the JDBC 3.0 specification. These methods, such as **updateClob**, are further defined in the J2SDK 1.4.x API, and later and require that the **ResultSet** object can be updated. The updateXXX() methods allow rows to be updated by using Java™ variables and objects and extend to include additional JDBC types.

These methods update JDBC types implemented with locators, not the data designated by the locators.

Informix® classes that extend the JDBC specification

To support the HCL Informix® implementation of SQL statements and data types, HCL Informix® JDBC Driver provides classes that extend the supported JDBC specification (see Java™ software development kit). The following table lists the Java™ classes and the Informix® classes that application programs can use to extend them.

JDBC interface or class	Informix® class	Adds methods or constants for...
java.lang.Object	com.informix.lang.IfxTypes	Representing data types
java.lang.Object	com.informix.jdbc.IfxStatementTypes	Representing SQL statements

JDBC interface or class	Informix® class	Adds methods or constants for...
java.lang.Object	com.informix.jdbc.Interval[1]	Interval qualifiers and some common methods for the next two classes (base class for the next two)
java.lang.Object	com.informix.jdbc.IntervalYM[1]	Interval year-to-month
java.lang.Object	com.informix.jdbc.IntervalDF[1]	Interval day-to-fraction
java.lang.Object	com.informix.jdbc.IfxSmartBlob	Access methods for smart large objects
java.lang.Object	com.informix.jdbc.IfxLocator	Large object locator pointer
java.lang.Object	com.informix.jdbc.IfxLoStat	Statistical information about smart large objects
java.lang.Object	com.informix.jdbc.IfxLobDescriptor	Internal characteristics of smart large objects
java.lang.Object	com.informix.jdbc.IfxUDTInfo	General information about opaque and distinct types, detailed information about complex types
java.sql.Blob	com.informix.jdbc.IfxBblob	Binary large objects
java.sql.CallableStatement	com.informix.jdbc.IfmxCallableStatement	Parameter processing with Informix® types
java.sql.Clob	com.informix.jdbc.IfxCblob	Character large objects
java.sql.Connection	com.informix.jdbc.IfmxConnection	Opaque, distinct, and complex types
java.sql.SQLData	com.informix.jdbc.IfxBSONObject [1]	Informix® BSON data type See the IfxBSONObjectDemo.java program in the `$INFORMIXDIR/demo/bson` directory for examples of how to insert and query JSON and BSON data and use the IfxBSONObject methods.
java.sql.PreparedStatement	com.informix.jdbc.IfmxPreparedStatement	Parameter processing with Informix® types
java.sql.ResultSet	com.informix.jdbc.IfmxResultSet	Informix® interval data types

JDBC interface or class	Informix® class	Adds methods or constants for...
java.sql.ResultSetMetaData	com.informix.jdbc.IfmxResultSetMetaData	Columns with Informix® data types
java.sql.SQLInput	com.informix.jdbc.IfmxComplexSQLInput	Opaque, distinct, and complex types
java.sql.SQLInput	com.informix.jdbc.IfmxUDTSQLInput	Opaque, distinct, and complex types
java.sql.SQLOutput	com.informix.jdbc.IfmxComplexSQLOutput	Opaque, distinct, and complex types
java.sql.SQLOutput	com.informix.jdbc.IfmxUDTSQLOutput	Opaque, distinct, and complex types
java.sql.Statement	com.informix.jdbc.IfmxStatement	Single result sets, autofree mode, statement types, and SERIAL data type processing

1. The server-side JDBC driver (`jdbc.jar`) does not support this Informix® class.

Informix® classes that provide support beyond the Java™ specification

A number of HCL Informix® classes provide support for functionality not present in the Java™ specification. These classes are listed in the following table.

JDBC interface or class	Informix® class	Provides support for...
java.lang.Object	UDTManager	Deploying opaque data types in the database server
java.lang.Object	UDTMetaData	Deploying opaque data types in the database server
java.lang.Object	UDRManager	Deploying user-defined routines in the database server
java.lang.Object	UDRMetaData	Deploying user-defined routines in the database server

UDTManager and UDRManager classes

The **UDTManager** and **UDRManager** helper classes are in the `ifxtools.jar` file in the **udtudrmgr** package.

To access a packaged class, use the following import statements in your program:

- `import udtudrmgr.UDTManager;`
- `import udtudrmgr.UDRManager;`

Files in HCL Informix® JDBC Driver

HCL Informix® JDBC Driver is available in the program file, `setup.jar`. For instructions on how to install the driver, see Installing the JDBC Driver on page 9.

After installation, the product consists of the following files, some of which are Java™ archive (JAR) files:

- `lib/ifxjdbc.jar`

  Optimized implementations of the JDBC API interfaces, classes, and methods

  The file is compiled with the **-O** option of the javac command.

- `lib/ifxtools.jar`

  Utilities: ClassGenerator, lightweight directory access protocol (LDAP) loader, and others

  The file is compiled with the **-O** option of the javac command.

- `lib/ifxlang.jar`

  Localized versions of all message text supported by the driver

  The file is compiled with the **-O** option of the javac command.

- `demo/basic/*`

  ```
  demo/rmi/*
  demo/stores7/*
  demo/clob-blob/*
  demo/complex-types/*
  demo/pickaseat/*
  demo/xml/*
  demo/proxy/*
  demo/connection-pool/*
  demo/udt-distinct/ *
  demo/hdr/*
  demo/tools/udtudrmgr/*
  ```

  Sample programs that use the JDBC API

  For descriptions of these sample files, see Sample code files on page 224.

- `proxy/IfxJDBCProxy.class`

  Http tunneling proxy class file

- `proxy/SessionMgr.class`

  Session manager class file supporting the http tunneling proxy
- `proxy/TimeoutMgr.class`

  Timeout manager class file supporting the http tunneling proxy
- `doc/release/*`

  Online release and documentation notes
- `doc/javadoc/*`

  The Javadoc™ pages for Informix® extension classes and interfaces

The `lib`, `demo`, `proxy`, and `doc` directories are subdirectories of the directory where you installed HCL Informix® JDBC Driver.

Obtain the JDBC driver

You can obtain the HCL Informix® JDBC Driver from the HCL Informix® JDBC Driver product CD, from the JDBC directory of the HCL Informix® product bundle CD, or you can download the driver from https://mvnrepository.com/artifact/com.ibm.informix/jdbc.

The CD or website download contain the following files:

- `setup.jar`
- `doc/jdbcrel.dita`
- `doc/install.txt`

The `setup.jar` file is the HCL Informix® JDBC Driver installation program.

The documentation directory, `<dir>/doc`, contains the release notes file in HTML format. See this document for any new information that is not available in these topics.

Installing the JDBC Driver

About this task

To install the HCL Informix® JDBC Driver when you install Informix®, follow the instructions in the *HCL® Informix® Installation Guide* for installing the database server.

To install the Informix® JDBC Driver by itself:

- If you downloaded the `.zip` file from the website, extract the file to a directory.
- If you are installing the driver from a CD, load the CD into the CD-ROM drive.

Installing JDBC driver in graphical or console mode

1. From a command prompt, start the installation program by using one of the following commands:
   Choose from:
   - To start in graphical mode:
     ```
     java -jar dir/setup.jar -i gui
     ```
   - To start in console mode:
     ```
     java -jar dir/setup.jar -i console
     ```
     Where *dir* is the location of the `setup.jar` file.
2. Read the license agreement and accept the terms. Respond to the prompts as the installation guides you.
3. When prompted, accept the default directory or specify a different directory.

   On a Windows™ platform, the default directory is `C:\Program Files\IBM\Informix_JDBC_Driver`

   To prevent errors in installation and uninstallation, do not use an exclamation point character (!) in the installation path.
4. When prompted, verify the location of the installation path.

   The installer installs the HCL Informix® JDBC Driver and notifies you that an uninstaller is being added to the installation directory.

   Tip: If the installation stalls when connected to an NFS-mounted file system, you should first try resolving NFS issues. In some cases, unmounting and remounting the share can resolve the issue. Otherwise, forcefully terminate the installation, clean up any orphaned processes, and restart the installation.
5. Specify **Finish** when you see this message:
   ```
   Congratulations. IBM Informix JDBC Driver has been successfully installed to:  <install dir>
   ```

Installing the JDBC driver in silent mode

About this task

From a command prompt, run the following command:

```
java -jar dir/setup.jar i -silent
   -DUSER_INSTALL_DIR=destination_dir-DLICENSE_ACCEPTED=TRUE
```

Where:

- *dir* is the location of the `setup.jar` file.
- `destination_dir` is the directory where you want to install the JDBC Driver.

The installation is complete when the command finishes running.

Using the driver in an application

To use HCL Informix® JDBC Driver in an application, you must set your **CLASSPATH** environment variable to point to the driver files. The **CLASSPATH** environment variable tells the Java™ virtual machine (JVM) and other applications where to find the Java™ class libraries used in a Java™ program.

UNIX™

There are two ways to set your **CLASSPATH** environment variable:

- Add the full path name of `ifxjdbc.jar` to **CLASSPATH**:

  ```
  setenv CLASSPATH /jdbcdriv/lib/ifxjdbc.jar:$CLASSPATH
  ```

  To add localized message support, specify `ifxlang.jar`:

  ```
  setenv CLASSPATH
  /jdbcdriv/lib/ifxjdbc.jar:/jdbcdriv/lib/ifxlang.jar:
     $CLASSPATH
  ```

- Unpack `ifxjdbc.jar` and add its directory to **CLASSPATH**:

  ```
  cd /jdbcdriv/lib
  jar xvf ifxjdbc.jar
  setenv CLASSPATH /jdbcdriv/lib:$CLASSPATH
  ```

  To add localized message support, specify `ifxlang.jar`:

  ```
  cd /jdbcdriv/lib
  jar xvf ifxjdbc.jar
  jar xvf ifxlang.jar
  setenv CLASSPATH /jdbcdriv/lib:$CLASSPATH
  ```

Windows™

There are two ways to set your **CLASSPATH** environment variable:

- Add the full path name of `ifxjdbc.jar` to **CLASSPATH**:

  ```
  set CLASSPATH=c:\jdbcdriv\lib\ifxjdbc.jar;%CLASSPATH%
  ```

  To add localized message support, specify `ifxlang.jar`:

  ```
  set CLASSPATH=c:\jdbcdriv\lib\ifxjdbc.jar;c:\
     jdbcdriv\lib\ifxlang.jar;%CLASSPATH%
  ```

- Unpack `ifxjdbc.jar` and add its directory to **CLASSPATH**:

  ```
  cd c:\jdbcdriv\lib
  jar xvf ifxjdbc.jar
  set CLASSPATH=c:\jdbcdriv\lib;%CLASSPATH%
  ```

  To add localized message support, specify `ifxlang.jar`:

  ```
  cd c:\jdbcdriv\lib
  jar xvf ifxjdbc.jar
  ```

```
jar xvf ifxlang.jar
set CLASSPATH=c:\jdbcdriv\lib;%CLASSPATH%
```

If you are using **javax.sql** classes (for example, Datasource), specify `ifxjdbcx.jar` in addition to `ifxjdbc.jar`.

Using the driver in an applet

About this task

You can use HCL Informix® JDBC Driver in an applet to connect to the Informix® database from a web browser. The following steps show how to specify HCL Informix® JDBC Driver in the applet and how to ensure that the driver is correctly downloaded from the web server.

To use HCL Informix® JDBC Driver in an applet:

1. Install `ifxjdbc.jar` in the same directory as your applet class file.
2. Specify `ifxjdbc.jar` in the ARCHIVE attribute of the APPLET tag in your HTML file, as shown in the following example:

   Example

   ```
   <APPLET ARCHIVE=ifxjdbc.jar CODE=my_applet.class
   CODEBASE=http://www.myhost.com WIDTH=460 HEIGHT=160>
   </APPLET>
   ```

Results

Important: Some browsers do not support the ARCHIVE attribute of the APPLET tag. If this is true of your browser, unpack and install the `ifxjdbc.jar` file in the root directory of your web server. If your browser also does not support the JDBC API, you must install the class files included in the `java.sql` package in the root directory of the web server. See your web server documentation for information about installing files in the root directory.

Because unsigned applets cannot access some system resources for security reasons, the following features of HCL Informix® JDBC Driver do not work for unsigned applets:

- **The `sqlhosts` file and LDAP server access**. The host name and port number properties or service name of the Informix® database server in the database URL are optional if you are referencing an `sqlhosts` file directly or through an LDAP server.

  For unsigned applets, however, the host name and the port number or service name of the Informix® database server are always required, unless your applet is using the HTTP proxy server. For more information about the HTTP proxy server, see An HTTP proxy server on page 44.
- **LOBCACHE=0**. Setting the **LOBCACHE** environment variable to `0` in the database URL specifies that a smart large object is always stored in a file. This setting is not supported for unsigned applets.

Tip: You can enable these features for unsigned applets by using Microsoft™ Internet Explorer, which provides an option to configure the applet permissions.

To access a database on a different host or behind a firewall from an applet, you can use the Informix® HTTP proxy servlet in a middle tier. For more information, see An HTTP proxy server on page 44.

Uninstall the JDBC Driver

When you install HCL Informix® JDBC Driver, the installation program creates an uninstall package in the directory in which you installed the JDBC Driver. Uninstalling Informix® JDBC Driver completely removes the driver and all of its components from your computer.

The following section describes how to uninstall Informix® JDBC Driver on all platforms.

Tip: If the *<destination-dir>* in which you installed the Informix® JDBC Driver includes spaces in its path name, enclose the entire path name in quotation marks when executing the uninstall command.

Uninstall in graphical or console mode

Run one of the following commands to start the uninstall program. The *destination-dir* is the directory in which you installed the HCL Informix® JDBC Driver.

If you installed the Informix® JDBC Driver by using the Informix® installation program, you must use run the uninstall executable program that was added to your computer when the driver was installed.

- To uninstall by using the graphical mode:

```
destination-dir/uninstall/uninstall_jdbc/uninstalljdbc -i gui
```

- To uninstall by using the console mode:

```
destination-dir/uninstall/uninstall_jdbc/uninstalljdbc -i console
```

If you installed the Informix® JDBC Driver separately, run the JAR file to uninstall the driver:

- To uninstall by using the graphical mode:

```
java -jar destination-dir/uninstall/uninstall_jdbc/uninstaller.jar -i gui
```

- To uninstall by using the console mode:

```
java -jar destination-dir/uninstall/uninstall_jdbc/uninstaller.jar -i console
```

Follow the prompts to uninstall the JDBC driver.

Uninstall in silent mode

When you uninstall the HCL Informix® JDBC Driver in the silent mode, you do not receive any messages about the uninstallation.

How you installed the Informix® JDBC Driver determines the program that you use to uninstall the driver.

Run one of the following commands to start the uninstall program in the silent mode. The *destination_dir* is the directory in which you installed the Informix® JDBC Driver.

If you installed the Informix® JDBC Driver by using the Informix® installation program, specify the following command to start the uninstall program in the silent mode:

```
destination_dir/uninstall/uninstall_jdbc/uninstalljdbc -i silent
```

If you installed the Informix® JDBC Driver separately from installing Informix®, specify the following command to run the JAR file in the silent mode:

```
java -jar destination_dir/uninstall/uninstall_jdbc/uninstaller.jar -i silent
```

Connect to the database

These topics explain the information you need to use HCL Informix® JDBC Driver to connect to the Informix® database.

You must first establish a connection to the Informix® database server or database before you can start sending queries and receiving results in your Java™ program.

You establish a connection by completing two actions:

1. Load Informix® JDBC Driver.
2. Create a connection to either a database server or a specific database in one of the following ways:
    - Use a **DataSource** object.
    - Use the DriverManager.getConnection() method.

Using a **DataSource** object is preferable to using the DriverManager.getConnection() method because a **DataSource** object is portable and allows the details about the underlying data source to be transparent to the application. The target data source implementation can be modified, or the application can be redirected to a different server without affecting the application code.

A **DataSource** object can also provide support for connection pooling and distributed transactions. In addition, Enterprise JavaBeans™ and J2EE require a **DataSource** object.

The following additional connection options are available:

- Setting environment variables
- Dynamically reading the Informix® `sqlhosts` file
- Using an HTTP proxy server
- Using password encryption
- Using network encryption

Load HCL Informix® JDBC Driver

To load HCL Informix® JDBC Driver, use the Class.forName() method, passing it the value `com.informix.jdbc.IfxDriver`:

```
try
    {
    Class.forName("com.informix.jdbc.IfxDriver");
    }
catch (Exception e)
    {
    System.out.println("ERROR: failed to load Informix JDBC driver.");
    e.printStackTrace();
    return;
    }
```

The Class.forName() method loads the Informix® implementation of the **Driver** class, **IfxDriver**. **IfxDriver** then creates an instance of the driver and registers it with the **DriverManager** class.

After you have loaded Informix® JDBC Driver, you are ready to connect to the Informix® database or database server.

If you are writing an applet to be viewed with Microsoft™ Internet Explorer, you might need to explicitly register Informix® JDBC Driver to avoid platform incompatibilities.

To explicitly register the driver, use the DriverManager.registerDriver() method:

```
DriverManager.registerDriver(com.informix.jdbc.IfxDriver)
   Class.forName("com.informix.jdbc.IfxDriver").newInstance());
```

This method might register Informix® JDBC Driver twice, which does not cause a problem.

A DataSource object

HCL Informix® JDBC Driver extends the standard **DataSource** interface to allow connection properties (both the standard properties and Informix® environment variables) to be defined in a **DataSource** object instead of through the URL.

The following table describes how Informix® connection properties correspond to **DataSource** properties.

Informix® connection property	DataSource property	Data type	Required?	Description
IFXHOST	None; see DataSource extensions on page 233 for how to set **IFXHOST**.	String	Yes for client-side JDBC, unless **SQLH_TYPE** is defined; no for server-side JDBC	The IP address or the host name of the computer running the Informix® database server
PORTNO	portNumber	int	Yes for client-side JDBC, unless **SQLH_TYPE** is defined; no for server-side JDBC	The port number of the Informix® database server. The port number is listed in the `/etc/services` file.

Informix® connection property	DataSource property	Data type	Required?	Description
DATABASE	databaseName	String	No, except for connections from web applications (such as a browser) running in the database server	The name of the Informix® database to which you want to connect If you do not specify the name of a database, a connection is made to the Informix® database server.
INFORMIXSERVER	serverName	String	Yes for client-side JDBC; ignored for server-side JDBC	The name of the Informix® database server to which you want to connect
USER	user	String	Yes	The user name controls (or determines) the session privileges when connected to the Informix® database or database server Normally, you must specify both user name and password; however, if the user running the JDBC application is trusted by the DBMS, you might omit both.
PASSWORD	password	String	Yes	The password of the user Normally, you must specify both the user name and the password; however, if the user running the JDBC application is trusted by the DBMS, you might omit both.
None	description	String	Yes	A description of the **DataSource** object
None	dataSourceName	String	No	The name of an underlying **ConnectionPoolDataSource** or **XADataSource** object for connection pooling or distributed transactions

Unsupported connection properties

The **networkProtocol** and **roleName** properties are not supported by HCL Informix® JDBC Driver.

Specify connection information

If an LDAP (Lightweight Directory Access Protocol) server or `sqlhosts` file provides the IP address, host name, or port number or service name of the Informix® database server through the **SQLH_TYPE** property, you do not have to specify them using the standard **DataSource** properties. For more information, see Dynamically reading the Informix sqlhosts file on page 32.

ConnectionPoolDataSource object

For information about the **ConnectionPoolDataSource** object, see A connection pool on page 219.

Environment variables

For a list of supported environment variables (properties), see Informix environment variables with the HCL Informix JDBC Driver on page 23. For a list of Informix® **DataSource** extensions, which allow you to define environment variable values and connection pool tuning parameters, see DataSource extensions on page 233. The driver does not consult the users environment to determine environment variable values.

High-availability data replication

You can use a **DataSource** object with High-Availability Data Replication. For more information, see Connections to the servers of a high-availability cluster on page 36.

Example: Use of a DataSource object in an example program

The following code from the **pickaseat** example program defines and uses a **DataSource** object:

```
IfxConnectionPoolDataSource cpds = null;
try
{
   Context initCtx = new InitialContext();
   cpds = new IfxConnectionPoolDataSource();
   cpds.setDescription("Pick-A-Seat Connection pool");
   cpds.setIfxIFXHOST("158.58.60.88");
   cpds.setPortNumber(179);
   cpds.setUser("demo");
   cpds.setPassword("demo");
   cpds.setServerName("ipickdemo_tcp");
   cpds.setDatabaseName("ipickaseat");
   cpds.setIfxGL_DATE("%B %d, %Y");
   initCtx.bind("jdbc/pooling/PickASeat", cpds);
}
catch (Exception e)
{
   System.out.println("Problem with registering the CPDS");
   System.out.println("Error: " + e.toString());
}
```

Example: Use of a DataSource object with the IFX_LOCK_MODE_WAIT connection property

The following are examples of the **IFX_LOCK_MODE_WAIT** connection property that use a **DataSource** object:

Example 1

```
IfxDataSource ds = new IfxDataSource ();
ds. setIfxIFX_LOCK_MODE_WAIT (65);    // wait for 65 seconds

int waitMode = ds.getIfxIFX_LOCK_MODE_WAIT ();
```

Example 2

```
An example Using DataSource:
IfxDataSource ds = new IfxDataSource ();
ds.setIfxIFX_ISOLATION_LEVEL ("0U");    // set isolation to dirty read with
             retain
     //  update locks.
.
String isoLevel = ds.getIfxIFX_ISOLATION_LEVEL ();
```

The DriverManager.getConnection() method

To create a connection to the HCL Informix® database or database server, you can use the DriverManager.getConnection() method. This method creates a **Connection** object, which is used to create SQL statements, send them to the Informix® database, and process the results.

The **DriverManager** class tracks the available drivers and handles connection requests between appropriate drivers and databases or database servers. The *url* parameter of the getConnection() method is a database URL that specifies the subprotocol (the database connectivity mechanism), the database or database server identifier, and a list of properties.

A second parameter to the getConnection() method, *property*, is the property list. See Specify properties on page 22 for an example of how to specify a property list.

The following example shows a database URL that connects to a database called **testDB** from a client application:

```
jdbc:informix-sqli://123.45.67.89:1533/testDB:
   INFORMIXSERVER=myserver;user=rdtest;password=test
```

The details of the database URL syntax are described in the next section.

The following partial example from the `CreateDB.java` program shows how to connect to database **testDB** by using DriverManager.getConnection(). In the full example, the *url* variable, described in the preceding example, is passed in as a parameter when the program is run at the command line.

```
try
   {
   conn = DriverManager.getConnection(url);
   }
catch (SQLException e)
   {
   System.out.println("ERROR: failed to connect!");
   System.out.println("ERROR: " + e.getMessage());
   e.printStackTrace();
   return;
   }
```

Important: The only Informix® connection type supported by HCL Informix® JDBC Driver is **tcp**. Shared memory and other connection types are not supported. For more information about connection types, see the *HCL® Informix® Administrator's Guide* for your database server.

Important: Not all methods of the **Connection** interface are supported by HCL Informix® JDBC Driver. For a list of unsupported methods, see Unsupported methods and methods that behave differently on page 77.

Client applications do not need to explicitly close a connection; the database server closes the connection automatically. However, if your application is running in the database server using server-side JDBC, you should explicitly close the connection.

Format of database URLs

The format of a database URL is determined by whether you are connecting from a client or on the database server.

For connections from a client, use the following format:

```
jdbc:informix-sqli://hostname:portnum/database_name:[USER=userid;PASSWORD=password][
INFORMIXSERVER=servername;][CSM=(SSO=database_server@realm,ENC=true)};name=value]
```

For connections on the database server, use the following format:

```
jdbc:informix-direct://database_name:;[user=userid;password=password][;name=value]
```

hostname

This required parameter specifies the host name of the computer that is running the Informix® database server.

This parameter is required for client-side JDBC, unless the SQLH_TYPE property is defined or the IFXHOST property is used. This parameter is optional for server-side JDBC.

portnum

This required parameter specifies the port number of the Informix® database server. The value of the port number and server name is listed in the `/etc/services` file.

This parameter is required for client-side JDBC unless the SQLH_TYPE property is defined. This parameter is optional for server-side JDBC.

database_name

This required parameter specifies the name of the Informix® database to connect to. If you do not specify the name of a database, a connection is made to the Informix® database server.

This parameter is not required except for connections from web applications running in the database server.

INFORMIXSERVER=*servername*

This optional parameter specifies the name of the database server to connect to. For example, `INFORMIXSERVER=lo_informix1210`.

USER=*userid*

This optional parameter specifies the user ID that is used in connections to the Informix® database server.

This parameter is optional, however, if you specify *userid* then you must also specify the PASSWORD. If you do not specify the USER and PASSWORD, the driver calls System.getProperty() to obtain the name of the user currently running the application, and the client is assumed to be trusted.

For SSO, you must specify the user and password or the CSM setting.

Note: Support for Communication Support Module (CSM) is removed starting Informix Server 14.10.xC9 . You should use Transport Layer Security (TLS)/Secure Sockets Layer (SSL) instead.

PASSWORD=*password*

This optional parameter specifies the password for the specified user ID.

This parameter is optional, however, if you specify *password* then you must also specify the USER. If you do not specify the USER and PASSWORD, the driver calls System.getProperty() to obtain the name of the user currently running the application, and the client is assumed to be trusted.

SSO=*database_server@realm*

This required parameter specifies the service principle for (SSO) access control. For more information, see Using single sign-on access control with the Informix JDBC Driver on page 54.

name=value

This optional parameter specifies the name-value pair that specifies a value for the Informix® environment variable that is contained in the *name* variable, which is recognized by either HCL Informix® JDBC Driver or by Informix® database servers. The *name* variable is not case-sensitive.

For more information, see Specify properties on page 22 and Informix environment variables with the HCL Informix JDBC Driver on page 23.

If an LDAP server or `sqlhosts` file provides the IP address, host name, or port number through the SQLH_TYPE property, you do not have to specify them in the database URL. For more information, see Dynamically reading the Informix sqlhosts file on page 32.

Example

In the following example, the connection syntax for a client-side connection is shown:

```
jdbc:informix-sqli://123.45.67.89:1533/testDB:
   INFORMIXSERVER=myserver;user=rdtest;password=test
```

In the following example, the connection syntax for a database server connection is shown:

```
jdbc:informix-direct://testDB;user=rdtest;password=test
```

IP address in connection URLs

The HCL Informix® JDBC Driver, Version 3.0 and later is IPv6 aware.

That is, the code that parses the connection URL can handle the longer (128-bit mode) IPv6 addresses (as well as IPv4 format). This IP address can be a IPv6 literal, for example:

```
3ffe:ffff:ffff:ffff:0:0:0:12
```

To connect to the IPv6 port with the Informix® server, use the system property, for example:

```
java -Djava.net.preferIPv6Addresses=true ...
```

With the HCL Informix® JDBC Driver, Version 3.0, or later handling of URLs without IPv6 literals is unchanged, and legacy behavior is unchanged.

The colon (:) is a key delimiter in a connection URL, especially in IPv6 literal addresses.

You must create a well-formed URL for the driver to recognize an IPv6 literal address. Note, in the following example:

- The `jdbc:informix-sqli://` is required.
- The colons surrounding the `8088`, (`:8088:`) are required.
- The `3ffe:ffff:ffff:ffff:0::12` is not validated by the driver.
- The `8088` must be a valid number < 32k.

```
jdbc:informix-sqli://3ffe:ffff:ffff:ffff:0::12:8088:informixserver=X...
```

Database versus database server connections

Using the DriveManager.getConnection() method, you can create a connection to either the HCL Informix® database or the Informix® database server.

To create a connection to the Informix® database, specify the name of the database in the *dbname* variable of the database URL. If you omit the name of a database, a connection is made to the database server specified by the **INFORMIXSERVER** environment variable of the database URL or the connection property list.

If you connect directly to the Informix® database server, you can execute an SQL statement that connects to a database in your Java™ program.

The example given in The DriverManager.getConnection() method on page 18 shows how to create a connection directly to the Informix® database called **testDB** with the database URL.

The following example from the `DBConnection.java` program shows how to first create a connection to the Informix® database server called **myserver** and then connect to the database **testDB** by using the Statement.executeUpdate() method.

The following database URL is passed in as a parameter to the program when the program is run at the command line; note that the URL does not include the name of a database:

```
jdbc:informix-sqli://123.45.67.89:1533:INFORMIXSERVER=myserver;
   user=rdtest;password=test
```

The code is:

```
String cmd = null;
int rc;
Connection conn = null;

try
{
   Class.forName("com.informix.jdbc.IfxDriver");
}
catch (Exception e)
{
   System.out.println("ERROR: failed to load Informix JDBC driver.");
}
try
{
   conn = DriverManager.getConnection(newUrl);
}
catch (SQLException e)
{
   System.out.println("ERROR: failed to connect!");
   e.printStackTrace();
   return;
}
try
{
   Statement stmt = conn.createStatement();
   cmd = "database testDB;";
   rc = stmt.executeUpdate(cmd);
   stmt.close();
}
catch (SQLException e)
{
   System.out.println("ERROR: execution failed - statement:
      " + cmd);
   System.out.println("ERROR: " + e.getMessage());        }
```

Specify properties

When you use the DriverManager.getConnection() method to create a connection, HCL Informix® JDBC Driver reads Informix® environment variables only from the name-value pairs in the connection database URL or from a connection property list. The driver does not consult the users environment for any environment variables.

To specify Informix® environment variables in the name-value pairs of the connection database URL, see Format of database URLs on page 19.

To specify Informix® environment variables via a property list, use the **java.util.Properties** class to build the list of properties. The list of properties might include Informix® environment variables, such as **INFORMIXSERVER**, as well as **user** and **password**.

After you have built the property list, pass it to the DriverManager.getConnection() method as a second parameter. You still need to include a database URL as the first parameter, although in this case you do not need to include the list of properties in the URL.

The following code from the `optofc.java` example shows how to use the **java.util.Properties** class to set connection properties. It first uses the Properties.put() method to set the environment variable **OPTOFC** to `1` in the connection property list; then it connects to the database.

The DriverManager.getConnection() method in this example takes two parameters: the database URL and the property list. The example creates a connection similar to the example given in The DriverManager.getConnection() method on page 18.

The following database URL is passed in as a parameter to the example program when the program is run at the command line:

```
jdbc:informix-sqli://myhost:1533:informixserver=myserver;
   user=rdtest;password=test
```

The code is:

```
try
{
   Class.forName("com.informix.jdbc.IfxDriver");
   }
catch (Exception e)
   {
   System.out.println("ERROR: failed to load Informix JDBC driver.");
   }

try
   {
   Properties pr = new Properties();
   pr.put("OPTOFC","1");
   conn = DriverManager.getConnection(newUrl, pr);
   }
catch (SQLException e)
   {
   System.out.println("ERROR: failed to connect!");
   }
```

Informix® environment variables with the HCL Informix® JDBC Driver

The following table lists most of the HCL Informix® environment variables supported by the client JDBC driver. For server-side JDBC, use property settings in the database URL rather than setting environment variables, because the environment variables would apply to all programs running in the database server. For more information about properties, see Specify properties on page 22.

For a list of environment variables that provide globalization features, see Globalization and date formats on page 195. For a list of environment variables useful for troubleshooting, see Tuning and troubleshooting on page 215

Supported Informix® environment variables	Description

APPENDISAM	When set to `TRUE`, the **APPENDISAM** environment variable appends the ISAM Error code and message (if present) to the SQL Exception message, which is shown when .toString() or .getMessage() of an SQL Exception is called. The exception message is shown in the following format:
	`<INFORMIX ERROR MESSAGE> (<INFORMIX CODE>)`
	`ISAM error: <ISAM MESSAGE>(<ISAM CODE>)`

AUTO_CASE_SCHEMA	To instruct JDBC to automatically case schema (default = true) or when set to false, to return the schema as it was saved.
CSM	To specify that Communication Support Module is to be used. HCL Informix® JDBC Driver 3.0 and later supports an encryption CSM. For more information, see Encryption options on page 49. **Note:** Support for Communication Support Module (CSM) is removed starting Informix Server 14.10.xC9 . You should use Transport Layer Security (TLS)/Secure Sockets Layer (SSL) instead.
DBANSIWARN	When set to `1`, checks for Informix® extensions to ANSI-standard syntax
DBSPACETEMP	Specifies the dbspaces in which temporary tables are built
DBTEMP	Specifies the full path name of the directory into which you want gateway products to place their temporary files and temporary tables. The driver does not use this variable; it just passes the value to the server.
DBUPSPACE	Specifies the amount of disk space and memory that the UPDATE STATISTICS statement can use for sorting rows when it constructs multiple-column distributions, whether to sort with indexes, and whether to save the plan for calculating the column distributions in the `sqexplain.out` file.
DELIMIDENT	When set to `Y`, specifies that strings set off by double quotation marks are delimited identifiers
ENABLE_TYPE_CACHE	When set to `TRUE`, caches the data type information for opaque, distinct, or row data types. When a **Struct** or **SQLData** object inserts data into a column and getSQLTypeName() returns the type name, the driver uses the cached information instead of querying the database server.

Supported Informix® environment variables	Description
ENABLE_HDRSWITCH	When set to `TRUE`, secondary server properties are used to connect to the secondary server if the primary server is unavailable.
FET_BUF_SIZE	Overrides the default setting for the size of the fetch buffer for all data except large objects. The default size is 4096 bytes. This variable is not supported in server-side JDBC.
IFX_AUTOFREE	When set to `1`, specifies that the Statement.close() method does not require a network round trip to free the database server cursor resources if the cursor has already been closed in the database server. The database server automatically frees the cursor resources after the cursor is closed, either explicitly by the ResultSet.close() method or implicitly through the **OPTOFC** environment variable. After the cursor resources have been freed, the cursor can no longer be referenced. For more information, see The Auto Free feature on page 87.
IFX_BATCHUPDATE_PER_SPEC	When set to `1` (the default), returns the number of rows affected by the SQL statements executed in a batch operation by the executeBatch() method
IFX_CODESETLOB	If set to a number greater than or equal to 0, automates code-set conversion for TEXT and CLOB data types between client and database locales. The value of this variable determines whether code-set conversion is done in memory in or in temporary files. If set to 0, code-set conversion uses temporary files. If set to a value greater than 0, code-set conversion occurs in the memory of the client computer, and the value represents the number of bytes of memory allocated for the conversion. For more information, see Convert with the IFX_CODESETLOB environment variable on page 206.
IFX_DIRECTIVES	Determines whether the optimizer allows query optimization directives from within a query. This variable is set on the client. The driver does not use this variable; it just passes the value to the server.
IFX_EXTDIRECTIVES	Specifies whether the query optimizer allows external query optimization directives from the **sysdirectives** system catalog table to be applied to queries in existing applications. The default is `OFF`. Possible values: **ON** External optimizer directives accepted **OFF** External optimizer directives not accepted

Supported Informix® environment variables	Description
	1 External optimizer directives accepted **0** External optimizer directives not accepted
IFX_GET_SMFLOAT_AS_FLOAT	When set to `0` (the default), maps the Informix® SMALLFLOAT data type to the JDBC REAL data type. This setting conforms to the JDBC specification. When set to `1`, maps the Informix® SMALLFLOAT data type to the JDBC FLOAT data type. This setting enables compatibility with earlier versions of HCL Informix® JDBC Driver.
IFX_ISOLATION_LEVEL	Defines the degree of concurrency among processes that attempt to access the same rows simultaneously. Gets the value of the**IFX_ISOLATION_LEVEL** variable, which is specific to Informix®. The default value is `2` (Committed Read). If the value has been set explicitly, it returns the set value. Returns: integer. Sets the value of the **IFX_ISOLATION_LEVEL** variable, which is specific to Informix®. Possible values: **0** Equivalent to `TRANSACTION_NONE` **1** Dirty Read (equivalent to `TRANSACTION_READ_UNCOMMITTED`), **2** Committed Read (equivalent to `TRANSACTION_READ_COMMITTED`), **3** Cursor Stability (equivalent to `TRANSACTION_READ_COMMITTED`), **4** Repeatable Read (equivalent to `TRANSACTION_REPEATABLE_READ`) **5** Committed Read LAST COMMITTED (equivalent to `TRANSACTION_LAST_COMMITTED`) **8** Equivalent to `TRANSACTION_SERIALIZABLE`

Supported Informix® environment variables	Description
	Specifying U after the mode means retain update locks. (See the **Important** note following table.) For example, a value could be: `2U` (equivalent to `SET ISOLATION TO COMMITTED READ RETAIN UPDATE LOCKS` The following example shows the code that you would use to specify an isolation level: `conn.setTransactionIsolation (IfxConnection.TRANSACTION_LAST_COMMITTED);`
IFX_FLAT_UCSQ	Overrides the global setting and directs the optimizer to use subquery flattening for all sessions. Default value is `1`.
IFX_LO_READONLY	Controls how smart large objects are opened by the database server during a query by a client application: **0** Default. Smart large objects are opened with read-write access. The database server locks the object until the object is closed by the client application or the transaction is completed. **1** Smart large objects are opened with read-only access. The database server does not lock the object during a query.
IFX_LOCK_MODE_WAIT	Application can use this property to override the default server process for accessing a locked row or table. Gets the value of the IFX_LOCK_MODE_WAIT variable, which is specific to Informix®. The default value is 0 (do not wait for the lock). If the value has been set explicitly, it returns the set value. Returns: integer. Sets the value of the IFX_LOCK_MODE_WAIT variable, which is specific to Informix®. Possible values: **-1** WAIT until the lock is released. **0** DO NOT WAIT, end the operation, and return with error. **nn** WAIT for nn seconds for the lock to be released.
IFX_PAD_VARCHAR	Controls how data associated with a VARCHAR data type is transmitted to and from the Informix® server. Can be set either on the connection

Supported Informix® environment variables	Description
	URL when using the Connection class or as a property when using the **DataSource** class. Valid values are 0 (the default) and `1`. • When set to 0, only the portion of the VARCHAR that contains data is transmitted (trailing spaces are stripped). • When set to 1, the entire VARCHAR data structure is transmitted to and from the server.
IFX_SET_FLOAT_AS_SMFLOAT	When set to `0` (the default), maps the JDBC FLOAT data type to the Informix® FLOAT data type. This setting conforms to the JDBC specification. When set to `1`, maps the JDBC FLOAT data type to the Informix® SMALLFLOAT data type. This setting enables compatibility with earlier versions of HCL Informix® JDBC Driver.
IFX_SOC_KEEPALIVE	When set to `true`, sets the TCP property SO_KEEPALIVE on the socket for open connections. This setting is useful to keep long running idle socket connections from timing out due to inactivity. Default is `false`.
IFX_TRIMTRAILINGSPACES	Removes trailing spaces. Default value is `1`.
IFX_USEPUT	When set to `1`, enables bulk inserts. For more information, see Perform bulk inserts on page 63.
IFX_XASPEC	When set to `y`, XA transactions with the same global transaction ID are tightly coupled and share the lock space. This only applies to XA connections and cannot be specified in a database URL. It can be specified by **DataSource** setter (See DataSource extensions on page 233.) or by setting a System (JVM) property with the same name. The **DataSource** property overrides the System property. Values for the properties other than `y`, `Y`, `n`, or `N` are ignored. IfxDataSource.getIfxIFX_XASPEC returns the final IFX_SPEC value, which is either `y` or `n`. For example if the value of DataSource **IFX_XASPEC** equals `n` and the value of the System **IFX_XASPEC** equals `Y` or `y`, `n` is returned.
IFX_XASTDCOMPLIANCE_XAEND	Specifies the behavior of XA_END when XA_RB* is returned. **0** XID is not forgotten. Transaction is in Rollback Only state. This is XA_SPEC+ compliant and is the default behavior with Informix®. **1** XID is forgotten. Transaction is Nonexistent.
IFXHOST	Sets the host name or host IP address

Supported Informix® environment variables	Description
IFXHOST_SECONDARY	Sets the secondary host name or host IP address for HDR connection redirection
INFORMIXCONRETRY	Specifies the maximum number of additional connection attempts that can be made to each database server by the client during the time limit specified by the value of **INFORMIXCONTIME**
INFORMIXCONTIME	Sets the timeout period for an attempt to connect to the database server. If a connection attempt does not succeed in this time, the attempt is aborted and a connection error is reported. The default value is `0` seconds. This variable adds timeouts for blocking socket methods and for socket connections.
INFORMIXOPCACHE	Specifies the size of the memory cache for the staging-area blobspace of the client application
INFORMIXSERVER	Specifies the default database server to which an explicit or implicit connection is made by a client application. **Note:** This parameter is optional for simple connections (connections that do not use a SQLHOST file) since 4.10.JC4.
INFORMIXSERVER_SECONDARY	Specifies the secondary database server in an HDR pair to which an explicit or implicit connection is made by a client application if the primary database server is unavailable.
INFORMIXSTACKSIZE	Specifies the stack size, in kilobytes, that the database server uses for a particular client session.
JDBCTEMP	Specifies where temporary files for handling smart large objects are created. You must supply an absolute path name.
LOBCACHE	Determines the buffer size for large object data that is fetched from the database server Possible values are: **A number greater than 0** The maximum number of bytes is allocated in memory to hold the data. If the data size exceeds the **LOBCACHE** value, the data is stored in a temporary file; if a security violation occurs during creation of this file, the data is stored in memory. **Zero** The data is always stored in a file. If a security violation occurs, the driver makes no attempt to store the data in memory.

Supported Informix® environment variables	Description
	A negative number The data is always stored in memory. If the required amount of memory is not available, an error occurs. If the **LOBCACHE** value is not specified, the default is 4096 bytes.
LOGINTIMEOUT	Determines whether the Informix® database server is running. If the server is running a connection to the server is established immediately. If the server is not running, this environment variable specifies how long, in milliseconds, that the server port is polled to establish a connection. If your application does not connect to the Informix® database server within the specified time, an error is returned.
METADATA_UPPERCASE_VALUES	Used to uppercase the ResultSet Metadata. When set to true, it returns ResultSet Metadata information in uppercase. This parameter is optional.
NEWNLSMAP	Allows new mappings to be defined between NLS and Java development kit locales and code sets. For more information, see User-defined locales on page 209.
NODEFDAC	Prevents the PUBLIC group from receiving table or routine privileges by default when new tables or UDRs are created in a database that was not created WITH LOG MODE ANSI. The required `yes` setting is case sensitive.
OPT_GOAL	Specifies the query performance goal for the optimizer. Set this variable in the user environment before you start an application. The driver does not use this variable; it just passes the value to the server.
OPTCOMPIND	Specifies the join method that the query optimizer uses
OPTOFC	When set to 1, the ResultSet.close() method does not require a network round trip if all the qualifying rows have already been retrieved in the clients tuple buffer. The database server automatically closes the cursor after all the rows have been retrieved. HCL Informix® JDBC Driver might not have additional rows in the clients tuple buffer before the next ResultSet.next() method is called. Therefore, unless HCL Informix® JDBC Driver has received all the rows from the database server, the ResultSet.close() method might still require a network round trip when **OPTOFC** is set to 1.
PATH	Specifies the directories that are to be searched for executable programs
PDQPRIORITY	Determines the degree of parallelism used by the database server
PLCONFIG	Specifies the name of the configuration file used by the high-performance loader

Supported Informix® environment variables	Description
PLOAD_LO_PATH	Specifies the path name for smart-large-object handles (which identify the location of smart large objects such as BLOB, CLOB, and BOOLEAN data types). The driver does not use this variable; it just passes the value to the server.
PORTNO_SECONDARY	Specifies the port number of the secondary database server in an HDR pair. The port number is listed in the `/etc/services` file.
PROXY	Specifies an HTTP proxy server. For more information, see An HTTP proxy server on page 44.
PSORT_DBTEMP	Specifies one or more directories to which the database server writes the temporary files it uses when performing a sort
PSORT_NPROCS	Enables the database server to improve the performance of the parallel-process sorting package by allocating more threads for sorting
SECURITY	Uses 56-bit encryption to send the password to the server. For more information, see Password encryption on page 50.
SQLH_TYPE	When set to FILE, specifies that database information (such as *host-name*, *port-number*, *user*, and *password*) is specified in an `sqlhosts` file. When set to LDAP, specifies that this information is specified in an LDAP server. For more information, see Dynamically reading the Informix sqlhosts file on page 32. **Note:** Starting Informix JDBC Driver version 4.50.JC10, the use of LDAP to retrieve Informix server connectivity information from a stored SQLHost files inside of an LDAP server has been removed.
SQLIDEBUG	Specifies the path name for the file to which a binary SQLI trace is to be written. A new trace file is generated for every connection and is suffixed with timestamp. Only use the SQLI trace facility when directed by the IBM® technical support representative.
SRV_FET_BUF_SIZE	Overrides the default setting for the size of the fetch buffer in distributed transactions with other database servers. That fetch buffer holds, for example, the data retrieved by a cross-server distributed query. For HCL Informix® 11.70.xC5 and later versions, the maximum value is 1048576 (1MiB).
SSLCONNECTION	When set to true, enables the connection to use SSL encryption for communication to the server.

Supported Informix® environment variables	Description
SSL_TRUSTSTORE	Specifies the location of the truststore to load by the JDBC driver.
SSL_TRUSTSTORE_PASSWORD	Specifies the password to the truststore that is being loaded by the JDBC driver.
STMT_CACHE	When set to 1, enables the use of the shared-statement cache in a session. This feature can reduce memory consumption and speed query processing among different user sessions. The driver does not use this variable; it just passes the value to the server.
TRUSTED_CONTEXT	When set to `TRUE`, a trusted connection request is sent from client. Either a successful trusted connection is established or the following error is returned from the server: `SQL Exception : -28021(Trusted Connection request rejected.)`

For a detailed description of a particular environment variable, see *HCL® Informix® Guide to SQL: Reference*.

Code example IFX_LOCK_MODE_WAIT environment variable

IFX_LOCK_MODE_WAIT

```
Connection conn = DriverManager.getConnection ( "jdbc:Informix-sqli://cleo:1550:
IFXHOST=cleo;PORTNO=1550;user=rdtest;password=my_passwd;IFX_LOCK_MODE_WAIT=1";);
```

Code example IFX_ISOLATION_LEVEL environment variable

IFX_ISOLATION_LEVEL

```
Connection conn = DriverManager.getConnection( "jdbc:Informix-sqli://cleo:1550:
IFXHOST=cleo;PORTNO=1550;user=rdtest;password=my_passwd;IFX_ISOLATION_LEVEL=1U";);
```

Important: The isolation property can be set in the URL only when it is an explicit connection to a database. For server-only connection, this property is ignored at connection time.

Code example IFX_ISOLATION_LEVEL environment variable

```
Connection conn = DriverManager.getConnection( "jdbc:informix-sqli://localhost:9088
/csdk_db:user=informix;password=inform123;LOGINTIMEOUT=60000";);
```

Dynamically reading the Informix® sqlhosts file

Note: Starting Informix JDBC Driver version 4.50.JC10, the use of LDAP to retrieve Informix server connectivity information from a stored SQLHost files inside of an LDAP server has been removed.

HCL Informix® JDBC Driver supports the JNDI (Java™ naming and directory interface). This support enables JDBC programs to access the Informix® `sqlhosts` file. The `sqlhosts` file lets a client application find and connect to the Informix®

database server anywhere on the network. For more information about this file, see the *HCL® Informix® Administrator's Guide* for your database server.

You can access `sqlhosts` data from a local file or from an LDAP server. The system administrator must load the `sqlhosts` data into the LDAP server using the Informix® utility.

Your **CLASSPATH** variable must reference the JNDI JAR (Java™ archive) files and the LDAP SPI (service provider interface) JAR files. You must use LDAP Version 3.0 or later, which supports the object class **extensibleObject**.

You can use the `sqlhosts` file **group** option to specify the name of a database server group for the value of `INFORMIXSERVER`. The **group** option is useful with High-Availability Data Replication (HDR); list the primary and secondary database servers in the HDR pair sequentially. For more information on about how to set or use groups in `sqlhosts` file, see the *HCL® Informix® Administrator's Guide*. For more information about HDR, see Connections to the servers of a high-availability cluster on page 36.

An unsigned applet cannot access the `sqlhosts` file or an LDAP server. For more information, see Using the driver in an applet on page 12.

Connection property syntax

Note: Starting Informix JDBC Driver version 4.50.JC10, the use of LDAP to retrieve Informix server connectivity information from a stored SQLHost files inside of an LDAP server has been removed.

You can let HCL Informix® JDBC Driver look up the host name or port number in an LDAP server instead of specifying them in a database URL or **DataSource** object directly. You must specify the following properties in the database URL or **DataSource** object for the LDAP server:

- **SQLH_TYPE**=LDAP
- **LDAP_URL**=ldap://*host-name*:*port-number*

  host-name and *port-number* are those of the LDAP server, not the database server.
- **LDAP_IFXBASE**=*Informix-base-DN*
- **LDAP_USER**=*user*
- **LDAP_PASSWD**=*password*

If **LDAP_USER** and **LDAP_PASSWD** are not specified, HCL Informix® JDBC Driver uses an anonymous search to search the LDAP server. The LDAP administrator must make sure that an anonymous search is allowed on the `sqlhosts` entry. For more information, see your LDAP server documentation.

Informix-base-DN has the following basic format:

```
cn=common-name,o=organization,c=country
```

If *common-name*, *organization*, or *country* consists of more than one word, you can use one entry for each word. For example:

```
cn=informix,cn=software
```

Here is an example database URL:

```
jdbc:informix-sqli:informixserver=value;SQLH_TYPE=LDAP;
   LDAP_URL=ldap://davinci:329;LDAP_IFXBASE=cn=informix,
   cn=software,o=kmart,c=US;LDAP_USER=abcd;LDAP_PASSWD=secret
```

You can also specify the `sqlhosts` file in the database URL or **DataSource** object. The host name and port number or the service name of the Informix® database server as specified in the `/etc/services` file are read from the `sqlhosts` file. You must specify the following properties for the file:

- **SQLH_TYPE**=FILE
- **SQLH_FILE**=*sqlhosts-filename*

The `sqlhosts` file can be local or remote, so you can refer to it in the local file system format or URL format. Here are some examples:

- `SQLH_FILE=http://host-name:port-number/sqlhosts.iusSQLH_FILE=http://host-name:service-name/sqlhosts.ius`

  The *host-name* and *port-number* or *service-name* of the Informix® database server (from the `etc/services` file) elements are those of the server on which the `sqlhosts` file resides.
- `SQLH_FILE=file://D:/local/myown/sqlhosts.ius`
- `SQLH_FILE=/u/local/sqlhosts.ius`

Here is an example database URL:

```
jdbc:informix-sqli:informixserver=value;SQLH_TYPE=FILE;
   SQLH_FILE=/u/local/sqlhosts.ius
```

If the database URL or **DataSource** object references the LDAP server or `sqlhosts` file but also directly specifies the IP address, host name, and port number, then the IP address, host name, and port number specified in the database URL or **DataSource** object take precedence. For information about how to set these connection properties by using a **DataSource** object, see DataSource extensions on page 233.

If you are using an applet or the database is behind a firewall, an HTTP proxy servlet, running in an extra tier, is required for communication. See An HTTP proxy server on page 44 for more information.

Administration requirements

Note: Starting Informix JDBC Driver version 4.50.JC10, the use of LDAP to retrieve Informix server connectivity information from a stored SQLHost files inside of an LDAP server has been removed. Hence, SqlhUpload and SqlhDelete utilities are also not supported.

If you want the LDAP server to store `sqlhosts` information that a JDBC program can look up, the following requirements must be met:

- The LDAP server must be installed on a computer that is accessible to the client. The LDAP administrator must create an **IFXBASE** entry in the LDAP server.

  For more information about LDAP directory servers, see:
  - www.oracle.com
  - www.openldap.org
- If you want to use the HCL Informix® SqlhUpload and SqlhDelete utilities, which can load or delete the `sqlhosts` entries from a flat ASCII file, the **servicename** field in the `sqlhosts` file must specify the database servers port number. For more information, see Utilities to update the LDAP server with sqlhosts data on page 35, next.
- The LDAP administrator must make sure that anonymous search is allowed on the `sqlhosts` entry. For more information, see the LDAP server documentation.

Utilities to update the LDAP server with sqlhosts data

Note: Starting Informix JDBC Driver version 4.50.JC10, the use of LDAP to retrieve Informix server connectivity information from a stored SQLHost files inside of an LDAP server has been removed. Hence, SqlhUpload and SqlhDelete utilities are not supported.

The SqlhUpload and SqlhDelete utilities are packaged in `ifxtools.jar`, so the **CLASSPATH** variable must point to `ifxtools.jar` (which, by default, is in the `lib` directory under the installation directory for HCL Informix® JDBC Driver). Make sure that the **CLASSPATH** variable also points to the JNDI JAR files and LDAP SPI JAR files.

The SqlhUpload utility

This utility loads the `sqlhosts` entries from a flat ASCII file to the LDAP server in the prescribed format.

Note: The SqlhUpload utility is not supported starting 4.50.JC10 .

Enter the following command:

```
java SqlhUpload sqlhfile.txt host-name:port-number [sqlhostsRdn]
```

The parameters have the following meanings:

- The `sqlhosts` file to be uploaded is `sqlhfile.txt`.
- The host name and port number of the LDAP server is *host-name*:*port-number*.
- The RDN (relative distinguished name) of the `sqlhosts` node under the HCL Informix® base in LDAP is *sqlhostsRdn*. The default name is `sqlhosts`.

The utility prompts for other required information, such as the Informix® base distinguished name (DN) in the LDAP server, the LDAP user, and the password.

You must convert the **servicename** field in the `sqlhosts` file to a string that represents an integer (the port number), because the **Java™.Socket** class cannot accept an alphanumeric **servicename** value for the port number. For more information about the **servicename** field, see the *HCL® Informix® Administrator's Guide* for your database server.

The SqlhDelete utility

Note: The SqlhDelete utility is not supported starting 4.50.JC10 .

This utility deletes the `sqlhosts` entries from the LDAP server. Enter the following command:

```
java SqlhDelete host-name:port-number [sqlhostsRdn]
```

The parameters of this command have the same meanings as the parameters listed for the SqlhUpload utility. See The SqlhUpload utility on page 35.

The utility prompts for other required information, such as the HCL Informix® base DN in the LDAP server, the LDAP user, and the password.

Connections to the servers of a high-availability cluster

Using the JDBC driver, Java™ applications can connect to HCL Informix® database servers in a high-availability cluster. Java™ applications can also connect to HCL Informix® Connection Managers, which can handle failover for high-availability clusters and redirect connections to cluster servers.

To connect your Java™ application to the servers of a high-availability cluster, you must set properties in the connection URL or DataSource. If the application performs update operations on secondary servers, configure the application to initially check for read-only server status.

When you configure HCL Informix® Connection Managers to handle connections between your Java™ application server and high-availability cluster, you get the following benefits:

- You can direct connection requests to the most appropriate secondary server through rule-based redirection policies.
- You can manage failover for your high-availability clusters, automatically promoting a secondary server to the role of primary server if the primary server fails.
- You can prioritize connections between a specific application server and the primary server of your high-availability cluster when you install and configure HCL Informix® Connection Managers on the same hosts as your Java™ application servers.
- When database servers are behind a firewall, HCL Informix® Connection Managers can act as proxy servers, and handle client/server communication.

You can use high-availability secondary servers with connection pooling. For more information, see High-Availability Data Replication with connection pooling on page 222.

Demonstration programs are available in the `hdr` directory within the `demo` directory where HCL Informix® JDBC Driver is installed. For details about the files, see Sample code files on page 224.

Related information

Connection management through the Connection Manager on page

High-availability cluster configuration on page

Properties for connecting to high-availability cluster servers through HCL Informix® Connection Managers

A JDBC application can connect to Connection Manager, just as the application might connect to a database server. Application connection requests are then redirected to the most appropriate server in a high-availability cluster.

You can configure multiple Connection Managers, and then create a Connection Manager group entry in `sqlhost` file that is used by the Java™ application server. If one Connection Manager fails, connection requests can be directed to working Connection Managers. The SQLH_FILE connection property directs the JDBC driver to search for group entries.

To connect to the HCL Informix® Connection Manager that then connects to the servers of a high-availability cluster, you must include the following properties in the connection URL or DataSource:

```
INFORMIXSERVER=CM_or_group_name
SQLH_TYPE=FILE
SQLH_FILE=sqlhosts
USER=user_name
PASSWORD=password
```

Include the following properties in the connection URL to prevent your Java™ applications from waiting indefinitely if a Connection Manager is running, but has a hung connection.

```
INFORMIXCONRETRY=value
INFORMIXCONTIME=value
LOGINTIMEOUT=value
```

The values are set based on the network environment.

Example

Example 1: Connecting to a high-availability cluster through the HCL Informix® Connection Manager

In this example, you have the following system setup:

- You have a high-availability cluster (**my_cluster**) that is composed of four servers.
- The user name on all cluster servers is **my_user**.
- The password on all cluster servers is **my_password**.
- **connection_manager**, on **cmhost1.example.com** uses the following configuration file:

```
NAME connection_manager

CLUSTER my_cluster
{
   INFORMIXSERVER my_servers
   SLA sla_primary     DBSERVERS=PRI
   SLA sla_secondaries DBSERVERS=SDS,HDR,RSS
```

```
   FOC ORDER=ENABLED \
       PRIORITY=1
}
```

- You have a Java™ application server on **host1.example.com**, and the Java™ application server uses the following `sqlhost` file entries:

```
#dbservername       nettype    hostname             servicename  options
 sla_primary      onsoctcp   cmhost1.example.com  cm_port_1
 sla_secondaries  onsoctcp   cmhost1.example.com  cm_port_1
```

- If the initial connection attempt by the client fails, you want it to retry two times.
- You want the CONNECT statement to wait 10 seconds to establish a connection.
- You want the connection to fail if the server port is polled and does not connect within 10 milliseconds.

To connect the Java™ application client to the primary server of **my_cluster**, use the following URL:

```
jdbc:informix-sqli://INFORMIXSERVER=sla_primary;
   SQLH_TYPE=FILE;SQLH_FILE=sqlhosts;
   USER=my_user_name;PASSWORD=my_password;
   INFORMIXCONRETRY=2;INFORMIXCONTIME=10;LOGINTIMEOUT=10
```

To connect the Java™ application client to a secondary server of **my_cluster**, use the following URL:

```
jdbc:informix-sqli://INFORMIXSERVER=sla_secondaries;
   SQLH_TYPE=FILE;SQLH_FILE=sqlhosts;
   USER=my_user_name;PASSWORD=my_password;
   INFORMIXCONRETRY=2;INFORMIXCONTIME=10;LOGINTIMEOUT=10
```

Example

Example 2: Connecting to a high-availability cluster through HCL Informix® Connection Managers

In this example, you have the following system setup:

- You have a high-availability cluster (**my_cluster**) that is composed of four servers.
- The user name on all cluster servers is **my_user**.
- The password on all cluster servers is **my_password**.
- **connection_manager_1**, on **cmhost1.example.com** uses the following configuration file for client redirection and failover:

```
NAME connection_manager_1

CLUSTER my_cluster
{
   INFORMIXSERVER my_servers
   SLA sla_primary_1 DBSERVERS=PRI
   FOC ORDER=ENABLED \
       PRIORITY=1
   CMALARMPROGRAM $INFORMIXDIR/etc/CMALARMPROGRAM.sh
}
```

- **connection_manager_2**, on **cmhost2.example.com** uses the following configuration file for client redirection and failover:

```
NAME connection_manager_2

CLUSTER my_cluster
{
   INFORMIXSERVER my_servers
   SLA sla_primary_1 DBSERVERS=PRI
   FOC ORDER=ENABLED \
       PRIORITY=2
   CMALARMPROGRAM $INFORMIXDIR/etc/CMALARMPROGRAM.sh
}
```

- You have a Java™ application server on **host1.example.com**, and the Java™ application server uses the following `sqlhost` file entries:

```
#dbservername       nettype    hostname             servicename  options
 g_primary          group      -                    -            c=1,e=sla_primary_2
 sla_primary_1      onsoctcp   cmhost1.example.com  cm_port_1    g=g_primary
 sla_primary_2      onsoctcp   cmhost2.example.com  cm_port_2    g=g_primary
```

- If the initial connection attempt by the client fails, you want it to retry two times.
- You want the CONNECT statement to wait 10 seconds to establish a connection.
- You want the connection to fail if the server port is polled and does not connect within 10 milliseconds.

To connect the Java™ application client to the primary server of **my_cluster** through either **connection_manager_1** or **connection_manager_2**, use the following URL:

```
jdbc:informix-sqli://INFORMIXSERVER=g_primary;
   SQLH_TYPE=FILE;SQLH_FILE=sqlhosts;
   USER=my_user_name;PASSWORD=my_password;
   INFORMIXCONRETRY=2;INFORMIXCONTIME=10;LOGINTIMEOUT=10
```

Related information

Connection management through the Connection Manager on page

The sqlhosts information on page

High-availability cluster configuration on page

Properties for connecting to high-availability cluster servers through SQLHOST file group entries

You can define `sqlhost` group entries, so that your application connection attempt is always directed to the primary server of a high-availability cluster, even if failover occurs.

To connect to the primary server of a high-availability cluster, include the following properties in the connection URL or DataSource:

```
INFORMIXSERVER=group_name
SQLH_TYPE=FILE
SQLH_FILE=sqlhosts
USER=user_name
PASSWORD=password
```

An exception is thrown if the JDBC driver cannot find a primary server in the group.

Enforcing connections to the primary server is enabled for HCL Informix®, Version 9.40.xC6 and later only.

Example

Example: Connecting to the primary server of a high-availability cluster through SQLHOST file group entries

In this example, you have the following system setup:

- You have a high-availability cluster (**my_cluster**) that is composed of four servers:
  - **server_1** (primary), on **host1.example.com**
  - **server_2** (shared-disk secondary), on **host1.example.com**
  - **server_3** (HDR), on **host2.example.com**
  - **server_4** (Remote-standalone secondary), on **host3.example.com**
- The user name on all cluster servers is **my_user**.
- The password on all cluster servers is **my_password**.
- You have a Java™ application server on **host4.example.com**. The server uses the following `sqlhost` file entries:

```
#dbservername   nettype    hostname            servicename   options
 my_servers     -          -                                 c=1,e=server_4
 server_1       onsoctcp   host1.example.com   port_1        g=my_servers
 server_2       onsoctcp   host1.example.com   port_2        g=my_servers
 server_3       onsoctcp   host2.example.com   port_3        g=my_servers
 server_4       onsoctcp   host3.example.com   port_4        g=my_servers
```

To connect the Java™ application client to the primary server of **my_cluster**, use the following URL:

```
jdbc:informix-sqli://INFORMIXSERVER=my_servers;
   SQLH_TYPE=FILE;SQLH_FILE=sqlhosts;
   USER=my_user_name;PASSWORD=my_password
```

Related information

The sqlhosts information on page

High-availability cluster configuration on page

Properties for connecting directly to an HDR pair of servers

You can define your client application's connection URL or DataSource so that your application connects directly to an HDR pair of servers. If a connection attempt to the primary server fails, the client application can attempt to connect to the HDR secondary server.

To connect directly to a primary server and HDR secondary server, include the following properties in the connection URL or DataSource:

```
INFORMIXSERVER=primary_server_name
INFORMIXSERVER_SECONDARY=secondary_server_name
```

```
IFXHOST_SECONDARY=secondary_host_name
PORTNO_SECONDARY=secondary_port_number
USER=user_name
PASSWORD=password
ENABLE_HDRSWITCH=true
```

If you are setting values in the DataSource, you must also include the following values:

```
IFXHOST=primary_host_name
PORTNO=primary_port_number
```

When you are using a **DataSource** object, you can set and get the secondary server connection properties with setXXX() and getXXX() methods. These methods are listed with their corresponding connection property in Get and set Informix connection properties on page 235.

You can manually redirect a connection to the secondary server in an HDR pair by editing the INFORMIXSERVER, PORTNO, and IFXHOST properties in the DataSource or by editing the INFORMIXSERVER property in the URL. Manual redirection requires editing the application code and then restarting the application.

Example

Example: Connecting to an HDR pair of servers

The following example shows a connection URL for a primary server that is named **server_1** and an HDR secondary server that is named **server_2**:

```
jdbc:informix-sqli://my_host:my_port/my_database:
   INFORMIXSERVER=server_1;INFORMIXSERVER_SECONDARY=server_2;
   IFXHOST_SECONDARY=host2.example.com;PORTNO_SECONDARY=port_2;
   user=my_name;password=my_password;
   ENABLE_HDRSWITCH=true
```

Related information

The sqlhosts information on page

High-availability cluster configuration on page

Checks for read-only status of high-availability secondary servers

You can write applications to check for read-only server status, so that update operations are not attempted on read-only secondary servers.

The HCL Informix® JDBC driver has extension methods to the java.sql.Connection class that provide a way to check the HDR secondary server's status. Users can type cast connection objects to 'com.informix.jdbc.IfmxConnection' to access the following extension methods.

Information obtained	Method signature	Additional information
Whether the server is read-only (a secondary server)	public boolean is ReadOnly() throws SQLException	Returns true if the active server is a secondary server

Information obtained	Method signature	Additional information
		Returns an exception if a database access error occurs If ENABLE_HDRSWITCH is set to false, isReadOnly() returns the value that is initially set after the last successful HDR connection was obtained.
Whether HDR is enabled	public boolean is HDREnabled()	Returns true if both servers in the HDR pair are available Returns false if one of the servers is unavailable
The type of the server (primary, secondary, or standard)	public string getHDRtype()	Returns primary or standard for a primary server, secondary for a secondary server The database administrator can manually reset the type of the server.

For example, you can use one of the following strategies:

- Use the isReadOnly() method before each SQL statement that might contain an update operation. If the value of isReadOnly() is `true`, perform an appropriate action, such as sending an error message to the user or notifying the server administrator.
- You call the isReadOnly() method after you establish a connection and then set a flag, like READ_ONLY, and then perform operations that are based on the flag value.

An administrator can manually switch a secondary server to a primary server to allow update operations. However, the server must be shut down in the process, which can cause uncommitted transactions to be lost.

Related information

High-availability cluster configuration on page

Connection retry attempts to HDR secondary servers

You can write applications so that if a connection is lost during query operations, HCL Informix® JDBC Driver returns a new connection to the secondary database server and the application reruns the queries.

The following example shows how to retry a connection with the secondary server information, and then rerun an SQL statement that received an error because the primary server connection failed:

```
public class HDRConnect {
  static IfmxConnection conn;

  public static void main(String[] args)
  {
     getConnection(args[0]);
     doQuery( conn );
     closeConnection();
  }

  static void getConnection( String url )
  {
        ..
     Class.forName("com.informix.jdbc.IfxDriver");
     conn = (IfmxConnection )DriverManager.getConnection(url);

  }
  static void closeConnection()
  {
     try
     {
         conn.close();
     }
     catch (SQLException e)
     {
          System.out.println("ERROR: failed to close the connection!");
          return;
     }
  }
  static void doQuery( Connection con )
  {
     int rc=0;
     String cmd=null;
     Statement stmt = null;

     try
      {
          // execute some sql statement
      }
     catch (SQLException e)
       {
          if (e.getErrorCode() == -79716 ) || (e.getErrorCode() == -79735)
          // system or internal error
       {
          // This is expected behavior when primary server is down
          getConnection(url);
          doQuery(conn);
       }
       else
         System.out.println("ERROR: execution failed - statement: " + cmd);
       return;
       }
  }
```

Related information

High-availability cluster configuration on page

An HTTP proxy server

Network security imposes certain restrictions on what client applications are allowed to do:

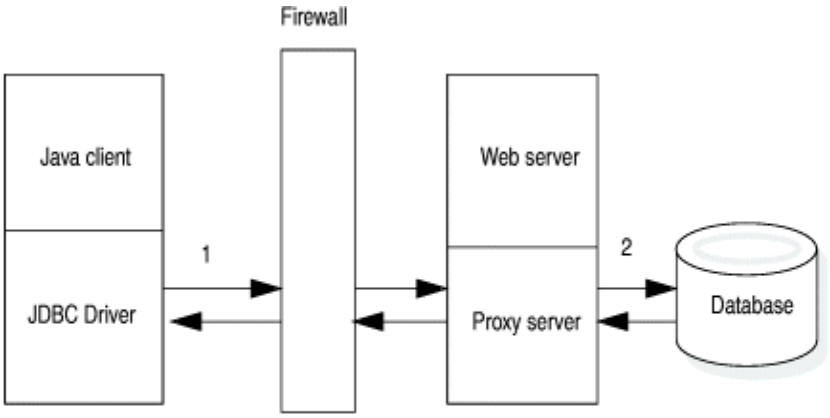
- Applets can only communicate back to the host from which they were downloaded.
- Direct IP connections between a JDBC client and database are not allowed when a firewall is between the client and the database server.

The HCL Informix® HTTP proxy handles both of these problems. The proxy is a servlet that runs in the middle tier between a JDBC client and the Informix® database server. The proxy extracts SQL requests from the JDBC client and transmits them to the database server. The client (the end user) is unaware of this middle tier.

The HTTP proxy feature is not part of the JDBC 2.0 specification.

The following figure illustrates how the proxy enables a connection to a database that is behind a firewall.

Figure 1. Connecting through a firewall



1 The driver sends the target IP address and port number to the proxy

2 The proxy uses the IP address and port to open a connection to the database.

Configuring your environment to use a proxy server

About this task

The HTTP proxy requires a web server that supports servlets, preferably a web server whose servlet engine uses a 2.1 or greater servlet API. The proxy is compatible with 2.0 and earlier servlet APIs, but the PROXYTIMEOUT feature is only enabled with a 2.1 or greater API.

To configure your environment for a proxy server:

1. Define a servlet alias or context for the proxy servlet in your web server configuration.

   The JDBC driver directs all client HTTP requests to:

   ```
   http://your-web-server:port/pathname/IfxJDBCProxy
   ```

   where *IfxJDBCProxy* is the proxy servlet and *pathname* is the path to the proxy servlet. Consult your web server documentation for the correct way to configure servlets.
2. Copy three class files—`IfxJDBCProxy.class`, `SessionMgr.class`, and `TimeoutMgr.class`—to the servlet directory you specified in the previous step.

   These class files reside in the directory `proxy`, which is under the installation directory for HCL Informix® JDBC Driver after the product bundle is installed.
3. Add the HCL Informix® JDBC Driver file, `ifxjdbc.jar`, to the CLASSPATH setting on your web server.

   Some web servers use the CLASSPATH of the environment under which the server is started, while others get their CLASSPATH from a web server-specific properties file. Consult your web server documentation for the correct place to update the CLASSPATH setting.
4. Start your web server and verify that the proxy is installed correctly by entering the following URL:

   ```
   http://server-host-name:port-number/servlet/
      IfxJDBCProxy
   ```

   The proxy replies with the following banner:

   ```
   -- Informix Proxy Servlet v220 Servlet API 2.1 --
   ```

   `v220` represents the Informix® proxy version. `Servlet API 2.1` represents the version of your web servers servlet API.

   If the servlet API is 2.0 or earlier, the banner says `Servlet API 0.0`.
5. After configuring the proxy, append the following to your applet or applications URL:

   ```
   PROXY=server-host-name:port-number
   ```

   Example

   For example:

   ```
   jdbc:informix-sqli://123.45.67.89:1533:INFORMIXSERVER=
      myserver;user=rdtest;password=test;
      PROXY=webserver:1462;
   ```

Results

Depending on your web server, the proxy servlet might be loaded when the web server is started or the first time it is referenced in the URL of your applet or application connection object.

The following websites offer more information about proxy servlets:

- www.oracle.com
- java.apache.org

Specify a timeout

You can specify a timeout value for the proxy by using the PROXYTIMEOUT keyword. The PROXYTIMEOUT value specifies how often the client-side JDBC driver sends a **keepalive** request to the proxy. A PROXYTIMEOUT value is represented in seconds; the value can be 60 or greater.

When PROXYTIMEOUT is specified by the client, the proxy sets the clients session expiration equal to `2 x PROXYTIMEOUT`. For example, if PROXYTIMEOUT is set to 60 seconds, the proxy sets the clients expiration time to 120 seconds. When the expiration time is reached, the proxy removes the clients session resources and closes its database connection.

The proxy resets the timeout interval each time a communication is received from the client. Here are some valid values for PROXYTIMEOUT:

`PROXYTIMEOUT=-1`

Disables the client timeout feature.

`PROXYTIMEOUT=nnn`

Client sends a **keepalive** request to proxy every *nnn* seconds. The *nnn* value must be 60 or greater.

`PROXYTIMEOUT=60`

Default value if PROXYTIMEOUT is not specified

The proxy timeout feature is helpful in determining if a client session has terminated without first sending the proxy a close request by closing the JDBC connection. The proxy maintains an open database connection on behalf of the client until the client either:

- Explicitly closes the database connection
- Exceeds its timeout interval

The onstat database utility shows an open session for any client sessions that have unexpectedly terminated and have set PROXYTIMEOUT to `-1`.

Here is an example that specifies PROXYTIMEOUT:

```
jdbc:informix-sqli://123.45.67.89:1533:informixserver=myserver;
   user=rdtest;password=test;
   PROXY=webserver:1462?PROXYTIMEOUT=180;
```

See the `demo/proxy` directory under the directory where your driver is installed for an example applet and application that uses the proxy.

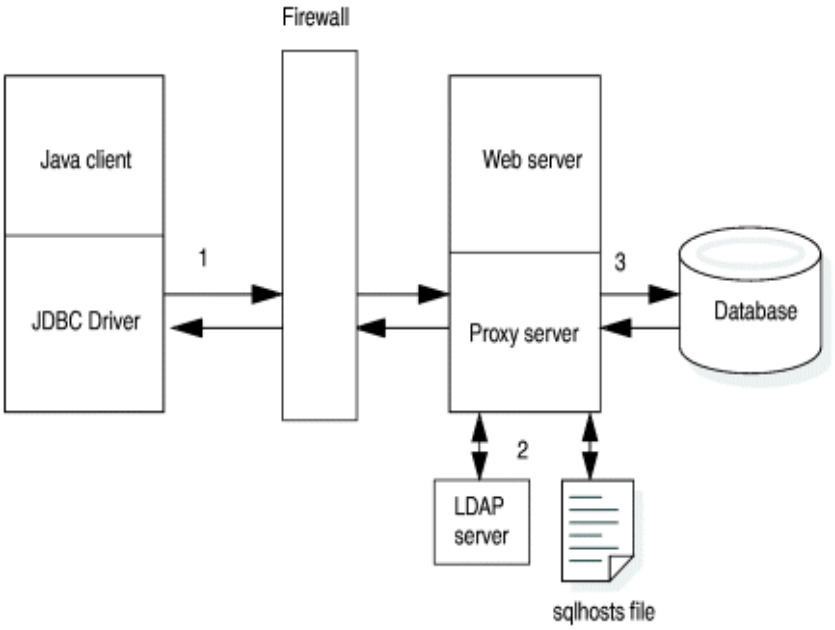
The proxy with an LDAP server

Note: Starting Informix JDBC Driver version 4.50.JC10, the use of LDAP to retrieve Informix server connectivity information from a stored SQLHost files inside of an LDAP server has been removed.

The proxy allows your JDBC applets and applications to alternatively get their database connection information from an LDAP server. If you plan to use this feature, you need to install an LDAP server. For general information about using an LDAP server with HCL Informix® JDBC Driver, see the topics beginning with Connection property syntax on page 33.

The following figure illustrates how the proxy works with an LDAP server. The figure also shows lookup from an `sqlhosts` file; for more information, see Specify sqlhosts file lookup on page 48.

Figure 2. Lookup by the proxy



1 The driver sends the LDAP or sqlhosts values to the proxy

2 The proxy gets the IP address and port from either the LDAP server or the sqlhosts file.

3 The proxy uses the IP address and port to open a connection to the database.

The proxy LDAP feature requires the JNDI class libraries and LDAP service provider files (`jndi.jar`, `ldap.jar`, and `providerutil.jar`). These JAR files can be downloaded from www.oracale.com.

After downloading and installing the files, add their full path names to the CLASSPATH setting on your web server. The files are in the `lib` directory under the installation directory.

Specify where LDAP lookup occurs

Note: Starting Informix JDBC Driver version 4.50.JC10, the use of LDAP to retrieve Informix server connectivity information from a stored SQLHost files inside of an LDAP server has been removed.

When used with other LDAP keywords, the SQLH_LOC keyword indicates where an LDAP lookup occurs.

SQLH_LOC can have a value of either `CLIENT` or `PROXY`. If the value is `CLIENT`, the driver performs the LDAP lookup on the client side. If the value is `PROXY`, the proxy performs the lookup on the server side. If no value is specified, the driver uses `CLIENT` as the default value.

Here is the format for an applet or application URL with LDAP keywords that specifies a server-side LDAP lookup:

```
jdbc:informix-sqli:informixserver=informix-server-name;
PROXY=proxy-hostname-or-ip-address:proxy-port-no?
PROXYTIMEOUT=60;SQLH_TYPE=LDAP;LDAP_URL=ldap:
//ldap-hostname-or-ip-address:ldap-port-no;LDAP_IFXBASE=dc=mydomain,dc=com;
SQLH_LOC=PROXY;
```

This example obtains the database server host name and port from an LDAP server:

```
jdbc:informix-sqli:informixserver=samsara;SQLH_TYPE=LDAP;
LDAP_URL=ldap://davinci:329;LDAP_IFXBASE=cn=informix,
o=kmart,c=US;LDAP_USER=abcd;LDAP_PASSWD=secret;SQLH_LOC=PROXY;
PROXY=webserver:1462
```

For a complete example of using an LDAP server with the proxy, see the `proxy` applet and application in the `demo` directory where your JDBC driver is installed.

Specify sqlhosts file lookup

The SQLH_LOC keyword also applies to `sqlhosts` file lookups when you are using the proxy. If the URL includes SQLH_LOC =PROXY, the driver reads the `sqlhosts` file on the server. If SQLH_LOC =PROXY is not specified, the driver reads the file on the client.

This example obtains the information from an `sqlhosts` file on the server:

```
jdbc:informix-sqli:informixserver=samsara;SQLH_TYPE=FILE;
   SQLH_FILE=/work/9.x/etc/sqlhosts;SQLH_LOC=PROXY;
   PROXY=webserver:1462
```

Other multitier solutions

Other ways to use HCL Informix® JDBC Driver in a multiple-tier environment are as follows:

Remote Method Invocation (RMI)

HCL Informix® JDBC Driver resides on an application server that is a middle tier between the Java™ applet or application and Informix® database machines. An example of RMI is included with HCL Informix® JDBC Driver; see Sample code files on page 224, for details.

Other communication protocols, such as CORBA

HCL Informix® JDBC Driver resides on an application server that is a middle tier between the Java™ applet or application and Informix® database computers.

Encryption options

You can use either password (SECURITY=PASSWORD) or network encryption to establish the security of your connection. To use either the password option or to use network encryption, you must have a Java™ Cryptography Extension (JCE)-compliant encryption services provider installed in your Java™ runtime environment.

It is recommended that you do not mix security packages on the same client. The following topics describe how to configure each package.

Encryption over the network and password encryption should not be used together. Thus, password encryption should not be enabled with the SECURITY environment variable when using JDBC encryption CSM. JDBC Encryption CSM does encrypt passwords before sending them over the network.

Note: Support for Communication Support Module (CSM) is removed starting Informix Server 14.10.xC9 . You should use Transport Layer Security (TLS)/Secure Sockets Layer (SSL) instead.

FIPS-compliant security package

IBM® SDK, Java™ Technology Edition includes a Federal Information Processing Standards (FIPS) 140-2 compliant package for JCE (Java™ Cryptographic Extension) called IBMJCEFIPS. IBMJCEFIPS supports FIPS-approved cryptographic operations through Java™ APIs.

The IBMJCEFIPS package can be used with the simple password CSM or with the encryption CSM.

To use the FIPS package, add the IBMJCEFIPS provider to the list of security providers in the Java™ virtual machine file `java.security`, which is in the `jre/lib/ext` directory where the Java™ runtime environment is installed

You must specify the IBMJCEFIPS provider at a higher preference order than any non-FIPS security providers in the `java.security` file. The order is 1-based, meaning that 1 is the most preferred, followed by 2, and so on.

For example:

```
security.provider.1=com.ibm.crypto.fips.provider.IBMJCEFIPS
security.provider.2=com.ibm.crypto.fips.provider.IBMJCE
```

Make sure that the IBMJCEFIPS has a higher preference order than the IBMJCE provider.

No changes to applications are needed for the HCL Informix® JDBC Driver to use the FIPS-compliant cryptographic package.

The certified JCE FIPS guide (http://csrc.nist.gov/groups/STM/cmvp/documents/140-1/140sp/140sp497.pdf) contains more information about the security policy that is provided by the cryptographic module, and describes how the module is designed to meet FIPS 140-2 compliance.

Password encryption

The **SECURITY** environment variable specifies the security operations that are performed when the Informix® JDBC client and Informix® database server exchange data. The only setting for the **SECURITY** environment variable supported in HCL Informix® JDBC Driver is `PASSWORD`.

If `PASSWORD` is specified, the user-provided password is encrypted using 56-bit encryption when it is passed from the client to the database server. There is no default setting.

Here is an example:

```
String URL = "jdbc:informix-sqli://158.58.10.171:1664:user=myname;
password=mypassord;INFORMIXSERVER=myserver;SECURITY=PASSWORD";
```

`PASSWORD` is not case sensitive.

Configure the database server

If the SECURITY=PASSWORD setting is specified in the Informix® JDBC client, the **SPWDCSM csm** option must be enabled on the Informix® database server. Otherwise, an error is returned during connection.

To use the **SPWDCSM csm** server option, which supports password encryption on the database server, you must configure the servers `sqlhosts` server name option. After this option is set on the server, only clients using the SECURITY=PASSWORD setting can connect to that server name.

Related information

Simple password encryption on page

Connecting JDBC applications with SSL

You can configure database connections for the HCL Informix® JDBC Driver to use the Secure Sockets Layer (SSL) protocol.

Before you begin

The client must use the same public key certificate file as the server.

1. **Create a truststore:** Use the **keytool** utility that comes with your Java™ runtime environment to import a client-side keystore database and add the public key certificate to the keystore.

   ```
   C:\work>keytool -importcert -file filename.extension -keystore .keystore
   ```

   Follow the prompts to enter a new keystore password and to trust the certificate.
2. **Define the truststore location:** Configure an SSL connection to the database from your Java™ application by using the following options:

   Option 1: Use system properties

   Set the location and password of the truststore using Java system properties.

Note: These settings apply to all the SSL connections made from this application.

```
C:\work>java -D javax.net.ssl.trustStore=/opt/ids/.keystore -D javax.net.ssl.trustStorePassword=password
 -jar yourapplication.jar
```

or set the location and password inside the java code using the System.setProperty API.

```
System.setProperty("javax.net.ssl.trustStore", "/opt/ids/.keystore");
 System.setProperty("javax.net.ssl.trustStorePassword", "password");
```

Option 2: Use a DataSource object

Define "per connection" the truststore location and password using a DataSource object by using the setTrustStore and setTrustStorePassword methods on the IfxDataSource object.

```
IfxDataSource cds = new IfxDataSource();
cds.setTrustStore("/opt/ids/.keystore");
cds.setTrustStorePassword("password");
//Add your additional connection details
```

Option 3: Pass in through the connection URL

If you do not use a DataSource object you can pass in the truststore and password via URL properties using SSL_TRUSTSTORE=/opt/ids/.keystore and SSL_TRUSTSTORE_PASSWORD=password

```
Connection c = DriverManager.getConnection("jdbc:informix-sqli:localhost:9089/mydatabase:
SSL_TRUSTSTORE=/opt/ids/.keystore;SSL_TRUSTSTORE_PASSWORD=password
```

3. **Declare the connection for SSL**: This is set per connection and can be done through the DataSource or the URL.

Option 1: Use a DataSource object

```
IfxDataSource cds = new IfxDataSource();
cds.setIfxSSLConnection("true");
```

Option 2: Pass in through the connection URL

```
Connection c = DriverManager.getConnection("jdbc:informix-sqli:localhost:9089/mydatabase:
 SSLCONNECTION=true
```

Example

JDBC sample for SSL connection

This sample Java™ program highlights the operations that are required to connect to the stores_demo database by using SSL.

```
import java.sql.Connection;
import java.sql.SQLException;

import com.informix.jdbc.IfxDriver;
import com.informix.jdbcx.IfxDataSource;

public class InformixSSLConnectionExample {
 public static void main(String[] args) {
```

```
  /* System properties for keystore */
  /* you can set this here for your whole system or you can set on */
  /* the data sourc (show below) or directly on your connection */
  /* properties using SSL_TRUSTSTORE and SSL_TRUSTSTORE_PASSWORD */
  System.setProperty("javax.net.ssl.trustStore", "/opt/ids/.keystore");
  System.setProperty("javax.net.ssl.trustStorePassword", "password");

  /* Instantiate Informix connection pooled data source */
  IfxDataSource cds = new IfxDataSource();

  /*
   * Set SSLConnection property to true and port pointing to SSL port on the
   * server
   */
  cds.setUser("dbuser");
  cds.setPassword("password");
  cds.setDatabaseName("stores_demo");
  cds.setPortNumber(9888);

  /* Enable SSL */
  cds.setIfxSSLCONNECTION("true");

  /* Optional if you don't set a system property */
  /* You can set the trust store and password in the data source */
  cds.setTrustStore("/opt/ids/.keystore");
  cds.setTrustStorePassword("password");

  try (Connection conn = cds.getConnection()) {
   System.out.println(" Successfully connected to Informix database using SSL Connection");
   System.out.println(" Database version  ...: " + conn.getMetaData().getDatabaseProductVersion());
   System.out.println(" JDBC Driver Version .: " + IfxDriver.getJDBCVersion());
  } catch (SQLException e) {
   System.err.println("Error Message : " + e.getMessage());
   System.err.println("Error Code    : " + e.getErrorCode());
  } catch (Exception e) {
   System.err.println("Error Message : " + e.getMessage());
  }
 }
}
```

CSM network encryption

HCL Informix® JDBC Driver enables encryption of data transmitted over a network by using an encryption communication support module (CSM).

The encryption module **com.informix.jdbc.Crypto** class is packaged in the JDBC `.jar` file. JDBC encryption **CSM** is a pure Java™ implementation that uses services from the Java™ Cryptography provider.

CSM network encryption syntax

To configure network encryption, set the **CSM** environment variable.

Use the following syntax to set the **CSM** environment variable and encryption options:

CSM environment variable syntax

```
CSM=("CLASSNAME=com.informix.jdbc.Crypto"[,{option tags | config=parameterfile}])
```

option tags

Specify the syntax of encryption tags. For more information, see Option tags on page 53.

config=parameterfile

Specify encryption options in a file. For more information, see Option parameters on page 53.

Option tags

The option tags that can be passed on to the encryption CSM are the same as the encryption option tags that are specified in the CSM configuration file used by the server or CSDK. There are three option tags:

cipher

Defines all ciphers that can be used by the session.

mac

Defines the message authentication code (MAC) key files to be used during the MAC generation and the level of MAC generation utilized.

switch

Defines the frequency at which ciphers or secret keys are renegotiated. The longer the secret key and encryption cipher remain in use, the more likely that the encryption rules might be broken by an attacker. To avoid this, cryptologists recommend periodically changing the secret key and cipher on long-term connections. The default for this renegotiation is once an hour. By using the switch tag, you can set the time for this renegotiation in minutes.

For the syntax of these tags, see the *HCL® Informix® Security Guide*.

The encryption CSM option parameters are separated by a comma and not by a semicolon. When using a DataSource, getIfxCSM() and setIfxCSM() methods can be used to get and set CSM as a property. When setting CSM as a property, make sure that you do not enclose the option string in parentheses. The following is an example that correctly sets the CSM as a property:

```
connProperties.put("CSM","classname=com.informix.jdbc.Crypto,cipher[all],
mac[<builtin>]");
```

Option parameters

You can configure encryption by creating a file with encryption parameters and then specifying the file name. The encryption parameters are:

- **ENCCSM_CIPHERS**: Ciphers to be used
- **ENCCSM_MAC**: MAC level
- **ENCCSM_MACFILES**: MAC file location
- **ENCCSM_SWITCH**: CIPHER and KEY change frequency, separated by a comma

For the syntax of these parameters, see the *HCL® Informix® Security Guide*.

The following is an example that specifies the CSM parameters in a configuration file:

```
  String newUrl = "jdbc:informix-sqli:
//beacon:8779/test:INFORMIXSERVER=danon950_beacon_encrypt;
```

```
user=rdtest;password=test;
csm=(classname=com.informix.jdbc.Crypto,config=test.cfg)";
        try
        {
            Class.forName( "com.informix.jdbc.IfxDriver" );
        }catch( Exception e )
        {
            System.out.println( "ERROR: failed to load
Informix JDBC driver." );
        }
        try
        {
            Connection con = DriverManager.getConnection( newUrl );
        }
        catch( SQLException e )
        {
            System.out.println( "ERROR: failed to connect." );
            e.printStackTrace();
            return;
        }
```

Using single sign-on access control with the Informix® JDBC Driver

Before you begin

Ensure that the database server is set up for SSO authentication. For information, see the "Single Sign-on Access Control" section in the *HCL® Informix® Security Guide*.

About this task

You can use single sign-on (SSO) access control with JDBC by using the DriverManager.getConnection() method and setting the Communication Support Module (CSM) in the connection URL to the service principal. Using SSO access control replaces the user ID and password option.

Note: Support for Communication Support Module (CSM) is removed starting Informix Server 14.10.xC9 . You should use Transport Layer Security (TLS)/Secure Sockets Layer (SSL) instead.

1. Modify the connection URL so that it includes the service principal.
   The service principal consists of the database server name and the SSO realm.

   ```
   CSM=(SSO=database_server@realm,ENC=true)
   ```

   The `ENC=true` setting means that Generic Security Services (GSS) encryption is enabled. The `ENC=true` setting is optional because by default, its value is `true`. If you do not want to enable GSS encryption, set the value to false: `ENC=false`.

   For complete syntax of the connection URL, see Format of database URLs on page 19.

2. Create a login configuration file with the following code:

   ```
   com.sun.security.jgss.initiate {
           com.sun.security.auth.module.Krb5LoginModule required
   useTicketCache=true
   ```

```
doNotPrompt=true;
}
```

3. Run the application with the **java.security.auth.login.config** property set to the login configuration files full path name, followed by the **TestSso** class.

   Example

   The following is an example where `IfmxLog.conf` is the login configuration file:

   ```
   java -Djava.security.auth.login.config=mydirectory/IfmxLog.conf TestSso
   ```

PAM authentication method

The HCL Informix® JDBC Driver, Version 2.21. JC5 and later, implements support for handling PAM (Pluggable Authentication Module)-enabled HCL Informix® server 9.40 and later servers. This implementation supports a challenge-response dialog between PAM and the end user. To facilitate this dialog, the JDBC developer must implement the **com.informix.jdbc.IfmxPAM** interface. The IfxPAM() method in the **IfmxPAM** interface acts as the gateway between PAM and the user.

The IfxPAM() method is called when the JDBC server encounters a PAM challenge method. The return value from the IfxPAM() method acts as the response to the challenge message and is sent to PAM.

The signature for the IfxPAM() method is:

```
public IfxPAMResponse IfxPAM(IfxPAMChallenge challengeMessage)
```

Two classes, **IfxPAMChallenge** and **IfxPAMResponse**, usher messages between the JDBC driver and PAM. The **IfxPAMChallenge** class contains the information that has been sent from PAM to the user.

The challenge message is obtained from the **IfxPAMChallenge** class by using the getChallenge() method. This message is what is sent directly from PAM running on HCL Informix® server to be routed to the end user. The challenge messages are listed in the following table.

Table 1. Types of challenge messages

Message	Description
PAM_PROMPT_ECHO_ON	The message is displayed to the user and the users response can be echoed back.
PAM_PROMPT_ECHO_OFF	The message is displayed to the user and the users response is hidden or masked (that is, when the user enters a password, asterisks are displayed instead of the exact characters the user types).
PAM_PROMPT_ERROR_MSG	The message is displayed to the user as an error, with no response required.
PAM_TEXT_INFO_MSG	The message is displayed to the user as an informational message, with no response required.

The challenge message type is governed by the PAM standard and can have vendor-specific values. See the PAM standard and vendor-specific information for possible values and interpretations.

The PAM standard defines the maximum size of a PAM message to be 512 bytes (**IfxPAMChallenge.PAM_MAX_MESSAGE_SIZE**).

The **IfxPAMResponse** class is similar to **IfxPAMChallenge**, but instead of being used by PAM to send a message to the user, the **IfxPAMResponse** class is used to send a message from the user to PAM. Use the IfxPAMResponse.setResponse() method to send the challenge-response string to PAM. However, set the response type (which is set by using the IfxPAMResponse.setResponseType() method) to zero, the default, as the response type is currently reserved for future use.

The challenge-response string is limited to the size of the challenge message: **IfxPAMResponse.PAM_MAX_MESSAGE_SIZE** or 512 bytes. If the response string exceeds this limit, an SQL exception is thrown.

Additionally, when the challenge message is of type PAM_INFO_TEXT or PAM_PROMPT_ERR_MSG (see PAM standards for meaning and integer values), PAM expects no user response. Thus, a null **IfxPAMResponse** object or one that has not been set with specific values can be returned to JDBC. The **IfxPAMResponse** class provides the following method to allow the JDBC developer to stop the connection attempt during a PAM session:

```
public void setTerminateConnection(boolean flag)
```

The value of the *flag* can be TRUE or FALSE. If the value of the parameter passed to **setTerminateConnection** is TRUE, then the connection to the PAM-enabled HCL Informix® server immediately terminates upon returning from IfxPAM(). If the value is set to FALSE, then the connection attempt to the PAM-enabled server continues as usual.

PAM in JDBC

JDBC developers using PAM to communicate with a PAM-enabled HCL Informix® server must implement the **com.informix.jdbc.IfmxPAM** interface. To do so, put the following on the class declaration line in a Java™ class file:

```
implements IfmxPAM
```

That Java™ class must then implement the **IfmxPAM** interface conforming to Java™ standards and the details provided previously. The next step is to inform the JDBC driver what Java™ class has implemented the **IfmxPAM** interface. There are two ways to do this:

- Add the key-value pair **IFX_PAM_CLASS**=*your.class.name* to the connection URL, where the value *your.class.name* is the path to the class that has implemented the **IfmxPAM** interface.

  This method is typically used when connecting to the Informix® server by using the **DriverManager.getConnection** (URL) approach.
- Add the property **IFX_PAM_CLASS** with the value *your.class.name* to your properties list before attempting the connection to the PAM-enabled server.

  This method is used when connecting to the Informix® server by using the DataSource.getConnection() approach.

JDBC developers have a wide latitude in implementing the **IfmxPAM** interface. The following actions happen during authentication that uses PAM:

1. The JDBC driver, when detecting communication with a PAM-enabled server, contacts the IfxPAM() method and passes it a **IfxPAMChallenge** object containing the PAM challenge question.
2. A dialog box you create appears with a text question containing the challenge message that was sent by PAM.
3. When the user furnishes the response, it is packaged into an **IfxPAMResponse** object, and it is returned to the JDBC driver by exiting the IfxPAM() method returning the **IfxPAMResponse** object.
4. When PAM receives the response from the challenge question, it can authorize the user, deny access to the user, or issue another challenge question, in which case the previous process is repeated.

This process continues until either the user is authorized or the user is denied access. The Java™ developer or user can terminate the PAM authorization sequence by calling the IfxPAMResponse.setTerminateConnection() method with a value of TRUE.

Close the connection

The following table contrasts the different effects of calling the Connection.close() and scrubConnection() methods in environments that use connection pooling and those that do not.

For more information about deallocating resources, see Deallocate resources on page 60. For more information about the scrubConnection() method, see Clean pooled connections on page 223.

Connection pooling status	Effect of calling Connection.close() method	Effect of calling scrubConnection() method
Non-connection pool setup	Closes database connection, all associated statement objects, and their result sets. Connection is no longer valid.	Returns connection to original state, keeps opened statements, but closes result sets Connection is still valid. Releases resources associated with result sets only.
Connection pool with HCL Informix® Implementation	Closes connection to the database and reopens it to close any statements associated with the connection object and reset the connection to its original state Connection object is then returned to the connection pool and is available when requested by a new application connection.	Returns a connection to original state and keeps all open statements, but closes all result sets Calling this method in this situation not recommended
Connection pool with application server implementation	Defined by your connection pooling implementation	Returns connection to original state and retains opened statements, but closes result sets This functionality can be useful if you are using the JDBC 3.0 feature of statement pooling with connections. When your application calls the Connection.close() method, your application servers connection-pool manager can call

Connection pooling status	Effect of calling Connection.close() method	Effect of calling scrubConnection() method
		scrubConnection() for the pooled connection object before returning the object to the connection pool.

Important: When calling the scrubConnection() method, your applications should be using server-only connections.

Perform database operations

These topics explain what you need to use HCL Informix® JDBC Driver to perform operations against the Informix® database.

Query the database

HCL Informix® JDBC Driver complies with the JDBC API specification for sending queries to a database and retrieving the results. The driver supports most of the methods of the **Statement**, **PreparedStatement**, **CallableStatement**, **ResultSet**, and **ResultSetMetaData** interfaces.

Example of sending a query to the Informix® database

The following example from the `SimpleSelect.java` program shows how to use the **PreparedStatement** interface to execute a SELECT statement that has one input parameter:

```
try
   {
   PreparedStatement pstmt = conn.prepareStatement("Select *
      from x "
      + "where a = ?;");
   pstmt.setInt(1, 11);
   ResultSet r = pstmt.executeQuery();
   while(r.next())
      {
      short i = r.getShort(1);
      System.out.println("Select: column a = " + i);
      }
   r.close();
   pstmt.close();
   }
catch (SQLException e)
   {
   System.out.println("ERROR: Fetch statement failed: " +
      e.getMessage());
   }
```

The program first uses the Connection.prepareStatement() method to prepare the SELECT statement with its single input parameter. It then assigns a value to the parameter by using the PreparedStatement.setInt() method and executes the query with the PreparedStatement.executeQuery() method.

The program returns resulting rows in a **ResultSet** object, through which the program iterates with the ResultSet.next() method. The program retrieves individual column values with the ResultSet.getShort() method, since the data type of the selected column is SMALLINT.

Finally, both the **ResultSet** and **PreparedStatement** objects are explicitly closed with the appropriate close() method.

For more information about which getXXX() methods retrieve individual column values, see Data type mapping for ResultSet.getXXX() methods on page 255.

Reoptimize queries

When you prepare SELECT, EXECUTE FUNCTION, or EXECUTE PROCEDURE statements, the database server uses a query plan to optimize the query. If you later modify the data that is associated with the prepared statement, you can compromise the effectiveness of the query plan for that statement. However, when you change the data, you can reoptimize your query.

You can reoptimize a query by setting the Informix® JDBC Driver extension method to reuse the **PreparedStatement** object but reoptimize the previously prepared query plan. Alternatively, you can create a new **PreparedStatement** object. Reoptimizing an existing **PreparedStatement** object, which rebuilds only the query plan, has the following advantages over creating a new **PreparedStatement** object, which rebuilds the whole statement:

- Uses fewer resources
- Reduces overhead
- Requires less time

To enable reoptimization, set the **withReoptimization** argument to the IfmxPreparedStatement.executeQuery() method to `true`. The executeQuery() method has the following format:

```
com.informix.jdbc.IfmxPreparedStatement.executeQuery(boolean withHold,
                              boolean withReOptimization)
```

The following query uses the IfmxPreparedStatement.executeQuery() method to enable reoptimization:

```
Connection conn = DriverManager.getConnection(URL);
com.informix.jdbc.IfmxPreparedStatement pStmt =
         (com.informix.jdbc.IfmxPreparedStatement)
         conn.prepareStatement("SELECT * FROM customer");
ResultSet rs =  pStmt.executeQuery(false, true);
```

Result sets

The HCL Informix® JDBC Driver implementation of the Statement.execute() method returns a single **ResultSet** object. Because the server does not support multiple **ResultSet** objects, this implementation differs from the JDBC API specification, which states that the Statement.execute() method can return multiple **ResultSet** objects.

Returning multiple Result Sets is not supported by the HCL Informix® JDBC Driver.

Scrollable result set for multiple rows

The Scrollable ResultSet fetches one row at a time from the server. A performance enhancement for Scrollable ResultSet allows multiple rows to be fetched at one time. In the following example, where the rows *m* through *n* are desired, the following fetches the rows into a ResultSet. As long as only rows between *m* and *n* inclusive are accessed, no further fetches occur. In this example, the rows 50 through 100 are desired and the ResultSet is SCROLL_INSENSITIVE:

```
rs.setFetchSize(51);
        rs.absolute(49); // one row will be fetched
        rs.next() // rs will contain 51 rows
```

HCL Informix® only fetches in the forward direction and only fetches one row, except when a DIR_NEXT fetch is used to fetch rows. For a DIR_NEXT operation, the server sends rows until the fetch buffer is filled or until the last row is sent. Only ResultSet.next() can generate a DIR_NEXT operation.

This performance enhancement does not change the behavior of FORWARD_ONLY ResultSets. The calculation of the size of the fetch buffer is unchanged.

For SCROLL_INSENTIVE ResultSets, the size of the fetch buffer is determined by the fetch size and row size. Statement.setFetchSize() and ResultSet.setFetchSize() can be used to set the fetch size. If fetch size is zero, the default fetch buffer size is used. The fetch buffer size is limited to 32 K.

Certain ResultSet methods require information about the number of rows generated by the query. The methods might result in fetching a row to obtain the information and then refetching the current row. The methods are isBeforeFirst(), isLast(), and absolute(-row).

Additionally, setMaxRows() can change the fetch buffer size for SCROLL_INSENSITIVE ResultsSets. Because additional server support is required to ensure efficient use of setMaxRows(), it is recommended that ResultSet.setMaxRows() is not used as this time.

Deallocate resources

Close a **Statement**, **PreparedStatement**, and **CallableStatement** object by calling the appropriate close() method in your Java™ program when you have finished processing the results of an SQL statement. This closure immediately deallocates the resources that have been allocated to execute your SQL statement. Although the ResultSet.close() method closes the **ResultSet** object, it does not deallocate the resources allocated to the **Statement**, **PreparedStatement**, or **CallableStatement** objects.

It is good practice to call ResultSet.close() and Statement.close() methods when you have finished processing the results of an SQL statement, to indicate to HCL Informix® JDBC Driver that you are done with the statement or result set. When you do so, your program releases all its resources on the database server. It is, however, not required to call ResultSet.close() and Statement.close() specifically, as long as you call to Connection.close(), which takes care of releasing these resources.

Execute across threads

The same **Statement** or **ResultSet** instance cannot be accessed concurrently across threads. You can, however, share a **Connection** object between multiple threads.

For example, if one thread executes the Statement.executeQuery() method on a **Statement** object, and another thread executes the Statement.executeUpdate() method on the same **Statement** object, the results of both methods are unexpected and depend on which method was executed last.

Similarly, if one thread executes the method ResultSet.next() and another thread executes the same method on the same **ResultSet** object, the results of both methods are unexpected and depend on which method was executed last.

Scroll cursors

The scroll cursors feature of HCL Informix® JDBC Driver follows the JDBC 3.0 specification, with these exceptions:

Scroll sensitivity

The HCL Informix® database server implementation of scroll cursors places the rows fetched in a temporary table. If another process changes a row in the original table (assuming the row is not locked) and the row is fetched again, the changes are not visible to the client.

This behavior is similar to the SCROLL_INSENSITIVE description in the JDBC 3.0 specification. HCL Informix® JDBC Driver does not support SCROLL_SENSITIVE cursors. To see updated rows, your client application must close and reopen the cursor.

Client-side scrolling

The JDBC specification implies that the scrolling can happen on the client-side result set. HCL Informix® JDBC Driver supports the scrolling of the result set only to the extent that the database server supports scrolling.

Result set updatability

The JDBC 3.0 API does not provide exact specifications for SQL queries that yield result sets that can be updated. Generally, queries that meet the following criteria can produce result sets that can be updated:

- The query references only a single table in the database.
- The query does not contain any JOIN operations.
- The query selects the primary key of the table it references.
- Every value expression in the select list must consist of a column specification, and no column specification can appear more than once.
- The WHERE clause of the table expression cannot include a subquery.

HCL Informix® JDBC Driver relaxes the primary key requirement, because the driver performs the following operations:

1. The driver looks for a column called ROWID.
2. The driver looks for a SERIAL, SERIAL8, or BIGSERIAL column in the table.
3. The driver looks for the tables primary key in the system catalogs.

If none of these is provided, the driver returns an error.

When you delete a row in a result set, the ResultSet.absolute() method is affected, because the positions of the rows change after the delete.

When the query contains a SERIAL column and the data is duplicated in more than one row, execution of updateRow() or deleteRow() affects all the rows containing that data.

The `ScrollCursor.java` example file shows how to retrieve a result set with a scroll cursor. For examples of how to use a scrollable cursor that can be updated, see the `UpdateCursor1.java`, `UpdateCursor2.java`, and `UpdateCursor3.java` files.

Hold cursors

When transaction logging is used, HCL Informix® generally closes all cursors and releases all locks when a transaction ends. In a multiuser environment, this behavior is not always desirable.

HCL Informix® JDBC Driver had already implemented holdable cursor support with Informix® extensions. Informix® database servers (5.x, 7.x, SE, 8.x, 9.x, and 10.x, or later) support adding keywords WITH HOLD in the declaration of the cursor. Such a cursor is referred to as a hold cursor and is not closed at the end of a transaction.

HCL Informix® JDBC Driver, in compliance with the JDBC 3.0 specifications, adds methods to JDBC interfaces to support holdable cursors.

For some of the applications, there might be a need to hold cursors over commit in ANSI databases. JDBC provides an API Connection.setHoldability(), but for more flexibility user can also set this via Connection URL or data source.

- Add CURSOR_HOLDABILITY = 1 or 2 {In Connection URL}
- Add setCursorHoldability(1 or 2) {In Informix datasource}

There are 2 constants for doing that:

- **HOLD_CURSORS_OVER_COMMIT:** Indicating that open ResultSet objects with this holdability will remain open when the current transaction is committed. Value is 1.
- **CLOSE_CURSORS_OVER_COMMIT:**Indicating that open ResultSet objects with this holdability will be closed when the current transaction is committed. Value is 2.

The default is CLOSE_CURSORS_OVER_COMMIT.

For more information about hold cursors, see the *HCL® Informix® Guide to SQL: Syntax*.

Update the database

You can issue batch update statements or perform bulk inserts to update the database.

Perform batch updates

The batch update feature is similar to multiple HCL Informix® SQL PREPARE statements. You can issue batch update statements as in the following example:

```
PREPARE stmt FROM "insert into tab values (1);
   insert into tab values (2);
   update table tab set col = 3 where col = 2";
```

The batch update feature in HCL Informix® JDBC Driver follows the JDBC 3.0 specification, with these exceptions:

- SQL statements
- Return value from Statement.executeBatch()

SQL statements and batch updates

The following commands cannot be put into multistatement PREPARE statements:

- SELECT (except SELECT INTO TEMP) statement
- DATABASE statements
- CONNECTION statements

For more details, see *HCL® Informix® Guide to SQL: Syntax*.

Return value from Statement.executeBatch() method

The return value differs from the JDBC 3.0 specification in the following ways:

- If the **IFX_BATCHUPDATE_PER_SPEC** environment variable is set to `0`, only the update count of the first statement executed in the batch is returned. If the **IFX_BATCHUPDATE_PER_SPEC** environment variable is set to `1` (the default), the return value equals the number of rows affected by all SQL statements executed by Statement.executeBatch(). For more information, see Informix environment variables with the HCL Informix JDBC Driver on page 23.
- When errors occur in a batch update executed in a **Statement** object, no rows are affected by the statement; the statement is not executed. Calling BatchUpdateException.getUpdateCounts() returns `0` in this case.
- When errors occur in a batch update executed in a **PreparedStatement** object, rows that were successfully inserted or updated on the database server do not revert to their pre-updated state. However, the statements are not always committed; they are still subject to the underlying autocommit mode.

The `BatchUpdate.java` example file shows how to send batch updates to the database server.

Perform bulk inserts

A bulk insert is the HCL Informix® extension to the JDBC 3.0 batch update feature. The bulk insert feature improves the performance of single INSERT statements that are executed multiple times, with multiple value settings. To enable this feature, set the **IFX_USEPUT** environment variable to `1`. (The default value is `0`.)

This feature does not work for multiple statements passed in the same **PreparedStatement** instance or for statements other than INSERT. If this feature is enabled and you pass in an INSERT statement followed by a statement with no parameters, the statement with no parameters is ignored.

The bulk insert feature requires the client to convert the Java™ type to match the target column type on the server for all data types except opaque types or complex types.

The `BulkInsert.java` example, which is installed in the `demo` directory where your JDBC driver is installed, shows how to perform a bulk insert.

Parameters, escape syntax, and unsupported methods

This section contains the following information:

- How to use OUT parameters
- How to use named parameters in a CallableStatement
- Support for the DESCRIBE INPUT statement
- How to use escape syntax to translate from JDBC to HCL Informix®

It also lists unsupported methods and methods that behave differently from the standard.

The CallableStatement OUT parameters

The **CallableStatement** methods handle OUT parameters in C function and Java™ user-defined routines (UDRs). Two registerOutParameter() methods specify the data type of OUT parameters to the driver. A series of getXXX() methods retrieves OUT parameters.

The OUT parameter routine makes available a valid blob descriptor and data to the JDBC client for a BINARY OUT parameter. Using receive methods, you can use these OUT parameter descriptors and data provided by the server.

Exchange of descriptor and data between Informix® and JDBC is consistent with the existing mechanism by which data is exchanged for the result set methods of JDBC, such as passing the blob descriptor and data through SQLI protocol methods. (SPL UDRs are the only type of UDRs supporting BINARY OUT parameters.)

For background information, see the following documentation:

- *HCL® Informix® User-Defined Routines and Data Types Developer's Guide* provides introductory and background information about opaque types and user-defined routines (UDRs) for use in the Informix® database.
- *HCL® J/Foundation Developer's Guide* describes how to write Java™ UDRs for use in the database server.
- The *HCL® Informix® Guide to SQL: Tutorial* describes how to write stored procedure language (SPL) routines.
- The *HCL® Informix® DataBlade® API Programmer's Guide* describes how to write external C routines.

Informix® database servers return one or multiple OUT parameter to HCL Informix® JDBC Driver.

For examples of how to use OUT parameters, see the `CallOut1.java`, `CallOut2.java`, `CallOut3.java`, and `CallOut4.java` example programs in the `basic` subdirectory of the `demo` directory where your HCL Informix® JDBC Driver is installed.

Server and driver restrictions and limitations

Server restrictions

This topic describes the restrictions imposed by different versions of the 9.x and later HCL Informix® server. It also describes enhancements made to the JDBC driver and the restrictions imposed by it.

Versions 9.2x and 9.3x of HCL Informix® have the following requirements and limitations concerning OUT parameters:

- Only a function can have an OUT parameter. A function is defined as a UDR that returns a value. A procedure is defined as a UDR that does not return a value.
- There can be only one OUT parameter per function.
- The OUT parameter has to be the last parameter.
- You cannot specify INOUT parameters.

  HCL Informix®, Version 10.0, or later allows you to specify INOUT parameters (C, SPL, or Java™ UDRs).
- The server does not correctly return the value `NULL` for external functions.
- You cannot specify OUT parameters that are complex types.
- You cannot specify C and SPL routines that use the RETURN WITH RESUME syntax.

These restrictions, for server versions 9.2x and 9.3x, are imposed whether users create C, SPL, or Java™ UDRs.

The functionality of the HCL Informix®, Version 9.4 allows:

- Any parameters to be OUT parameters for C, SPL, or Java™ UDRs
- User-defined procedures with no return value to have OUT parameters
- Multiple OUT parameters

You cannot specify INOUT parameters.

For more information about UDRs, see *HCL® Informix® User-Defined Routines and Data Types Developer's Guide* and *HCL® J/Foundation Developer's Guide*.

Driver enhancement

The **CallableStatement** object provides a way to call or execute UDRs in a standard way for all database servers. Results from the execution of these UDRs are returned as a result set or as an OUT parameter.

The following is a program that creates a user-defined function, myudr, with two OUT parameters and one IN parameter, and then executes the myudr() function. The example requires server-side support for multiple OUT parameters; hence it only works for HCL Informix®, Version 9.4 or above. For more information about UDRs, see *HCL® Informix® User-Defined Routines and Data Types Developer's Guide* and *HCL® J/Foundation Developer's Guide*.

```
import java.sql.*;
public class myudr {

  public myudr() {
  }

  public static void main(String args[]) {
```

```
Connection myConn = null;
try {
  Class.forName("com.informix.jdbc.IfxDriver");
  myConn = DriverManager.getConnection(
      "jdbc:informix-sqli:MYSYSTEM:18551/testDB:"
      +"INFORMIXSERVER=patriot1;user=USERID;"
      +"password=MYPASSWORD");
}
catch (ClassNotFoundException e) {
  System.out.println(
   "problem with loading Ifx Driver\n" + e.getMessage());
}
catch (SQLException e) {
  System.out.println(
    "problem with connecting to db\n" + e.getMessage());
}
try {
  Statement stmt = myConn.createStatement();
  stmt.execute("DROP FUNCTION myudr");
}
catch (SQLException e){
}
try
{
  Statement stmt = myConn.createStatement();

  stmt.execute(
    "CREATE FUNCTION myudr(OUT arg1 int, arg2 int, OUT arg3 int)"
    +" RETURNS boolean; LET arg1 = arg2; LET arg3 = arg2 * 2;"
    +"RETURN 't'; END FUNCTION;");
}
catch (SQLException e) {
  System.out.println(
    "problem with creating function\n" + e.getMessage());
}

Connection conn = myConn;

try
{
  String command = "{? = call myudr(?, ?, ?)}";
  CallableStatement cstmt = conn.prepareCall (command);

  // Register arg1 OUT parameter
  cstmt.registerOutParameter(1, Types.INTEGER);

  // Pass in value for IN parameter
  cstmt.setInt(2, 4);

  // Register arg3 OUT parameter
  cstmt.registerOutParameter(3, Types.INTEGER);

  // Execute myudr
  ResultSet rs = cstmt.executeQuery();

  // executeQuery returns values via a resultSet
  while (rs.next())
```

```
      {
        // get value returned by myudr
        boolean b = rs.getBoolean(1);
        System.out.println("return value from myudr = " + b);
      }

      // Retrieve OUT parameters from myudr
      int i = cstmt.getInt(1);
      System.out.println("arg1 OUT parameter value = " + i);

      int k = cstmt.getInt(3);
      System.out.println("arg3 OUT parameter value = " + k);

      rs.close();
      cstmt.close();
      conn.close();
    }
    catch (SQLException e)
    {
      System.out.println("SQLException: " + e.getMessage());
      System.out.println("ErrorCode: " + e.getErrorCode());
      e.printStackTrace();
    }
  }
}
- - -
.../j2sdk1.4.0/bin/java ... myudr
return value from myudr = true
arg1 OUT parameter value = 4
arg3 OUT parameter value = 8
```

Driver restrictions and limitations

HCL Informix® JDBC Driver has the following requirements and limitations concerning OUT parameters:

- With HCL Informix®, Version 9.2, the driver always returns a -9752 error if a function contains an OUT parameter. The driver creates an **SQLWarning** object and chains this to the **CallableStatement** object.

  You can determine if a function contains an OUT parameter by calling the CallableStatement.getWarnings() method or by calling the IfmxCallableStatement.hasOutParameter() method, which return `TRUE` if the function has an OUT parameter.

  If a function contains an OUT parameter, you must use the CallableStatement.registerOutParameter() method to register the OUT parameter, the setXXX() methods to register the IN and OUT parameter values, and the getXXX() method to retrieve the OUT parameter value.
- The CallableStatement.getMetaData() method returns `NULL` until the executeQuery() method has been executed. After executeQuery() has been called, the **ResultSetMetaData** object contains information only for the return value, not the OUT parameter.
- You must specify all IN parameters by using setXXX() methods. You cannot use literals in the SQL statement. For example, the following statement produces unreliable results:

```
CallableStatement cstmt = myConn.prepareCall("{call
   myFunction(25, ?)}");
```

Instead, use a statement that does not specify literal parameters:

```
CallableStatement cstmt = myConn.prepareCall("{call
   myFunction(?, ?)}");
```

Call the setXXX() methods for both parameters.

- Do not close the **ResultSet** returned by the CallableStatement.executeQuery() method until you have retrieved the OUT parameter value by using a getXXX() method.
- You cannot cast the OUT parameter to a different type in the SQL statement. For example, the following cast is ignored:

```
CallableStatement cstmt = myConn.prepareCall("{call
   foo(?::lvarchar, ?)}";
```

- The setMaxRows() and registerOutParameter() methods both take **java.sql.Types** values as parameters. There are some one-to-many mappings from **java.sql.Types** values to Informix® types.

  In addition, some Informix® types do not map to **java.sql.Types** values. Extensions for setMaxRows() and registerOutParameter() fix these problems. See IN and OUT parameter type mapping on page 68.

These restrictions apply to a JDBC application that handles C, SPL, or Java™ UDRs.

IN and OUT parameter type mapping

An exception is thrown by the registerOutParameter(int, int), registerOutParameter(int, int, int), or setNull(int, int) method if the driver cannot find a matching HCL Informix® type or finds a mapping ambiguity (more than one matching Informix® type). The table that follows shows the mappings the **CallableStatement** interface uses. Asterisks (*) indicate mapping ambiguities.

java.sql.Types	com.informix.lang.IfxTypes
Array*	IFX_TYPE_LIST
	IFX_TYPE_MULTISET
	IFX_TYPE_SET
Bigint	IFX_TYPE_INT8
Binary	IFX_TYPE_BYTE
Bit	Not supported
Blob	IFX_TYPE_BLOB
Char	IFX_TYPE_CHAR (*n*)
Clob	IFX_TYPE_CLOB

java.sql.Types	com.informix.lang.IfxTypes
Date	IFX_TYPE_DATE
Decimal	IFX_TYPE_DECIMAL
Distinct*	Depends on base type
Double	IFX_TYPE_FLOAT
Float	IFX_TYPE_FLOAT[1]
Integer	IFX_TYPE_INT
Java_Object*	IFX_TYPE_UDTVAR
	IFX_TYPE_UDTFIX
Long	IFX_TYPE_BIGINT
	IFX_TYPE_BIGSERIAL
Longvarbinary*	IFX_TYPE_BYTE
	IFX_TYPE_BLOB
Longvarchar*	IFX_TYPE_TEXT
	IFX_TYPE_CLOB
	IFX_TYPE_LVARCHAR
Null	Not supported
Numeric	IFX_TYPE_DECMIAL
Other	Not supported
Real	IFX_TYPE_SMFLOAT
Ref	Not supported
Smallint	IFX_TYPE_SMINT
Struct	IFX_TYPE_ROW
Time	IFX_TYPE_DTIME (hour to second)
Timestamp	IFX_TYPE_DTIME (year to fraction(5))
Tinyint	IFX_TYPE_SMINT
Varbinary	IFX_TYPE_BYTE
Varchar	IFX_TYPE_VCHAR (*n*)

java.sql.Types	com.informix.lang.IfxTypes
Nothing*	IFX_TYPE_BOOL

[1] This mapping is JDBC compliant. You can map the JDBC FLOAT data type to the Informix® SMALLFLOAT data type for compatibility with earlier versions by setting the IFX_SET_FLOAT_AS_SMFLOAT connection property to 1.

To avoid mapping ambiguities, use the following extensions to **CallableStatement**, defined in the **IfmxCallableStatement** interface:

```
public void IfxRegisterOutParameter(int parameterIndex,
   int ifxType) throws SQLException;

public void IfxRegisterOutParameter(int parameterIndex,
   int ifxType, String name) throws SQLException;

public void IfxRegisterOutParameter(int parameterIndex,
   int ifxType, int scale) throws SQLException;

public void IfxSetNull(int i, int ifxType) throws SQLException;

public void IfxSetNull(int i, int ifxType, String name) throws
   SQLException;
```

Possible values for the *ifxType* parameter are listed in The IfxTypes class on page 249.

HCL Informix®, Version 10.0, or later makes available to the JDBC client valid BLOB descriptors and data to support binary OUT parameters for SPL UDRs.

HCL Informix® JDBC Driver, Version 3.0, or later can receive the OUT parameter descriptor and data provided by the server and use it in Java™ applications.

The single correct return value for any JDBC binary type (BINARY, VARBINARY, LONGVARBINARY) retrieved via method getParameterType (ParameterMetaData) is -4, which is associated with **java.sql.Type.LONGVARBINARY** data type. This reflects the fact that all the JDBC binary types are mapped to the same Informix® SQL data type, BYTE.

Named parameters in a CallableStatement

A CallableStatement provides a way to call a stored procedure on the server from a Java™ program. You can use named parameters in a CallableStatement to identify the parameters by name instead of by ordinal position. This enhancement was introduced in the JDBC 3.0 specification. If the procedure is unique, you can omit parameters that have default values and you can enter the parameters in any order. Named parameters are especially useful for calling stored procedures that have many arguments and some of those arguments have default values.

The JDBC driver ignores case for parameter names. If the stored procedure does not have names for all the arguments, the server passes an empty string for missing names.

Requirements and restrictions for named parameters in a CallableStatement

HCL Informix® JDBC Driver has the following requirements and restrictions for named parameters in a CallableStatement:

- Parameters for the CallableStatement must be specified by either name or by the ordinal format within a single invocation of a routine. If you name a parameter for one argument, for example, you must use parameter names for all of the arguments.
- Named parameters are not supported for a remote CallableStatement.
- Support for named parameters is subject to existing limitations for calling stored procedures.

Verify support for named parameters in a CallableStatement

The JDBC specification provides the DatabaseMetaData.supportsNamedParameters() method to determine if the driver and the RDMS support named parameters in a CallableStatement. For example:

```
      Connection myConn = . . .   // connection to the RDBMS for Database
      . . .
      DatabaseMetaData dbmd = myConn.getMetaData();
      if (dbmd.supportsNamedParameters() == true)
      {
          System.out.println("NAMED PARAMETERS FOR CALLABLE"
                             + "STATEMENTS IS SUPPORTED");
          . . .
      }
```

The system returns `true` if named parameters are supported.

Retrieve parameter names for stored procedures

To retrieve the names of parameters for stored procedures, use **DatabaseMetaData** methods defined by the JDBC specification as shown in the following example.

```
Connection myConn = ...   // connection to the RDBMS for Database
. . .
      DatabaseMetaData dbmd = myConn.getMetaData();
      ResultSet rs = dbmd.getProcedureColumns(
       "myDB", schemaPattern, procedureNamePattern, columnNamePattern);
      rs.next() {
          String parameterName = rs.getString(4);
 - - - or - - -
 String parameterName = rs.getString("COLUMN_NAME");
 - - -
          System.out.println("Column Name: " + parameterName);
```

The names of all columns that match the parameters of the getProcedureColumns() method are displayed.

Parameter names are not part of the **ParameterMetaData** interface and cannot be retrieved from a **ParameterMetaData** object.

When you use the getProcedureColumns() method, the query retrieves all procedures owned by **informix** (including system-generated routines) from the **sysprocedures** system catalog table. To prevent errors, verify that the stored procedures you are using have been configured with correct permissions on the server.

See Unsupported methods and methods that behave differently on page 77 for important differences in JDBC API behavior for the getProcedureColumns() method.

Named parameters and unique stored procedures

A unique stored procedure has a unique name and a unique number of arguments. Named parameters are supported for unique stored procedures when the number of parameters in the CallableStatement is equal to or less than the number of arguments in the stored procedure.

Example of number of named parameters equals the number of arguments

The following stored procedure has five arguments

```
create procedure createProductDef(productname   varchar(64),
                 productdesc  varchar(64),
                 listprice    float,
                 minprice     float,
             out prod_id      float);
. . .
  let prod_id = <value for prod_id>;
end procedure;
```

The following Java™ code with five parameters corresponds to the stored procedure. The question mark characters (`?`) within the parentheses of a JDBC call refer to the parameters. (In this case five parameters for five arguments.) Set or register all the parameters. Name the parameters by using the format `cstmt.setString("arg", name);`, where *arg* is the name of the argument in the corresponding stored procedure. You do not need to name parameters in the same order as the arguments in the stored procedure.

```
String sqlCall = "{call CreateProductDef(?,?,?,?,?)}";
      CallableStatement cstmt = conn.prepareCall(sqlCall);

      cstmt.setString("productname", name);     // Set Product Name.
      cstmt.setString("productdesc", desc);     // Set Product Description.
      cstmt.setFloat("listprice", listprice);   // Set Product ListPrice.
      cstmt.setFloat("minprice", minprice);     // Set Product MinPrice.

      // Register out parameter which should return the product is created.

      cstmt.registerOutParameter("prod_id", Types.FLOAT);

      // Execute the call.
      cstmt.execute();

      // Get the value of the id from the OUT parameter: prod_id
      float id = cstmt.getFloat("prod_id");
```

The Java™ code and the stored procedure show the following course of events:

1. A call to the stored procedure is prepared.
2. Parameter names indicate which arguments correspond to which parameter value or type.
3. The values for the input parameters are set and the type of the output parameter is registered.
4. The stored procedure executes with the input parameters as arguments.
5. The stored procedure returns the value of an argument as an output parameter and the value of the output parameter is retrieved.

Example of number of named parameters Is less than the number of arguments

If the number of parameters in CallableStatement is less than the number of arguments in the stored procedure, the remaining arguments must have default values. You do not need to set values for arguments that have default values because the server automatically uses the default values. You must, however, indicate the arguments that have non-default values or override default values with a question mark character (?) in the CallableStatement.

For example, if a stored procedure has 10 arguments of which 4 have non-default values and 6 have default values, you must have at least four question marks in the CallableStatement. Alternatively, you can use 5, 6, or up to 10 question marks.

If the CallableStatement is prepared with more parameters than non-default values, but less than the number of stored procedure arguments, it must set the values for non-default arguments. The remaining parameters can be any of the other arguments and they can be changed with each execution.

In the following unique stored procedure, the arguments `listprice` and `minprice` have default values:

```
create procedure createProductDef(productname   varchar(64),
                 productdesc  varchar(64),
                 listprice    float default 100.00,
                 minprice     float default  90.00,
             out prod_id      float);
. . .
  let prod_id = <value for prod_id>;
end procedure;
```

The following Java™ code calls the stored procedure with fewer parameters than arguments in the stored procedure (four parameters for five arguments). Because `listprice` has a default value, it can be omitted from the CallableStatement.

```
String sqlCall = "{call CreateProductDef(?,?,?,?)}";
                                          // 4 params for 5 args
      CallableStatement cstmt = conn.prepareCall(sqlCall);

      cstmt.setString("productname", name);   // Set Product Name.
      cstmt.setString("productdesc", desc);   // Set Product Description.

      cstmt.setFloat("minprice", minprice);   // Set Product MinPrice.

      // Register out parameter which should return the product id created.

      cstmt.registerOutParameter("prod_id", Types.FLOAT);

      // Execute the call.
      cstmt.execute();

      // Get the value of the id from the OUT parameter: prod_id
      float id = cstmt.getFloat("prod_id");
```

Alternatively, for the same stored procedure you can omit the parameter for the `minprice` argument. You do not need to prepare the CallableStatement again.

```
      cstmt.setString("productname", name); // Set Product Name.
      cstmt.setString("productdesc", desc); // Set Product Description.
```

```
      cstmt.setFloat("listprice", listprice); // Set Product ListPrice.

      // Register out parameter which should return the product id created.

     cstmt.registerOutParameter("prod_id", Types.FLOAT);

     // Execute the call.
     cstmt.execute();

     // Get the value of the id from the OUT parameter: prod_id
     float id = cstmt.getFloat("prod_id");
```

Or you can omit the parameters for both of the default arguments:

```
cstmt.setString("productname", name);
cstmt.setString("productdesc", desc);
cstmt.registerOutParameter("prod_id", Types.FLOAT);
cstmt.execute();
float id = cstmt.getFloat("prod_id");
```

Named parameters and overloaded stored procedures

If multiple stored procedures have the same name and the same number of arguments, the procedures are overloaded (also known as overloaded UDRs).

The JDBC driver throws an SQLException for overloaded stored procedures because the call cannot resolve to a single stored procedure. To prevent an SQLException, specify the HCL Informix® server data type of the named parameters in the parameter list by appending `::data_type` to the question mark characters where *data_type* is the Informix® server data type. For example `?::varchar` or `?::float`. You must also enter the named parameters for all the arguments and in the same order as the overloaded stored arguments of procedure.

For example, the following two procedures have the same name (**createProductDef**) and the same number of arguments. The data type for the **prod_id** argument is a different data type in each procedure.

Procedure 1

```
create procedure createProductDef(productname   varchar(64),
          productdesc  varchar(64),
          listprice    float default 100.00,
          minprice     float default  90.00,
          prod_id      float);
 ...
 let prod_id = <value for prod_id>;
end procedure;
```

Procedure 2

```
create procedure createProductDef(productname   varchar(64),
          productdesc  varchar(64),
          listprice    float default 100.00,
          minprice     float default  90.00,
          prod_id      int);
 ...
```

```
 let prod_id = <value for prod_id>;
end procedure;
```

If you use the following Java™ code, it returns an SQLException because it cannot resolve to only one procedure:

```
String sqlCall = "{call CreateProductDef(?,?,?,?,?)}";
CallableStatement cstmt = con.prepareCall(sqlCall);
cstmt.setString("productname", name);   // Set Product Name.
```

If you specify the Informix® data type for the argument that has a different data type, the Java™ code resolves to one procedure. The following Java™ code resolves to Stored Procedure 1 because the code specifies the FLOAT data type for the **prod_id** argument:

```
String sqlCall = "{call CreateProductDef(?,?,?,?,?::float)}";
CallableStatement cstmt = con.prepareCall(sqlCall);
cstmt.setString("productname", name);   // Set Product Name
```

JDBC support for DESCRIBE INPUT

The SQL 92 and 99 standards specify a DESCRIBE INPUT statement for Dynamic SQL. Version 9.4 of HCL Informix® provides support for this statement. (For more information about SQL standards, syntax, and this statement, see *HCL® Informix® Guide to SQL: Syntax*.)

The JDBC 3.0 specification introduces a **ParameterMetaData** class and methods that correspond to DESCRIBE INPUT support.

The HCL Informix® JDBC Driver implements the **java.sql.ParameterMetaData** class. This interface is used for describing input parameters in prepared statements. The method getParameterMetaData() has been implemented to retrieve the metadata for a particular statement.

The **ParameterMetaData** class and the getParameterMetaData() method are part of the JDBC 3.0 API and are included as interfaces in J2SDK1.4.0. Details of these interfaces are specified in the JDBC 3.0 specification.

The HCL Informix® JDBC Driver has implemented additional methods to the **ParameterMetaData** interface to extend its functionality, as shown in the following table.

Return type	Method	Description
int	getParameterLength (int param)	Retrieves parameters length
int	getParameterExtendedId (int param)	Retrieves parameters extended ID
java.lang.String	getParameterExtendedName (int param)	Retrieves parameters extended name
java.lang.String	getParameterExtendedOwnerName (int param)	Retrieves parameters extended owner name of the type
int	getParameterSourceType (int param)	Retrieves parameters SourceType
int	getParameterAlignment (int param)	Retrieves parameters alignment

The following is an example of using the **ParameterMetaData** interface in the HCL Informix® JDBC Driver:

```
. . .
try
{
    PreparedStatement pstmt = null;

    pstmt = myConn.prepareStatement(
      "select * from table_1 where int_col = ? "
      +"and string_col = ?");
    ParameterMetaData paramMeta = pstmt.getParameterMetaData();
    int count = paramMeta.getParameterCount();
    System.out.println("Count : "+count);

    for (int i=1; i <= count; i++)
    {
        System.out.println("Parameter type name : "
                        +paramMeta.getParameterTypeName(i));
        System.out.println("Parameter type : "
                        +paramMeta.getParameterType(i));
        System.out.println("Parameter class name : "
                        +paramMeta.getParameterClassName(i));
        System.out.println("Parameter mode : "
                        +paramMeta.getParameterMode(i));
        System.out.println("Parameter precision : "
                        +paramMeta.getPrecision(i));
        System.out.println("Parameter scale : "
                        +paramMeta.getScale(i));
        System.out.println("Parameter nullable : "
                        +paramMeta.isNullable(i));
        System.out.println("Parameter signed : "
                        +paramMeta.isSigned(i));
    }
. . .
```

The escape syntax

Escape syntax indicates information that must be translated from JDBC format to HCL Informix® native format. Valid escape syntax for SQL statements is as follows.

Type of statement	Escape syntax
Procedure	{call *procedure*}
Function	{*var* = call *function*}
Date	{d '*yyyy-mm-dd*'}
Time	{t '*hh:mm:ss*'}
Limit	{limit *number-to-limit*}
Skip	{limit *number-to-limit number-to-skip*}
Timestamp (Datetime)	{ts '*yyyy-mm-dd hh:mm:ss*[.*fffff*]'}
Function call	{fn *func*[(*args*)]}
Escape character	{escape '*escape-char*'}

Type of statement	Escape syntax
Outer join	{oj *outer-join-statement*}

You can put any of this syntax in an SQL statement, as follows:

```
executeUpdate("insert into tab1 values( {d '1999-01-01'} )");
```

Everything inside the brackets is converted into a valid Informix® SQL statement and returned to the calling function.

Unsupported methods and methods that behave differently

The following JDBC API methods are not supported by HCL Informix® JDBC Driver and cannot be used in a Java™ program that connects to Informix® databases:

- CallableStatement.getRef(int)
- Connection.setCatalog()
- Connection.setReadOnly()
- PreparedStatement.addBatch(String)
- PreparedStatement.setRef(int, Ref)
- PreparedStatement.setUnicodeStream(int, java.io.InputStream, int)
- ResultSet.getRef(int)
- ResultSet.getRef(String)
- ResultSet.getUnicodeStream(int)
- ResultSet.getUnicodeStream(String)
- ResultSet.refreshRow()
- ResultSet.rowDeleted()
- ResultSet.rowInserted()
- ResultSet.rowUpdated()
- ResultSet.setFetchSize()
- Statement.setMaxFieldSize()

The Connection.setCatalog() and Connection.setReadOnly() methods return with no error. The other methods throw the exception: `Method not Supported`.

The following JDBC API methods behave other than specified by the JavaSoft specification:

- CallableStatement.execute()

  Returns a single result set

- DatabaseMetaData.getProcedureColumns()

  Example:

  ```
  DBMD.getProcedureColumns(String catalog,
              String schemaPattern,
  ```

```
            String procedureNamePattern,
            String columnNamePattern)
```

Ignores the **columnNamePattern** field; returns `NULL` when used with any server version older than 9.x.

When you use the getProcedureColumns() method, the query retrieves all procedures owned by **informix** (including system-generated routines) from the **sysprocedures** system catalog table. To prevent errors, verify that the stored procedures you are using have been configured with correct permissions on the server.

For example, if you use one of the following statements:

```
getProcedureColumns("","","","")
```

```
getProcedureColumns("",informix,"","")
```

The DatabaseMetaData.getProcedureColumns() method loads all server UDRs or all UDRs owned by user **informix**. If you chose not to install J/Foundation, or if the configuration parameters for J/Foundation are not set to valid values in your `onconfig` file, the method fails. Also, if any one UDR is not set up correctly on the server, the method fails.

For information about how to set up J/Foundation on Informix® servers and how to run Java™ UDRs on Informix® servers, see the *HCL® J/Foundation Developer's Guide*. For information about how to set up and run C UDRs, see the *HCL® Informix® User-Defined Routines and Data Types Developer's Guide*.

- DatabaseMetaData.othersUpdatesAreVisible()

  Always returns `FALSE`
- DatabaseMetaData.othersDeletesAreVisible()

  Always returns `FALSE`
- DatabaseMetaData.othersInsertsAreVisible()

  Always returns `FALSE`
- DatabaseMetaData.ownUpdatesAreVisible()

  Always returns `FALSE`
- DatabaseMetaData.ownDeletesAreVisible()

  Always returns `FALSE`
- DatabaseMetaData.ownInsertsAreVisible()

  Always returns `FALSE`
- DatabaseMetaData.deletesAreDetected()

  Always returns `FALSE`
- DatabaseMetaData.updatesAreDetected()

  Always returns `FALSE`

- DatabaseMetaData.insertsAreDetected()

  Always returns `FALSE`
- PreparedStatement.execute()

  Returns a single result set
- ResultSet.getFetchSize()

  Always returns `0`
- ResultSetMetaData.getCatalogName()

  Always returns a **String** object containing one blank space
- ResultSetMetaData.getTableName()

  Returns the table name for SELECT, INSERT, and UPDATE statements

  SELECT statements with more than one table name and all other statements return a **String** object containing one blank space.
- ResultSetMetaData.getSchemaName()

  Always returns a **String** object containing one blank space
- ResultSetMetaData.isDefinitelyWriteable()

  Always returns `TRUE`
- ResultSetMetaData.isReadOnly()

  Always returns `FALSE`
- ResultSetMetaData.isWriteable()

  Always returns `TRUE`
- Statement.execute()

  Returns a single result set
- Connection.isReadOnly()

  Returns `TRUE` only when connecting to a secondary server in HDR scenario (see the following **Important** note)

Important: HCL® Informix® servers do not currently support read-only connections. For the HCL Informix® JDBC Driver, Version 2.21.JC4, the implementation of the setReadOnly() method from the **java.sql.Connection** interface has been changed to accept the value passed to it by the calling process. The setReadOnly() method simply returns to the calling process without any interaction to the Informix® database server. (Previous versions of the JDBC driver threw an unsupported method exception.) This change has been made to synchronize the functionality present in the

HCL Informix® JDBC Driver to the HCL Data Server JDBC driver and also to achieve a higher level of compliance in the Sun Conformance Test (CTS).

Handle transactions

By default, all new **Connection** objects are in autocommit mode. When autocommit mode is on, a COMMIT statement is automatically executed after each statement that is sent to the database server. To turn off autocommit mode , explicitly call Connection.setAutoCommit(false).

When autocommit mode is off, HCL Informix® JDBC Driver implicitly starts a new transaction when the next statement is sent to the database server. This transaction lasts until the user issues a COMMIT or ROLLBACK statement. If the user has already started a transaction by executing setAutoCommit(false) and then calls setAutoCommit(false) again, the existing transaction continues unchanged. The Java™ program must explicitly terminate the transaction by issuing either a COMMIT or a ROLLBACK statement before it drops the connection to the database or the database server.

In a database that has been created with logging, if a COMMIT statement is sent to the database server and autocommit mode is on, the error `-255: Not in transaction` is returned by the database server because there is currently no user transaction started. This occurs whether the COMMIT statement was sent with the Connection.commit() method or directly with an SQL statement.

In a database created in ANSI mode, explicitly sending a COMMIT statement to the database server commits an empty transaction. No error is returned because the database server automatically starts a transaction before it executes the statement if there is no user transaction currently open.

For an **XAConnection** object, autocommit mode is off by default and must remain off while a distributed transaction is occurring. The transaction manager performs commit and rollback operations; therefore, you avoid performing these operations directly.

For HCL Informix® releases later than 11.50.xC2, two JDBC classes support SQL transactions that can be rolled back to a savepoint (rather than canceled in its entirety) after an adverse event is encountered:

- **IfmxSavepoint** (Interface)
- **IfxSavepoint** (Savepoint class)

JDBC applications can create, destroy, or rollback to savepoint objects through the following standard JDBC methods:

Table 2. JDBC savepoint classes and methods

Class	Method
IfxConnection	setSavepoint() releaseSavepoint() rollback(*savepoint*)

Table 2. JDBC savepoint classes and methods (continued)

Class	Method
IfxSavepoint	getSavepointId() getSavepointName() These two methods are not interchangeable. A call to getSavepointName() fails with an error unless the savepoint object is declared with a string argument to the setSavepoint() method or to the setSavepointUnique() method. Similarly, an error is returned if you call getSavepointId() for a named savepoint object.

In addition, the setSavepointUnique() method can set a named savepoint whose identifier is unique. While the unique savepoint is active,Informix® issues an exception if the application attempts to reuse its name within the same connection.

The following restrictions apply to savepoint objects in JDBC:

- Savepoints are not valid within XA transactions.
- Savepoints cannot be used unless the current connection sets autocommit mode off.
- Savepoints are not valid in connections to unlogged databases.
- Savepoints cannot be referenced in a triggered action.
- In cross-server distributed queries in which any participating subordinate server does not support savepoint objects, a warning is issued if you set a savepoint after connecting to a server that does not support savepoints, and any call to **rollback***savepoint* fails with an error.

See the descriptions of the SAVEPOINT, RELEASE SAVEPOINT, and ROLLBACK WORK TO SAVEPOINT statements in *HCL® Informix® Guide to SQL: Syntax* for more information about using savepoint objects in SQL transactions.

Autocommit

By default, all new **Connection** objects are in autocommit mode. When autocommit mode is on, a COMMIT statement is automatically executed after each statement that is sent to the database server. To turn off autocommit mode, explicitly call Connection.setAutoCommit(false).

When autocommit mode is off the JDBC Driver implicitly starts a new transaction when the next statement is sent to the database server. This transaction lasts until the user issues a COMMIT or ROLLBACK statement. If the user has already started a transaction by executing setAutoCommit(false) and then calls setAutoCommit(false) again, the existing transaction continues unchanged. The Java™ program must explicitly terminate the transaction by issuing either a COMMIT or a ROLLBACK statement before it drops the connection to the database or the database server.

Logged Database

In a database that has been created with logging, if a COMMIT statement is sent to the database server and autocommit mode is enabled, the error -255: Not in transaction is returned by the database server because there is currently no user

transaction started. This occurs whether the COMMIT statement was sent with the Connection.commit() method or directly with an SQL statement.

ANSI Databases

In a database created in ANSI mode, explicitly sending a COMMIT statement to the database server commits an empty transaction. No error is returned because the database server automatically starts a transaction before it executes the statement if there is no user transaction currently open.

Non-logged Databases

You cannot turn off autocommit on non-logged databases. Because NONLOGGED databases do not support transactions you cannot disable auto-commit which forces JDBC to attempt to start a transaction.

Transactions with Large Objects

Large objects are a special consideration when dealing with database transactions. Manipulating a large object (BLOB/CLOB) is considered a distinct step in a transaction. This has the following implications:

Autocommit is enabled

When autocommit is enabled, creating and inserting a large object is considered two steps. Consider the following example:

```
ByteArrayInputStream byteStream = new ByteArrayInputStream(buffer);
PreparedStatement p = c.prepareStatement("INSERT INTO blobTestValues(?)")) {
   p.setBinaryStream(1, byteStream);
   p.execute();
}
```

In this example we are inserting a single row into the table. Since the column we are inserting is a BLOB, this is two operations. First, JDBC needs to create the BLOB object on the server. This is a single operation and with auto-commit enabled, this is commited and the BLOB is now present on the server. Second, we insert the BLOB pointer into the table row. This operation is then committed. Any error on the INSERT does **NOT** rollback or dispose of the BLOB object that was created. Since the BLOB was dynamically created by the JDBC driver, you will lose all references to the object in the system. It can be cleaned up by a DBA running on the database system, but not by the JDBC application.

If you want to ensure the BLOB is not lost in this scenario you **MUST** using an explicit transaction like the following example shows:

```
ByteArrayInputStream byteStream = new ByteArrayInputStream(buffer);
c.setAutoCommit(false);
PreparedStatement p = c.prepareStatement("INSERT INTO blobTestValues(?)")) {
   p.setBinaryStream(1, byteStream);
   p.execute();
}
c.commit();
```

Autocommit is disabled

If autocommit is disabled then you are using explicit transactions and most large object operations will work as expected in between your transaction boundaries. However, you are free to commit/rollback the intermediate large object operations if you use an explicit Blob/Clob object.

```
c.setAutoCommit(false);
PreparedStatement p = c.prepareStatement("INSERT INTO blobTestValues(?)")) {
   Blob blob = c.createBlob();
   c.commit(); //Commits the blob creation
   p.setBlob(1, blob);
   p.execute();
}
c.rollback(); //rollback the insert, the blob survives
```

Transactions with XA

For a **XAConnection** object, autocommit mode is off by default and must remain off while a distributed transaction is occurring. The transaction manager performs commit and rollback operations; therefore, you avoid performing these operations directly.

Transactions with Savepoints

Since JDBC 3.00.JC2 and Informix server 11.50.xC2, Informix supports SQL transactions that can be rolled back to a Savepoint. A Savepoint is a marker created at any point during a transaction that you can rollback to rather than completely rolling back the entire transaction.

JDBC applications can create, destroy, or rollback to Savepoint objects through the following standard JDBC methods:

Table 3. JDBC Savepoint classes and methods

Class	Method
Connection	**setSavepoint()** **setSavepoint**(*String name*) **releaseSavepoint**(*Savepoint*) **rollback**(*Savepoint*)
Savepoint	**getSavepointId()** **getSavepointName()** These two methods are not interchangeable. A call to **getSavepointName()** fails with an error unless the savepoint object is declared with a string argument to the **setSavepoint()** method or to the **setSavepointUnique()**

Table 3. JDBC Savepoint classes and methods (continued)

Class	Method
	method. Similarly, an error is returned if you call **getSavepointId()** for a named savepoint object.

In addition, the **setSavepointUnique(***String name***)** method can set a named savepoint whose identifier is unique. If the application attempts to reuse its name within the same connection JDBC will throw a SQLException.

The following restrictions apply to Savepoint objects in JDBC:

- Savepoints are not valid within XA transactions.
- Savepoints cannot be used unless the current connection sets autocommit mode off.
- Savepoints are not valid in connections to unlogged databases.
- Savepoints cannot be referenced in a triggered action.
- In cross-server distributed queries in which any participating subordinate server does not support savepoint objects, a warning is issued if you set a savepoint after connecting to a server that does not support savepoints, and any call to **rollback***savepoint* fails with an error.

Form more information, see *IBM Informix Guide to SQL: Syntax*.

Handle errors

Use the JDBC API **SQLException** class to handle errors in your Java™ program. The HCL Informix®-specific **com.informix.jdbc.Message** class can also be used outside a Java™ program to retrieve the HCL Informix® error text for a given error number.

Handle errors with the SQLException class

Whenever an error occurs from either HCL Informix® JDBC Driver or the database server, an **SQLException** is raised. Use the following methods of the **SQLException** class to retrieve the text of the error message, the error code, and the **SQLSTATE** value:

getMessage()

Returns a description of the error

SQLException inherits this method from the **java.util.Throwable** class.

getErrorCode()

Returns an integer value that corresponds to the Informix® database server or HCL Informix® JDBC Driver error code

getSQLState()

Returns a string that describes the **SQLSTATE** value

The string follows the X/Open **SQLSTATE** conventions.

All HCL Informix® JDBC Driver errors have error codes of the form `-79xxx`, such as `-79708: Can't take null input.`

For a list of Informix® database server errors, see *IBM® Informix® Error Messages*. For a list of HCL Informix® JDBC Driver errors, see Error messages.

The following example from the `SimpleSelect.java` program shows how to use the **SQLException** class to catch HCL Informix® JDBC Driver or database server errors by using a try-catch block:

```
try
   {
   PreparedStatement pstmt = conn.prepareStatement("Select *
      from x "
      + "where a = ?;");
   pstmt.setInt(1, 11);
   ResultSet r = pstmt.executeQuery();
   while(r.next())
      {
      short i = r.getShort(1);
      System.out.println("Select: column a = " + i);
      }
   r.close();
   pstmt.close();
   }
catch (SQLException e)
   {
   System.out.println("ERROR: Fetch statement failed: " +
      e.getMessage());
   }
```

Retrieve the syntax error offset

To determine the exact location of a syntax error, use the getSQLStatementOffset() method to return the syntax error offset.

The following example shows how to retrieve the syntax error offset from an SQL statement (which is `10` in this example):

```
try {
  Statement stmt = conn.createStatement();
  String command = "select * fom tt";
  stmt.execute( command );
}
catch(Exception e)
{
  System.out.println
  ("Error Offset :"+((IfmxConnection conn).getSQLStatementOffset() );
  System.out.println(e.getMessage() );
}
```

Catch RSAM error messages

RSAM messages are attached to SQLCODE messages. For example, if an SQLCODE message says that a table cannot be created, the RSAM message states the reason, which might be insufficient disk space.

You can use the SQLException.getNextException() method to catch RSAM error messages. For an example of how to catch these messages, see the `ErrorHandling.java` program, which is included in HCL Informix® JDBC Driver.

Handle errors with the com.informix.jdbc.Message class

HCL Informix® provides the class **com.informix.jdbc.Message** for retrieving Informix® error message text based on the Informix® error number. To use this class, call the Java™ interpreter **java** directly, passing it the Informix® error number, as shown in the following example:

```
java com.informix.jdbc.Message 100
```

The example returns the message text for Informix® error 100:

```
100: ISAM error: duplicate value for a record with unique key.
```

A positive error number is returned if you specify an unsigned number when using the **com.informix.jdbc.Message** class. This differs from the finderr utility, which returns a negative error number for an unsigned number.

Access database metadata

To access information about the HCL Informix® database, use the JDBC API **DatabaseMetaData** interface.

HCL Informix® JDBC Driver implements all the JDBC 3.0 specifications for **DatabaseMetaData** methods.

The following methods in **DatabaseMetaData** are included in HCL Informix® JDBC Driver for JDBC 3.0 compliance:

- getSuperTypes()
- getSuperTables()
- getAttributes()
- getResultSetHoldability()
- getDatabaseMajorVersion()
- getDatabaseMinorVersion()
- getJDBCMajorVersion()
- getJDBCMinorVersion()
- getSQLStateType()
- locatorsUpdateCopy()
- supportsGetGeneratedKeys()
- supportsMultipleOpenResults()
- supportsNamedParameters()
- supportsGetGeneratedKeys()
- supportsMultipleOpenResults()

Methods retrieve server-generated keys. Retrieving autogenerated keys involves the following actions:

1. The JDBC application programmer provides an SQL statement to be executed.
2. The server executes the SQL statement and an indication that autogenerated keys can be retrieved is returned.
3. Before the server executes the SQL statement, **columnNames** or **columnIndexes** (if provided) are validated. An **SQLException** is thrown if they are invalid.

4. If requested, the JDBC driver and server returns a **resultSet** object. If no keys were generated, the **resultSet** is empty, containing no rows or columns.
5. The user can request metadata for the **resultSet** object, and the JDBC driver and server returns a **resultSetMetaData** Object.

For more information about retrieving autogenerated keys, see the JDBC 3.0 Specification, Section 13.6, "Retrieving Auto Generated Keys."

HCL Informix® JDBC Driver uses the **sysmaster** database to get database metadata. If you want to use the **DatabaseMetaData** interface in your Java™ program, the **sysmaster** database must exist in the Informix® database server to which your Java™ program is connected.

HCL Informix® JDBC Driver interprets the JDBC API term *schemas* to mean the names of Informix® users who own tables. The DatabaseMetaData.getSchemas() method returns all the users found in the **owner** column of the **systables** system catalog.

Similarly, HCL Informix® JDBC Driver interprets the JDBC API term *catalogs* to mean the names of Informix® databases. The DatabaseMetaData.getCatalogs() method returns the names of all the databases that currently exist in the Informix® database server to which your Java™ program is connected.

The example `DBMetaData.java` shows how to use the **DatabaseMetaData** and **ResultSetMetaData** interfaces to gather information about a new procedure. Refer to Sample code files on page 224 for more information about this example.

Other Informix® extensions to the JDBC API

This section describes the HCL Informix®-specific extensions to the JDBC API not already discussed in this guide. These extensions handle information that is specific to HCL Informix® databases.

Another Informix® extension, the **com.informix.jdbc.Message** class, is fully described in Handle errors on page 84.

The Auto Free feature

If you enable the HCL Informix® Auto Free feature, the database server automatically frees the cursor when it closes the cursor. Therefore, your application does not have to send two separate requests to close and then free the cursor—closing the cursor is sufficient.

You can enable the Auto Free feature by setting the **IFX_AUTOFREE** variable to `TRUE` in the database URL, as in this example:

```
jdbc:informix-sqli://123.45.67.89:1533:INFORMIXSERVER=myserver;
   user=rdtest;password=test;ifx_autofree=true;
```

You can also use one of the following methods:

```
public void setAutoFree (boolean flag)
public boolean getAutoFree()
```

The setAutoFree() method should be called before the executeQuery() method, but the getAutoFree() method can be called before or after the executeQuery() method.

To use these methods, your applications must import classes from the Informix® package `com.informix.jdbc` and cast the **Statement** class to the **IfmxStatement** class, as shown here:

```
import com.informix.jdbc.*;
...
(IfmxStatement)stmt.setAutoFree(true);
```

The Auto Free feature is available for the following database server versions:

- Version 7.23 and later
- Version 9.0 and later

Obtaining driver version information

About this task

There are two ways to obtain version information about HCL Informix® JDBC Driver: from your Java™ program or from the UNIX™ or MS-DOS command prompt.

To get version information from your Java™ program:

1. Import the Informix® package `com.informix.jdbc.*` into your Java™ program by adding the following line to the import section:

   ```
   import com.informix.jdbc.*;
   ```

2. Invoke the static method IfxDriver.getJDBCVersion().
   This method returns a **String** object that contains the complete version of the current HCL Informix® JDBC Driver.

   An example of a version of HCL Informix® JDBC Driver is 2.00.JC1.

   The IfxDriver.getJDBCVersion() method returns only the version, not the serial number you provided during installation of the driver.

Results

Important: For version X.Y of HCL Informix® JDBC Driver, the JDBC API methods Driver.getMajorVersion() and DatabaseMetaData.getDriverMajorVersion() always return the value X. Similarly, the methods Driver.getMinorVersion() and DatabaseMetaData.getDriverMinorVersion() always return the value Y.

To get the version of HCL Informix® JDBC Driver from the command line, enter the following command at the UNIX™ shell prompt or the Windows™ command prompt:

```
java com.informix.jdbc.Version
```

The command also returns the serial number you provided when you installed the driver.

Store and retrieve XML documents

Extensible Markup Language (XML), as defined by the World Wide Web Consortium (W3C) provides rules, guidelines, and conventions for describing structured data in a plain text, editable file (called an *XML document*). XML uses tags only to delimit pieces of data, leaving the interpretation of the data to the application that uses it. XML is an method of representing data in an open, platform-independent format.

The currently available API for accessing XML documents is called JAXP (Java™ API for XML Parsing). The API has the following two subsets:

- **Simple API for XML** (SAX) is an event-driven protocol, with the programmer providing the callback methods that the XML parser invokes when it analyzes a document.
- **Document Object Model** (DOM) is a random-access protocol, which converts an XML document into a collection of objects in memory that can be manipulated at the programmers discretion. DOM objects have the data type Document.

JAXP also contains a *plugability layer* that standardizes programmatic access to SAX and DOM by providing standard factory methods for creating and configuring SAX parsers and creating DOM objects.

HCL Informix® extensions to the JDBC API facilitate storage and retrieval of XML data in database columns. The methods used during data storage assist in parsing the XML data, verify that well-formed and valid XML data is stored, and ensure that invalid XML data is rejected. The methods used during data retrieval assist in converting the XML data to DOM objects and to type **InputSource**, which is the standard input type to both SAX and DOM methods. The Informix® extensions are designed to support XML programmers while still providing flexibility regarding which JAXP package the programmer is using.

Set up your environment to use XML methods

This section contains information you need to know to prepare your system to use the JDBC driver XML methods.

Set your CLASSPATH

To use the XML methods, add the path names of the following files to your CLASSPATH setting:

- `ifxtools.jar`
- `xerces.jar`

All of these files are located in the `lib` directory where you installed your driver.

The Xerces XML library `xerces.jar` has been removed from distribution with the HCL Informix® JDBC Driver, Version 3.00.

The XML methods are not part of the `ifxjdbc.jar` file. Instead, they are released in a separate `.jar` file named `ifxtools.jar`. To use the methods, you must add this file to your CLASSPATH setting along with `ifxjdbc.jar`.

In addition, building `ifxtools.jar` requires that you use code from a `.jar` file that supports the SAX, DOM, and JAXP methods. To use `ifxtools.jar`, you must add these `.jar` files to your CLASSPATH setting.

The Java development kit uses the default XML parser even if the xml4j parser is in the CLASSPATH. To use the xml4j implementation of the SAX parser, set the following system properties in the application code or use the **-D** command-line option:

- The property **javax.xml.parsers.SAXParserFactory** must be set to **org.apache.xerces.jaxp.SAXParserFactoryImpl**.
- For the Document Object Model, the property **javax.xml.parsers.DocumentBuilderFactory** must be set to **org.apache.xerces.jaxp.DocumentBuilderFactoryImpl**.

For more info about how to set the properties, see Specify a parser factory on page 90.

Specify a parser factory

By default, the xml4j xerces parser (and as a result, `ifxtools.jar`) uses the non-validating XML parser. To use an alternative SAX parser factory, run your application from the command line as follows:

```
% java -Djavax.xml.parsers.SAXParserFactory=new-factory
```

If you are not running from the command line, the factory name must be enclosed in double quotation marks:

```
% java -Djavax.xml.parsers.SAXParserFactory="new-factory"
```

You can also set a system property in your code:

```
System.setProperty("javax.xml.parsers.SAXParserFactory",
   "new-factory")
```

In this code, *new-factory* is the alternative parser factory. For example, if you are using the xerces parser, then *new-factory* is replaced by **org.apache.xerces.jaxp.SAXParserFactoryImpl**.

It is also possible to use an alternative document factory for DOM methods. Run your application from the command line as follows:

```
% java -Djavax.xml.parsers.DocumentBuilderFactory=new-factory
```

If you are not running from the command line, the factory name must be enclosed in double quotation marks:

```
% java -Djavax.xml.parsers.DocumentBuilderFactory="new-factory"
```

You can also set a system property in your code:

```
System.setProperty("javax.xml.parsers.DocumentBuilderFactory",
   "new-factory")
```

For example, if you are using the xerces parser, then *new-factory* is replaced by **jorg.apache.xerces.jaxp.DocumentBuilderFactoryImpl**.

Insert data

You can use the methods in this section to insert XML data into a database column.

The parameters in method declarations in this section have the following meanings:

- The *file* parameter is an XML document. The document can be referenced by a URL (such as **http://server/file.xml** or **file:///path/file.xml**) or a path name (such as `/tmp/file.xml` or `c:\\work\\file.xml`).
- The *handler* parameter is an optional class you supply, containing callback routines that the SAX parser invokes as it is parsing the file. If no value is specified, or if *handler* is set to `NULL`, the driver uses empty callback routines that echo success or failure (the driver reports failure in the form of an **SQLException**).
- The *validating* parameter tells the SAX parser factory to use a validating parser instead of a parser that only checks form.

  If you do not specify *nsa* or *validating*, the driver uses the xml4j nonvalidating XML parser. To change the default, see Specify a parser factory on page 90.
- The *nsa* parameter tells the SAX parser factory whether it can use a parser that can handle namespaces.

The following methods parse a file by using SAX and convert it to a string. You can then use the string returned by these methods as input to the PreparedStatement.setString() method to insert the data into a database column.

```
public String XMLtoString(String file, String handler, boolean
   validating,boolean nsa) throws SQLException

public String XMLtoString(String file, String handler) throws
   SQLException

public String XMLtoString(String file) throws SQLException
```

The following methods parse a file by using SAX and convert it to an object of class **InputStream**. You can then use the **InputStream** object as input to the PreparedStatement.setAsciiStream(), PreparedStatement.setBinaryStream(), or PreparedStatement.setObject() methods to insert the data into a database column.

```
public InputStream XMLtoInputStream(String file, String handler,
   boolean validating,boolean nsa) throws SQLException;

public InputStream XMLtoInputStream(String file, String handler)
   throws SQLException;

public InputStream XMLtoInputStream(String file) throws
   SQLException;
```

For examples of using these methods, see Insert data examples on page 92.

If no value is specified, or if *handler* is set to `NULL`, the driver uses the default HCL Informix® handler.

Important: The driver truncates any input data that is too large for a column. For example, if you insert the `x.xml` file into a column of type char (55) instead of a column of type char (255), the driver inserts the truncated file with no

errors (the driver throws an SQLWarn exception, however). When the truncated row is selected, the parser throws a SAXParseException because the row contains invalid XML.

Retrieve data

You can use the methods in this section to convert XML data that has been fetched from a database column. These methods help you either convert selected XML text to DOM or parse the data with SAX. The **InputSource** class is the input type to JAXP parsing methods.

For information about the *file*, *handler*, *nsa*, and *validating* parameters, see Insert data on page 90.

The following methods convert objects of type String or InputStream to objects of type InputSource. You can use the ResultSet.getString(), ResultSet.getAsciiStream(), or ResultSet.getBinaryInputStream() methods to retrieve the data from the database column and then pass the retrieved data to getInputSource() for use with any of the SAX or DOM parsing methods. (For an example, see Retrieve data examples on page 93.)

```
public InputSource getInputSource(String s) throws SQLException;

public InputSource getInputSource(InputStream is) throws
   SQLException;
```

The following methods convert objects of type String or InputStream to objects of type Document:

```
public Document StringtoDOM(String s, String handler, boolean
   validating,boolean nsa) throws SQLException

public Document StringtoDOM(String s, String handler) throws
   SQLException

public Document StringtoDOM(String s) throws SQLException

public Document InputStreamtoDOM(String s, String handler, boolean
   validating,boolean nsa) throws SQLException

public Document InputStreamtoDOM(String file, String handler)
   throws SQLException

public Document InputStreamtoDOM(String file) throws SQLException
```

For examples of using these methods, see Retrieve data examples on page 93.

Insert data examples

The examples in this section illustrate converting XML documents to formats acceptable for insertion into HCL Informix® database columns.

The XMLtoString() examples

The following example converts three XML documents to character strings and then uses the strings as parameter values in an SQL INSERT statement:

```
PreparedStatement p = conn.prepareStatement("insert into tab
   values(?,?,?)");
p.setString(1, UtilXML.XMLtoString("/home/file1.xml"));
p.setString(2, UtilXML.XMLtoString("http://server/file2.xml");
p.setString(3, UtilXML.XMLtoString("file3.xml");
```

The following example inserts an XML file into an LVARCHAR column. In this example, **tab1** is a table created with the SQL statement:

```
create table tab1 (col1 lvarchar);
```

The code is:

```
try
   {
   String cmd = "insert into tab1 values (?)";
   PreparedStatement pstmt = conn.prepareStatement(cmd);
   pstmt.setString(1, UtilXML.XMLtoString("/tmp/x.xml"));
   pstmt.execute();
   pstmt.close();
   }
    catch (SQLException e)
   {
   // Error handling
   }
```

The XMLtoInputStream() example

The following example inserts an XML file into a text column. In this example, table **tab2** is created with the SQL statement:

```
create table tab2 (col1 text);
```

The code is:

```
try
   {
   String cmd = "insert into tab2 values (?)";
   PreparedStatement pstmt = conn.prepareStatement(cmd);
   pstmt.setAsciiStream(1, UtilXML.XMLtoInputStream("/tmp/x.xml"),
      (int)(new File("/tmp/x.xml").length()));
    pstmt.execute();
    pstmt.close();
    }
    catch (SQLException e)
    {
    // Error handling
    }
```

Retrieve data examples

The following examples illustrate retrieving data from HCL Informix® database columns and converting the data to formats acceptable to XML parsers.

The StringtoDOM() example

This example operates under the assumption that **xmlcol** is a column of type **lvarchar** that contains XML data. The data could be fetched and converted to DOM with the following code:

```
ResultSet r = stmt.executeQuery("select xmlcol from table where
      ...");
while (r.next()
    {
    Document doc= UtilXML.StringtoDOM(r.getString("xmlcol"));
    // Process ‘doc'
    }
```

The InputStreamtoDOM() example

The following example fetches XML data from a text column into a DOM object:

```
try
    {
    String sql = "select col1 from tab2";
    Statement stmt = conn.createStatement();
    ResultSet r = stmt.executeQuery(sql);
     while(r.next())
        {
        Document doc = UtilXML.InputStreamtoDOM(r.getAsciiStream(1));
        }
    r.close();
    }
    catch (Exception e)
    {
    // Error handling
    }
```

The getInputSource() examples

This example retrieves the XML data stored in column **xmlcol** and converts it to an object of type InputSource; the InputSource object `i` can then be used with any SAX or DOM parsing methods:

```
InputSource i = UtilXML.getInputSource
   (resultset.getString("xmlcol"));
```

This example uses the implementation of JAXP API, in `xerces.jar`, to parse fetched XML data in column **xmlcol**:

```
InputSource input = UtilXML.getInputSource(resultset.getString("xmlcol"));
SAXParserFactory f = SAXParserFactory.newInstance();
SAXParser parser = f.newSAXParser();
parser.parse(input);
```

In the examples that follow, **tab1** is a table created with the SQL statement:

```
create table tab1 (col1 lvarchar);
```

The following example fetches XML data from an LVARCHAR column into an **InputSource** object for parsing. This example uses SAX parsing by invoking the parser at **org.apache.xerces.parsers.SAXParser**.

```
try
    {
```

```
    String sql = "select col1 from tab1";
    Statement stmt = conn.createStatement();
    ResultSet r = stmt.executeQuery(sql);
    Parser p = ParserFactory.makeParser("org.apache.xerces.parsers.SAXParser");
    while(r.next())
        {
        InputSource i = UtilXML.getInputSource(r.getString(1));
        p.parse(i);
        }
    r.close();
    }
    catch (SQLException e)
    {
    // Error handling
    }
```

The following example fetches XML data from a text column into an **InputSource** object for parsing. This example is the same example as the previous one, but it uses JAXP factory methods instead of the SAX parser to analyze the data.

```
try
    {
    String sql = "select col1 from tab2";
    Statement stmt = conn.createStatement();
    ResultSet r = stmt.executeQuery(sql);
    SAXParserFactory factory = SAXParserFactory.newInstance();
    Parser p = factory.newSAXParser();
    while(r.next())
        {
        InputSource i = UtilXML.getInputSource(r.getAsciiStream(1));
        p.parse(i);
        }
    r.close();
    }
    catch (Exception e)
    {
    // Error handling
    }
```

Work with Informix® types

These topics explain the data types that are specific to HCL Informix® (other than opaque types) supported in HCL Informix® JDBC Driver. For information about opaque types, see Work with opaque types on page 154.

Distinct data types

A distinct type can map to the underlying base type or to a user-defined Java™ object. For example, a distinct type of INT can map to int or to a Java™ object that encapsulates the data representation. This Java™ object must implement the java.sql.SQLData interface. You must provide a custom type map as described in Mapping data types on page 241, to map this Java™ object to the corresponding SQL type name.

Insert data examples

The following example shows an SQL statement that defines a distinct type:

```
CREATE DISTINCT TYPE mymoney AS NUMERIC(10, 2);
CREATE TABLE distinct_tab (mymoney_col mymoney);
```

The following is an example of mapping to the base type:

```
String s = "insert into distinct_tab (mymoney_col) values (?)";
System.out.println(s);
pstmt = conn.prepareStatement(s);

...
BigDecimal bigDecObj = new BigDecimal(123.45);
pstmt.setBigDecimal(1, bigDecObj);
System.out.println("setBigDecimal...ok");
pstmt.executeUpdate();
```

When you map to the underlying type, HCL Informix® JDBC Driver performs the mapping on the client side because the database server provides implicit casting between the underlying type and the distinct type.

You can also map distinct types to Java™ objects that implement the **SQLData** interface. The following example shows an SQL statement that defines a distinct type:

```
CREATE DISTINCT TYPE mymoney AS NUMERIC(10,2)
```

The following code maps the distinct type to a Java™ object named **MyMoney**:

```
import java.sql.*;
import com.informix.jdbc.*;
public class myMoney implements SQLData
{
     private String sql_type = "mymoney";
     public java.math.BigDecimal value;
     public myMoney() { }

     public myMoney(java.math.BigDecimal value)

         this.value = value;

     public String getSQLTypeName()
     {
         return sql_type;
     {

     public void readSQL(SQLInput stream, String type) throws
   SQLException
     {
         sql_type = type;
         value = stream.readBigDecimal();
     {

     public void writeSQL(SQLOutput stream) throws SQLException
     {
         stream.writeBigDecimal(value);
     }
     // overides Object.equals()
     public boolean equals(Object b)

         return value.equals(((myMoney)b).value);
```

```
     }
     public String toString()
     {
         return "value=" + value;
     }
}
...
String s - "insert into distinct_tab (mymoney_col) values (?)";
pstmt = conn.prepareStatement(s);
myMoney mymoney = new myMoney();
mymoney.value = new java.math.BigDecimal(123.45);
pstmt.setObject(1, mymoney);
System.out.println("setObject(myMoney)...ok");
pstmt.executeUpdate();
```

In this case, you use the setObject() method instead of the setBigDecimal() method to insert data.

Retrieve data example

You can fetch a distinct type as its underlying base type or as a Java™ object, if the mapping is defined in a custom type map. Using the previous example, you can fetch the data as a Java™ object, as shown in the following example:

```
java.util.Map customtypemap = conn.getTypeMap();
System.out.println("getTypeMap...ok");
if (customtypemap == null)
{
   System.out.println("\n***ERROR: typemap is null!");
   return;
}
customtypemap.put("mymoney", Class.forName("myMoney"));

...
String s = "select mymoney_col from distinct_tab order by 1";
try
{
   Statement stmt = conn.createStatement();
   ResultSet rs = stmt.executeQuery(s);
   System.out.println("Fetching data ...");
   int curRow = 0;
   while (rs.next())
   {
      curRow++;
      myMoney mymoneyret = (myMoney)rs.getObject("mymoney_col");
   }
   System.out.println("total rows expected: " + curRow);
   stmt.close();
}
catch (SQLException e)
{
   System.out.println("***ERROR: " +   e.getErrorCode() + " " +
                                    e.getMessage());
   e.printStackTrace();
}
```

In this case, you use the getObject() method instead of the getBigDecimal() method to retrieve data.

Unsupported methods

The following methods of the **SQLInput** and **SQLOutput** interfaces are not supported for distinct types:

- **java.sql.SQLInput**
  - readArray()
  - readCharacterStream()
  - readRef()
- **java.sql.SQLOutput**
  - writeArray()
  - writeCharacterStream(Reader x)
  - writeRef(Ref x)

BYTE and TEXT data types

This section describes the HCL Informix® BYTE and TEXT data types and how to manipulate columns of these data types with the JDBC API.

The BYTE data type is a data type for a simple large object that stores any data in an undifferentiated byte stream. Examples of this binary data include spreadsheets, digitized voice patterns, and video clips. The TEXT data type is a data type for a simple large object that stores any text data. It can contain both single and multibyte characters.

Columns of either data type have a theoretical limit of 2^{31} bytes and a practical limit determined by your disk capacity.

For more detailed information about the Informix® BYTE and TEXT data types, see *HCL® Informix® Guide to SQL: Reference* and *HCL® Informix® Guide to SQL: Syntax*.

Cache large objects

Whenever an object of type BLOB, CLOB, text, or byte is fetched from the database server, the data is cached in client memory. If the size of the large object is bigger than the value in the **LOBCACHE** environment variable, the large object data is stored in a temporary file. For more information about the **LOBCACHE** variable, see Manage memory for large objects on page 217.

Example: Inserting or updating data

To insert into or update BYTE and TEXT columns, read a stream of data from a source, such as an operating system file, and transmit it to the database as a **java.io.InputStream** object. The **PreparedStatement** interface provides methods for setting an input parameter to this Java™ input stream. When the statement is executed, HCL Informix® JDBC Driver makes repeated calls to the input stream, reading its contents and transmitting those contents as the actual parameter data to the database.

For BYTE data types, use the PreparedStatement.setBinaryStream() method to set the input parameter to the **InputStream** object. For TEXT data types, use the PreparedStatement.setAsciiStream() method.

The following example from the `ByteType.java` program shows how to insert the contents of the operating system file `data.dat` into a column of data type BYTE:

```
try
{
        stmt = conn.createStatement();
        stmt.executeUpdate("create table tab1(col1 byte)");
}
catch (SQLException e)
{
        System.out.println("Failed to create table ..." + e.getMessage());
}

try
{
        pstmt = conn.prepareStatement("insert into tab1 values (?)");
}
catch (SQLException e)
{
        System.out.println("Failed to Insert into tab: " + e.toString());
}

File file = new File("data.dat");
int fileLength = (int) file.length();
InputStream value = null;
FileInputStream fileinp = null;
int row = 0;
String str = null;
int        rc = 0;
ResultSet rs = null;

System.out.println("Inserting data ...\n");

try
{
        fileinp =  new FileInputStream(file);
        value = (InputStream)fileinp;
}
catch (Exception e) {}

try
{
        pstmt.setBinaryStream(1,value,10); //set 1st column
}
catch (SQLException e)
{
        System.out.println("Unable to set parameter");
}

set_execute();




...
public static void set_execute()
{
try
{
        pstmt.executeUpdate();
}
```

```
catch (SQLException e)
{
   System.out.println("Failed to Insert into tab: " + e.toString());
   e.printStackTrace();
}
}
```

The example first creates a **java.io.File** object that represents the operating system file `data.dat`. The example then creates a **FileInputStream** object to read from the object of type **File**. The object of type **FileInputStream** is cast to its superclass **InputStream**, which is the expected data type of the second parameter to the PreparedStatement.setBinaryStream() method. The setBinaryStream() method executes on the already prepared INSERT statement, which sets the input stream parameter. Finally, the PreparedStatement.executeUpdate() method executes, which inserts the contents of the `data.dat` operating system file into the column of type BYTE.

The `TextType.java` program shows how to insert data into a column of type TEXT. It is similar to inserting into a column of type BYTE, except the method setAsciiStream() is used to set the input parameter instead of setBinaryStream().

Example: Selecting data

After you select from a table into a **ResultSet** object, you can use the ResultSet.getBinaryStream() method to retrieve a stream of binary or ASCII data from the columns of type BYTE. You can also use the ResultSet.getAsciiStream() method to retrieve a stream of binary or ASCII data from the columns of type TEXT. Both methods return an **InputStream** object, which can be used to read the data in chunks.

All the data in the returned stream in the current row must be read before you call the next() method to retrieve the next row.

The following example from the `ByteType.java` program shows how to select data from a column of type BYTE and print out the data to the standard output device:

```
try
{
        stmt = conn.createStatement();
        rs =  stmt.executeQuery("Select * from tab1");
        while( rs.next() )
       {
            row++;
            value = rs.getBinaryStream(1);
            dispValue(value);
       }
}
catch (Exception e) { }

...

public static void dispValue(InputStream in)
{
        int size;
        byte buf;
        int count = 0;
        try
        {
             size = in.available();
             byte ary[] = new byte[size];
```

```
             buf = (byte) in.read();
             while(buf!=-1)
            {
                  ary[count] = buf;
                  count++;
                  buf = (byte) in.read();
            }
        }
        catch (Exception e)
        {
             System.out.println("Error occured while reading stream ... \n");
        }
}
```

The example first puts the result of a SELECT statement into a **ResultSet** object. It then executes the method ResultSet.getBinaryStream() to retrieve the BYTE data into a Java™ **InputStream** object.

The method dispValue(), whose Java™ code is also included in the example, is used to print out the contents of the column to the standard output device. The dispValue() method uses byte arrays and the InputStream.read() method to systematically read the contents of the column of type BYTE.

The `TextType.java` program shows how to select data from a column of type TEXT. It is similar to selecting from a column of type BYTE, except the getAsciiStream() method is used instead of getBinaryStream().

Binary data types

The binary18 and binaryvar data types allow user to store binary-encoded strings which can be indexed for quick retrieval.

You can use string manipulation functions to validate the data types and bitwise operation functions that allow you to perform bitwise logical AND, OR, XOR comparisons or apply a bitwise logical NOT to a string.

Since the binary data types are unstructured types, they can store many different types of information, for example, IP addresses, MAC addresses, or device identification numbers from RFID tags. The binary data types can also store encrypted data in binary format, which saves disk space. Instead of storing an IP address like xxx.xxx.xxx.xxx as a CHAR(15) data type, you can store it as a binaryvar data type, which uses only 6 bytes.

You can store and index binary data by using the binaryvar and binary18 data types.

Binaryvar data type

The binaryvar data type is a variable-length opaque type with a maximum length of 255 bytes.

Binary18 data type

The binary18 data type is a fixed-length opaque data type that holds 18 bytes. Input strings shorter than 18 bytes are right padded with zeros (00). Strings longer than 18 bytes are truncated.

The binary18 data type has the advantage of not having its length stored as part of the byte stream. When inserting data into the binaryvar data type, the first byte must be the length of the byte array. The binary18 data type does not have this restriction.

Related information

Binary data types on page

SERIAL and SERIAL8 data types

HCL Informix® JDBC Driver provides support for the Informix® SERIAL and SERIAL8 data types through the methods getSerial() and getSerial8(), which are part of the implementation of the **java.sql.Statement** interface.

Because the SERIAL and SERIAL8 data types do not have an obvious mapping to any JDBC API data types from the **java.sql.Types** class, you must import classes that are specific to Informix® into your Java™ program to handle SERIAL and SERIAL8 columns. To do this, add the following import line to your Java™ program:

```
import com.informix.jdbc.*;
```

Use the getSerial() method after an INSERT statement to return the serial value that was automatically inserted into the SERIAL column of a table. Use the getSerial8() method after an INSERT statement to return the serial value that was automatically inserted into the SERIAL8 column of a table. The methods return `0` if any of the following conditions are true:

- The last statement was not an INSERT statement.
- The table being inserted into does not contain a SERIAL or SERIAL8 column.
- The INSERT statement has not executed yet.

If you execute the getSerial() or getSerial8() method after a CREATE TABLE statement, the method returns `1` by default (assuming the new table includes a SERIAL or SERIAL8 column). If the table does not contain a SERIAL or SERIAL8 column, the method returns `0`. If you assign a new serial starting number, the method returns that number.

If you want to use the getSerial() and getSerial8() methods, you must cast the **Statement** or **PreparedStatement** object to **IfmxStatement**, the implementation of the **Statement** interface, which is specific to Informix®. The following example shows how to perform the cast:

```
cmd = "insert into serialTable(i) values (100)";
stmt.executeUpdate(cmd);
System.out.println(cmd+"...okay");
int serialValue = ((IfmxStatement)stmt).getSerial();
System.out.println("serial value: " + serialValue);
```

If you want to insert consecutive serial values into a column of data type SERIAL or SERIAL8, specify a value of `0` for the SERIAL or SERIAL8 column in the INSERT statement. When the column is set to `0`, the database server assigns the next-highest value.

For more detailed information about the Informix® SERIAL and SERIAL8 data types, see the *HCL® Informix® Guide to SQL: Reference* and the *HCL® Informix® Guide to SQL: Syntax*.

BIGINT and BIGSERIAL data types

The BIGINT and BIGSERIAL data types have the same range of values as INT8 and SERIAL8 data types. However, BIGINT and BIGSERIAL have advantages for storage and computation over INT8 and SERIAL8.

Both the BIGINT and BIGSERIAL data types map to the to BIGINT Java™ type in the class **java.sql.Types**. When data is retrieved from the database, the BIGINT and BIGSERIAL data types map to long Java™ Type.

The Informix® JDBC Driver provides support for the Informix® BIGSERIAL and BIGINT data types through the getBigSerial() method, which is a part of the **java.sql.Statement** interface

Because the BIGSERIAL and BIGINT data types do not have an obvious mapping to any JDBC API data types from the **java.sql.Types** class, you must import classes that are specific to Informix® into your Java™ program to handle BIGSERIAL and BIGINT columns. To do this, add the following import line to your Java™ program:

```
import com.informix.jdbc.*;
```

Use the getBigSerial() method after an INSERT statement to return the value that was inserted into the BIGSERIAL or BIGINT column of a table.

If you want to use the getBigSerial() method, you must cast the **Statement** or **PreparedStatement** object to **IfmxStatement**, the implementation of the **Statement** interface, which is specific to Informix®. The following example shows how to perform the cast:

```
cmd = "insert into bigserialTable(i) values (100)";
stmt.executeUpdate(cmd);
System.out.println(cmd+"...okay");
long serialValue = ((IfmxStatement)stmt).getBigSerial();
System.out.println("serial value: " + serialValue);
```

These types are part of the **com.informix.lang.IfxTypes** class. See the The IfxTypes class on page 249 table for the IfxTypes constants and the corresponding Informix® data types.

INTERVAL data type

The HCL Informix® INTERVAL data type stores a value that represents a span of time. INTERVAL data types comprise two types: year-month intervals and day-time intervals. A year-month interval can represent a span of years and months, and a day-time interval can represent a span of days, hours, minutes, seconds, and fractions of a second. For more information about the INTERVAL data type and definitions of *qualifier*, *precision*, and *fraction*, see the following publications:

- *HCL® Informix® Guide to SQL: Tutorial*
- *HCL® Informix® Guide to SQL: Reference*
- *HCL® Informix® Guide to SQL: Syntax*

The Interval class

The **com.informix.lang.Interval** class is the HCL Informix®-specific extension to the JDBC specification. Interval is the base class for the INTERVAL data type. Interval has two subclasses: IntervalYM (for year-month qualifiers) and IntervalDF (for day-time qualifiers). You use these subclasses to create and manipulate INTERVAL data types.

Tip: Many of the **Interval**, **IntervalYM**, and **IntervalDF** constructors take a **Connection** object as a parameter. This passes the value of the **CLIENT_LOCALE** environment variable to the **Interval**, **IntervalYM**, or **IntervalDF** object, which allows the display of localized error messages if an exception is thrown. For more information, see Support for globalized error messages on page 210.

For information about the string INTERVAL formats in this section, see the *HCL® Informix® Guide to SQL: Syntax*.

This section discusses many of the methods you can use with the INTERVAL data types. For complete reference information, see the online reference documentation in the directory `doc/javadoc/*` after you install your software. (The `doc` directory is a subdirectory of the directory where you installed HCL Informix® JDBC Driver.)

Variables for binary qualifiers

You can use string qualifiers to manipulate INTERVAL data types, but using binary qualifiers results in faster performance. The following variables are defined in the **Interval** base class and represent the time unit (start and end code) of a field in the binary qualifier. To use these variables, instantiate objects of the **IntervalYM** and **IntervalDF** classes, which inherit these variables from the **Interval** base class.

TU_YEAR

Time unit for the YEAR qualifier field

TU_MONTH

Time unit for the MONTH qualifier field

TU_DAY

Time unit for the DAY qualifier field

TU_HOUR

Time unit for the HOUR qualifier field

TU_MINUTE

Time unit for the MINUTE qualifier field

TU_SECOND

Time unit for the SECOND qualifier field

TU_FRAC

Time unit for the leading FRACTION qualifier field

TU_F1

Time unit for the ending field of the first position of FRACTION

TU_F2

Time unit for the ending field of the second position of FRACTION

TU_F3

Time unit for the ending field of the third position of FRACTION

TU_F4

Time unit for the ending field of the fourth position of FRACTION

TU_F5

Time unit for the ending field of the fifth position of FRACTION

Interval methods

You can use the **Interval** methods to extract information about binary qualifiers. To use these methods, instantiate objects of the **IntervalYM** and **IntervalDF** classes, which inherit these variables from the **Interval** base class.

Some of the tasks you can perform and the methods you can use follow:

- Extracting the length of a qualifier:

```
public static byte getLength(short qualifier)
```

- Extracting the starting field code (one of the TU_*XXX* variables) from a qualifier:

```
public static byte getStartCode(short qualifier)
```

- Extracting the ending field code (one of the TU_*XXX* variables) from a qualifier:

```
public static byte getEndCode(short qualifier)
```

- Obtaining the string value that corresponds to the TU_*XXX* value of part of an interval (for example, `getFieldName(TU_YEAR)` returns the string `year`):

```
public static String getFieldName(byte code)
```

- Obtaining the entire name of the interval as a character string, taking a qualifier as input:

```
public static String getIfxTypeName(int type,
   short qualifier)
```

- Obtaining the number of digits in the FRACTION part of the INTERVAL data type:

```
public static byte getScale(short qualifier)
```

- Creating a binary qualifier from a length, start code (TU_*XXX*), and end code (TU_*XXX*):

```
public static short getQualifier(byte length, byte
   startCode, byte endCode) throws SQLException
```

  For example, `getQualifier(4, TU_YEAR, TU_MONTH)` creates a binary representation of the YEAR TO MONTH qualifier.

The IntervalYM class

The **com.informix.lang.IntervalYM** class allows you to manipulate year-month intervals.

The IntervalYM constructors

The default constructor is defined as follows:

```
public IntervalYM() throws SQLException
```

Use this second version of the constructor to display localized error messages if an exception is thrown:

```
public IntervalYM(Connection conn) throws SQLException
```

Use the following constructors to create year-month intervals from specific input values:

- Two time stamps, returning the IntervalYM value that equals *Timestamp1* - *Timestamp2*:

  ```
  public IntervalYM(Timestamp t1, Timestamp t2) throws
     SQLException
  public IntervalYM (Timestamp t1, Timestamp t2, Connection
     conn) throws SQLException
  ```

  The second version allows you to support localized error messages.

- Year and month values (large month values are converted to year):

  ```
  public IntervalYM(int years, int months) throws
     SQLException

  public IntervalYM(int years, int months,
     Connection conn) throws SQLException
  ```

  The second version allows you to support localized error messages.

- A month value and the encoded qualifier:

  ```
  public IntervalYM(int months, short qualifier,
     Connection conn) throws SQLException
  ```

  To specify the qualifier, you can use the getQualifier() method described in Interval methods on page 105. This constructor supports localized error messages.

- A string:

  ```
  public IntervalYM(String string) throws SQLException
  public IntervalYM(String string, Connection conn) throws
     SQLException
  ```

  The second version allows you to support localized error messages.

- A string and qualifier:

  ```
  public IntervalYM(String string, short qualifier,
     Connection conn) throws SQLException
  ```

  To specify the qualifier, you can use the getQualifier() method described in Interval methods on page 105. This constructor supports localized error messages.

- A string and qualifier information:

  ```
  public IntervalYM(String string, int length,
     byte startCode, byte endCode) throws SQLException

  public IntervalYM(String string, int length,
  ```

```
byte startCode, byte endCode, Connection conn) throws
SQLException
```

The second version allows you to support localized error messages.

The IntervalYM methods

The following methods allow you to manipulate year-month intervals. (You can also use the **Interval** methods, described previously.) Some of the tasks you can perform with **IntervalYM** methods include the following:

- Comparing two intervals:

```
boolean equals(Object other)
boolean greaterThan(IntervalYM other)
boolean lessThan(IntervalYM other)
```

- Setting a value for an interval from:
  - A string:

```
void fromString(String other)
void set(String string)
```

  - Year and month values (large month values are converted to years):

```
void set(int years, int months)
```

  - Two time stamps:

```
void set(Timestamp t1, Timestamp t2)
```

- Setting the qualifier for an interval:
  - From the length, start code, and end code:

```
void setQualifier(int length, byte startcode, byte
      endcode)
```

  - Using an existing qualifier:

```
void setQualifier(short qualifier)
```

- Obtaining the number of months in the interval:

```
long getMonths()
```

- Creating a string representation of the interval in the format `yyyy-mm`:

```
String toString()
```

The fields present depend on the qualifier. Blanks replace leading zeros.

The IntervalDF class

The **com.informix.lang.IntervalDF** class allows you to manipulate intervals.

The IntervalDF constructors

The default constructor is defined as follows:

```
public IntervalDF() throws SQLException
```

Use this second version of the default constructor to display localized error messages if an exception is thrown:

```
public IntervalDF(Connection conn) throws SQLException
```

Use the following constructors to create intervals from specific input values:

- Two time stamps *t1* and *t2*, returning the IntervalDF value that equals *t1* - *t2*:

  ```
  public IntervalDF(Timestamp t1, Timestamp t2)
                       throws SQLException

  public IntervalDF(Timestamp t1, Timestamp t2, Connection conn)
                       throws SQLException
  ```

  The second version allows you to support localized error messages.

- A number of seconds and nanoseconds (large second values are converted to minutes, hours, or days):

  ```
  public IntervalDF(long seconds, long nanos)
                       throws SQLException

  public IntervalDF(long seconds, long nanos, Connection conn)
                       throws SQLException
  ```

  The second version allows you to support localized error messages.

- A number of seconds, a number of nanoseconds, and qualifier:

  ```
  public IntervalDF(long seconds, long nanos, short qualifier)
                       throws SQLException

  public IntervalDF(long seconds, long nanos, short qualifier, Connection conn)
                       throws SQLException
  ```

  To specify the qualifier, you can use the getQualifier() method described in Interval methods on page 105. The second version allows you to support localized error messages.

- A string:

  ```
  public IntervalDF(String string)
                       throws SQLException
  public IntervalDF(String string, Connection conn)
                       throws SQLException
  ```

  The second version allows you to support localized error messages.

  When you use these constructors, the default qualifier is set to the following values:

  leading field precision: `2` start code: `TU_DAY` end code: `TU_F5`

  For information about string INTERVAL formats, see the *HCL® Informix® Guide to SQL: Syntax*.

- A string and a qualifier:

  ```
  public IntervalDF(String string, short qualifier)
                       throws SQLException
  ```

```
public IntervalDF(String string, short qualifier, Connection conn)
                      throws SQLException
```

To specify the qualifier, you can use the getQualifier() method described in Interval methods on page 105. The second version allows you to support localized error messages.

- A string and qualifier information:

```
public IntervalDF(String string, int length, byte startcode, byte endcode)
                        throws SQLException

public IntervalDF(String string, int length, byte startcode,
byte endcode, Connection conn) throws SQLException
```

The second version allows you to support localized error messages.

The IntervalDF methods

The following methods allow you to manipulate intervals. (You can also use the **Interval** methods, described previously.) The tasks you can perform, and the methods you can use, are as follows:

- Comparing two intervals:

```
boolean equals(Object other)
boolean greaterThan(IntervalDF other)
boolean lessThan(IntervalDF other)
```

- Setting a value for an interval from:
  - A string:

```
void fromString(String other)
void set(String string)
```

  - Second and nanosecond values (large second values are converted to minutes, hours, or days):

```
void set(long seconds, long nanos)
```

  - Two time stamps:

```
void set(Timestamp t1, Timestamp t2)
```

- Setting the qualifier from the length, start code, and end code:

```
void setQualifier(int length, byte startcode, byte endcode)
```

- Obtaining the number of nanoseconds in the interval:

```
long getNanoSeconds()
```

- Obtaining the number of seconds in the interval:

```
long getSeconds()
```

- Creating a string representation of the interval in the format `ddddd hh:mm:ss.nano`:

```
String toString()
```

The fields present depend on the qualifier. Blanks replace leading zeros.

Interval example

The `Intervaldemo.java` program, which is included in HCL Informix® JDBC Driver, shows how to insert into and select from the two types of INTERVAL data types.

Collections and arrays

The JDBC 3.0 specification describes only one method to exchange collection data between a Java™ client and a relational database: an array.

Because the array interface does not include a constructor, HCL Informix® JDBC Driver includes an extension that allows a **java.util.Collection** object to be used in the PreparedStatement.setObject() and ResultSet.getObject() methods.

If you prefer to use an **Array** object, use the PreparedStatement.setArray() and ResultSet.getArray() methods. A **Collection** object is easier to use, but an **Array** object conforms to JDBC 3.0 standards.

By default, the driver maps LIST columns to **java.util.ArrayList** objects and SET and MULTISET columns to **java.util.HashSet** objects during a fetch. You can override these defaults, but the class you use must implement the **java.util.Collection** interface.

To override this default mapping, you can use other classes in the **java.util.Collection** interface, such as the **TreeSet** class. You can also create your own classes that implement the **java.util.Collection** interface. In either case, you must provide a customized type map using the Connection.setTypeMap() method.

During an INSERT operation, any **java.util.Collection** object that is an instance of the **java.util.Set** interface is mapped to the Informix® MULTISET data type. An instance of the **java.util.List** interface is mapped to the Informix® LIST data type. You can override these defaults by creating a customized type mapping.

For information about customized type mappings, see Mapping data types on page 241.

Important: Sets are by definition unordered. If you select collection data using a **HashSet** object, the order of the elements in the **HashSet** object might not be the same as the order specified when the set was inserted. For example, if the data on the database server is the **set** {1, 2, 3}, it might be retrieved into the **HashSet** object as {3, 2, 1} or any other order.

The complete versions of all of the examples in the following sections are in the `complex-types` directory where you installed the driver. For more information, see Sample code files on page 224.

Collection examples

Following is a sample database schema:

```
create table tab ( a set(integer not null), b integer);
insert into tab values ("set{1, 2, 3}", 10);
```

The following is a fetch example using a **java.util.HashSet** object:

```
java.util.HashSet set;

PreparedStatement pstmt;
ResultSet rs;
pstmt = conn.prepareStatement("select * from tab");
System.out.println("prepare ... ok");
rs = pstmt.executeQuery();
System.out.println("executeQuery ... ok");

rs.next();
set = (HashSet) rs.getObject(1);
System.out.println("getObject() ... ok");

/* The user can now use HashSet.iterator() to extract
 * each element in the collection.
 */
Iterator it = set.iterator();
Object obj;
Class cls = null;
int i = 0;
while (it.hasNext())
   {
   obj = it.next();
   if (cls == null)
      {
      cls = obj.getClass();
      System.out.println("    Collection class: " + cls.getName());
      }
   System.out.println("    element[" + i + "] = " +
   obj.toString());
   i++;
   }
pstmt.close();
```

In the `set = (HashSet) rs.getObject(1)` statement of this example, HCL Informix® JDBC Driver gets the type for column 1. Because it is a SET type, a **HashSet** object is instantiated. Next, each collection element is converted into a Java™ object and inserted into the collection.

The following fetch example uses a **java.util.TreeSet** object:

```
java.util.TreeSet set;

PreparedStatement pstmt;
ResultSet rs;

/*
 * Fetch a SET as a TreeSet instead of the default
 * HashSet. In this example a new java.util.Map object has
 * been allocated and passed in as a parameter to getObject().
 * Connection.getTypeMap() could have been used as well.
 */
java.util.Map map = new HashMap();
map.put("set", Class.forName("java.util.TreeSet"));
System.out.println("mapping ... ok");

pstmt = conn.prepareStatement("select * from tab");
System.out.println("prepare ... ok");
```

```
rs = pstmt.executeQuery();
System.out.println("executeQuery ... ok");

rs.next();
set = (TreeSet) rs.getObject(1, map);
System.out.println("getObject(Map) ... ok");

/* The user can now use HashSet.iterator() to extract
 * each element in the collection.
 */
Iterator it = set.iterator();
Object obj;
Class cls = null;
int i = 0;
while (it.hasNext())
   {
   obj = it.next();
   if (cls == null)
      {
      cls = obj.getClass();
      System.out.println("    Collection class: " + cls.getName());
      }
   System.out.println("    element[" + i + "] = " +
   obj.toString());
   i++;
   }
pstmt.close();
```

In the `map.put("set", Class.forName( "java.util.TreeSet" ));` statement, the default mapping of **set = HashSet** is overridden.

In the `set = (TreeSet) rs.getObject(1, map)` statement, HCL Informix® JDBC Driver gets the type for column 1 and finds that it is a SET object. Then the driver looks up the type mapping information, finds **TreeSet**, and instantiates a **TreeSet** object. Next, each collection element is converted into a Java™ object and inserted into the collection.

The following example shows an insert. This example inserts the set (0, 1, 2, 3, 4) into a SET column:

```
java.util.HashSet set = new HashSet();
Integer intObject;
int i;

/* Populate the Java collection */
for (i=0; i < 5; i++)
   {
   intObject = new Integer(i);
   set.add(intObject);
   }
System.out.println("populate java.util.HashSet...ok");

PreparedStatement pstmt = conn.prepareStatement
   ("insert into tab values (?, 20)");
System.out.println("prepare...ok");

pstmt.setObject(1, set);
System.out.println("setObject()...ok");
pstmt.executeUpdate();
```

```
System.out.println("executeUpdate()...ok");
pstmt.close();
```

The `pstmt.setObject(1, set)` statement in this example first serializes each element of the collection. Next, the type information is constructed as each element is converted into a Java™ object. If the types of any elements in the collection do not match the type of the first element, an exception is thrown. The type information is sent to the database server.

Array example

Following is a sample database schema:

```
CREATE TABLE tab (a set(integer not null), b integer);
INSERT INTO tab VALUES ("set{1,2,3}", 10);
```

The following example fetches data using a **java.sql.Array** object:

```
PreparedStatement pstmt = conn.prepareStatement("select a from tab");
System.out.println("prepare ... ok");
ResultSet rs = pstmt.executeQuery();
System.out.println("executeQuery ... ok");

rs.next();
java.sql.Array array = rs.getArray(1);
System.out.println("getArray() ... ok");
pstmt.close();

/*
 * The user can now materialize the data into either
 * an array or else a ResultSet. If the collection elements
 * are primitives then the array should be an array of primitives,
 * not Objects. Mapping data can be provided at this point.
 */

Object obj = array.getArray((long) 1, 2);

int [] intArray = (int []) obj;   // cast it to an array of ints
int i;
for (i=0; i < intArray.length; i++)
   {
   System.out.println("integer element = " + intArray[i]);
   }
pstmt.close();
```

The `java.sql.Array array = rs.getArray(1)` statement instantiates a **java.sql.Array** object. Data is not converted at this point.

The `Object obj = array.getArray((long) 1, 2)` statement converts data into an array of integers (**int** types, not **Integer** objects). Because the getArray() method has been called with index and count values, only a subset of data is returned.

Named and unnamed rows

The JDBC 3.0 specification refers to an SQL type called a *structured type* or *struct*, which is equivalent to the HCL Informix® named row. The specification defines two approaches to exchange structured-type data between a Java™ client and a relational database:

- **Using the SQLData interface**. A single Java™ class per named row type implements the **SQLData** interface. The class has a member for each element in the named row.
- **Using the Struct interface**. This interface instantiates the necessary Java™ object for each element in the named row and constructs an array of **java.util.Object** Java™ objects.

Whether HCL Informix® JDBC Driver instantiates a Java™ object or a **Struct** object for a fetched named row depends on whether there is a customized type-mapping entry or not, as follows:

- If there is an entry for a named row in the Connection.getTypeMap() map, or if you provided a type mapping using the getObject() method, a single Java™ object is instantiated.
- If there is no entry for a named row in the Connection.getTypeMap() map, and if you have not provided a type mapping using the getObject() method, a **Struct** object is instantiated.

Unnamed rows are always fetched into **Struct** objects.

Important: Regardless of whether you use the **SQLData** or **Struct** interface, if a named or unnamed row contains an opaque data type column, there must be a type-mapping entry for it. If you are using the **Struct** interface to access a row that contains an opaque data type column, you need a customized type map for the opaque data type column, but not for the row as a whole.

For more information about custom type mapping, see Mapping data types on page 241.

Interval and collection support

The java.sql.SQLOutput and java.sql.SQLInput methods are extended to support **Collection** and **Interval** objects in named and unnamed rows. These extensions include the following methods:

- The com.informix.jdbc.IfmxComplexSQLInput.readObject() method returns the appropriate **java.util.Collection** object if the data is a set, list, or multiset data type.
- The com.informix.jdbc.IfmxComplexSQLInput.readInterval() method returns the appropriate **IntervalYM** or **IntervalDF** object for an interval data type, depending on the qualifier.
- The com.informix.jdbc.IfmxComplexSQLOutput.writeObject() method accepts objects derived from the **java.util.Collection** interface or from **IntervalYM** and **IntervalDF** objects.

Unsupported methods

The following **SQLInput** methods are not supported for selecting a ROW column into a Java™ object that implements **SQLData**:

- readByte()
- readCharacterStream()
- readRef()

The following **SQLOutput** methods are not supported for inserting a Java™ object that implements **SQLData** into a ROW column:

- writeByte(byte)
- writeCharacterStream(java.io.Reader x)
- writeRef(Ref x)

The SQLData interface

The Java™ class for the named row must implement the **SQLData** interface. The class must have a member for each element in the named row but can have other members in addition to these. The members can be in any order and need not be public.

The Java™ class must implement the writeSQL(), readSQL(), and getSQLTypeName() methods for the named row as defined in the **SQLData** interface, but can implement additional methods. You can use the ClassGenerator utility to create the class; for more information, see The ClassGenerator utility on page 122.

To link this Java™ class with the named row, create a customized type mapping using the Connection.setTypeMap() method or the getObject() method. For more information about type mapping, see Mapping data types on page 241.

You cannot use the **SQLData** interface to access unnamed rows.

SQLData examples

The complete versions of all of the examples in this section are in the `demo/complex-types` directory where you installed the driver. For more information, see Sample code files on page 224.

The following example includes a Java™ class that implements the **java.sql.SQLData** interface.

Here is a sample database schema:

```
CREATE ROW TYPE fullname_t (first char(20), last char(20));
CREATE ROW TYPE person_t (id int, name fullname_t, age int);
CREATE TABLE teachers (person person_t, dept char (20));
INSERT INTO teachers VALUES ("row(100, row('Bill', 'Smith'), 27)", "physics");
```

This is the **fullname** Java™ class:

```
import java.sql.*;
public class fullname implements SQLData
{
   public String first;
   public String last;
   private String sql_type = "fullname_t";

   public String getSQLTypeName()
   {
      return sql_type;
   }
   public void readSQL (SQLInput stream, String type) throws
      SQLException
   {
      sql_type = type;
```

```
      first = stream.readString();
      last = stream.readString();
   }
   public void writeSQL (SQLOutput stream) throws SQLException
   {
      stream.writeString(first);
      stream.writeString(last);
   }
   /*
     * Function not required by SQLData interface, but makes
     * it easier for displaying results.
     */
   public String toString()
   {
      String s = "fullname: ";
      s += "first: " + first + " last: " + last;
      return s;
   }
}
```

This is the **person** Java™ class:

```
import java.sql.*;
public class person implements SQLData
{
   public int id;
    public fullname name;
    public int age;
    private String sql_type = "person_t";

   public String getSQLTypeName()
   {
      return sql_type;
   }
   public void readSQL (SQLInput stream, String type) throws SQLException
   {
      sql_type = type;
      id = stream.readInt();
      name = (fullname)stream.readObject();
      age = stream.readInt();
   }
   public void writeSQL (SQLOutput stream) throws SQLException
   {
      stream.writeInt(id);
      stream.writeObject(name);
      stream.writeInt(age);
   }
   public String toString()
   {
      String s = "person:";
      s += "id: " + id + "\n";
      s += "     name: " + name.toString() + "\n";
      s += "     age: " + age + "\n";
      return s;
   }
}
```

Here is an example of fetching a named row:

```
java.util.Map map = conn.getTypeMap();
conn.setTypeMap(map);
map.put("fullname_t", Class.forName("fullname"));
map.put("person_t", Class.forName("person"));

...
PreparedStatement pstmt;
ResultSet rs;
pstmt = conn.prepareStatement("select person from teachers");
System.out.println("prepare ...ok");

rs = pstmt.executeQuery();
System.out.println("executetQuery()...ok");

while (rs.next())
   {
   person who = (person) rs.getObject(1);
   System.out.println("getObject()...ok");
   System.out.println("Data fetched:");
   System.out.println("row: " + who.toString());
   }
pstmt.close();
```

The conn.getTypeMap() method returns the named row mapping information from the **java.util.Map** object through the **Connection** object.

The map.put() method registers the mappings between the nested named row on the database server, **fullname_t**, and the Java™ class **fullname**, and between the named row on the database server, **person_t**, and the Java™ class **person**.

The `person who = (person) rs.getObject(1)` statement retrieves the named row into the Java™ object **who**. HCL Informix® JDBC Driver recognizes that this object **who** is a named row, a distinct type, or an opaque type, because the information sent by the database server has an extended name of **person_t**.

The driver looks up **person_t** and finds it is a named row. The driver calls the map.get() method with the key **person_t**, which returns the **person** class object. An object of class **person** is instantiated.

The readSQL() method in the **person** class calls methods defined in the **SQLInput** interface to convert each field in the ROW column into a Java™ object and assign each to a member in the **person** class.

The following example shows a method for inserting a Java™ object into a named row column using the setObject() method:

```
java.util.Map map = conn.getTypeMap();
map.put("fullname_t", Class.forName("fullname"));
map.put("person_t", Class.forName("person"));

...
PreparedStatement pstmt;
System.out.println("Populate person and fullname objects");
person who = new person();
fullname name = new fullname();
name.last = "Jones";
name.first = "Sarah";
who.id = 567;
who.name = name;
```

```
who.age = 17;

String s = "insert into teachers values (?, 'physics')";
pstmt = conn.prepareStatement (s);
System.out.println("prepared...ok");

pstmt.setObject(1, who);
System.out.println("setObject()...ok");

int rowcount = pstmt.executeUpdate();
System.out.println("executeUpdate()...ok");
pstmt.close();
```

The conn.getTypeMap() method returns the named row mapping information from the **java.util.Map** object through the **Connection** object.

The map.put() method registers the mappings between the nested named row on the database server, **fullname_t**, and the Java™ class **fullname** and between the named row on the database server, **person_t**, and the Java™ class **person**.

HCL Informix® JDBC Driver recognizes that the object **who** implements the **SQLData** interface, so it is either a named row, a distinct type, or an opaque type. HCL Informix® JDBC Driver calls the getSQLTypeName() method for this object (required for classes implementing the **SQLData** interface), which returns **person_t**. The driver looks up **person_t** and finds it is a named row.

The writeSQL() method in the **person** class calls the corresponding SQLOutput.writeXXX() method for each member in the class, each of which maps to one field in the named row **person_t**. The writeSQL() method in the class contains calls to the SQLOutput.writeObject(name) and SQLOutput.writeInt(id) methods. Each member of the class **person** is serialized and written into a stream.

The Struct interface

The JDBC documentation does not specify that **Struct** objects can be parameters to the PreparedStatement.setObject() method. However, HCL Informix® JDBC Driver can handle any object passed by the PreparedStatement.setObject() or ResultSet.getObject() method that implements the **java.sql.Struct** interface.

You must use the **Struct** interface to access unnamed rows.

You do not need to create your own class to implement the **java.sql.Struct** interface. However, you must perform a fetch to retrieve the ROW data and type information before you can insert or update the ROW data. HCL Informix® JDBC Driver automatically calls the getSQLTypeName() method, which returns the type name for a named row or the row definition for an unnamed row.

If you create your own class to implement the **Struct** interface, the class you create must implement all the **java.sql.Struct** methods, including the getSQLTypeName() method. You can choose what the getSQLTypeName() method returns.

Although you must return the row definition for unnamed rows, you can return either the row name or the row definition for named rows. Each has advantages:

- **Row definition**. The driver does not need to query the database server for the type information. In addition, the row definition returned does not have to match the named row definition exactly, because the database server provides casting, if needed. This is useful if you want to use strings to insert into an opaque type in a row, for example.
- **Row name**. If a user-defined routine takes a named row as a parameter, the signature has to match, so you must pass in a named row.

  For more information about user-defined routines, see the following publications: *HCL® J/Foundation Developer's Guide* (for information specific to Java™); *HCL® Informix® User-Defined Routines and Data Types Developer's Guide* and *HCL® Informix® Guide to SQL: Reference* (both for general information about user-defined routines); and *HCL® Informix® Guide to SQL: Syntax* (for the syntax to create and invoke user-defined routines).

Important: If you use the **Struct** interface for a named row and provide type-mapping information for the named row, a **ClassCastException** message is generated when the ResultSet.getObject() method is called, because Java™ cannot cast between an **SQLData** object and a **Struct** object.

Struct examples

The complete versions of all of the examples in this section are in the `demo/complex-types` directory where you installed the driver. For more information, see Sample code files on page 224.

This example fetches an unnamed ROW column. Here is a sample database schema:

```
CREATE TABLE teachers
   (
   person row(
      id int,
      name row(first char(20), last char(20)),
      age int
      ),
   dept char(20)
   );
INSERT INTO teachers VALUES ("row(100, row('Bill', 'Smith'), 27)", "physics");
```

This is the rest of the example:

```
PreparedStatement pstmt;
ResultSet rs;
pstmt = conn.prepareStatement("select person from teachers");
System.out.println("prepare ...ok");
rs = pstmt.executeQuery();
System.out.println("executetQuery()...ok");

rs.next();
Struct person = (Struct) rs.getObject(1);
System.out.println("getObject()...ok");
System.out.println("\nData fetched:");

Integer id;
Struct name;
Integer age;
Object[] elements;
```

```
/* Get the row description */
String personRowType = person.getSQLTypeName();
System.out.println("person row description: " + personRowType);
System.out.println("");

/* Convert each element into a Java object */
elements = person.getAttributes();

/*
 * Run through the array of objects in 'person' getting out each structure
 * field. Use the class Integer instead of int, because int is not an object.
 */
id = (Integer) elements[0];
name = (Struct) elements[1];
age = (Integer) elements[2];
System.out.println("person.id: " + id);
System.out.println("person.age: " + age);
System.out.println("");

/* Convert 'name' as well. */
/* get the row definition for 'name' */
String nameRowType = name.getSQLTypeName();
System.out.println("name row description: " + nameRowType);

/* Convert each element into a Java object */
elements = name.getAttributes();

/*
 * run through the array of objects in 'name' getting out each structure
 * field.
 */
String first = (String) elements[0];
String last = (String) elements[1];
System.out.println("name.first: " + first);
System.out.println("name.last: " + last);
pstmt.close();
```

The `Struct person = (Struct) rs.getObject(1)` statement instantiates a **Struct** object if column 1 is a ROW type and there is no extended data type name (if it is a named row).

The `elements = person.getAttributes();` statement performs the following actions:

- Allocates an array of **java.lang.Object** objects with the correct number of elements
- Converts each element in the row into a Java™ object

  If the element is an opaque type, you must provide type mapping in the **Connection** object or pass in a **java.util.Map** object in the call to the getAttributes() method.

The `String personrowType = person.getSQLTypeName();` statement returns the row type information. If this type is a named row, the statement returns the name. Because the type is not a named row, the statement returns the row definition: **row(int id, row(first char(20), last char(20)) name, int age)**.

The example then goes through the same steps for the unnamed row **name** as it did for the unnamed row **person**.

The following example uses a user-created class, **GenericStruct**, which implements the **java.sql.Struct** interface. As an alternative, you can use a **Struct** object returned from the ResultSet.getObject() method instead of the **GenericStruct** class.

```
import java.sql.*;
import java.util.*;
public class GenericStruct implements java.sql.Struct
{
   private Object [] attributes = null;
   private String typeName = null;

   /*
    * Constructor
    */
   GenericStruct() { }

   GenericStruct(String name, Object [] obj)
   {
      typeName = name;
      attributes = obj;
   }
   public String getSQLTypeName()
   {
      return typeName;
   }
   public Object [] getAttributes()
   {
      return attributes;
   }
   public Object [] getAttributes(Map map) throws SQLException
   {
      // this class shouldn't be used if there are elements
      // that need customized type mapping.
      return attributes;
   }
   public void setAttributes(Object [] objArray)
   {
      attributes = objArray;
   }
   public void setSQLTypeName(String name)
   {
      typeName = name;
   }
}
```

The following Java™ program inserts a ROW column:

```
PreparedStatement pstmt;
ResultSet rs;
GenericStruct gs;
String rowType;

pstmt = conn.prepareStatement("insert into teachers values (?, 'Math')");
System.out.println("prepare insert...ok\n");

System.out.println("Populate name struct...");
Object[] name = new Object[2];

// populate inner row first
```

```
name[0] = new String("Jane");
name[1] = new String("Smith");

rowType = "row(first char(20), last char(20))";
gs = new GenericStruct(rowType, name);
System.out.println("Instantiate GenericStructObject...okay\n");

System.out.println("Populate person struct...");
// populate outer row next
Object[] person = new Object[3];
person[0] = new Integer(99);
person[1] = gs;
person[2] = new Integer(56);

rowType = "row(id int, " +
   "name row(first char(20), last char(20)), " +
   "age int)";
gs = new GenericStruct(rowType, person);
System.out.println("Instantiate GenericStructObject...okay\n");

pstmt.setObject(1, gs);
System.out.println("setObject()...okay");
pstmt.executeUpdate();
System.out.println("executeUpdate()...okay");
pstmt.close();
```

At the `pstmt.setObject(1, gs)` statement in this example, HCL Informix® JDBC Driver determines that the information is to be transported from the client to the database server as a ROW column, because the **GenericStruct** object is an instance of the **java.sql.Struct** interface.

Each element in the array is serialized, verifying that each element matches the type as defined by the getSQLTypeName() method.

The ClassGenerator utility

The ClassGenerator utility generates a Java™ class for a named row type defined in the system catalog. The utility is the HCL Informix® extension to the JDBC specification.

The created Java™ class implements the **java.sql.SQLData** interface. The class has members for each field in the named row. The readSQL(), writeSQL(), and SQLData.readSQL() methods read the attributes in the order in which they appear in the definition of the named row type in the database. Similarly, writeSQL() writes the data to the stream in that order.

ClassGenerator is packaged in the `ifxtools.jar` file, so the **CLASSPATH** environment variable must point to `ifxtools.jar`.

The syntax for using ClassGenerator is as follows:

```
java ClassGenerator rowtypename [-u URL] [-c classname]
```

The default value for *classname* is the value for *rowtypename*.

If the *URL* parameter is not specified, the required information is retrieved from the `setup.std` file in the home directory.

The structure of `setup.std` is as follows:

```
URL jdbc:host-name:port-number
informixserver informixservername
database database
user user
passwd password
```

Simple named row example

To use ClassGenerator, you first create the named row on the database server as shown in this example:

```
create row type employee (name char (20), age int);
```

Next, run ClassGenerator:

```
java ClassGenerator employee
```

The class generator generates `employee.java`, as shown next, and retrieves the database URL information from `setup.std`, which has the following contents:

```
URL jdbc:davinci:1528
database test
user scott
passwd tiger
informixserver picasso_ius
```

Following is the generated `.java` file:

```
import java.sql.*;
import java.math.*;
public class employee implements SQLData
{
   public String name;
   public int age;
   private String sql_type;

   public String getSQLTypeName() { return "employee"; }

   public void readSQL (SQLInput stream, String type) throws
      SQLException
   {
      sql_type = type;
      name = stream.readString();
      age = stream.readInt();
   }

   public void writeSQL (SQLOutput stream) throws SQLException
   {
      stream.writeString(name);
      stream.writeInt(age);
   }
}
```

Nested named row example

To use ClassGenerator for a nested row, you first create the named row on the database server:

```
create row type manager (emp employee, salary int);
```

Next, run ClassGenerator. In this case, the `setup.std` file is not consulted, because you provide all the needed information at the command line:

```
java ClassGenerator manager -c Manager -u "jdbc:davinci:1528/test:user=scott;
password=tiger;informixserver=picasso_ius"
```

The **-c** option defines the Java™ class you are creating, which is **Manager** (with uppercase M).

The preceding command generates the following Java™ class:

```
import java.sql.*;
import java.math.*;
public class Manager implements SQLData
{
   public employee emp;
   public int salary;
   private String sql_type;

   public String getSQLTypeName() { return "manager"; }

   public void readSQL (SQLInput stream, String type) throws
      SQLException
   {
      sql_type = type;
      emp = (employee)stream.readObject();
      salary = stream.readInt();
   }

   public void writeSQL (SQLOutput stream) throws SQLException
   {
      stream.writeObject(emp);
      stream.writeInt(salary);
   }
}
```

Type cache information

When objects of some data types insert data into columns of certain other data types, HCL Informix® JDBC Driver verifies that the data provided matches the data the database server expects by calling the SQLData.getSQLTypeName() method. The driver asks the database server for the type information with each insertion.

This occurs in the following cases:

- When an **SQLData** object inserts data into an opaque type column and getSQLTypeName() returns the name of the opaque type
- When a **Struct** or **SQLData** object inserts data into a row column and getSQLTypeName() returns the name of a named row
- When an **SQLData** object inserts data into a DISTINCT type column.

In the database URL, you can set the environment variable **ENABLE_TYPE_CACHE**=TRUE to have the driver cache the data type information the first time it is retrieved. The driver then asks the cache for the type information before requesting the data from the database server.

Smart large object data types

A smart large object is a large object with the following features:

- A smart large object can hold a very large amount of data.

  Currently, a single smart large object can hold up to four terabytes of data. This data is stored in a separate disk space called an sbspace.

- A smart large object is recoverable.

  The database server can log changes to smart large objects and therefore can recover smart-large-object data in the event of a system or hardware failure. Logging of smart large objects is not the default behavior.

- A smart large object supports random access to its data.

  Access to a simple large object (BYTE or TEXT) is on an "all or nothing" basis; that is, the database server returns all of the simple large-object data that you request at one time. With smart large objects, you can seek to a desired location and read or write the desired number of bytes.

- You can customize storage characteristics of a smart large object.

  When you create a smart large object, you can specify storage characteristics for the smart large object such as:
  - Whether the database server logs the smart large object in accordance with the current database log mode
  - Whether the database server keeps track of the last time the smart large object was accessed
  - Whether the database server uses page headers to detect data corruption

Smart large objects are stored in the database as BLOB and CLOB data types, which you can access in two ways:

- In HCL Informix® JDBC Driver 3.0, and later, and HCL Informix® servers that support smart large object data types, you can use the standard JDBC API methods described in the JDBC 3.0 specifications. This is the simpler approach.

  The following JDBC 3.0 methods for BLOB and CLOB internal update have already been implemented in previous releases:

  ```
  int setBytes(long, byte[]) throws SQLException
  ```

  ```
  void truncate(long) throws SQLException
  ```

  The following JDBC 3.0 methods from the BLOB interface are implemented in HCL Informix® JDBC Driver, Version 3.0, or later:

  ```
  OutputStream setBinaryStream(long) throws SQLException
  ```

  ```
  int setBytes(long, byte[], int, int) throws SQLException
  ```

The following JDBC 3.0 methods from the CLOB interface are implemented in HCL Informix® JDBC Driver, Version 3.0, or later:

```
OutputStream setAsciiStream(long) throws SQLException
Writer setCharacterStream(long) throws SQLException
```

```
int setString(long, String) throws SQLException
```

```
int setString(long, String, int, int) throws SQLException
```

- You can use Informix® extensions that are based on smart-large-object support within HCL Informix®. This approach offers more options.

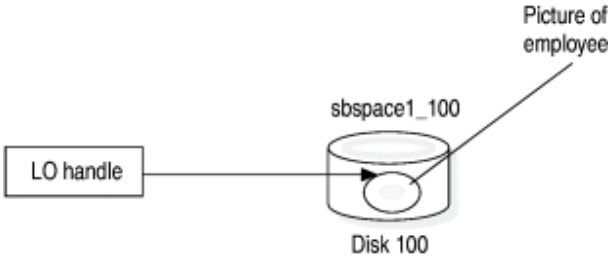
Smart large objects in the database server

In the HCL Informix® database server, a smart large object has two parts:

- The data, which is stored in an *sbspace*
- A *large-object handle*, known as an *LO handle*, which identifies the location of the smart-large-object data in its sbspace

Suppose you store the picture of an employee as a smart large object. The following figure shows how the LO handle contains information about the location of the actual employee picture in the **sbspace1_100** sbspace.

Figure 4. Smart large object in the database server



In the figure, the sbspace holds the actual employee image that the LO handle identifies. For more information about the structure of an sbspace, and the onspaces database utility that creates and drops sbspaces, see the *HCL® Informix® Administrator's Guide*.

Important: Smart large objects can only be stored in sbspaces. You must create an sbspace before you attempt to insert smart large objects into the database.

Because a smart large object is potentially very large, the database server stores only its LO handle in a database table; it can then use this handle to find the actual data of the smart large object in the sbspace. This arrangement minimizes the table size.

Applications obtain the LO handle from the database and use it to locate the smart-large-object data and to open the smart large object for read and write operations.

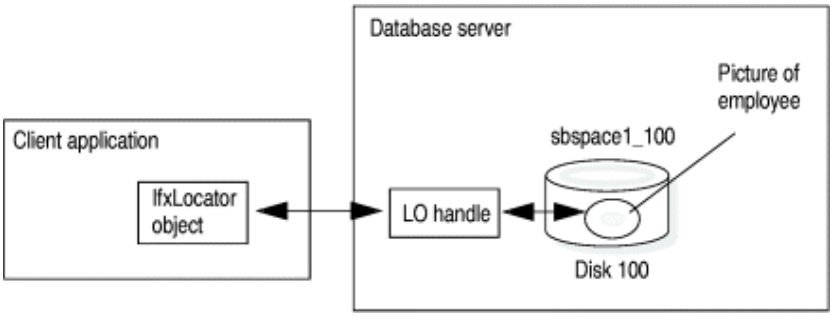
Smart large objects in a client application

On the client, your JDBC application can use **ResultSet** methods to access smart-large-object data, such as:

- getClob() and getAsciiStream() for CLOB data
- getBlob() and getBinaryStream() for BLOB data
- getString() for both CLOB and BLOB data

On the client side, the JDBC driver references the LO handle through an **IfxLocator** object. Your JDBC application obtains an instance of the **IfxLocator** class to contain the smart-large-object locator handle, as shown in the following figure. Your application creates a smart large object independently and then inserts the smart large object into different columns, even in multiple tables. Using multiple threads, an application can write or read data from various portions of the smart large object in parallel, which is very efficient.

Figure 5. Locating a smart large object In a client application



In HCL Informix®, support for Informix® smart large object data types is available only with 9.x and later versions of the database server.

Creating smart large objects

About this task

The HCL Informix® smart large object implementation is based on the following classes:

- **IfxLobDescriptor** stores attributes for the large object.
- **IfxLocator** contains the handle to the large object in the database server.
- **IfxSmartBlob** contains methods for working with the smart large object, such as positioning within the object, reading data from the object, and writing data to the object.
- **IfxBblob** and **IfxCblob** implement the **java.sql.Blob** and **java.sql.Clob** interfaces from the JDBC 3.0 specification.
- **IfxLoStat** stores status information about the large object.

Tip: This section describes how to use the Informix® smart-large-object interface, but it does not currently document every method and parameter in the interface. For a comprehensive reference to all the methods in the interface and

their parameters, see the javadoc files for HCL Informix® JDBC Driver, located in the `doc/javadoc` directory where your driver is installed.

To create a smart large object:

1. For a new smart large object, ensure that the smart large object has an sbspace specified for its data.

   For detailed documentation about the onspaces utility that creates sbspaces, see the *HCL® Informix® Administrator's Guide*. For an example of creating an sbspace, see Example of setting sbspace characteristics on page 141.
2. Create an **IfxLobDescriptor** object.

   This allows you to set storage characteristics for the smart large object. The driver passes the **IfxLobDescriptor** object to the database server when the IfxSmartBlob.IfxLoCreate() method creates the large object.
3. If desired, call methods in the **IfxLobDescriptor** object to specify storage characteristics.

   For most smart large objects, the sbspace name is the only storage characteristic that you need to specify. The database server can calculate values for all other storage characteristics. You can set particular storage characteristics to override these calculated values. However, most applications do not need to set storage characteristics at this level of detail. For more information, see Work with storage characteristics on page 139.
4. Create an **IfxLocator** object.

   This is the pointer to the smart large object on the client.
5. Create an **IfxSmartBlob** object.

   This lets you perform various common operations on the smart large object.
6. Execute the IfxSmartBlob.IfxLoCreate() method to create the large object in the database server.

   IfxLoCreate() takes the **IfxLocator** and **IfxLobDescriptor** objects as parameters to identify the smart large object in the database server.
7. Execute IfxSmartBlob.IfxLoWrite() to write data to the smart large object in the database server.
8. Execute additional **IfxSmartBlob** methods to position within the object, read from the object, and so forth.
9. Execute IfxSmartBlob.IfxLoClose() to close the large object.
10. Insert the smart large object into the database (see Inserting a smart large object into a column on page 132).
11. Execute IfxSmartBlob.IfxLoRelease() to release the locator pointer.

Create an IfxLobDescriptor object

The **IfxLobDescriptor** class stores the internal storage characteristics for a smart large object. Before you can create a smart large object on the database server, you must create an **IfxLobDescriptor** object, as follows:

```
IfxLobDescriptor loDesc = new IfxLobDescriptor(conn);
```

The *conn* parameter is a **java.sql.Connection** object. The IfxLobDescriptor() constructor sets all the default values for the object.

For more information about the internal storage characteristics, see Work with storage characteristics on page 139.

Create an IfxLocator object

The **IfxLocator** object (usually known as the *locator pointer* or *large object locator*) identifies the location of the smart large object, as shown in Figure 5: Locating a smart large object In a client application on page 127; the locator pointer is the communication link between the database server and the client for a particular large object. Before it creates a large object or opens a large object for reading or writing, an application must create an **IfxLocator** object:

```
IfxLocator loPtr = new IfxLocator();
IfxLocator loPtr = new IfxLocator(Connection conn);
```

Use the second of these constructors to display localized error messages if an exception is thrown. For more information, see Support for globalized error messages on page 210.

Create an IfxSmartBlob object

To create a smart large object and obtain access to the methods for performing operations on the object, call the **IfxSmartBlob** constructor, passing a reference to the JDBC connection:

```
IfxSmartBlob smb = new IfxSmartBlob(myConn)
```

Once you have written all the methods that perform operations you need in the smart large object, you can then use the IfxSmartBlob.IfxLoCreate() method to create the large object in the database server and open it for access within your application. The method signature is as follows:

```
public int  IfxLoCreate(IfxLobDescriptor loDesc, int flag,
   IfxLocator loPtr) throws SQLException
public int  IfxLoCreate(IfxLobDescriptor loDesc, int flag,
   IfxBblob blob)throws SQLException
public int  IfxLoCreate(IfxLobDescriptor loDesc, int flag,
   IfxCblob clob throws SQLException
```

The return value is the locator handle, which you can use in subsequent read, write, seek, and close methods (you can pass it as the locator file descriptor (*lofd)* parameter to the methods that operate on open smart large objects; these methods are described beginning with Position within a smart large object on page 134).

The *flag* parameter is an integer value that specifies the access mode in which the new smart large object is opened in the server. The access mode determines which read and write operations are valid on the open smart large object. If you do not specify a value, the object is opened in read-only mode.

Use the access mode *flag* values in the following table with the IfxLoCreate() and IfxLoOpen() methods to open or create smart large objects with specific access modes.

Access mode	Purpose	Flag value in IfxSmartBlob
Read only	Allows read operations only	LO_RDONLY
Write only	Allows write operations only	LO_WRONLY
Write/Append	Appends data you write to the end of the smart large object By itself, it is equivalent to write-only mode followed by a seek to the end of the smart large object. Read operations fail. When you open a smart large object in write/append mode only, the smart large object is opened in write-only mode. Seek operations move the seek position, but read operations to the smart large object fail, and the seek position remains unchanged from its position just before the write. Write operations occur at the seek position, and then the seek position is moved.	LO_APPEND
Read/Write	Allows read and write operations	LO_RDWR

The following example shows how to use a LO_RDWR *flag* value:

```
IfxSmartBlob smb = new IfxSmartBlob(myConn);
int loFd = smb.IfxLoCreate(loDesc, smb.LO_RDWR, loPtr);
```

The **loDesc** and **loPtr** objects are previously created **IfxLobDescriptor** and **IfxLocator** objects, respectively.

The database server uses the following system defaults when it opens a smart large object.

Open-mode information

Default open mode

Access mode

Read-only

Access method

Random

Buffering

Buffered access

Locking

Whole-object locks

For more information about locking, see Work with locks on page 149.

The following table provides the full set of open-mode flags:

Open-mode flag	Description
LO_APPEND	Appends data you write to the end of the smart large object By itself, it is equivalent to write-only mode followed by a seek to the end of the smart large object. Read operations fail. When you open a smart large object in write/append mode only, the smart large object is opened in write-only mode. Seek operations move the seek position, but read operations to the smart large object fail, and the seek position remains unchanged from its position just before the write. Write operations occur at the seek position, and then the seek position is moved.
LO_WRONLY	Allows write operations only
LO_RDONLY	Allows read operations only
LO_RDWR	Allows read and write operations
LO_DIRTY_READ	For open only Allows you to read uncommitted data pages for the smart large object You cannot write to a smart large object after you set the mode to LO_DIRTY_READ. When you set this flag, you reset the current transaction isolation mode to Dirty Read for the smart large object. Do not base updates on data that you obtain from a smart large object in Dirty Read mode.
LO_RANDOM	Overrides optimizer decision Indicates that I/O is random and that the database server should not read ahead. Default open mode.
LO_SEQUENTIAL	Overrides optimizer decision Indicates that reads are sequential in either forward or reverse direction.
LO_FORWARD	Used only for sequential access to indicate forward direction

Open-mode flag	Description
LO_REVERSE	Used only for sequential access to indicate reverse direction
LO_BUFFER	Use standard database server buffer pool.
LO_NOBUFFER	Do not use the standard database server buffer pool. Use private buffers from the session pool of the database server.
LO_NODIRTY_READ	Do not allow dirty reads on smart large object. See LO_DIRTY_READ flag for more information.
LO_LOCKALL	Specifies that locking will occur on entire smart large object
LO_LOCKRANGE	Specifies that locking will occur for a range of bytes You specify the range of bytes through the IfxSmartBlob.IfxLoLock() method when you place the lock.

Inserting a smart large object into a column

About this task

After creating a smart large object, you must insert it into a BLOB or CLOB column to save it in the database. To do this, you must convert the **IfxLocator** object to an **IfxBblob** or **IfxCblob** object, depending upon the column type.

To insert a smart large object into a BLOB or CLOB column:

1. Create an **IfxBblob** or **IfxCblob** object, as follows:

   ```
   IfxBblob blb = new IfxBblob(loPtr);
   ```

   The *loPtr* parameter is an **IfxLocator** object obtained from one of the previous sets of steps.
2. Use the PreparedStatement.setBlob() or setClob() method to insert the object into the column.

Results

Important: The sbspace for the smart large object must exist in the database server before the insertion executes.

Accessing smart large objects

About this task

Follow these steps to use the HCL Informix® extensions to select a smart large object from a database column.

To access a smart large object:

1. Cast the **java.sql.Blob** or **java.sql.Clob** object to an **IfxBblob** or **IfxCblob** object.
2. Use the IfxBblob.getLocator() or IfxCblob.getLocator() method to extract an **IfxLocator** object.
3. Create an **IfxSmartBlob** object.
4. Use the IfxSmartBlob.IfxLoOpen() method to open the smart large object.
5. Use the IfxSmartBlob.IfxLoRead() method to read the data from the smart large object.
6. Close the smart large object using the IfxSmartBlob.IfxLoClose() method.
7. Release the locator pointer in the server by calling the IfxSmartBlob.IfxLoRelease() method.

Results

Standard JDBC ResultSet methods such as ResultSet.getBinaryStream(), getAsciiStream(), getString(), getBytes(), getBlob(), and getClob() can fetch BLOB or CLOB data from a table. The Informix® extension classes can then access the data.

Perform operations on smart large objects

In the database server, you can store a smart large object directly in a column that has one of the following data types:

- The CLOB data type holds text data.
- The BLOB data type can store any kind of binary data in an undifferentiated byte stream.

The CLOB or BLOB column holds an LO handle for the smart large object. Therefore, when you select a CLOB or BLOB column, you do not obtain the actual data of the smart large object, but the LO handle that identifies this data. Columns for smart large objects have a theoretical limit of 4 terabytes and a practical limit determined by your disk capacity.

You can use either of the following ways to store a smart large object in a column:

- For direct access to the smart large object, create a column of the CLOB or BLOB data type.
- To hide the smart large object within an atomic data type, create an opaque type that holds a smart large object.

In a client application, the **IfxBblob** and **IfxCblob** classes are bridges between the way of handling smart large object data described in the JDBC 3.0 specification and the HCL Informix® extensions. The **IfxBblob** class implements the **java.sql.Blob** interface, and the **IfxCblob** class implements the **java.sql.Clob** interface. The Informix® extensions require an **IfxLocator** object to identify the smart large object in the database server.

When you query a table containing a column of type BLOB or CLOB, an object of type Blob or Clob is returned, depending upon the column type. You can then use the JDBC 3.0 supporting methods for objects of type Blob or Clob to access the smart large object.

The constructors create an **IfxBblob** or **IfxCblob** object from the **IfxLocator** object *loPtr*:

```
public IfxBblob(IfxLocator loPtr)
public IfxCblob(IfxLocator loPtr)
```

The following locator method returns an **IfxLocator** object from an **IfxBblob** or **IfxCblob** object. You can then open, read, and write to the smart large object using the IfxSmartBlob.IfxLoOpen(), IfxLoRead(), and IfxLoWrite() methods:

```
public IfxLocator getLocator() throws SQLException
```

Open a smart large object

The following methods in the **IfxSmartBlob** class open an existing smart large object in the database server:

```
public int IfxLoOpen(IfxLocator loPtr, int flag) throws
   SQLException
public int IfxLoOpen(IfxBblob blob, int flag) throws SQLException
public int IfxLoOpen(IfxCblob clob, int flag) throws SQLException
```

The first version opens the smart large object that is referenced by the locator pointer *loPtr*. The second and third versions open the smart large objects that are referenced by the specified **IfxBblob** and **IfxCblob** objects, respectively. The *flag* parameter is a value from the table in Create an IfxSmartBlob object on page 129.

Position within a smart large object

The IfxLoTell() method in the **IfxSmartBlob** class returns the current seek position, which is the offset for the next read or write operation on the smart large object. The IfxLoSeek() method in the **IfxSmartBlob** class sets the read or write position within an already opened large object.

```
public long       IfxLoTell(int lofd)
public long IfxLoSeek(int lofd, long offset, int whence) throws
   SQLException
```

The absolute position depends on the value of the second parameter, *offset*, and the value of the third parameter, *whence*.

The *lofd* parameter is the locator file descriptor returned by the IfxLoCreate() or IfxLoOpen() method. The *offset* parameter is an offset from the starting seek position.

The *whence* parameter identifies the starting seek position. Use the *whence* values in the following table to define the position within a smart large object to start a seek operation.

Starting seek position	Whence value
Beginning of the smart large object	IfxSmartBlob.LO_SEEK_SET
Current® location in the smart large object	IfxSmartBlob.LO_SEEK_CUR

Starting seek position	Whence value
End of the smart large object	IfxSmartBlob.LO_SEEK_END

The return value is a long integer representing the absolute position within the smart large object.

The following example shows how to use a LO_SEEK_SET *whence* value:

```
IfxLobDescriptor loDesc = new IfxLobDescriptor(myConn);
IfxLocator loPtr = new IfxLocator();
IfxSmartBlob smb = new IfxSmartBlob(myConn);
int loFd = smb.IfxLoCreate(loDesc, smb.LO_RDWR, loPtr);
int n = smb.IfxLoWrite(loFd, fin, fileLength);
smb.IfxLoClose(loFd);
loFd = smb.IfxLoOpen(loPtr, smb.LO_RDWR);
long m = smb.IfxLoSeek(loFd, 200, smb.LO_SEEK_SET);
```

The writing position is set at an offset of 200 bytes from the beginning of the smart large object.

Read data from a smart large object

You can read data from a smart large object in the following ways:

- Read the data from the object into a **byte[ ]** buffer.
- Read the data from the object into a file output stream.
- Read the data from the object into a file.

Use the IfxLoRead() method in the **IfxSmartBlob** class, which has the following signatures, to read from a smart large object into a buffer or file output stream:

```
public byte[] IfxLoRead(int lofd, int nbytes) throws SQLException
public int IfxLoRead(int lofd, byte[] buffer, int nbytes) throws
   SQLException
public int IfxLoRead(int lofd, FileOutputStream fout, int nbytes
   throws SQLException
public int IfxLoRead(int lofd, byte[] buffer, int nbytes, int
  offset throws SQLException
```

The *lofd* parameter is a locator file descriptor returned by the IfxLoRead() or IfxLoOpen() method.

The first version returns *nbytes* bytes of data into a byte buffer. This version of the method allocates the memory for the buffer. The second version reads *nbytes* bytes of data into an already allocated buffer. The third version reads *nbytes* bytes of data into a file output stream. The fourth version reads *nbytes* bytes of data into a byte buffer starting at the *current seek position plus offset* into the smart large object. The return values for the last three versions indicate the number of bytes read.

Use the IfxLoToFile() method in the **IfxSmartBlob** class, which has the following signatures, to read from a smart large object into a file:

```
public int IfxLoToFile(IfxLocator loPtr, String filename, int flag
   , int whence) throws SQLException
```

```
public int IfxLoToFile(IfxBblob blob, String filename, int flag ,
   int whence) throws SQLException
public int IfxLoToFile(IfxCblob clob, String filename, int flag ,
   int whence) throws SQLException
```

The first version reads the smart large object that is referenced by the locator pointer *loPtr*. The second and third versions read the smart large objects that are referenced by the specified **IfxBblob** and **IfxCblob** objects, respectively.

The *flag* parameter indicates whether the file is on the client or the server. The value is either `IfxSmartBlob.LO_CLIENT_FILE` or `IfxSmartBlob.LO_SERVER_FILE`. The *whence* parameter identifies the starting seek position. For the values, see Position within a smart large object on page 134.

Tip: There has been a change in the signature of the following function:

```
IfxSmartBlob.IfxLoToFile().
```

This function used to accept four parameters, but now only accepts three parameters. All three overloaded functions for IfxLoToFile() accept three parameters.

Write data to a smart large object

You can write data to a smart large object in the following ways:

- Write the data from a **byte[ ]** buffer to the object.
- Write the data from a file input stream to the object.
- Write the data from a file to the object.

Use the IfxLoWrite() methods in the **IfxSmartBlob** class to write to a smart large object from a **byte[ ]** buffer or file input stream:

```
public int IfxLoWrite(int lofd, byte[] buffer) throws SQLException
public int IfxLoWrite(int lofd, InputStream fin, int length)
   throws SQLException
```

The first version of the method writes *buffer.length* bytes of data from the buffer into the smart large object. The second version writes *length* bytes of data from an **InputStream** object into the smart large object.

The *lofd* parameter is a locator file descriptor returned by the IfxLoCreate() or IfxLoOpen() method. The *buffer* parameter is the **byte[]** buffer where the data is read. The *fin* parameter is the **InputStream** object from which data is written into the smart large object. The *length* parameter is the number of bytes written into the smart large object. The driver returns the number of bytes written.

Use the IfxLoFromFile() method in the **IfxSmartBlob** class to write data to a smart large object from a file:

```
public int IfxLoFromFile (int lofd, String filename, int flag, int
  offset, int amount) throws SQLException
```

The *lofd* parameter is a locator file descriptor returned by the IfxLoCreate() or IfxLoOpen() method. The *flag* parameter indicates whether the file is on the client or the server. The value is either `IfxSmartBlob.LO_CLIENT_FILE` or `IfxSmartBlob.LO_SERVER_FILE`.

The driver returns the number of bytes written.

Truncate a smart large object

Use the IfxLoTruncate() method in the **IfxSmartBlob** class to truncate a large object at an offset you specify. The method signature is as follows:

```
public void IfxLoTruncate(int lofd, long offset) throws
   SQLException
```

The *offset* parameter is the absolute position at which the smart large object is truncated.

Measure a smart large object

Use the IfxLoSize() method in the **IfxSmartBlob** class to return the size of a smart large object. This method returns a long integer representing the size of the large object.

The method signature is as follows:

```
public long IfxLoSize(int lofd) throws SQLException
```

Close and release a smart large object

After you have performed all the operations your application needs, you must close the object and then release the resources in the server. The methods in the **IfxSmartBlob** class that perform these tasks are as follows:

```
public void IfxLoClose(int lofd) throws SQLException
public void IfxLoRelease(IfxLocator loPtr) throws SQLException
public void IfxLoRelease(IfxBblob blob) throws SQLException
public void IfxLoRelease(IfxCblob clob) throws SQLException
```

For any further access to the same large object, you must reopen it with the IfxLoOpen() method.

Convert IfxLocator to a hexadecimal string

Some applications, for example, web browsers, can only process ASCII data; they require IfxLocator to be converted to hexadecimal string format. In a typical web-based application, the web server queries the database table and sends the results to the browser. Instead of sending the entire smart large object, the web server converts the locator into hexadecimal string format and sends it to the browser. If the user requests the browser to display the smart large object, the browser sends the locator in hexadecimal format back to the web server. The web server then reconstructs the binary locator from the hexadecimal string and sends the corresponding smart large object data to the browser.

To convert between the IfxLocator byte array and a hexadecimal number, use the methods listed in the following table.

Task performed	Method signature	Additional information
Converts a byte array to a hexadecimal character string	public static String toHexString(byte[] *byteBuf*);	Works on data other than IfxLocator Provided in the **com.informix.util.stringUtil** class

Task performed	Method signature	Additional information
Converts a hexadecimal character string to a byte array	public static byte[] fromHexString(String *str*) throws NumberFormatException;	Works on data other than IfxLocator Provided in the **com.informix.util.stringUtil** class
Constructs an IfxLocator object using a byte array	public IfxLocator(byte[] *byteBuf*) throws SQLException;	Provided in the **IfxLocator** class
Converts an IfxLocator byte array to a hexadecimal character string	public String toString();	Provided in the **IfxLocator** class
Converts a hexadecimal character string to an IfxLocator byte array	public byte[] toBytes();	Provided in the **IfxLocator** class

The following example uses the toString() and toBytes() methods to fetch the locator from a smart large object and then convert it into a hexadecimal string:

```
...

String hexLoc = "";
byte[] blobBytes;
byte[] rawLocA = null;
IfxLocator loc;
try
{
     ResultSet rs = stmt.executeQuery("select b1 from btab");
     while(rs.next())
     {
         IfxBblob b=(IfxBblob)rs.getBlob(1);
         loc =b.getLocator();
         hexLoc = loc.toString();
         rawLocA = loc.toBytes();
     }
}
catch(SQLException e)
{}
```

The following example uses the IfxLocator() method to construct an IfxLocator, which is then used to read a smart large object:

```
...

try
{
     IfxLocator loc2 = new IfxLocator(rawLoc);
     IfxSmartBlob b2 = new IfxSmartBlob((IfxConnection)myConn);
     int lofd = b2.IfxLoOpen(loc2, b2.LO_RDWR);
     blobBytes = b2.IfxLoRead(lofd, fileLength);
}
```

```
catch(SQLException e)
  {}
```

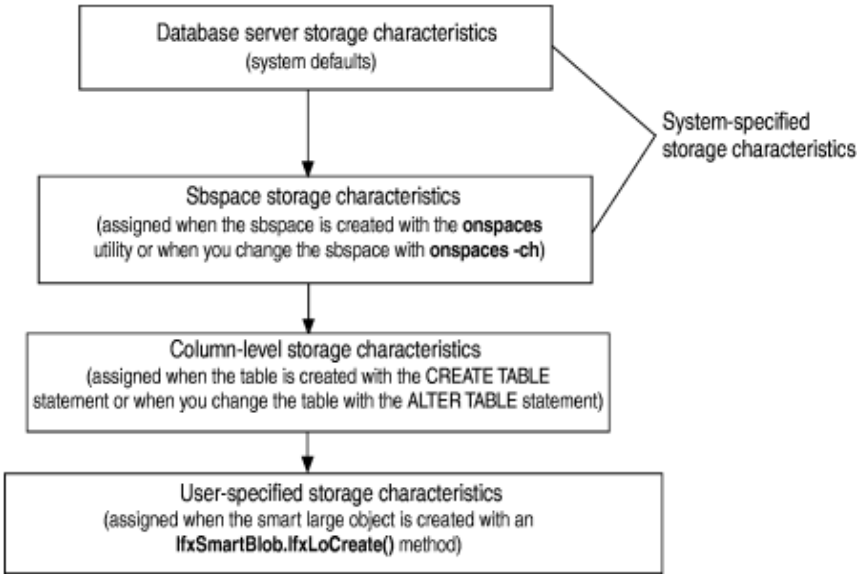
Work with storage characteristics

Storage characteristics tell the database server how to manage a smart large object. These characteristics include such areas as sizing, logging, locking, and open modes. You have the following options with respect to storage characteristics:

- Use the system-specified storage characteristics as a basis for obtaining the storage characteristics of a smart large object.
- Override the system defaults with one of the following:
  - Storage characteristics defined for a particular CLOB or BLOB column in which you want to store the smart large object
  - Storage characteristics that are unique to a particular CLOB or BLOB column called *column-level storage characteristics*
  - Special storage characteristics that you define for this smart large object only called *user-specified storage characteristics*

The database server uses a hierarchy, which the following figure shows, to obtain the storage characteristics for a new smart large object.

Figure 6. Storage-characteristics hierarchy



For a given storage characteristic, any value defined at the column level overrides the system-specified value, and any user-level value overrides the column-level value. You can specify storage characteristics at the three points shown in the following table.

When specified	How specified	For more information
When an sbspace is created	Options of onspaces utility	System-specified storage characteristics on page 140

When specified	How specified	For more information
		HCL® Informix® Administrator's Guide
When a database table is created	Keywords in PUT clause of CREATE TABLE statement	*HCL® Informix® Guide to SQL: Syntax*
When a smart large object is created	Create flags and methods in the **ifxLobDescriptor** class	Set create flags on page 147

System-specified storage characteristics

The database administrator establishes system-specified storage characteristics when they initialize the database server and creates an sbspace with the onspaces utility, as follows:

- If the onspaces utility has specified a value for a particular storage characteristic, the database server uses the onspaces value as the system-specified storage characteristic.
- If the onspaces utility has not specified a value for a particular storage characteristic, the database server uses the system default as the system-specified storage characteristic.

The system-specified storage characteristics apply to all smart large objects that are stored in the sbspace, unless a smart large object specifically overrides them with column-level or user-specified storage characteristics.

For the storage characteristics that onspaces can set, as well as the system defaults, see Table 5: Specifying disk-storage information on page 143 and Table 6: Specifying attribute information on page 144.

For most applications, it is recommended that you use the system-specified default values for the storage characteristics. Note the following exceptions:

- Your application needs to obtain extra performance.

  You can use setXXX() methods in **ifxLobDescriptor** to change the disk-storage information of a new smart large object. For more information, see Set create flags on page 147.
- You want to use the storage characteristics of an existing smart large object.

  The IfxLoStat.getLobDescriptor() method can obtain the large-object descriptor of an open smart large object. You can then create a new object and use the IfxSmartBlob.ifxLoAlter() method to set its characteristics to the new descriptor. For more information, see Changing the storage characteristics on page 147.
- You are working with more than one smart large object and do not want to use the default sbspace.

  The DBA can specify a default sbspace name with the SBSPACENAME configuration parameter in the `onconfig` file. However, you must ensure that the location (the name of the sbspace) is correct for the smart large object that

you create. If you do not specify an sbspace name for a new smart large object, the database server stores it in this default sbspace. This arrangement can lead to space constraints.

- If you know the size of the smart large object, specify this size in your application using the IfxLobDescriptor.setEstBytes() method instead of in the onspaces utility (system level) or the CREATE TABLE or the ALTER TABLE statement (column level).

Obtain information about storage characteristics

To obtain the column-level storage characteristics of a smart large object, your application can call the following method in the **IfxSmartBlob** class, passing the name of the column for the *colname* parameter:

```
IfxLobDescriptor IfxLoColInfo(java.lang.String colname) throws
   SQLException
```

Most applications only need to ensure correct storage characteristics for an sbspace name (the location of the smart large object). You can get information for this and other storage characteristics by calling the various getXXX() methods in the **ifxLobDescriptor** class before creating the **IfxSmartBlob** object. The following table summarizes the getXXX() methods.

Method signature in ifxLobDescriptor	Purpose
int getCreateFlags()	Obtains the create flags for the object
long getEstSize()	Obtains the estimated size, in bytes, of the object
int getExtSize()	Obtains the extent size of the object
long getMaxBytes()	Obtains the maximum size, in bytes, of the object
java.lang.String getSbspace()	Obtains the name of the sbspace in the database server in which the object is stored

Example of setting sbspace characteristics

The following call to the onspaces utility creates an sbspace called **sb1** in the **/dev/sbspace1** partition:

```
onspaces -c -S sb1 -p /dev/sbspace1 -o 500 -s 2000
   -Df "AVG_LO_SIZE=32"
```

The following table shows the resulting system-specified storage characteristics for all smart large objects in the **sb1** sbspace.

Table 4. System-specified storage characteristics for the sb1 sbspace

Disk-storage information	System-specified value	Specified by onspaces utility
Size of extent	Calculated by database server	System default
Size of next extent	Calculated by database server	System default
Minimum extent size	Calculated by database server	System default

Table 4. System-specified storage characteristics for the sb1 sbspace (continued)

Disk-storage information	System-specified value	Specified by onspaces utility
Size of smart large object	32 kilobytes (database server uses as size estimate)	AVG_LO_SIZE
Maximum size of I/O block	Calculated by database server	System default
Name of sbspace	**sb1**	-S option
Logging	OFF	System default
Last-access time	OFF	System default

Work with disk-storage information

Disk-storage information helps the database server determine how to manage the smart large object most efficiently on disk.

Important: For most applications, use the values that the database server calculates for the disk-storage information. Methods provided in HCL Informix® JDBC Driver are intended for special situations.

This disk-storage information includes:

- Allocation-extent information:
  - Extent size:

    An *allocation extent* is a collection of contiguous bytes within an sbspace that the database server allocates to a smart large object at one time. The database server performs storage allocations for smart large objects in increments of the extent size.

    You can specify an extent size by calling the ifxLobDescriptor.setExtSize() method.
  - Next-extent size:

    The database server tries to allocate an extent as a single, contiguous region in a chunk. However, if no single extent is large enough, the database server must use multiple extents as necessary to satisfy the current write request. After the initial extent fills, the database server attempts to allocate another extent of contiguous disk space. This process is called *next-extent allocation*.

  For more information about extents, see the topics on disk structure and storage in the *HCL® Informix® Administrator's Guide*.
- Sizing information:
  - Estimated number of bytes in a new smart large object
  - Maximum number of bytes to which the smart large object can grow

  To specify sizing information, you can use the setMaxBytes() and setEstBytes() methods in the **ifxLobDescriptor** class.

If you know the size of the smart large object, specify this size using the setEstBytes() method. This is the best way to set the extent size because the database server can allocate the entire smart large object as one extent.

- Location:

  The name of the sbspace identifies the location at which to store the smart large object. To set this name, you can use the vifxLobDescriptor.setSbSpace() method.

The database server uses the disk-storage information to determine how best to size, allocate, and manage the extents of the sbspace. It can calculate all disk-storage information for a smart large object except the sbspace name.

The following table summarizes the ways to specify disk-storage information for a smart large object.

Table 5. Specifying disk-storage information

Disk-storage information	**System-specified storage characteristics**		**Column-level storage characteristics**	**User-specified storage characteristics**
	System default value	**Specified by onspaces utility**	**Specified by PUT clause of CREATE TABLE**	**Specified by the HCL Informix® JDBC Driver method**
Size of extent	Calculated by database server	EXTENT_SIZE	EXTENT SIZE	Yes
Size of next extent	Calculated by database server	NEXT_SIZE	No	No
Minimum extent size	4 kilobytes	MIN_EXT_SIZE	No	No
Size of smart large object	Calculated by database server	Average size of all smart large objects in sbspace: AVG_LO_SIZE	No	Estimated size of a particular smart large object Maximum size of a particular smart large object
Maximum size of I/O block	Calculated by database server	MAX_IO_SIZE	No	No
Name of sbspace	SBSPACENAME	-S *option*	Name of an existing sbspace in which a smart large object: IN clause	Yes

Work with logging, last-access time, and data integrity

Database administrators and applications can affect some additional smart-large-object attributes:

- Whether to log changes to the smart large object in the system log file
- Whether to save the last-access time for a smart large object
- How to format the pages in the sbspace of the smart large object

The following table summarizes how you can alter these attributes at the system, column, and application levels.

Table 6. Specifying attribute information

Attribute information	System-specified storage characteristics default value	System-specified storage characteristics, specified by onspaces utility	Column-level storage characteristics, specified by PUT clause of CREATE TABLE	User-specified storage characteristics, specified by a JDBC driver method
Logging	OFF	LOGGING	LOG, NO LOG	Yes
Last-access time	OFF	ACCESSTIME	KEEP ACCESS TIME, NO KEEP ACCESS TIME	Yes
Buffering mode	OFF	BUFFERING	No	No
Lock mode	Lock entire smart large object	LOCK_MODE	No	Yes
Data integrity	High integrity	No	HIGH INTEG, MODERATE INTEG	Yes

Logging

By default, the database server does not log the user data of a smart large object. You can control the logging behavior for a smart large object as part of its create flags. For more information, see Set create flags on page 147.

When a database performs logging, smart large objects might result in long transactions for the following reasons:

- Smart large objects can be very large, even several gigabytes in size.

  The amount of log storage needed to log user data can easily overflow the log.
- Smart large objects might be used in situations where data collection can be quite long.

  For example, if a smart large object holds low-quality audio recording, the amount of data collection might be modest but the recording session might be quite long.

A simple workaround is to divide a long transaction into multiple smaller transactions. However, if this solution is not acceptable, you can control when the database server performs logging of smart large objects. (Table 6: Specifying attribute information on page 144 shows how you can control the logging behavior for a smart large object.)

When logging is enabled, the database server logs changes to the user data of a smart large object. It performs this logging in accordance with the current database log mode.

For a database that is not ANSI compliant, the database server does not guarantee that log records that pertain to smart large object are flushed at transaction commit. However, the metadata is always restorable to an action-consistent state; that is, to a state that ensures no structural inconsistencies exist in the metadata (control information of the smart large object, such as reference counts).

An ANSI-compliant database uses unbuffered logging. When smart-large-object logging is enabled, all log records (metadata and user data) that pertain to smart large objects are flushed to the log at transaction commit. However, user data is not guaranteed to be flushed to its stable storage location at commit time.

When logging is disabled, the database server does not log changes to user data even if the database server logs other database changes. However, the database server always logs changes to the metadata. Therefore, the database server can still restore the metadata to an action-consistent state.

Important: Consider carefully whether to enable logging for a smart large object. The database server incurs considerable overhead to log smart large objects. You must also ensure that the system log file is large enough to hold the value of the smart large object. The logical log size must exceed the total amount of data that the database server logs while the update transaction is active.

Write your application so that any transactions with smart large objects that have potentially long updates do not cause other transactions to wait. Multiple transactions can access the same smart-large-object instance if the following conditions are satisfied:

- The transaction can access the database row that contains an LO handle for the smart large object.

  Multiple references can exist on the same smart large object if more than one column holds an LO handle for the same smart large object.
- Another transaction does not hold a conflicting lock on the smart large object.

  For more information about smart large object locks, see Work with locks on page 149.

The best update performance and fewest logical-log problems result when you disable the logging feature when you load a smart large object and re-enable it after the load operation completes. If logging is turned on, you might want to turn logging off before a bulk load and then perform a level-0 backup.

Last-access time

The last-access time of a smart large object is the system time at which the database server last read or wrote the smart large object. The last-access time records access to the user data and metadata of a smart large object. This system time is

stored as number of seconds since January 1, 1970. The database server stores this last-access time in the metadata area of the sbspace.

By default, the database server does not save the last access time. You can specify saving the last-access time by setting the LO_KEEP_LASTACCESS_TIME create flag and calling the IfxLobDescriptor.setCreateFlags() method. For more information, see Set create flags on page 147.

The database server also tracks the last-modification time and the last change in status for a smart large object. For more information, see Work with status characteristics on page 148.

Important: Consider carefully whether to track last-access time for a smart large object. The database server incurs considerable overhead in logging and concurrency to maintain last-access times for smart large objects.

Data integrity

You can specify data integrity with the LO_HIGH_INTEG and LO_MODERATE_INTEG create flags, by calling the IfxLobDescriptor.setCreateFlags() method. For more information, see Set create flags on page 147.

An sbpage is the unit of allocation for smart large object data, which is stored in the user-data area of an sbspace. The structure of an sbpage in the sbspace determines how much data integrity the database server can provide. The database server uses the page header and trailer to detect incomplete writes and data corruption.

The database server supports the following levels of data integrity:

- High integrity tells the database server to use both a page header and a page trailer in each sbpage.
- Moderate integrity tells the database server to use only a page header in each sbpage.

Moderate integrity provides the following benefits:

- It eliminates an additional data copy operation that is necessary when an sbpage has page headers and page trailers.
- It preserves the user data alignments on pages because no page header and page trailer are present.

Moderate integrity might be useful for smart large objects that contain large amounts of audio or video data that is moved through the database server and that do not require a high data integrity. By default, the database server uses high integrity (page headers and page trailers) for sbspace pages. You can control the data integrity for a smart large object as part of its storage characteristics.

Important: Consider carefully whether to use moderate integrity for sbpages of a smart large object. Although moderate integrity takes less disk space per page, it also reduces the ability of the database server to recover information if disk errors occur.

For information about the structure of sbspace pages, see the *HCL® Informix® Administrator's Guide*.

Changing the storage characteristics

About this task

The IfxLoAlter() methods in the **IfxSmartBlob** class let you change the storage characteristics of a smart large object.

To change smart-large-object characteristics:

1. Create a new large-object descriptor.
   Example
   For example:
   ```
   IfxLobDescriptor loDesc = new IfxLobDescriptor(conn);
   ```
2. Call IfxLobDescriptor.setCreateFlags(), setEstBytes(), IfxLobDescriptor.setMaxBytes(), **setExtSize**, and setSbspace() to specify the new characteristics:
   ```
   public void setCreateFlags( int flags )
   public void setEstBytes(long estSize)
   public void setMaxBytes (long maxSize)
   public void setExtSize (long extSize)
   public void setSbspace(java.lang.String sbspacename)
   ```
   The *flag* parameter is a constant from Set create flags on page 147.
3. Call IfxLoAlter() to alter the existing smart large object to contain the new descriptor:
   ```
   public int IfxLoAlter(IfxLocator loPtr, IfxLobDescriptor loDesc)
      throws SQLException
   public int IfxLoAlter(IfxBblob blob, IfxLobDescriptor loDesc)
      throws SQLException
   public int IfxLoAlter(IfxCblob clob, IfxLobDescriptor loDesc)
      throws SQLException
   ```

Results

IfxLoAlter() obtains an exclusive lock in the server for the entire smart large object before it proceeds with the update. It holds this lock until the update completes.

Set create flags

You can change the following characteristics by calling the IfxLobDescriptor.setCreateFlags() method:

- Logging characteristics

  You can specify the LO_LOG or LO_ NOLOG constant.

  LO_LOG causes the server to follow the logging procedure used with the current database log for the corresponding smart large object. This option can generate large amounts of log traffic and increase the risk that the logical log fills up.

  Instead of full logging, you might turn off logging when you load the smart large object initially and then turn logging back on once the smart large object is loaded. If you use NO LOG, you can restore the smart-large-object metadata

later to a state in which no structural inconsistencies exist. In most cases, no transaction inconsistencies will exist either, but that result is not guaranteed.

For more usage details on logging, see Logging on page 144.

- Last-access time characteristics

  You can specify the LO_ KEEP_LASTACCESS_TIME or LO NOKEEP_LASTACCESS_TIME constant. LO_ KEEP_LASTACCESS_TIME records, in the smart-large-object metadata, the system time at which the corresponding smart large object was last read or written.

  For more usage details on last-access time, see Last-access time on page 145.

- Whether to detect incomplete writes and data corruption by producing user-data pages with a page header and page trailer

  You can specify the LO_ HIGH_INTEG or LO_moderate_integ constant. LO_ HIGH_INTEG is the default data-integrity behavior.

  For more usage details on data integrity, see Data integrity on page 146.

The following example sets multiple flags:

```
loDesc.setCreateFlags
   (IfxSmartBlob.LO_LOG+IfxSmartBlob.LO_TEMP+...)
```

A parallel getXXX() method lets you obtain the current storage characteristics for the large object:

```
public int getCreateFlags()
```

For more detailed information about all of the characteristics, see the section describing the PUT clause for the CREATE TABLE statement, in the *HCL® Informix® Guide to SQL: Syntax*.

Work with status characteristics

The **IfxLoStat** class stores some statistical information about a smart large object such as the size, last access time, last modified time, last status change, and so on. The following table shows the status information that you can obtain.

Table 7. Status information for a smart large object

Status information	Description
Last-access time	The time, in seconds, that the smart large object was last accessed This value is available only if the last-access time attribute is enabled for the smart large object. For more information, see Last-access time on page 145.
Last-change time	The time, in seconds, of the last change in status for the smart large object

Table 7. Status information for a smart large object (continued)

Status information	Description
	A change in status includes changes to metadata and user data (data updates and changes to the number of references). This system time is stored as number of seconds since January 1, 1970.
Last-modification time	The time, in seconds, that the smart large object was last modified A modification includes only changes to user data (data updates). This system time is stored as the number of seconds since January 1, 1970. On some platforms, the last-modification time might also have a microseconds component, which can be obtained separately from the seconds component.
Size	The size, in bytes, of the smart large object
Storage characteristics	See Work with storage characteristics on page 139.

To obtain a reference to the status structure, call the following method in the **IfxSmartBlob** class:

```
IfxLoStat IfxLoGetStat(int lofd)
```

To obtain particular categories of status information, call the methods shown in the following table.

Table 8. Methods for obtaining status information

Status information	Method signature in ifxLoStat class
Last-access time	int getLastAccessTime()
Last-change time	int getLastStatusTime()
Last-modification time	int getLastModifyTimeM() - time in microseconds int getLastModifyTimeS() - time rounded to seconds
Size	int getSize()
Storage characteristics	ifxLobDescriptor getLobDescriptor()

Work with locks

To prevent simultaneous access to smart-large-object data, the database server obtains a lock on this data when you open the smart large object. This smart-large-object lock is distinct from the following kinds of locks:

- Row locks

  A lock on a smart large object does not lock the row in which the smart large object resides. However, if you retrieve a smart large object from a row and the row is still current, the database server might hold a row lock as well as a smart-large-object lock. Locks are held on the smart large object instead of on the row because many columns could be accessing the same smart-large-object data.

- Locks of different smart large objects in the same row of a table

  A lock on one smart large object does not affect other smart large objects in the row.

The following table shows the lock modes that a smart large object can support.

Table 9. Lock modes for a smart large object

Lock mode	Purpose	Description
Lock-all	Lock the entire smart large object	Indicates that lock requests apply to all data for the smart large object
Byte-range	Lock only specified portions of the smart large object	Indicates that lock requests apply only to the specified number of bytes of smart-large-object data

When the server opens a smart large object, it uses the following information to determine the lock mode of the smart large object:

- The access mode of the smart large object

  The database server obtains a lock as follows:

  - In *share mode*, when you open a smart large object for reading (read-only)
  - In *update mode*, when you open a smart large object for writing (write-only, read/write, write/append)

    When a write operation (or some other update) is actually performed on the smart large object, the server upgrades this lock to an *exclusive lock*.

- The isolation level of the current transaction

  If the database table has an isolation mode of Repeatable Read, the server does not release any locks that it obtains on a smart large object until the end of the transaction.

By default, the server chooses the lock-all lock mode.

The server retains the lock as follows:

- It holds share-mode locks and update locks (which have not yet been upgraded to exclusive locks) until one of the following events occurs:
  - The close of the smart large object
  - The end of the transaction
  - An explicit request to release the lock (for a byte-range lock only)
- It holds exclusive locks until the end of the transaction even if you close the smart large object.

When one of the preceding conditions occurs, the server releases the lock on the smart large object.

Important: You lose the lock at the end of a transaction even if the smart large object remains open. When the server detects that a smart large object has no active lock, it automatically obtains a new lock when the first access occurs to the smart large object. The lock that it obtains is based on the original access mode of the smart large object.

The server releases the lock when the current transaction terminates. However, the server obtains the lock again when the next function that needs a lock executes. If this behavior is undesirable, the server-side SQL application can use BEGIN WORK transaction blocks and place a COMMIT WORK or ROLLBACK WORK statement after the last statement that needs to use the lock.

Byte-range locking

By default, the database server uses whole lock-all locks when it needs to lock a smart large object. Lock-all locks are an “all or nothing lock; that is, they lock the entire smart large object. When the database server obtains an exclusive lock, no other user can access the data of the smart large object as long as the lock is held.

If this locking is too restrictive for the concurrency requirements of your application, you can use byte-range locking instead of lock-all locking. With byte-range locking, you can specify the range of bytes to lock in the smart-large-object data. If other users access other portions of the data, they can still acquire their own byte-range lock.

Use the IfxLoLock() method in the **IfxSmartBlob** class to specify byte-range locking:

```
public long IfxLoLock(int lofd, long offset, int whence, long
  range, int lockmode) throws SQLException
```

To unlock a range of bytes in the object, use the IfxLoUnLock() method:

```
public long IfxLoUnLock( int lofd, long offset, int whence, long
  range) throws SQLException
```

The *lofd* parameter is the locator file descriptor returned by the IfxLoCreate() or IfxLoOpen() method. The *offset* parameter is an offset from the starting seek position. The *whence* parameter identifies the starting seek position. The values are described in the table in Position within a smart large object on page 134.

The *range* parameter indicates the number of bytes to lock or unlock within the smart large object. The *lockmode* parameter indicates what type of lock to create. The values can be either `IfxSmartBlob.LO_EXCLUSIVE_MODE` or `IfxSmartBlob.LO_SHARED_MODE.`

Cache large objects

Whenever an object of type BLOB, CLOB, text, or byte is fetched from the database server, the data is cached in client memory. If the size of the large object is bigger than the value in the **LOBCACHE** environment variable, the large object data is stored in a temporary file. For more information about the **LOBCACHE** variable, see Manage memory for large objects on page 217.

Avoid errors transferring large objects

The **IFX_LOB_XFERSIZE** environment variable is used to specify the number of bytes in a CLOB or BLOB to transfer from a client application to the database server before checking whether an error has occurred. The error check occurs each time the specified number of bytes is transferred. If an error occurs, the remaining data is not sent and an error is reported. If no error occurs, the file transfer will continue until it finishes.

For example, if the value of **IFX_LOB_XFERSIZE** is set to 10485760 (10 MB), then error checking will occur after every 10485760 bytes of the CLOB or BLOB is sent. If the **IFX_LOB_XFERSIZE** environment variable is not set, the error check occurs after the entire BLOB or CLOB is transferred.

The valid range for the **IFX_LOB_XFERSIZE** environment variable is from 1 to 9223372036854775808 bytes. The **IFX_LOB_XFERSIZE** environment variable is set on the client.

You should adjust the value of **IFX_LOB_XFERSIZE** to suit your environment. Set the **IFX_LOB_XFERSIZE** environment variable low enough so that transmission errors of large BLOB or CLOB data types are detected early, but not so low that excessive network resources are consumed.

Smart large object examples

The following examples illustrate some of the tasks discussed in this section.

Create a smart large object

This example illustrates the steps shown in Creating smart large objects on page 127.

```
file = new File("data.dat");
FileInputStream fin = new FileInputStream(file);

byte[] buffer = new byte[200];;

IfxLobDescriptor loDesc = new IfxLobDescriptor(myConn);
IfxLocator loPtr = new IfxLocator();
IfxSmartBlob smb = new IfxSmartBlob(myConn);

// Now create the large object in server. Read the data from the
   file
// data.dat and write to the large object.
int loFd = smb.IfxLoCreate(loDesc, smb.LO_RDWR, loPtr);
System.out.println("A smart-blob is created ");
int n = fin.read(buffer);
if (n > 0)
n = smb.IfxLoWrite(loFd, buffer);
```

```
System.out.println("Wrote: " + n +" bytes into it");

// Close the large object and release the locator.
smb.IfxLoClose(loFd);
System.out.println("Smart-blob is closed " );
smb.IfxLoRelease(loPtr);
System.out.println("Smart Blob Locator is released ");
```

The contents of the file `data.dat` are written to the smart large object.

Insert data into a smart large object

The following code inserts data into a smart large object:

```
String s = "insert into large_tab (col1, col2) values (?,?)";
pstmt = myConn.prepareStatement(s);

file = new File("data.dat");
FileInputStream fin = new FileInputStream(file);

byte[] buffer = new byte[200];;

IfxLobDescriptor loDesc = new IfxLobDescriptor(myConn);
IfxLocator loPtr = new IfxLocator();
IfxSmartBlob smb = new IfxSmartBlob(myConn);

// Create a smart large object in server
int loFd = smb.IfxLoCreate(loDesc, smb.LO_RDWR, loPtr);
System.out.println("A smart-blob has been created ");
int n = fin.read(buffer);
if (n > 0)
n = smb.IfxLoWrite(loFd, buffer);
smb.IfxLoClose(loFd);

System.out.println("Wrote: " + n +" bytes into it");
System.out.println("Smart-blob is closed " );

Blob blb = new IfxBblob(loPtr);
pstmt.setInt(1, 2);  // set the Integer column
pstmt.setBlob(2, blb); // set the blob column
pstmt.executeUpdate();
System.out.println("Binding of smart large object to table is
   done");

pstmt.close();
smb.IfxLoRelease(loPtr);
System.out.println("Smart Blob Locator is released ");
```

The contents of the file `data.dat` are written to the BLOB column of the **large_tab** table.

Retrieve data from a smart large object

The example in this topic illustrates the steps in Accessing smart large objects on page 133.

The following code example shows how to access the smart large object data using HCL Informix® extension classes:

```
byte[] buffer  = new byte[200];
System.out.println("Reading data now ...");
try
   {
   int row = 0;
   Statement stmt = myConn.createStatement();
   ResultSet rs =  stmt.executeQuery("Select * from demo_14");
   while( rs.next() )
      {
      row++;
      String str = rs.getString(1);
      InputStream value = rs.getAsciiStream(2);
      IfxBblob b = (IfxBblob) rs.getBlob(2);
      IfxLocator loPtr = b.getLocator();
      IfxSmartBlob smb = new IfxSmartBlob(myConn);
      int loFd = smb.IfxLoOpen(loPtr, smb.LO_RDONLY);

      System.out.println("The Smart Blob is Opened for reading ..");
      int number = smb.IfxLoRead(loFd, buffer, buffer.length);
      System.out.println("Read total " + number  + " bytes");
      smb.IfxLoClose(loFd);
      System.out.println("Closed the Smart Blob ..");
      smb.IfxLoRelease(loPtr);
      System.out.println("Locator is released ..");
      }
   rs.close();
   }
catch(SQLException e)
   {
   System.out.println("Select Failed ...\n" +e.getMessage());
         }
```

First, the ResultSet.getBlob() method gets an object of type BLOB. The casting is required to convert the returned object to an object of type **IfxBblob**. Next, the IfxBblob.getLocator() method gets an **IfxLocator** object from the **IfxBblob** object. After the **IfxLocator** object is available, you can instantiate an **IfxSmartBlob** object and use the IfxLoOpen() and IfxLoRead() methods to read the smart large object data. Fetching **CLOB** data is similar, but it uses the methods ResultSet.getClob(), IfxCblob.getLocator(), and so on.

If you use getBlob() or getClob() to fetch data from a column of type BLOB, you do not need to use the Informix® extensions to retrieve the actual BLOB content as outlined in the preceding sample code. You can simply use Java.Blob.getBinaryStream() or Java.Clob.getAsciiStream() to retrieve the content. HCL Informix® JDBC Driver implicitly gets the content from the database server for you, using basically the same steps as the sample code. This approach is simpler than the approach of the preceding example but does not provide as many options for reading the contents of the BLOB column.

Work with opaque types

An *opaque data type* is an atomic data type that you define to extend the database server. The database server has no information about the opaque data type until you provide routines that describe it.

Extending the database server also frequently requires that you create *user-defined routines* (UDRs) to support the extensions. A UDR is a routine that you create that can be invoked in an SQL statement, by the database server, or from another UDR. UDRs can be part of opaque types, or they can be separate.

The JDBC 3.0 standard provides the java.sql.SQLInput and java.sql.SQLOutput methods to access opaque types. The definition of these interfaces is extended to fully support HCL Informix® fixed binary and variable binary opaque types. This extension includes the following interfaces:

- **IfmxUdtSQLInput**
- **IfmxUdtSQLOutput**

In addition, the following classes simplify creating Java™ opaque types and UDRs in the database server from a JDBC client application:

- **UDTManager**
- **UDTMetaData**
- **UDRManager**
- **UDRMetaData**

The **UDTManager** and **UDRManager** classes provide an infrastructure for mapping client-side Java™ classes as opaque data types and UDRs and storing their instances in the database.

This facility works only in client-side JDBC. For details about the features and limitations of server-side JDBC, see the *HCL® J/Foundation Developer's Guide*.

For detailed information about opaque types and UDRs, see the following publications:

- *HCL® Informix® User-Defined Routines and Data Types Developer's Guide* discusses the terms and concepts about opaque types and UDRs that you need to use the information in this section, including the internal data structure, support functions, and implicit and explicit casts.
- The *HCL® J/Foundation Developer's Guide* discusses information specific to writing UDRs in Java™.

The IfmxUDTSQLInput interface

The **com.informix.jdbc.IfmxUdtSQLInput** interface extends **java.sql.SQLInput** with several added methods. To use these methods, you must cast the **SQLInput** references to **IfmxUdtSQLInput**. The methods allow you to perform the following functions:

- Read data.
- Position in the data stream.
- Set or obtain attributes of the data.

Read data

The readString() method reads the next attribute in the stream as a Java™ string. The readBytes() method reads the next attribute in the stream as a Java™ byte array. Both methods are similar to the SQLInput.readBytes() method except that a fixed length of data is read in:

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

In both methods, you must supply a length for HCL Informix® JDBC Driver to read the next attribute properly, because the characteristics of the opaque type are unknown to the driver. The *maxlen* parameter specifies the maximum length of data to read in.

Position in the data stream

The getCurrentPosition() method retrieves the current position in the input stream. The setCurrentPosition() method changes the position in the input stream to the position specified by the *position* parameter:

```
public int getCurrentPosition();
public void setCurrentPosition(int position) throws SQLException;
public void skipBytes(int len) throws SQLException;
```

The *position* parameter must be a positive integer. The skipBytes() method changes the position in the input stream by the number of bytes specified by the *len* parameter, relative to the current position. The *len* parameter must be a positive integer.

In both setCurrentPosition() and skipBytes(), HCL Informix® JDBC Driver generates an **SQLException** if the new position specified is after the end of the input stream.

Set or obtain data attributes

The length() method returns the total length of the entire data stream. The getAutoAlignment() method retrieves the `TRUE` or `FALSE` (on or off) state of the auto alignment feature. The setAutoAlignment() method sets the state to `TRUE` or `FALSE`:

```
public int length();
public boolean getAutoAlignment();
public void setAutoAlignment(boolean value);
```

Important: Setting the auto alignment feature might result in discarded bytes from the input stream if the data is not already aligned. JDBC applications should provide aligned data or set the auto alignment feature to `FALSE`.

The IfmxUDTSQLOutput interface

The **com.informix.jdbc.IfmxUdtSQLOutput** interface extends **java.sql.SQLOutput** with the following added methods:

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

To use these methods, you must cast the **SQLOutput** references to **IfmxUdtSQLOutput**.

Use the writeString() method to write the next attribute to the stream as a Java™ string. If the string passed in is shorter than the specified length, HCL Informix® JDBC Driver pads the string with zeros.

Use the writeBytes() method to write the next attribute to the stream as a Java™ byte array.

Both methods are similar to the SQLOutput.writeBytes() method except that a fixed length of data is written to the stream. If the array or string passed in is shorter than the specified length, HCL Informix® JDBC Driver pads the array or string with zeros. In both methods, you must supply a length for HCL Informix® JDBC Driver to write the next attribute properly, because the opaque type is unknown to the driver.

Map opaque data types

HCL Informix® opaque types map to Java™ objects, which must implement the **java.sql.SQLData** interface. These Java™ objects describe all the data members that make up the opaque type. These Java™ objects are strongly typed; that is, each read or write method in the **readSQL** or **writeSQL** method of the Java™ object must match the corresponding data member in the opaque type definition.HCL Informix® JDBC Driver cannot perform any type conversion because the type structure is unknown to it.

HCL Informix® JDBC Driver also requires that all opaque data be transported as Informix® DataBlade® API data types, as defined in `mitypes.h` (this file is included in all HCL Informix® installations). All opaque data is stored in the database server table in a C struct, which is made up of various DataBlade® API types, as defined in the opaque type.

You do not need to handle mapping between Java™ and C if you use the UDT and UDR Manager facility to create opaque types. For more information, see Creating opaque types and UDRs on page 158.

Type cache information

When objects of some data types insert data into columns of certain other data types, HCL Informix® JDBC Driver verifies that the data provided matches the data the database server expects by calling the SQLData.getSQLTypeName() method. The driver asks the database server for the type information with each insertion.

This occurs in the following cases:

- When an **SQLData** object inserts data into an opaque type column and getSQLTypeName() returns the name of the opaque type
- When a **Struct** or **SQLData** object inserts data into a row column and getSQLTypeName() returns the name of a named row
- When an **SQLData** object inserts data into a DISTINCT type column,

In the database URL, you can set the environment variable **ENABLE_TYPE_CACHE**`=TRUE` to have the driver cache the data type information the first time it is retrieved. The driver then asks the cache for the type information before requesting the data from the database server.

Unsupported methods

The following methods of the **SQLInput** and **SQLOutput** interfaces are not supported for opaque types:

- **java.sql.SQLInput**
  - readAsciiStream()
  - readBinaryStream()
  - readBytes()
  - readCharacterStream()
  - readObject()
  - readRef()
  - readString()
- **java.sql.SQLOutput**
  - writeAsciiStream(InputStream x)
  - writeBinaryStream(InputStream x)
  - writeBytes(byte[] x)
  - writeCharacterStream(Reader x)
  - writeObject(Object x)
  - writeRef(Ref x)
  - writeString(String x)

Creating opaque types and UDRs

About this task

The **UDTManager** and **UDRManager** classes allow you to easily create and deploy opaque types and user-defined routines (UDRs) in the database server.

Before using the information in this section, read the following two additional publications:

- For information about configuring your system to support Java™ UDRs, see the *HCL® J/Foundation Developer's Guide*.
- For detailed information about developing opaque types, see *HCL® Informix® User-Defined Routines and Data Types Developer's Guide*.

Overview of creating opaque types and UDRs

In the database server, any Java™ class that implements the **java.sql.SQLData** interface and is accessible to the Java™ Virtual Machine can be stored as an opaque type. The **UDTManager** and **UDRManager** classes, together with their supporting **UDTMetaData** and **UDRMetaData** classes, extend this facility to client applications: your Java™ client application can use these classes to create opaque types and user-defined routines and transfer their class definitions to the database server. The client does not need to be accessible to the database server to use this functionality.

Important: This functionality is tightly coupled with server support for creating and using Java™ opaque types and user-defined routines. Any limitations on using Java™ opaque types and user-defined routines that exist in

your version of the database server apply equally to Java™ opaque types and routines you create in your client applications.

When you use the **UDTManager** and **UDTMetaData** classes, HCL Informix® JDBC Driver performs all of the following actions for your application:

1. Obtains the JAR file you specify
2. Transports the JAR file from the client local area to the server local area

   You define the server local area using the UDTManager.setJarFileTmpPath() method. The default is `/tmp` on UNIX™ systems and `C:\temp` on Windows™ systems.
3. Installs the JAR file in the server
4. Registers the opaque data type in the database with the CREATE OPAQUE TYPE SQL statement, taking input from the **UDTMetaData** class
5. Registers the support functions and casts you provide for the opaque type using the CREATE Function and CREATE CAST SQL statements

   You define support functions and casts using the setSupportUDR() and setXXXCast() methods in the **UDTMetaData** class.

   If you do not provide input and output functions for the opaque type, the driver registers the default functions (see the release notes for any limitations on this feature).
6. Registers any other nonsupport routines or casts (if any) that you specified, taking input from the UDTMetaData.setUDR() and UDTMetaData.setXXXCast() method calls in your application
7. Creates a mapping between an SQL OPAQUE type and a Java™ object (using the sqlj.setUDTExtName() method)

When you use the **UDRManager** and **UDRMetaData** classes, HCL Informix® JDBC Driver performs the following actions:

1. Obtains the JAR file you specify
2. Transports the JAR file from the client local area to the server local area
3. Installs the JAR file in the server
4. Registers the UDRs in the database with the CREATE FUNCTION SQL statement, taking input from the UDRMetaData.setUDR() method calls in your application

The methods in the UDT and UDR Manager facility perform the following main functions:

- Creating opaque types in Java™ without preexisting Java™ classes, using the default input and output methods the server provides
- Converting existing Java™ classes on the client to opaque types and UDRs in the database server
- Converting Java™ static methods to UDRs

Preparing to create opaque types and UDRs

Before you begin

Before using the UDT and UDR Manager facility, perform the following setup tasks:

- Make sure your database server supports Java™.

  The UDT and UDR Manager facility does not work in legacy servers that do not include Java™ support.
- Include either the `ifxtools.jar` or `ifxtools_g.jar` file in your CLASSPATH setting.
- Create a directory named `/usr/informix` in the database server, with owner and group set to user **informix** and permissions set to `777`.
- Add the following entry to the `/etc/group` file in the database server:

  ```
  informix::unique-id-number:
  ```
- Check the release notes for the driver and database server for any further limitations in this release.

Creating opaque types

About this task

Using UDT Manager, you can create a Java™ opaque type from an existing Java™ class that implements the **SQLData** interface. UDT Manager can also help you create a Java™ opaque type without requiring that you have the Java™ class ready; you specify the characteristics of the opaque type you want to create, and the UDT Manager facility creates the Java™ class and then the Java™ opaque type.

Follow the steps in this section to use the **UDTManager** classes.

Creating an opaque type from an existing Java™ class

About this task

To create an opaque type from an existing Java™ class:

1. Ensure that the class meets the requirements for conversion to an opaque type.

   For the requirements, see Requirements for the Java class on page 164.
2. If you do not want to use the default input and output routines provided by the server, write support UDRs for input and output.

   For general information about writing support UDRs, see *HCL® Informix® User-Defined Routines and Data Types Developer's Guide*.
3. Create a default sbspace on the database server to hold the JAR file that contains the code for the opaque type.

   For information about creating an sbspace, see the *HCL® Informix® Administrator's Guide* for your database server and the *HCL® J/Foundation Developer's Guide*.
4. Open a JDBC connection.

Make sure a database object is associated with the connection object. The driver cannot create an opaque type without a database object. For details about creating a connection with a database object, see Connect to the database on page 14.

5. Instantiate an **UDTManager** object and an **UDTMetaData** object:

```
UDTManager udtmgr = new UDTManager(connection);
UDTMetaData mdata = new UDTMetaData();
```

6. Set properties for the opaque type by calling methods in the **UDTMetaData** object.

   At a minimum, you must specify the SQL name, UDT length, and JAR file SQL name. For an explanation of SQL names, see SQL names on page 165.

   You can also specify the alignment, implicit and explicit casts, and any support UDRs:

```
mdata.setSQLName("circle2");
mdata.setLength(24);
mdata.setAlignment(UDTMetaData.EIGHT_BYTE)
mdata.setJarFileSQLName("circle2_jar");
mdata.setUDR(areamethod, "area");
mdata.setSupportUDR(input, "input", UDTMetaData.INPUT)
mdata.setSupportUDR(output, "output",UDTMetaData.OUTPUT)
mdata.SetImplicitCast(com.informix.lang.IfxTypes.IFX_TYPE_
   LVARCHAR, "input");
mdata.SetExplicitCast(com.informix.lang.IfxTypes.IFX_TYPE_
   LVARCHAR, "output");
```

7. If desired, specify a path name where the driver should place the JAR file in the database server file system:

```
String pathname = "/work/srv93/examples";
udtmgr.setJarFileTmpPath(pathname);
```

   Make sure the path exists in the server file system. For more information, see Specify a JAR file temporary path on page 170.

8. Create the opaque type:

```
udtmgr.createUDT(mdata, "Circle2.jar", "Circle2", 0);
```

Results

For additional information about creating an opaque type from existing code, see Creating an opaque type from existing code on page 170.

For a complete code example of using the preceding steps to create an opaque type, see Create an opaque type from an existing Java class with UDTManager on page 181.

Creating an opaque type without an existing Java™ class

About this task

To create an opaque type without an existing Java™ class:

1. Create a default sbspace on the database server to hold the JAR file that contains the code for the opaque type.

   For information about creating an sbspace, see the *HCL® Informix® Administrator's Guide* for your database server and the *HCL® J/Foundation Developer's Guide*.

2. Open a JDBC connection.

   Make sure the connection object has a database object associated with it. For details, see Connect to the database on page 14.

3. Instantiate a **UDTManager** object and a **UDTMetaData** object:

   ```
   UDTManager udtmgr = new UDTManager(connection);
   UDTMetaData mdata = new UDTMetaData();
   ```

4. Specify the characteristics of the opaque type by calling methods in the **UDTMetaData** class:

   ```
   mdata.setSQLName("acircle");
   mdata.setLength(24);
   mdata.setFieldCount(3);
   mdata.setFieldName(1, "x");
   mdata.setFieldName(2, "y");
   mdata.setFieldName(3, "radius");
   mdata.setFieldType
      (1,com.informix.lang.IfxTypes.IFX_TYPE_INT);
   mdata.setFieldType
      (2,com.informix.lang.IfxTypes.IFX_TYPE_INT);
   mdata.setFieldType
      (3,com.informix.lang.IfxTypes.IFX_TYPE_INT);
   mdata.setJarFileSQLName("ACircleJar");
   ```

   For more information about setting characteristics for opaque types, see Specify characteristics for an opaque type on page 165.

5. Create the Java™ file, the class file, and the JAR file:

   ```
   mdata.keepJavaFile(true);
   String classname = udtmgr.createUDTClass(mdata);
   String jarfilename = udtmgr.createJar(mdata, new String[]
      {classname + .class"});
   ```

   For more information, see Creating the JAR and class files on page 168.

6. If desired, specify a path name where the driver should place the JAR file in the database server file system:

   ```
   String pathname = "/work/srv93/examples";
   udtmgr.setJarFileTmpPath(pathname);
   ```

   Make sure the path exists in the server file system. For more information, see Specify a JAR file temporary path on page 170.

7. Send the class definition to the database server:

   ```
   udtmgr.createUDT(mdata, jarfilename, classname, 0);
   ```

   For more information, see Send the class definition to the database server on page 169.

Results

For a complete code example of using the preceding steps to create an opaque type, see Create an opaque type without an existing Java class on page 190.

Creating a UDR

About this task

The following topics shows you how to create a UDR from a Java™ class.

To create a UDR:

1. Write a Java™ class with one or more static method to be registered as UDRs.

   For more information, see Requirements for the Java class on page 164.

2. Create an sbspace on the database server to hold the JAR file that contains the code for the UDR.

   For information about creating an sbspace, see the *HCL® Informix® Administrator's Guide* for your database server and the *HCL® J/Foundation Developer's Guide*.

3. Open a JDBC connection.

   Make sure the connection object has a database object associated with it. For details, see Connect to the database on page 14.

4. Instantiate a **UDRManager** object and a **UDRMetaData** object:

   ```
   UDRManager udrmgr = new UDRManager(myConn);
   UDRMetaData mdata = new UDRMetaData();
   ```

5. Create **java.lang.Reflect.Method** objects for the static methods to be registered as UDRs.
   Example
   In the following example, **method1** is an instance that represents the udr1(string, string) method in the Group1 java class; **method2** is an instance that represents the udr2(Integer, String, String) method in the **Group1** Java™ class:

   ```
   Class gp1 = Class.forName("Group1");
   Method method1 = gp1.getMethod("udr1",
          new Class[]{String.class, String.class});
   Method method2 = gp1.getMethod("udr2",
          new Class[]{Integer.class, String.class, String.class});
   ```

6. Specify which methods to register as UDRs.

   The second parameter specifies the SQL name of the UDR:

   ```
   mdata.setUDR(method1, "group1_udr1");
   mdata.setUDR(method2, "group1_udr2");
   ```

   For more information, see Create UDRs on page 173.

7. Specify the JAR file SQL name:

   ```
   mdata.setJarFileSQLName("group1_jar");
   ```

8. If desired, specify a path name where the driver should place the JAR file in the database server file system:

   ```
   String pathname = "/work/srv93/examples";
   udrmgr.setJarFileTmpPath(pathname);
   ```

   Make sure the path exists in the database server file system. For more information, see Specify a JAR file temporary path on page 170.

9. Install the UDRs in the database server:

   ```
   udrmgr.createUDRs(mdata, "Group1.jar", "Group1", 0);
   ```

   For more information, see Create UDRs on page 173.

Results

For complete code examples of creating UDRs, see Create UDRs with UDRManager on page 194.

Requirements for the Java™ class

To qualify for converting into an opaque type, your Java™ class must meet the following conditions:

- The class must implement the **java.sql.SQLData** interface. For an example, see Examples on page 176.
- If the class contains another opaque type, the additional opaque type must be implemented in a similar way and the additional `.class` file must be packaged as part of the same JAR file as the original opaque type.
- If the class contains DISTINCT types, the class can either implement the **SQLData** interface for the DISTINCT types or let the driver map the DISTINCT types to the base types. For more information, see Distinct data types on page 95.
- The class cannot contain complex types.
- If you are creating an opaque type from an existing Java™ class and using the default support functions in the database server, you must cast the **SQLInput** and **SQLOutput** streams in SQLData.readSQL() and SQLData.writeSQL() to **IfmxUDTSQLInput** and **IfmxUDTSQLOutput**.

  For a code example that shows how to do this, see Create an opaque type using default support functions on page 181.
- All Java™ methods for the opaque type must be in the same `.java` file with the class that defines the opaque type.

Additional requirements for UDRs are as follows:

- All class methods to be registered as UDRs must be static.
- The method argument types and the return types must be valid Java™ data types.
- The methods can use all basic nongraphic Java™ packages that are included in the Java™ development kit, such as `java.util`, `java.io`, `java.net`, `java.rmi`, `java.sql`, and so forth.
- Data types of method arguments and return types must conform to the data type mapping tables shown in Data type mapping for UDT manager and UDR manager on page 259.
- The following SQL argument or return types are not supported:
  - MONEY
  - DATETIME with qualifier other than hour to second or year to fraction(5)

- INTERVAL with qualifier other than year to month or day to fraction(5)
- Any data type not shown in the mapping tables for method arguments and return types; for the tables, see Data type mapping for UDT manager and UDR manager on page 259.

SQL names

Some of the methods in the **UDTMetaData** class set an *SQL name* for an opaque type or a JAR file that contains the opaque type or UDR code. The SQL name is the name of the object as referenced in SQL statements. For example, assume your application makes the following call:

```
mdata.setSQLName("circle2");
```

The name as used in an SQL statement is as follows:

```
CREATE TABLE tab (c circle2);
```

Similarly, assume the application sets the JAR file name as follows:

```
mdata.setJarFileSQLname("circle2_jar");
```

The JAR file name as referenced in SQL is as follows:

```
CREATE FUNCTION circle2_output (...)
RETURNS circle2
EXTERNAL NAME
   'circle2_jar: circle2.fromString (...)'
LANGUAGE JAVA
NOT VARIANT
END FUNCTION;
```

Important: There is no default value for an SQL name. Use the setSQLname() or setJarFileSQLName() method to specify a name, otherwise an SQL exception will be thrown.

Specify characteristics for an opaque type

The following topics provide additional information about creating an opaque type without a preexisting Java™ class. Details about creating an opaque type from an existing Java™ class begin with Creating an opaque type from existing code on page 170.

Using the methods in the **UDTMetaData** class, you can specify characteristics for a new opaque type. These settings apply for new opaque types; for opaque types created from existing files, see Creating an opaque type from existing code on page 170.

You can set the following characteristics:

- The number of fields in the internal data structure that defines the opaque type
- Additional characteristics, such as data type, name, and scale, of each field in the internal structure that defines the opaque type
- The length of the opaque type
- The alignment of the opaque type

- The SQL name of the opaque type and the JAR file
- The name of the generated Java™ class
- Whether to keep the generated `.java` file

Specify field count

The setFieldCount() method specifies the number of fields in the internal data structure that defines the opaque type:

```
public void setFieldCount(int fieldCount) throws SQLException
```

Specify additional field characteristics

The following methods set additional characteristics for fields in the internal data structure:

```
public void setFieldName (int field, String name) throws SQLException
public void setFieldType (int field, int ifxtype) throws SQLException
public void setFieldTypeName(int field, String sqltypename) throws SQLException
public void setFieldLength(int field, int length) throws SQLException
```

The *field* parameter indicates the field for which the driver should set or obtain a characteristic. The first field is `1`; the second field is `2`, and so forth.

The name you specify with setFieldName() appears in the Java™ class file. The following example sets the first field name to `IMAGE`.

```
mdata.setFieldName(1, "IMAGE");
```

The setFieldType() method sets the data type of a field using a constant from the file **com.informix.lang.IfxTypes**. For more information, see Mapping for field types on page 261. The following example specifies the CHAR data type for values in the third field:

```
mdata.setFieldType(3, com.informix.lang.IfxTypes.IFX_TYPE_CHAR);
```

The setFieldTypeName() method sets the data type of a field using the SQL data type name:

```
mdata.setFieldTypeName(1, "IMAGE_UDT");
```

This method is valid only for opaque and distinct types; for other types, the driver ignores the information.

The *length* parameter has the following meanings, depending on the data type of the field:

Character types

Maximum length in characters

DATETIME

Encoded length

INTERVAL

Encoded length

Other data type or no type specified

Driver ignores the information

The possible values for encoded length are those in the JDBC 2.20 specification: hour to second; year to second; and year to fraction(1), year to fraction(2), up through year to fraction(5).

The following example specifies that the third (VARCHAR) field in an opaque type cannot store more than 24 characters:

```
mdata.setFieldLength(3, 24);
```

Specify length

The setLength() method specifies the total length of the opaque type:

```
public void setLength(int length) throws SQLException
```

If you are creating an opaque type from an existing Java™ class and do not specify a length, the driver creates a variable-length opaque type. If you are creating an opaque type without an existing Java™ class, you must specify a length; UDT Manager creates only fixed-length opaque types in this case.

Specify alignment

The setAlignment() method specifies the opaque types alignment:

```
public void setAlignment(int alignment)
```

The *alignment* parameter is one of the alignment values shown in the next section. If you do not specify an alignment, the database server aligns the opaque type on 4-byte boundaries.

Alignment values

Alignment values are shown in the following table.

Value	Constant	Structure begins with	Boundary aligned on
1	SINGLE_BYTE	1-byte quantity	single-byte
2	TWO_BYTE	2-byte quantity (such as SMALLINT)	2-byte
4	FOUR_BYTE	4-byte quantity (such as FLOAT or UNSIGNED INT)	4-byte
8	EIGHT_BYTE	8-byte quantity	8-byte

Specify SQL names

Specify SQL names with the setSQLName() and setJarFileSQLName() methods:

```
public void setSQLName(String name) throws SQLException
public void setJarFileSQLName(String name) throws SQLException
```

By default, the driver uses the name you set through the setSQLName() method as the file names of the Java™ class and JAR files generated when you call the UDTManager.createUDTCclass() and UDTManager.createJar() methods. For example, if you called `setSQLName("circle")` and then called createUDTCclass() and createJar(), the class file name generated would

be `circle.class` and the JAR file name would be `circle.jar`. You can specify a Java™ class file name other than the default by calling the setClassName() method.

The JAR file SQL name is the name as it will be referenced in the SQL CREATE FUNCTION statement the driver uses to register a UDR.

Important: The JAR file SQL name is the name of the JAR file in SQL statements; it has no relationship to the contents of the JAR file.

Specify the Java™ class name

Use setClassName() to specify the Java™ class name:

```
public void setClassName(String name)throws SQLException
```

If you do not set a class name with setClassName(), the driver uses the SQL name of the opaque type (set through setSQLName()) as the name of the Java™ class and the file name of the `.class` file generated by the createUDTCclass() method.

Specifying Java™ source file retention

Use keepJavaFile() to specify whether to retain the `.java` source file:

```
public void keepJavaFile(boolean value)
```

The *value* parameter indicates whether the createUDTClass() method should retain the `.java` file that it generates when it creates the Java™ class file for the new opaque type. The default is to remove the file. The following example specifies keeping the `.java` file:

```
mdata.keepJavaFile(true);
```

Creating the JAR and class files

About this task

Once you have specified the characteristics of the opaque type through the **UDTMetaData** methods, you can use the methods in the **UDTManager** class to create opaque types and their class and JAR files in the following order:

1. Instantiate the **UDTManager** object.

   The constructor is defined as follows:

   ```
   public UDTManager(Connection conn) throws SQLException
   ```

2. Create the `.class` and `.java` files with the createUDTClass() method.
3. Create the `.jar` file with the createJar() method.
4. Create the opaque type with the createUDT() method.

Create the .class and .java files

The createUDTClass() method has the following signature:

```
public String createUDTClass(UDTMetaData mdata) throws SQLException
```

The createUDTClass() method causes the driver to perform all of the following actions for your application:

1. Creates a Java™ class with the name you specified in the UDTMetaData.setClassName() method

   If no class name was specified, the driver uses the name specified in the UDTMetaData.setSQLName() method.
2. Puts the Java™ class code into a `.java` file and then compile the file to a `.class` file
3. Returns the name of the newly created class to your application

If you specified `TRUE` by calling the UDTMetaData.keepJavaFile() method, the driver retains the generated `.java` file. The default is to delete the `.java` file.

Your application should call the createUDTClass() method only to create new `.class` and `.java` files to define an opaque type, not to generate an opaque type from existing files.

Create the .jar file

The createJar() method compiles the class files you specify in the *classnames* list. The files in the list must have the `.class` extension.

```
public String createJar(UDTMetaData mdata, String[] classnames)
   throws SQLException;
```

The driver creates a JAR file named *sqlname*`.jar` (where *sqlname* is the name you specified by calling UDTMetaData.setSQLName()) and returns the file name to your application.

Send the class definition to the database server

After you have created the JAR file, use the UDTManager.createUDT() method to create the opaque type by sending the class definition to the database server:

```
public void createUDT(UDTMetaData mdata, String jarfile, String
   classname, int deploy) throws SQLException;
```

The *jarfile* parameter is the path name of a JAR (`.jar`) file that contains the class definition for the opaque type. By default, the classes in the `java.io` package resolve relative path names against the current user directory as named by the system property **user.dir**; it is typically the directory in which the Java™ Virtual Machine was invoked. The file name must be included in your CLASSPATH setting if you use an absolute path name.

The *classname* parameter is the name of the class that implements the opaque type.

The SQL name of the opaque type defaults to the class name if your application does not call setClassName(). You can specify an SQL name by calling the UDTMetaData.setSQLName() method.

Important: If your application calls createUDT() within a transaction or your database is ANSI or enables logging, some extra guidelines apply. For more information, see Execute in a transaction on page 176.

Specify deployment descriptor actions

In the **UDTManager** and **UDRManager** methods, the *deploy* parameter indicates whether install_actions should be executed if a deployment descriptor is present in the JAR file. The *undeploy* parameter indicates whether remove_actions should be executed.

0

Execute install_actions or remove_actions.

Nonzero

Do not execute install_actions or remove_actions.

A deployment descriptor allows you to include the SQL statements for creating and dropping UDRs in a JAR file. For more information about the deployment descriptor, see the *HCL® J/Foundation Developer's Guide* and the SQLJ specification.

Specify a JAR file temporary path

When the driver ships the JAR file for an opaque type or UDR, it places the file by default in `/tmp` (on UNIX™) or in `C:\temp` (on Windows™). You can specify an alternative path name by calling the setJarTmpPath() method in either the **UDTManager** or **UDRManager** class:

```
public void setJarTmpPath(String path) throws SQLException
```

You can call this method at any point before calling createUDT() or createUDR(), the **UDTManager** or **UDRManager** objects. The *path* parameter must be an absolute path name, and you must ensure that the path exists on the server file system.

Creating an opaque type from existing code

About this task

The preceding topics describe methods you use to create a new opaque type without an existing Java™ class. When you create an opaque type from existing Java™ code, you specify the SQL name, JAR file SQL name, support UDRs (if any), and any additional nonsupport UDRs that are included in the opaque type. (For an explanation of SQL names, see SQL names on page 165.) You can also specify the length, alignment, and implicit and explicit casts.

To create an opaque type from existing code, use the following methods:

- UDTMetaData.setSQLName() to specify the SQL name of the opaque type as referenced in SQL statements
- UDTMetaData.setSupportUDR() for each support UDR in the opaque type

  Support UDRs are input/output, send/receive, and so forth.
- UDTMetaData.setUDR() for each nonsupport UDR in the opaque type
- UDTMetaData.setJarFileSQLName() to specify an SQL name for the JAR file

- UDTMetaData.setImplicitCast() or UDTMetaData.setExplicitCast() to specify each cast
- UDTMetaData.setLength() if the opaque type is fixed length (the driver defaults to variable length)
- UDTMetaData.setAlignment() to specify the byte boundary on which the opaque type is aligned (necessary only if you do not want the database server to default to a 4-byte boundary)
- UDTManager.createJar() to create a JAR (`.jar`) file if you do not already have one
- UDTManager.createUDT() to create the opaque type

In addition, the setXXXCast(), setSupportUDR(), and **setUDR()** methods are used only for creating an opaque type from existing code:

```
public void setImplicitCast(int ifxtype, String methodsqlname)
   throws SQLException

public void setExplicitCast(int ifxtype, String methodsqlname)
   throws SQLException

public void setSupportUDR(Method method, String sqlname, int type)
   throws SQLException
public void setUDR(Method method, String sqlname)
   throws SQLException
```

The setXXXCast() methods

The setXXXCast() methods specify the implicit or explicit cast to convert data from an opaque type to the data type specified.

The *ifxtype* parameter is a type code from the class **com.informix.lang.IfxTypes**. Data type mapping between the *ifxtype* parameter and the SQL type in the database server is detailed in Mapping for casts on page 260. The *methodsqlname* parameter is the SQL name of the Java™ method that implements the cast.

The following example sets an implicit cast implemented by a Java™ method with the SQL name **circle2_input**:

```
setImplicitCast(com.informix.lang.IfxTypes.IFX_TYPE_LVARCHAR,
   "circle2_input");
```

The following example sets an explicit cast implemented by a Java™ method with the SQL name **circle_output**:

```
setExplicitCast(com.informix.lang.IfxTypes.IFX_TYPE_LVARCHAR,
   "circle2_output");
```

The following example sets an explicit cast for converting a **circle2** opaque type to an integer:

```
setExplicitCast(com.informix.lang.IfxTypes.IFX_TYPE_INT,
   "circle2_to_int");
```

The setSupportUDR() and setUDR() methods

The setSupportUDR() method specifies a Java™ method in an existing Java™ class that will be registered as a support UDR for the opaque type.

The *method* parameter specifies an object from **java.lang.reflect.Method** to be registered as a Java™ support UDR for the opaque type in the database server. Support UDRs are Input, Output, Send, Receive, and so forth (for more information, see *HCL® Informix® User-Defined Routines and Data Types Developer's Guide*.)

The *sqlname* parameter specifies the SQL name of the method. For more information, see SQL names on page 165.

The *type* parameter specifies the kind of support UDR, as follows:

```
UDTMetaData.INPUT
UDTMetaData.OUTPUT
UDTMetaData.SEND
UDTMetaData.RECEIVE
UDTMetaData.IMPORT
UDTMetaData.EXPORT
UDTMetaData.BINARYIMPORT
UDTMetaData.BINARYEXPORT
```

For step-by-step information about creating an opaque type from existing code, see Creating an opaque type from an existing Java class on page 160.

Tip: It is not necessary to register the methods in the SQLData interface. For example, you do not need to register SQLData.getSQLTypeName(), SQLData.readSQL(), or SQLData.writeSQL().

To specify other UDRs, use setUDR() as described in Create UDRs on page 173.

Remove opaque types and JAR files

You can remove opaque types and their JAR files using the following methods:

```
public static void removeUDT(String sqlname) throws SQLException
public static void removeJar(String jarfilesqlname, int undeploy)
   throws SQLException
```

The removeUDT() method removes the opaque type, with all its casts and UDRs, from the database server. It does not remove the JAR file itself because other opaque types or UDRs could be using the same JAR file.

Important: If your application calls removeUDT() within a transaction or if your database is ANSI or enables logging, some extra guidelines apply. For more information, see Execute in a transaction on page 176.

The removeJar() method removes the JAR file from the system catalog. The *jarfilesqlname* parameter is the name you specified with the setJarFileSQLName() method.

For the *undeploy* parameter, see Specify deployment descriptor actions on page 170.

Important: Before calling removeJar(), you must first remove all functions and procedures that depend on the JAR file. Otherwise, the database server fails to remove the file.

Create UDRs

Using UDR Manager to create UDRs in the database server involves:

- Coding the UDRs and packaging the code in a JAR file

  For details about coding UDRs, see the *HCL® J/Foundation Developer's Guide*.
- Creating a default sbspace in the database server to hold the JAR file that contains the code for the UDR

  For information about creating an sbspace, see the *HCL® Informix® Administrator's Guide* for your database server and the *HCL® J/Foundation Developer's Guide*.
- Calling methods in the **UDRMetaData** class to specify the information necessary for HCL Informix® JDBC Driver to register the UDRs in the database server
- If desired, specifying a path name where the driver should place the JAR file in the database server file system
- Installing the UDRs in the server

Creating a UDR for a C-language opaque type is not supported; the opaque type must be in Java™.

To specify a UDR for the driver to register, use this method in **UDRMetaData**:

```
public void setUDR(Method method, String sqlname) throws SQLException
```

The *method* parameter specifies an object from **java.lang.Reflect.Method** to be registered as a Java™ UDR in the database server. The *sqlname* parameter is the name of the method as used in SQL statements.

Once you have specified the UDRs to be registered, you can set the JAR file SQL name using UDRMetaData.setJarFileSQLName() and then use the UDRManager.createUDRs() method to install the UDRs in the database server, as follows:

```
public void createUDRs(UDRMetaData mdata, String jarfile, String
   classname, int deploy) throws SQLException
```

The *jarfile* parameter is the absolute or relative path name of the client-side JAR file that contains the Java™ method definitions. If you use the absolute path name, the JAR file name must be included in your CLASSPATH setting.

The *classname* parameter is the name of a Java™ class that contains the methods you want to register as UDRs in the database server. Requirements for preparing the Java™ methods are described on 1 on page 163.

For the *deploy* parameter, see Specify deployment descriptor actions on page 170.

The createUDRs() method causes the driver to perform all of the following steps for your application:

1. Obtain the JAR file designated by the first parameter.
2. Transport the JAR file from the client local area to the server local area.

3. Register the UDRs specified in the **UDRMetaData** object (set through one or more calls to UDRMetaData.setUDR()).
4. Install the JAR file and create the UDRs in the server.

After createUDRs() executes, your application can use the UDRs in SQL statements.

Important: If your application calls createUDRs() within a transaction, or if your database is ANSI or enables logging, some extra guidelines apply. For more information, see Execute in a transaction on page 176.

Remove UDRs and JAR files

You can remove UDRs using the following methods:

```
public void removeUDR(String sqlname) throws SQLException
public void removeJar(String jarfilesqlname, int undeploy) throws
   SQLException
```

Tip: The removeUDR() method removes the UDR from the server but does not remove the JAR file, because other opaque types or UDRs could be using the same JAR file.

The removeJar() method is described in Remove opaque types and JAR files on page 172.

Remove overloaded UDRs

To remove overloaded UDRs, use the removeUDR() method with an additional parameter:

```
public void removeUDR(String sqlname, Class[] methodparams) throws
   SQLException
```

The *methodparams* parameter specifies the data type of each parameter in the UDR. Specify NULL to indicate no parameters. For example, assume a UDR named print() is overloaded with two additional method signatures.

Java™ method signature	Correspo nding SQL name
void print()	print1
void print(String x, String y, int r)	print2
void print(int a, int b)	print3

The code to remove all three UDRs is:

```
udrmgr.removeUDR("print1", null );
udrmgr.removeUDR("print2",
   new Class[] {String.class, String.class, int.class} );
udrmgr.removeUDR("print3", new Class[] {int.class, int.class} );
```

Obtain information about opaque types and UDRs

Many of the setXXX() methods in the **UDTMetaData** and **UDRMetaData** classes have parallel getXXX() methods for obtaining characteristics of existing opaque types and UDRs.

The getXXX() methods in the UDTMetaData class

The following table summarizes the available getXXX() methods in the **UDTMetaData** class. For the *field* parameter, `1` designates the first field in the internal data structure, `2` is the second, and so forth. For details about SQL names, see SQL names on page 165.

Information obtained	Method signature	Additional information
Number of fields in the internal data structure	public int getFieldCount()	Returns `0` if no fields are present
Name of a field in the internal data structure	public String getFieldName int *field*) throws SQLException	Returns NULL if no name exists
Data type code of a field in the internal data structure	public int getFieldType (int *field*) throws SQLException	Data type codes come from the class **com.informix.lang.IfxTypes**. Returns `-1` if no data type exists
Data type name of a field in the internal data structure	public String getFieldTypeName (int *field*) throws SQLException	Returns NULL if no name exists
For character type: maximum number of characters in the field; for date-time or interval type: encoded qualifier	public int getFieldLength (int *field*) throws SQLException	Returns `-1` if no length was set
SQL name of the opaque type	public String getSQLName()	Returns NULL if no name was set
SQL name of the JAR file	public String getJarFileSQLName()	Returns NULL if no name was set
Name of the Java™ class for the opaque type	public String getClassName()	If no class name was set through setClassName(), *sqlname* is returned (this is the default). If no SQL name was set through setSQLName(), returns NULL
Length of a fixed-length opaque type	public int getLength()	Returns`-1` if no length was set
Alignment of an opaque type	public int getAlignment()	Returns `-1` if no alignment was set For the alignment codes, see Alignment values on page 167.

Information obtained	Method signature	Additional information
An array of Method objects that have been specified as support UDRs through setSupportUDR()	public Method[] getSupportUDRs()	For details about support UDRs, see the description of setSupportUDR() in Creating an opaque type from existing code on page 170. Returns NULL if no support UDRs were specified
SQL name of a Java™ method that was specified as a support UDR through setSupportUDR()	public String getSupportUDRSQLName (Method *method*) throws SQLException	Returns NULL if no name was set

The getXXX() methods in the UDRMetaData class

To obtain information about UDRs, use the methods in the following table.

Information obtained	Method signature	Additional information
An array of **java.lang.Method.Reflect** methods that have been specified as UDRs for an opaque type.	public Method[] getUDRs()	To specify a UDR for an opaque type, call the UDTMetaData.setUDR() method. Returns NULL if no UDRs were specified
SQL name of a Java™ method	public String getUDRSQLName(Method *method*) throws SQLException	Returns NULL if no SQL name was specified for the UDR Method object

Execute in a transaction

If your database is ANSI or has logging enabled, and the application is not already in a transaction, the driver executes the SQL statements to create opaque types and UDRs on the server within a transaction. This means that either all the steps will succeed, or all will fail. If the opaque type or UDR creation fails at any point, the driver rolls back the transaction and throws an SQLException.

If the application is already in a transaction when the UDTManager.createUDT() or UDRManager.createUDRs() method calls are issued, the SQL statements are executed within the existing transaction. This means that if the driver returns an SQLException to your application during the creation of the opaque type or UDR, your application must roll back the transaction to ensure the integrity of the database. Otherwise, the opaque type, parts of its casts, or UDRs could be left in the database.

Examples

The rest of this section contains examples for creating and using opaque types and UDRs.

The first four examples are released with your JDBC driver software in the `demo/udt-distinct` directory; the last two are in the `demo/tools/udtudrmgr` directory. See the `README` file in each directory for a description of the files.

Class definition

The class for the C opaque type, **charattrUDT** in the following example, must implement the **SQLData** interface:

```
import java.sql.*;
import com.informix.jdbc.*;
/*
 * C struct of charattr_udt:
 *
 * typedef struct charattr_type
 * {
 *     char          chr1[4+1];
 *     mi_boolean    bold;       // mi_boolean (1 byte)
 *     mi_smallint   fontsize;   // mi_smallint (2 bytes)
 * }
 * charattr;
 *
 * typedef charattr charattr_udt;
 *
 */
public class charattrUDT implements SQLData
{
        private String sql_type = "charattr_udt";
        // an ASCII character/a multibyte character, and is null-terminated.
        public String chr1;
        // Is the character in boldface?
        public boolean bold;
        // font size of the character
        public short fontsize;

   public charattrUDT() { }

   public charattrUDT(String chr1, boolean bold, short fontsize)
   {
        this.chr1 = chr1;
        this.bold = bold;
        this.fontsize = fontsize;
   }

    public String getSQLTypeName()
   {
                return sql_type;
   }
   // reads a stream of data values and builds a Java object
   public void readSQL(SQLInput stream, String type) throws SQLException
   {
        sql_type = type;
        chr1 = ((IfmxUDTSQLInput)stream).readString(5);
        bold = stream.readBoolean();
        fontsize = stream.readShort();
   }
   // writes a sequence of values from a Java object to a stream
   public void writeSQL(SQLOutput stream) throws SQLException
   {
```

```
                ((IfmxUDTSQLOutput)stream).writeString(chr1, 5);
                stream.writeBoolean(bold);
                stream.writeShort(fontsize);
   }
   // overides Object.equals()
   public boolean equals(Object b)
   {
                return (chr1.equals(((charattrUDT)b).chr1) &&
                bold == ((charattrUDT)b).bold &&
                fontsize == ((charattrUDT)b).fontsize);
   }

   public String toString()
   {
                return "chr1=" + chr1 + " bold=" + bold + " fontsize=" + fontsize;
   }
}
```

In your JDBC application, a custom type map must map the SQL-type name **charattr_udt** to the **charattrUDT** class:

```
java.util.Map customtypemap = conn.getTypeMap();
if (customtypemap == null)
        {
        System.out.println("\n***ERROR: typemap is null!");
        return;
        }
customtypemap.put("charattr_udt", Class.forName("charattrUDT"));
```

Insert data

You can insert an opaque type as either its original type or its cast type. The following example shows how to insert opaque data using the original type:

```
String s = "insert into charattr_tab (int_col, charattr_col)
   values (?, ?)";
System.out.println(s);
pstmt = conn.prepareStatement(s);
...
charattrUDT charattr = new charattrUDT();
charattr.chr1 = "a";
charattr.bold = true;
charattr.fontsize = (short)1;

pstmt.setInt(1, 1);
System.out.println("setInt...ok");

pstmt.setObject(2, charattr);
System.out.println("setObject(charattrUDT)...ok");

pstmt.executeUpdate();
```

If a casting function is defined, and you would like to insert data as the casting type instead of the original type, you must call the setXXX() method that corresponds to the casting type. For example, if you have defined a function casting CHAR or LVARCHAR to a **charattrUDT** column, you can use the setString() method to insert data, as follows:

```
// Insert into UDT column using setString(int,String) and Java
   String object.
```

```
String s =
        "insert into charattr_tab " +
        "(decimal_col, date_col, charattr_col, float_col) " +
        "values (?,?,?,?)";
writeOutputFile(s);
PreparedStatement pstmt = myConn.prepareStatement(s);

...
String strObj = "(A, f, 18)";
pstmt.setString(3, strObj);
...
```

Retrieve data

To retrieve HCL Informix® opaque types, you must use ResultSet.getObject(). HCL Informix® JDBC Driver converts the data to a Java™ object according to the custom type map you provide. Using the previous example of the **charattrUDT** type, you can fetch the opaque data, as in the following example:

```
String s = "select int_col, charattr_col from charattr_tab order by 1";
System.out.println(s);

Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(s);
System.out.println("execute...ok");

System.out.println("Fetching data ...");
int curRow = 0;
while (rs.next())
        {
        curRow++;
        System.out.println("currentrow=" + curRow + " : ");

        int intret = rs.getInt("int_col");
        System.out.println("   int_col       " + intret);

        charattrUDT charattrret = (charattrUDT)rs.getObject("charattr_col");
        System.out.print("   charattr_col  ");
        if (curRow == 2 || curRow == 6)
                {
                        if (rs.wasNull())
                                System.out.println("<null>");
                        else
                                System.out.println("***ERROR: " + charattrret);
                }
                else
                System.out.println(charattrret+"");
        } //while

System.out.println("total rows expected: " + curRow);
stmt.close();
```

Smart large objects within an opaque type

A smart large object can be a data member within an opaque type, although you are most likely to create a large object on the database server, outside of the opaque type context, using the HCL Informix® extension classes.

For more information about smart large objects, see Smart large object data types on page 125.

A large object is stored as an **IfxLocator** object within the opaque type; in the C struct that defines the opaque type internally, the large object is referenced through a locator pointer of type MI_LO_HANDLE. The object is created using the methods provided in the **IfxSmartBlob** class, and the large object handle obtained from these methods becomes the data member within the opaque type. Both BLOB and CLOB objects use the same large object handle, as shown in the following example:

```
import java.sql.*;
import com.informix.jdbc.*;
/*
 * C struct of large_bin_udt:
 *
 * typedef struct LARGE_BIN_TYPE
 * {
 *     MI_LO_HANDLE lb_handle;   // handle to large object (72 bytes)
 * }
 * large_bin_udt;
 *
 */
public class largebinUDT implements SQLData
{
    private String sql_type = "large_bin_udt";
    public Clob lb_handle;

    public largebinUDT() { }

    public largebinUDT(Clob clob)
        {
                lb_handle = clob;
        }

    public String getSQLTypeName()
        {
                return sql_type;
        }
    // reads a stream of data values and builds a Java object
    public void readSQL(SQLInput stream, String type) throws SQLException
        {
                sql_type = type;
                lb_handle = stream.readClob();
        }
        // writes a sequence of values from a Java object to a stream
    public void writeSQL(SQLOutput stream) throws SQLException
        {
                stream.writeClob(lb_handle);
        }
}
```

In a JDBC application, you create the MI_LO_HANDLE object using the methods provided by the **IfxSmartBlob** class:

```
String s = "insert into largebin_tab (int_col, largebin_col, lvc_col) " +
        "values (?,?,?)";
System.out.println(s);
pstmt = conn.prepareStatement(s);

...
// create a large object using IfxSmartBlob's methods
```

```
String filename = "lbin_in1.dat";
File file = new File(filename);
int fileLength = (int) file.length();
FileInputStream fin = new FileInputStream(file);

IfxLobDescriptor loDesc = new IfxLobDescriptor(conn);
System.out.println("create large object descriptor...ok");

IfxLocator loPtr = new IfxLocator();
IfxSmartBlob smb = new IfxSmartBlob((IfxConnection)conn);
int loFd = smb.IfxLoCreate(loDesc, 8, loPtr);
System.out.println("create large object...ok");

int n = smb.IfxLoWrite(loFd, fin, fileLength);
System.out.println("write file content into large object...ok");

pstmt.setInt(1, 1);
System.out.println("setInt...ok");

// initialize largebin object using the large object created
// above, before doing setObject for the large_bin_udt column.
largebinUDT largebinObj = new largebinUDT();
largebinObj.lb_handle = new IfxCblob(loPtr);
pstmt.setObject(2, largebinObj);
System.out.println("setObject(largebinUDT)...ok");

pstmt.setString(3, "Sydney");
System.out.println("setString...ok");

pstmt.executeUpdate();
System.out.println("execute...ok");

// close/release large object
smb.IfxLoClose(loFd);
System.out.println("close large object...ok");
smb.IfxLoRelease(loPtr);
System.out.println("release large object...ok");
```

See Smart large object data types on page 125 for details.

Create an opaque type from an existing Java™ class with UDTManager

The following example shows how an application can use the **UDTManager** and **UDTMetaData** classes to convert an existing Java™ class on the client (inaccessible to the database server) to an SQL opaque type in the database server.

Create an opaque type using default support functions

The following example creates an opaque type named **Circle**, using an existing Java™ class and using the default support functions provided in the database server:

```
*/

import java.sql.*;
import com.informix.jdbc.IfmxUDTSQLInput;
import com.informix.jdbc.IfmxUDTSQLOutput;
```

```
public class Circle implements SQLData
{
    private static double PI = 3.14159;

    double x;           // x coordinate
    double y;           // y coordinate
    double radius;

    private String type = "circle";

    public String getSQLTypeName() { return type; }

    public void readSQL(SQLInput stream, String typeName)
        throws SQLException
    {
        // To be able to use the DEFAULT support functions supplied
        // by the server, you must cast the stream to IfmxUDTSQLInput.
        // (Server requirement)

        IfmxUDTSQLInput in = (IfmxUDTSQLInput) stream;
        x = in.readDouble();
        y = in.readDouble();
        radius = in.readDouble();
    }

    public void writeSQL(SQLOutput stream) throws SQLException
    {
        // To be able to use the DEFAULT support functions supplied
        // by the server, have to cast the stream to IfmxUDTSQLOutput.
        // (Server requirement)

        IfmxUDTSQLOutput out = (IfmxUDTSQLOutput) stream;
        out.writeDouble(x);
        out.writeDouble(y);
        out.writeDouble(radius);
    }

    public static double area(Circle c)
    {
        return PI * c.radius * c.radius;
    }

}
```

The opaque type

The following JDBC client application installs the class **Circle** (which is packaged in `Circle.jar`) as an opaque type in the system catalog. Applications can then use the opaque type **Circle** as a data type in SQL statements:

```
import java.sql.*;
import java.lang.reflect.*;


public class PlayWithCircle
{
```

```
String dbname = "test";
String url = null;
Connection conn = null;

public static void main (String args[])
{
    new PlayWithCircle(args);
}

PlayWithCircle(String args[])
{
    System.out.println("----------------");
    System.out.println("- Start - Demo 1");
    System.out.println("----------------");

    // -----------
    // Getting URL
    // -----------
    if (args.length == 0)
        {
        System.out.println("\n***ERROR: connection URL must be provided " +
                           "in order to run the demo!");
        return;
        }
    url = args[0];

    // --------------
    // Loading driver
    // --------------
    try
        {
        System.out.print("Loading JDBC driver...");
        Class.forName("com.informix.jdbc.IfxDriver");
        System.out.println("ok");
        }
    catch (java.lang.ClassNotFoundException e)
        {
        System.out.println("\n***ERROR: " + e.getMessage());
        e.printStackTrace();
        return;
        }

    // ------------------
    // Getting connection
    // ------------------
    try
        {
        System.out.print("Getting connection...");
        conn = DriverManager.getConnection(url);
        System.out.println("ok");
        }
    catch (SQLException e)
        {
        System.out.println("URL = '" + url + "'");
        System.out.println("\n***ERROR: " + e.getMessage());
        e.printStackTrace();
        return;
        }
```

```
        System.out.println();

        // -------------------
        // Setup UDT meta data
        // -------------------
        Method areamethod = null;
        try
            {
            Class c = Class.forName("Circle");
            areamethod = c.getMethod("area", new Class[] {c});
            }
        catch (ClassNotFoundException e)
            {
            System.out.println("Cannot get Class: " + e.toString());
            return;
            }
        catch (NoSuchMethodException e)
            {
            System.out.println("Cannot get Method: " + e.toString());
            return;
            }

        UDTMetaData mdata = null;
        try
            {
            System.out.print("Setting mdata...");
            mdata = new UDTMetaData();
            mdata.setSQLName("circle");
            mdata.setLength(24);
            mdata.setAlignment(UDTMetaData.EIGHT_BYTE);
            mdata.setUDR(areamethod, "area");
            mdata.setJarFileSQLName("circle_jar");
            System.out.println("ok");
            }
        catch (SQLException e)
            {
            System.out.println("\n***ERROR: " + e.getMessage());
            return;
            }

        // -------------------------------
        // Install the UDT in the database
        // -------------------------------
        UDTManager udtmgr = null;
        try
            {
            udtmgr = new UDTManager(conn);

            System.out.println("\ncreateJar()");
            String jarfilename = udtmgr.createJar(mdata,
                new String[] {"Circle.class"}); // jarfilename = circle.jar
            System.out.println("   jarfilename = " + jarfilename);

            System.out.println("\nsetJarTmpPath()");
            udtmgr.setJarTmpPath("/tmp");

            System.out.print("\ncreateUDT()...");
            udtmgr.createUDT(mdata,
```

```
            "/vobs/jdbc/demo/tools/udtudrmgr/" + jarfilename, "Circle", 0);
            System.out.println("ok");
            }
        catch (SQLException e)
            {
            System.out.println("\n***ERROR: " + e.getMessage());
            return;
            }
        System.out.println();

        // ---------------
        // Now use the UDT
        // ---------------
        try
        {
            String s = "drop table tab";
            System.out.print(s + "...");
            Statement stmt = conn.createStatement();
            int count = stmt.executeUpdate(s);
            stmt.close();
            System.out.println("ok");
        }
        catch ( SQLException e)
        {
            // -206 The specified table (%s) is not in the database.
            if (e.getErrorCode() != -206)
                {
                System.out.println("\n***ERROR: " + e.getMessage());
                return;
                }
            System.out.println("ok");
        }

        executeUpdate("create table tab (c circle)");

        // test DEFAULT Input function
        executeUpdate("insert into tab values ('10 10 10')");

        // test DEFAULT Output function
        try
            {
            String s = "select c::lvarchar from tab";
            System.out.println(s);
            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery(s);
            if (rs.next())
                {
                String c = rs.getString(1);
                System.out.println("   circle = '" + c + "'");
                }
            rs.close();
            stmt.close();
            }
        catch (SQLException e)
            {
            System.out.println("***ERROR: " + e.getMessage());
            }
        System.out.println();
```

```
// test DEFAULT Send function
try
    {
    // setup type map before using getObject() for UDT data.
    java.util.Map customtypemap = conn.getTypeMap();
    System.out.println("getTypeMap...ok");
    if (customtypemap == null)
        {
        System.out.println("***ERROR: map is null!");
        return;
        }
    customtypemap.put("circle", Class.forName("Circle"));
    System.out.println("put...ok");

    String s = "select c from tab";
    System.out.println(s);
    Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery(s);
    if (rs.next())
        {
        Circle c = (Circle)rs.getObject(1, customtypemap);
        System.out.println("   c.x = " + c.x);
        System.out.println("   c.y = " + c.y);
        System.out.println("   c.radius = " + c.radius);
        }
    rs.close();
    stmt.close();
    }
catch (SQLException e)
    {
    System.out.println("***ERROR: " + e.getMessage());
    }
catch (ClassNotFoundException e)
    {
    System.out.println("***ERROR: " + e.getMessage());
    }
System.out.println();

// test user's non-support UDR
try
    {
    String s = "select area(c) from tab";
    System.out.println(s);
    Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery(s);
    if (rs.next())
        {
        double a = rs.getDouble(1);
        System.out.println("   area = " + a);
        }
    rs.close();
    stmt.close();
    }
catch (SQLException e)
    {
    System.out.println("***ERROR: " + e.getMessage());
    }
```

```
        System.out.println();

        executeUpdate("drop table tab");

        // ------------------
        // Closing connection
        // ------------------
        try
            {
            System.out.print("Closing connection...");
            conn.close();
            System.out.println("ok");
            }
        catch (SQLException e)
            {
            System.out.println("\n***ERROR: " + e.getMessage());
            }
}
```

Create an opaque type using support functions you supply

In this example, the Java™ class **Circle2** on the client is mapped to an SQL opaque type named **circle2**. The **circle2** opaque type uses support functions provided by the programmer.

```
import java.sql.*;
import java.text.*;
import com.informix.jdbc.IfmxUDTSQLInput;
import com.informix.jdbc.IfmxUDTSQLOutput;

public class Circle2 implements SQLData
{
    private static double PI = 3.14159;

    double x;           // x coordinate
    double y;           // y coordinate
    double radius;

    private String type = "circle2";

    public String getSQLTypeName() { return type; }

    public void readSQL(SQLInput stream, String typeName)
        throws SQLException
    {
/* commented out - because the first release of the UDT/UDR Manager feature
 *                 does not support mixing user-supplied support functions
 *                 with server DEFAULT support functions.
 * However, once the mix is supported, this code needs to be used to
 * replace the existing code.
 *
        // To be able to use the DEFAULT support functions (other than
        // Input/Output) supplied by the server, you must cast the stream
        // to IfmxUDTSQLInput.

        IfmxUDTSQLInput in = (IfmxUDTSQLInput) stream;
        x = in.readDouble();
        y = in.readDouble();
```

```
        radius = in.readDouble();
 */

        x = stream.readDouble();
        y = stream.readDouble();
        radius = stream.readDouble();
    }

    public void writeSQL(SQLOutput stream) throws SQLException
    {
/* commented out - because the 1st release of UDT/UDR Manager feature
 *                 doesn't support the mixing of user support functions
 *                 with server DEFAULT support functions.
 * However, once the mix is supported, this code needs to be used to
 * replace the existing code.
 *
        // To be able to use the DEFAULT support functions (other than
        // Input/Output) supplied by the server, you must cast the stream
        // to IfmxUDTSQLOutput.

        IfmxUDTSQLOutput out = (IfmxUDTSQLOutput) stream;
        out.writeDouble(x);
        out.writeDouble(y);
        out.writeDouble(radius);
 */

        stream.writeDouble(x);
        stream.writeDouble(y);
        stream.writeDouble(radius);
    }

    /**
     * Input function - return the object from the String representation -
     * 'x y radius'.
     */
    public static Circle2 fromString(String text)
    {
        Number a = null;
        Number b = null;
        Number r = null;

        try
            {
            ParsePosition ps = new ParsePosition(0);
            a = NumberFormat.getInstance().parse(text, ps);
            ps.setIndex(ps.getIndex() + 1);
            b = NumberFormat.getInstance().parse(text, ps);
            ps.setIndex(ps.getIndex() + 1);
            r = NumberFormat.getInstance().parse(text, ps);
            }
        catch (Exception e)
            {
            System.out.println("In exception : " + e.getMessage());
            }

        Circle2 c = new Circle2();
        c.x = a.doubleValue();
        c.y = b.doubleValue();
```

```
        c.radius = r.doubleValue();

        return c;
    }

    /**
     * Output function - return the string of the form 'x y radius'.
     */
    public static String makeString(Circle2 c)
    {
        StringBuffer sbuff = new StringBuffer();
        FieldPosition fp = new FieldPosition(NumberFormat.INTEGER_FIELD);
        NumberFormat.getInstance().format(c.x, sbuff, fp);
        sbuff.append(" ");
        NumberFormat.getInstance().format(c.y, sbuff, fp);
        sbuff.append(" ");
        NumberFormat.getInstance().format(c.radius, sbuff, fp);

        return sbuff.toString();
    }

    /**
     * user function - get the area of a circle.
     */
    public static double area(Circle2 c)
    {
        return PI * c.radius * c.radius;
    }

}
```

The opaque type

The following JDBC client application installs the class **Circle2** (which is packaged in `Circle2.jar`) as an opaque type in the system catalog. Applications can then use the opaque type **Circle2** as a data type in SQL statements:

```
import java.sql.*;
import java.lang.reflect.*;


public class PlayWithCircle2
{
    String dbname = "test";
    String url = null;
    Connection conn = null;

    public static void main (String args[])
    {
        new PlayWithCircle2(args);
    }

    PlayWithCircle2(String args[])
    {

        // -----------
        // Getting URL
```

```
        // -----------
        if (args.length == 0)
            {
            System.out.println("\n***ERROR: connection URL must be provided " +
                               "in order to run the demo!");
            return;
            }

        url = args[0];

        // --------------
        // Loading driver
        // --------------
        try
            {
            System.out.print("Loading JDBC driver...");
            Class.forName("com.informix.jdbc.IfxDriver");
            }
        catch (java.lang.ClassNotFoundException e)
            {
            System.out.println("\n***ERROR: " + e.getMessage());
            e.printStackTrace();
            return;
            }



        try
            {
            conn = DriverManager.getConnection(url);
            }
        catch (SQLException e)
            {
            System.out.println("URL = '" + url + "'");
            System.out.println("\n***ERROR: " + e.getMessage());
            e.printStackTrace();
            return;
            }
        System.out.println();
```

Create an opaque type without an existing Java™ class

In this example, the Java™ class **MyCircle** on the client is used to create a fixed-length opaque type in the database server named **ACircle**. The **ACircle** opaque type uses the default support functions provided by the database server:

```
import java.sql.*;

public class MyCircle
{
    String dbname = "test";
    String url = null;
    Connection conn = null;

    public static void main (String args[])
    {
        new MyCircle(args);
    }
```

```
    MyCircle(String args[])
    {
        System.out.println("----------------");
        System.out.println("- Start - Demo 3");
        System.out.println("----------------");

        // -----------
        // Getting URL
        // -----------
        if (args.length == 0)
            {
            System.out.println("\n***ERROR: connection URL must be provided " +
                               "in order to run the demo!");
            return;
            }
        url = args[0];

        // --------------
        // Loading driver
        // --------------
        try
            {
            System.out.print("Loading JDBC driver...");
            Class.forName("com.informix.jdbc.IfxDriver");
            System.out.println("ok");
            }
        catch (java.lang.ClassNotFoundException e)
            {
            System.out.println("\n***ERROR: " + e.getMessage());
            e.printStackTrace();
            return;
            }

        // ------------------
        // Getting connection
        // ------------------
        try
            {
            System.out.print("Getting connection...");
            conn = DriverManager.getConnection(url);
            System.out.println("ok");
            }
        catch (SQLException e)
            {
            System.out.println("URL = '" + url + "'");
            System.out.println("\n***ERROR: " + e.getMessage());
            e.printStackTrace();
            return;
            }
         // --------------------
        // Setup UDT meta data
        // -------------------
        UDTMetaData mdata = null;
        try
            {
            mdata = new UDTMetaData();
            System.out.print("Setting fields in mdata...");
            mdata.setSQLName("acircle");
```

```
            mdata.setLength(24);
            mdata.setFieldCount(3);
            mdata.setFieldName(1, "x");
            mdata.setFieldName(2, "y");
            mdata.setFieldName(3, "radius");
            mdata.setFieldType(1, com.informix.lang.IfxTypes.IFX_TYPE_INT);
            mdata.setFieldType(2, com.informix.lang.IfxTypes.IFX_TYPE_INT);
            mdata.setFieldType(3, com.informix.lang.IfxTypes.IFX_TYPE_INT);
            // set class name if don't want to use the default name
            // <udtsqlname>.class
            mdata.setClassName("ACircle");
            mdata.setJarFileSQLName("ACircleJar");
            mdata.keepJavaFile(true);
            System.out.println("ok");
            }
        catch (SQLException e)
            {
            System.out.println("***ERROR: " + e.getMessage());
            return;
            }

        // ---------------------------------------------------------
        // create java file for UDT and install UDT in the database
        // ---------------------------------------------------------
        UDTManager udtmgr = null;
        try
            {
            udtmgr = new UDTManager(conn);

            System.out.println("Creating .class/.java files - " +
                               "createUDTClass()");
            String classname = udtmgr.createUDTClass(mdata); // generated
                               //java file is kept
            System.out.println("   classname = " + classname);

            System.out.println("\nCreating .jar file - createJar()");
            String jarfilename = udtmgr.createJar(mdata,
                new String[]{"ACircle.class"}); // jarfilename is
                                                // <udtsqlname>.jar
                                                // ie. acircle.jar

            System.out.println("\nsetJarTmpPath()");
            udtmgr.setJarTmpPath("/tmp");

            System.out.print("\ncreateUDT()...");
            udtmgr.createUDT(mdata,
                "/vobs/jdbc/demo/tools/udtudrmgr/" + jarfilename, "ACircle", 0);
            System.out.println("ok");
            }
        catch (SQLException e)
            {
            System.out.println("\n***ERROR: " + e.getMessage());
            return;
            }
        System.out.println();


        // ---------------
```

```
        // Now use the UDT
        // ---------------
        try
        {
            String s = "drop table tab";
            System.out.print(s + "...");
            Statement stmt = conn.createStatement();
            int count = stmt.executeUpdate(s);
            stmt.close();
            System.out.println("ok");
        }
        catch ( SQLException e)
        {
            // -206 The specified table (%s) is not in the database.
            if (e.getErrorCode() != -206)
                {
                System.out.println("\n***ERROR: " + e.getMessage());
                return;
                }
            System.out.println("ok");
        }

        executeUpdate("create table tab (c acircle)");

        // test DEFAULT Input function
        executeUpdate("insert into tab values ('10 10 10')");

        // test DEFAULT Output function
        try
            {
            String s = "select c::lvarchar from tab";
            System.out.println(s);
            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery(s);
            if (rs.next())
                {
                String c = rs.getString(1);
                System.out.println("   acircle = '" + c + "'");
                }
            rs.close();
            stmt.close();
            }
        catch (SQLException e)
            {
            System.out.println("***ERROR: " + e.getMessage());
            }
        System.out.println();

        executeUpdate("drop table tab");

        // ------------------
        // Closing connection
        // ------------------
        try
            {
            System.out.print("Closing connection...");
            conn.close();
            System.out.println("ok");
```

```
            }
        catch (SQLException e)
            {
            System.out.println("\n***ERROR: " + e.getMessage());
            }

        System.out.println("------------------");
        System.out.println("- End - UDT Demo 3");
        System.out.println("------------------");

   }
```

Create UDRs with UDRManager

The following code shows how an application can use the **UDRManager** and **UDRMetaData** classes to convert methods in a Java™ class on the client (inaccessible to the database server) to Java™ UDRs in the database server. Applications can later reference the UDRs in SQL statements. In this example, the Java™ class on the client is named **Group1**. The class has two routines, **udr1** and **udr2**.

The following code creates methods in the **Group1** class to be registered as UDRs in the database server:

```
import java.sql.*;

public class Group1
{
   public static String udr1 (String s1, String s2)
        throws SQLException
   {
   return s1 + s2;
   }
    // Return a formatted string with all inputs
   public static String udr2 (Integer i, String s1,
                  String s2) throws SQLException
   {
   return "{" + i + "," + s1 + "," + s2 +"}";
   }
}
```

The following code creates Java™ methods **udr1** and **udr2** as UDRs **group1_udr1** and **group1_udr2** in the database server and then uses the UDRs:

```
import java.sql.*;
import java.lang.reflect.*;

public class PlayWithGroup1
{
// Open a connection...
url = "jdbc:informix-sqli://hostname:portnum:db/:
   informixserver=servname;user=scott;password=tiger;
myConn = DriverManager.getConnection(url);

//Install the routines in the database.
UDRManager udtmgr = new UDRManager(myConn);
UDRMetaData mdata = new UDRMetaData();
Class gp1 = Class.forName("Group1");
Method method1 = gp1.getMethod("udr1",
```

```
   new Class[]{String.class, String.class});
Method method2 = gp1.getMethod("udr2",
   new Class[]{Integer.class, String.class, String.class});
mdata.setUDR(method1, "group1_udr1");
mdata.setUDR(method2, "group1_udr2");
mdata.setJarFileSQLName("group1_jar");
udtmgr.createUDRs(mdata, "Group1.jar", "Group1", 0);

// Use the UDRs in SQL statements:
Statement stmt = myConn.createStatement();
stmt.executeUpdate("create table tab (c1 varchar(10),
   c2 char(20)", c3 int);
stmt.close();
Statement stmt = myConn.createStatement();
stmt.executeUpdate("insert into tab values ('hello', 'world',
   222)");
stmt.close();

Statement stmt = myConn.createStatement();
ResultSet r = stmt.executeQuery("select c3, group1_udr2(c3, c1, c2)
   from tab where group1_udr1(c1, c2) = 'hello world'");

...

}
```

Globalization and date formats

HCL Informix® JDBC Driver extends the Java™ globalization features by providing access to Informix® databases that are based on different locales and code sets.

Globalization allows you to develop software independently of the countries or languages of its users and then to localize your software for multiple countries or regions.

For general information about setting up Global Language Support (GLS), see the *HCL® Informix® GLS User's Guide*.

Support for Java™ and globalization

The Java™ development kit provides a rich set of APIs for developing global applications. These globalization APIs are based on the Unicode 2.0 code set and can adapt text, numbers, dates, currency, and user-defined objects to any country conventions.

The globalization APIs are concentrated in three packages:

- The `java.text` package contains classes and interfaces for handling text in a locale-sensitive way.
- The `java.io` package contains new classes for importing and exporting non-Unicode character data.
- The `java.util` package contains the **Locale** class, the globalization support classes, and new classes for date and time handling.

Important: There is no connection between Java™ development kit locales and code sets; you must keep these code sets in agreement. For example, if you select the Japanese locale **ja_JP**, there is no Java™ method that tells you that the SJIS code set is the most appropriate.

Support for HCL Informix® GLS variables

Globalization adds several environment variables to HCL Informix® JDBC Driver, which are summarized in the following table.

Supported Informix® environment variables	Description
CLIENT_LOCALE	Specifies the locale of the client that is accessing the database. Provides defaults for user-defined formats such as the **GL_DATE** format. User-defined data types can use it for code-set conversion. Together with the **DB_LOCALE** variable, the database server uses this variable to establish the server processing locale. The **DB_LOCALE** and **CLIENT_LOCALE** values must be the same, or their code sets must be convertible.
DBCENTURY	Enables you to specify the appropriate expansion for one- or two-digit year DATE values
DBDATE	Specifies the end-user formats of values in DATE columns. Supported for compatibility with earlier versions; **GL_DATE** is preferred.
DB_LOCALE	Specifies the locale of the database. HCL Informix® JDBC Driver uses this variable to perform code-set conversion between Unicode and the database locale. Together with the **CLIENT_LOCALE** variable, the database server uses this variable to establish the server processing locale. The **DB_LOCALE** and **CLIENT_LOCALE** values must be the same, or their code sets must be convertible.
GL_DATE	Specifies the end-user formats of values in DATE columns
GL_USEGLU	To enable Unicode collation by Java/JDBC client applications with the International Components for Unicode (ICU), specify GL_USEGLU=1 in the connection string before connecting to the Informix® instance. This enables the server to use advanced Unicode converters that are required to work with Java™. The GL_USEGLU environment variable must be set to a value of 1 (one) in the database server environment before the server is started, and before the database is created.
NEWCODESET	Allows new code sets to be defined between releases of HCL Informix® JDBC Driver.
NEWLOCALE	Allows new locales to be defined between releases of HCL Informix® JDBC Driver.

The HCL Informix® JDBC Driver does not change the decimal format, even if there is a **CLIENT_LOCALE** setting available. Globalization should be done within the Java™ application with the **DecimalFormat** class.

Important: The **DB_LOCALE**, **CLIENT_LOCALE**, and **GL_DATE** variables are supported only if the database server supports the HCL® Informix® GLS feature.

Support for DATE end-user formats

The end-user format is the format in which a DATE value appears in a string variable. This section describes the **GL_DATE**, **DBDATE**, and **DBCENTURY** variables, which specify DATE end-user formats. These variables are optional.

Important: HCL Informix® JDBC Driver does not support ALS 6.0, 5.0, or 4.0 formats for the **DBDATE** or **GL_DATE** environment variables.

For more information about **GL_DATE**, see *HCL® Informix® GLS User's Guide*.

The GL_DATE variable

The **GL_DATE** environment variable specifies the end-user formats of values in DATE columns. A **GL_DATE** format string can contain the following characters:

- One or more white space characters
- An ordinary character (other than the percent symbol (%) or a white space character)
- A formatting directive, which is composed of the percent symbol (%) followed by one or two conversion characters that specify the required replacement

Date formatting directives are defined in the following table.

Directive	Replaced by
%a	The abbreviated weekday name as defined in the locale
%A	The full weekday name as defined in the locale
%b	The abbreviated month name as defined in the locale
%B	The full month name as defined in the locale
%C	The century number (the year divided by 100 and truncated to an integer) as a decimal number (00 through 99)
%d	The day of the month as a decimal number (01 through 31) A single digit is preceded by a zero (0).
%D	Same as the %m/%d/%y format
%e	The day of the month as a decimal number (1 through 31) A single digit is preceded by a space.

Directive	Replaced by
%h	Same as the %b formatting directive
%iy	The year as a two-digit decade (00 through 99) It is the formatting directive that is specific to Informix® for %y.
%iY	The year as a four-digit decade (0000 through 9999) It is the formatting directive that is specific to Informix® for %Y.
%m	The month as a decimal number (01 through 12)
%n	A `newline` character
%t	The `TAB` character
%w	The weekday as a decimal number (0 - 6) The 0 represents the locale equivalent of Sunday.
%x	A special date representation that the locale defines
%y	The year as a two-digit decade (00 - 99)
%Y	The year as a four-digit decade (0000 - 9999)
%%	% (to allow % in the format string)

Important: GL_DATE optional date format qualifiers for field specifications are not supported.

For example, by using *%4m* to display a month as a decimal number with a maximum field width of 4 is not supported.

The **GL_DATE** conversion modifier O, which indicates use of alternative digits for alternative date formats, is not supported.

White space or other nonalphanumeric characters must appear between any two formatting directives. If a **GL_DATE** variable format does not correspond to any of the valid formatting directives, errors can result when the database server attempts to format the date.

For example, for a U.S. English locale, you can format an internal DATE value for 09/29/1998 using the following format:

```
* Sep 29, 1998 this day is:(Tuesday), a fine day *
```

To create this format, set the **GL_DATE** environment variable to this value:

```
* %b %d, %Y this day is:(%A), a fine day *
```

To insert this date value into a database table that has a date column, you can perform the following types of inserts:

- Nonnative SQL, in which SQL statements are sent to the database server unchanged

  Enter the date value exactly as expected by the **GL_DATE** setting.

- Native SQL, in which escape syntax is converted to a format that is specific to Informix®

  Enter the date value in the JDBC escape format `yyyy-mm-dd`; the value is converted to the **GL_DATE** format automatically.

The following example shows both types of inserts:

To retrieve the formatted **GL_DATE** DATE value from the database, call the getString() method of the **ResultSet** class.

To enter strings that represent dates into database table columns of char, varchar, or lvarchar type, you can also build date objects that represent the date string value. The date string value must be in **GL_DATE** format.

The following example shows both ways of selecting DATE values:

```
PreparedStatement pstmt = conn.prepareStatement("Select * from
   tablename "
   + "where col2 like ?;");
pstmt.setString(1, "%Tue%");
ResultSet r = pstmt.executeQuery();
while(r.next())
   {
   String s = r.getString(1);
   java.sql.Date d = r.getDate(2);
   System.out.println("Select: column col1 (GL_DATE format) = <"
      + s + ">");
   System.out.println("Select: column col2 (JDBC Escape format) = <"
      + d + ">");
   }
r.close();
pstmt.close();
```

The DBDATE variable (deprecated)

Support for the **DBDATE** environment variable provides compatibility with earlier versions for client applications that are based on HCL Informix® database server versions before 7.2x, 8.x, or 9.x. Use the **GL_DATE** environment variable for new applications.

The **DBDATE** environment variable specifies the end-user formats of values in DATE columns. End-user formats are used in the following ways:

- When you input DATE values, HCL® Informix® products use the **DBDATE** environment variable to interpret the input. For example, if you specify a literal DATE value in an INSERT statement, Informix® database servers require this literal value to be compatible with the format specified by the **DBDATE** variable.
- When you display DATE values, HCL® Informix® products use the **DBDATE** environment variable to format the output.

With standard formats, you can specify the following attributes:

- The order of the month, day, and year in a date
- Whether the year is printed with two digits (Y2) or four digits (Y4)
- The separator between the month, day, and year

The format string can include the following characters:

- Hyphen (-), dot (.), and slash (/) are separator characters in a date format. A separator appears at the end of a format string (for example `Y4MD-`).
- A 0 indicates that no separator is displayed.
- D and M are characters that represent the day and the month.
- Y2 and Y4 are characters that represent the year and the number of digits in the year.

The following format strings are valid standard **DBDATE** formats:

- DMY2
- DMY4
- MDY4
- MDY2
- Y4MD
- Y4DM
- Y2MD
- Y2DM

The separator always goes at the end of the format string (for example, `DMY2/`). If no separator or an invalid character is specified, the slash (/) character is the default.

For the U.S. ASCII English locale, the default setting for **DBDATE** is `Y4MD-`, where Y4 represents a four-digit year, M represents the month, D represents the day, and hyphen (-) is the separator (for example, 1998-10-08).

To insert a date value into a database table with a date column, you can perform the following types of inserts:

- **Nonnative SQL**. SQL statements are sent to the database server unchanged. Enter the date value exactly as expected by the **DBDATE** setting.
- **Native SQL**. Escape syntax is converted to a format that is specific to Informix®. Enter the date value in the JDBC escape format `yyyy-mm-dd`; the value is converted to the **DBDATE** format automatically.

The following example shows both types of inserts (the **DBDATE** value is `MDY2-`):

```
stmt = conn.createStatement();
cmd = "create table tablename (col1 date, col2 varchar(20));";
rc = stmt.executeUpdate(cmd);..
.String[] dateVals = {"'08-10-98'", "{d '1998-08-11'}" };
String[] charVals = {"'08-10-98'", "'08-11-98'" };
int numRows = dateVals.length;
for (int i = 0; i < numRows; i++)
    {
    cmd = "insert into tablename values(" + dateVals[i] + ", " +
```

```
        charVals[i] + ")";
    rc = stmt.executeUpdate(cmd);
    System.out.println("Insert: column col1 (date) = " + dateVals[i]);
    System.out.println("Insert: column col2 (varchar) = " + charVals[i]);
    }
```

To retrieve the formatted **DBDATE** DATE value from the database, call the **getString** method of the **ResultSet** class.

To enter strings that represent dates into database table columns of char, varchar, or lvarchar type, you can build date objects that represent the date string value. The date string value needs to be in **DBDATE** format.

The following example shows both ways to select DATE values:

```
PreparedStatement pstmt = conn.prepareStatement("Select * from tablename "
    + "where col1 = ?;");
GregorianCalendar gc = new GregorianCalendar(1998, 7, 10);
java.sql.Date dateObj = new java.sql.Date(gc.getTime().getTime());
pstmt.setDate(1, dateObj);
ResultSet r = pstmt.executeQuery();
while(r.next())
    {
    String s = r.getString(1);
    java.sql.Date d = r.getDate(2);
    System.out.println("Select: column col1 (DBDATE format) = <"
        + s + ">");
    System.out.println("Select: column col2 (JDBC Escape format) = <"
        + d + ">");
    }
r.close();
pstmt.close();
```

The DBCENTURY variable

If a **String** value represents a DATE value that has less than a three-digit year and **DBCENTURY** is set, HCL Informix® JDBC Driver converts the **String** value to a DATE value and uses the **DBCENTURY** property to determine the correct four-digit expansion of the year.

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

Method	**Condition**
PreparedStatement.setString(int, String)	The target column is DATE.
PreparedStatement.setObject(int, String)	The target column is DATE.
IfxPreparedStatement.IfxSetObject(String)	The target column is DATE.
ResultSet.getDate(int) ResultSet.getDate(int, Calendar) ResultSet.getDate(String) ResultSet.getDate(String, Calendar)	The source column is a **String** type.

Method	Condition
ResultSet.getTimestamp(int) ResultSet. getTimestamp(int, Calendar) ResultSet.getTimestamp(String) ResultSet.getTimestamp(String, Calendar)	The source column is a **String** type.
ResultSet.updateString(int, String) ResultSet.updateString(String, String)	The target column is DATE.
ResultSet.updateObject(int, String) ResultSet.updateObject(int, String, int) ResultSet.updateObject(String, String) ResultSet.updateObject(String, String, int)	The target column is DATE.

The following table describes the four possible settings for the **DBCENTURY** environment variable.

Setting	Meaning	Description
P	Past	Uses past and present centuries to expand the year value.
F	Future	Uses present and next centuries to expand the year value.
C	Closest	Uses past, present, and next centuries to expand the year value.
R	Present	Uses present century to expand the year value.

See the "Environment Variables" section in the *HCL® Informix® Guide to SQL: Reference* for a discussion of the algorithms used for each setting and examples of each setting.

Here is an example of a URL that sets the **DBCENTURY** value:

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

A URL must not have a line break.

HCL Informix® JDBC Driver always includes four-digit years when it sends **java.sql.Date** and **java.sql.Timestamp** values to the server. Similarly, the server always includes four-digit years when it sends Informix® date values to HCL Informix® JDBC Driver.

For examples of how to use **DBCENTURY** with HCL Informix® JDBC Driver, see the `DBCENTURYSelect.java`, `DBCENTURYSelect2.java`, `DBCENTURYSelect3.java`, `DBCENTURYSelect4.java`, and `DBCENTURYSelect5.java` example programs.

Precedence rules for end-user formats

The precedence rules that define how to determine an end-user format for an internal DATE value are listed here:

- If a **DBDATE** format is specified, this format is used.
- If a **GL_DATE** format is specified, a locale must be determined:
  - If a **CLIENT_LOCALE** value is specified, it is used with the **GL_DATE** format string to display DATE values.
  - If a **DB_LOCALE** value is specified but a **CLIENT_LOCALE** value is not, the **DB_LOCALE** value is compared with the database locale (read from the **systables** table of the user database) to verify that the **DB_LOCALE** value is valid. If the **DB_LOCALE** value is valid, it is used with the **GL_DATE** format string to display DATE values. If the **DB_LOCALE** value is not valid, the database locale is used with the **GL_DATE** format string.
  - If the **CLIENT_LOCALE** or **DB_LOCALE** values are not specified, the database locale is used with the **GL_DATE** format string to display DATE values.
- If a **CLIENT_LOCALE** value is specified, the DATE formats conform to the default formats associated with this locale.
- If a **DB_LOCALE** value is specified but no **CLIENT_LOCALE** value is specified, the **DB_LOCALE** value is compared with the database locale to verify that the **DB_LOCALE** value is valid.

  If the **DB_LOCALE** value is valid, the **DB_LOCALE** default formats are used. If the **DB_LOCALE** value is not valid, the default formats for dates associated with the database locale are used.
- If the **CLIENT_LOCALE** or **DB_LOCALE** values are not specified, all DATE values are formatted in U.S. English format, `Y4MD-`.

Support for code-set conversion

Code-set conversion converts character data from one code set to another. In a client/server environment, character data might need to be converted from one code set to another if the client and database server computers use different code sets to represent the same characters. For detailed information about code-set conversion, see the *HCL® Informix® GLS User's Guide*.

You must specify code-set conversion for the following types of character data:

- SQL data types (char, varchar, nchar, nvarchar)
- SQL statements
- Database objects such as database names, column names, table names, statement identifier names, and cursor names
- Stored procedure text
- Command text
- Environment variables

HCL Informix® JDBC Driver converts character data as it is sent between client and database server. The code set (encoding) used for the conversion is specified in the **systables** catalog for the opened database. You set the **DB_LOCALE** and **CLIENT_LOCALE** values in the connection properties or database URL.

Unicode to database code set

Java™ is Unicode based, so HCL Informix® JDBC Driver converts data between Unicode and the Informix® database code set. The code-set conversion value is extracted from the **DB_LOCALE** value specified at the time the connection is made. If the **DB_LOCALE** value is incorrect, a `Database Locale information mismatch` error occurs.

The **DB_LOCALE** value must be a valid Informix® locale, with a valid Informix® code-set name or number as shown in the compatibility table that follows. The following table maps the supported Java™ development kit encodings to Informix® code sets.

Informix® code set name	Informix® code set number	JDK code set
8859-1	819	8859_1
8859-2	912	8859_2
8859-3	57346	8859_3
8859-4	57347	8859_4
8859-5	915	8859_5
8859-6	1089	8859_6
8859-7	813	8859_7
8859-8	916	8859_8
8859-9	920	8859_9
8859-13	57390	ISO8859_13
ASCII	364	ASCII
sjis-s	932	SJIS
sjis	57350	SJIS
utf8	57372	UTF8
big5	57352	Big5
CP1250	1250	Cp1250
CP1251	1251	Cp1251
CP1252	1252	Cp1252
CP1253	1253	Cp1253
CP1254	1254	Cp1254
CP1255	1255	Cp1255
CP1256	1256	Cp1256
CP1257	1257	Cp1257
cp936	57357	ISO2022CN_GB
cp_949	57356	Cp949
GB18030-2000	5488	GB18030

Informix® code set name	Informix® code set number	JDK code set
KS5601	57356	Cp949
ksc	57356	Cp949
gb	57357	ISO2022CN_GB
GB2312-80	57357	ISO2022CN_GB
GB18030-2000	5488	GB18030
ujis	57351	EUC_JP

You cannot use the Informix® locale with a code set for which there is no JDK-supported encoding. This incorrect usage results in an `Encoding or code set not supported` error message.

The following table shows the supported locales.

Supported Locales				
ar_ae	ar_bh	ar_kw	ar_om	ar_qa
ar_sa	bg_bg	ca_es	cs_cz	da_dk
de_at	de_ch	de_de	el_gr	en_au
en_ca	en_gb	en_ie	en_nz	en_us
es_ar	es_bo	es_cl	es_co	es_cr
es_ec	es_es	es_gt	es_mx	es_pa
es_pe	es_py	es_sv	es_uy	es_ve
et_ee	fi_fi	fr_be	fr_ca	fr_ch
fr_fr	hr_hr	hu_hu	is_is	it_ch
it_it	iw_il	ja_jp	ko_kr	mk_mk
nl_be	nl_nl	no_no	pl_pl	pt_br
pt_pt	ro_ro	ru_ru	sh_yu	sk_sk
sv_se	th_th	tr_tr	uk_ua	zh_cn
zh_tw				

Unicode to client code set

Because the Unicode code set includes all existing code sets, the Java™ virtual machine (JVM) must render the character with the platforms local code set. Inside the Java™ program, you must always use Unicode characters. The JVM on that platform converts input and output between Unicode and the local code set.

For example, you specify button labels in Unicode, and the JVM converts the text to display the label correctly. Similarly, when the getText() method gets user input from a text box, the client program gets the string in Unicode, no matter how the user entered it.

Never read a text file one byte at a time. Always use the InputStreamReader() or OutputStreamWriter() methods to manipulate text files. By default, these methods use the local encoding, but you can specify an encoding in the constructor of the class, as follows:

```
InputStreamReader = new InputStreamReader (in, "SJIS");
```

You and the JVM are responsible for getting external input into the correct Java™ Unicode string. Thereafter, the database locale encoding is used to send the data to and from the database server.

Connect to a database with non-ASCII characters

If you do not specify the database name at connection time, the connection must be opened with the correct **DB_LOCALE** value for the specified database.

If close database and database *dbname* statements are issued, the connection continues to use the original **DB_LOCALE** value to interpret the database name. If the **DB_LOCALE** value of the new database does not match, an error is returned. In this case, the client program must close and reopen the connection with the correct **DB_LOCALE** value for the new database.

If you supply the database name at connection time, the **DB_LOCALE** value must be set to the correct database locale.

You can connect to an NLS database by defining a locale with **NEWCODESET** and **NEWLOCALE** connection properties. For information about their formats, see Connecting with the NEWLOCALE and NEWCODESET Environment Variables on page 209.

Code-set conversion for TEXT and CLOB data types

HCL Informix® JDBC Driver does not automatically convert between code sets for TEXT, BYTE, CLOB, and BLOB data types.

You can convert between code sets for TEXT and CLOB data types in one of the following ways:

- You can automate code-set conversion for TEXT or CLOB data between the client and database locales by using the **IFX_CODESETLOB** environment variable.
- You can convert between code sets for TEXT data by using the getBytes(), getString(), InputStreamReader(), and OutputStreamWriter() methods.

Convert with the IFX_CODESETLOB environment variable

You can automate the following pair of code-set conversions for TEXT and CLOB data types:

- Convert from client locale to database locale before the data is sent to the database server.
- Convert from database locale to client locale before the data is retrieved by the client.

To automate code-set conversion for TEXT and CLOB data types, set the **IFX_CODESETLOB** environment variable in the connection URL. For example: `IFX_CODESETLOB = 4096`. You can also use the following methods of the **IfxDataSource** class to set and get the value of **IFX_CODESETLOB**:

```
public void setIfxIFX_CODESETLOB(int codesetlobFlag);
public int getIfxIFX_CODESETLOB();
```

IFX_CODESETLOB can have the following values:

none

Default

Automatic code-set conversion is not enabled.

0

Automatic code-set conversion takes place in internal temporary files.

> 0

Automatic code-set conversion takes place in the memory of the client computer. The value indicates the number of bytes allocated for the conversion.

If the number of allocated bytes is less than the size of the large object, an error is returned.

To perform conversion in memory, you must specify an amount that is smaller than the memory limits of the client machines and larger than the possible size of any converted large object.

When you are using any of the following java.sql.Clob interface methods or Informix® extensions to the Clob interface, no code-set conversion is performed, even if the **IFX_CODESETLOB** environment variable is set. These methods include:

```
IfxCblob::setAsciiStream(long)
Clob::setAsciiStream(long position, InputStream fin, int length)
```

IFX_CODESETLOB takes effect only for methods from the java.sql.PreparedStatement interface.

However when using any of following java.sql.Clob interface methods or Informix® extensions to Clob interface, Unicode characters are always converted automatically to the database locale code set. Here is a list of those methods:

```
Clob::setCharacterStream(long) throws SQLException
Clob::setString(long, String) throws SQLException
Clob:: setString(long pos, String str, int offset, int len)
IfxCblob::setSubString(long position, String str, int length)
```

Convert with Java™ methods

The Java™ methods getBytes(), getString(), InputStreamReader(), and OutputStreamWriter() take a code-set parameter that converts to and from Unicode and the specified code set.

Here is sample code that shows how to convert a file from the client code set to Unicode and then from Unicode to the database code set:

```
File infile = new File("data_jpn.dat");
File outfile = new File ("data_conv.dat");..
```

```
.pstmt = conn.prepareStatement("insert into t_text values (?)");..
.// Convert data from client encoding to database encoding
System.out.println("Converting data ...\n");
try
    {
    String from = "SJIS";
    String to = "8859_1";
    convert(infile, outfile, from, to);
    }
catch (Exception e)
    {
    System.out.println("Failed to convert file");
    }

System.out.println("Inserting data ...\n");
try
    {
    int fileLength = (int) outfile.length();
    fin = new FileInputStream(outfile);
    pstmt.setAsciiStream(1 , fin, fileLength);
    pstmt.executeUpdate();
    }
catch (Exception e)
    {
    System.out.println("Failed to setAsciiStream");
    }..
.public static void convert(File infile, File outfile, String from, String to)
    throws IOException
    {
    InputStream in = new FileInputStream(infile);
    OutputStream out = new FileOutputStream(outfile);

    Reader r = new BufferedReader( new InputStreamReader( in, from));
    Writer w = new BufferedWriter( new OutputStreamWriter( out, to));

    //Copy characters from input to output. The InputStreamReader converts
    // from the input encoding to Unicode, and the OutputStreamWriter
    // converts from Unicode to the output encoding.  Characters that can
    // not be represented in the output encoding are output as '?'

    char[] buffer = new char[4096];
    int len;
    while ((len = r.read(buffer)) != -1)
        w.write(buffer, 0, len);
    r.close();
    w.flush();
    w.close();
    }
```

When you retrieve data from the database, you can use the same approach to convert the data from the database code set to the client code set.

Code-set conversion for BLOB and BYTE data types

When you use java.sql.PreparedStatement::setCharacterStream() to insert in a CLOB column, Java™ Unicode characters are converted automatically to the database locale code set. If the environment variable **IFX_CODESETLOB** is set, its value

determine whether to perform code set conversion using temporary files or to perform the code set conversion in memory. If **IFX_CODESETLOB** is not set, the **LOBCACHE** environment variable determines whether the code set conversion takes place in temporary files or in memory.

However, you are discouraged from using java.sql.PreparedStatement::setCharacterStream() to insert BLOB or BYTE columns. The JDBC driver cannot insert Java™ characters in a database and consequently attempts code set conversion of the characters. Using java.sql.PreparedStatement::setBinaryStream() is the preferred way to insert BLOB or BYTE columns.

User-defined locales

HCL Informix® JDBC Driver uses the Java™ globalization API to manipulate international data.

The classes and methods in that API take a Java™ development kit locale or encoding as a parameter, but 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 Java™ development kit names. These mappings are kept in internal tables, which are updated periodically.

For example, the Informix® and Java™ development kit names for the ASCII code set are 8859-1 and 8859_1, respectively. HCL Informix® JDBC Driver maps 8859-1 to 8859_1 in its internal tables and uses the appropriate name in the Java™ development kit classes and methods.

Connect with the NEWLOCALE and NEWCODESET environment variables

Because new locales may be created between updates of these tables, two connection properties, **NEWLOCALE** and **NEWCODESET**, let you specify a locale or code set that is not specified in the tables. Here is an example URL using these properties:

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

A URL must be on one line.

The **NEWLOCALE** and **NEWCODESET** properties have the following formats:

```
NEWLOCALE=JDK-locale,Ifx-locale:JDK-locale,Ifx-locale...

NEWCODESET=JDK-encoding,Ifx-codeset,Ifx-codeset-number:JDK-
  encoding, Ifx-codeset,Ifx-codeset-number...
```

There is no limit to the number of locale or code-set mappings you can specify.

You can connect to an NLS database by defining a locale using **NEWCODESET** and **NEWLOCALE** connection properties.

If you specify an incorrect number of parameters or values, you get a `Locale Not Supported` or `Encoding or Code Set Not Supported` message.

If these properties are set in the URL or a **DataSource** 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 internally, but you specify `NEWCODESET=8888,8859-1,819` instead, the new value `8888` is used for the code-set conversion.

Connect with the NEWNLSMAP environment variable

To support connecting to NLS databases, HCL Informix® JDBC Driver maintains a table for mapping NLS locale to the corresponding Java™ development kit locale and code set. Locales and code sets that are not supported in a particular version of the development kit might be supported in later versions of the development kit. Use the **NEWNLSMAP** connection property to specify mappings for an NLS locale that is not specified in the table.

The **NEWNLSMAP** property has the following format:

```
NEWNLSMAP=NLS-locale,JDK-locale,JDK-codeset:NLS-locale,JDK-locale,
JDK-codeset,....
```

Here is an example URL using these properties:

```
jdbc:informix-sqli://myhost:1533:informixserver=myserver;
```

```
user=myname;password=mypasswd;NEWNLSMAP=rumanian,ro_RO,ISO8859_2;
```

There is no limit to the number of mappings you can specify. If you specify an incorrect number of parameters or values, you get a `Locale Not Supported` or `Encoding or Code Set Not Supported` message.

Support for globalized error messages

Message text is usually the text of an **SQLException** object, but can also be an **SQLWarn** object or any other text output from the driver.

There are two requirements to enable globalized message text output, as follows:

- You must add the full path of the `ifxlang.jar` file to the **$CLASSPATH** (UNIX™) or **%CLASSPATH%** (Windows™) environment variable. This JAR file contains globalized versions of all message text supported by HCL Informix® JDBC Driver. Supported languages are English, German, French, Spanish, Russian, Polish, Czech, Slovak, Chinese (simplified and traditional), Korean, and Japanese.
- The **CLIENT_LOCALE** environment variable value must be passed through the property list to the connection object at connection time if you are using a nondefault locale. For more information about **CLIENT_LOCALE** and GLS features in general, see Support for HCL Informix GLS variables on page 196.

Several public classes have constructors that take the current connection object as a parameter so they have access to the **CLIENT_LOCALE** value. If you want access to non-English error messages, you must use the constructors that include the connection object. Otherwise, any error message text from those classes is in English only. Affected public classes are Interval, IntervalYM, IntervalDF, and IfxLocator. For more information about the constructors to use for these classes, see Work with Informix types on page 95.

For an example of how to use the globalized error message support feature, see the `locmsg.java` program, which is included with HCL Informix® JDBC Driver.

Push data feature

Push data feature lets clients register for changes in a dataset using simple SELECT statements and WHERE clauses. Once the server captures data for push data event conditions which evaluates to true for WHERE clause condition, the server pushes committed data to the client, based on registered events. Scaling is achieved by clients not having to poll for data, and not having to parse, prepare, and execute SQL queries. Database servers with parallel architecture - Enterprise Replication log snooper and grouper -- feed the data to all clients by asynchronously reading logical log file changes. This design lets client applications scale linearly without adding significant overhead to the database server or any OLTP applications making changes to the database. Data that is returned to the client is in a developer-friendly JSON format.

Table 10.

Input attribute name	Description
table	Table name to be registered
owner	Table owner
database	Database name
query	SELECT statement including projection list and WHERE clause to register for changes in a data set.
label	User defined string to be returned along with an event document. This attribute is useful to differentiate between events when more than one push-data event is registered within the same session
timeout	The amount of time a client is blocked in the smartblob read API for an event data. The server returns timeout json document when a timeout condition is triggered. Supported range of values are: • -1 to wait forever • >=0 to wait for a specified amount of time in seconds.
commit_time	Returns event data that is committed after the stated transaction commit time.
txnid	A unique 8 byte ID: • Higher order 4 bytes: commit work log ID • Lower order 4 bytes: commit work log position
max_pending_ops	Maximum number of event records to be kept in the pending session .

Table 10.

(continued)

Input attribute name	Description
maxrecs	Maximum number of records to be returned by the smartblob API read call.

Grant replication permission on sysadmin database for the user registering push data events:

```
execute function task('grant admin', ‘user1', 'replication');
```

Register client as a push data session by using the sysadmin task command:

```
execute function informix.task('pushdata open')
```

The above command registers the client as a push-data session, and returns a unique session ID. This ID is needed for reading event documents using the smartblob readAPI.

This command also auto-registers enterprise replication, when it has not been defined earlier.

To internally define enterprise replication automatically, the pushdata open command relies on the existence of at least one storagepool entry to create the dbspace and subspace required for defining enterprise replication. You must create a storagepool entry using the task API.

For example:

```
Execute function task( 'storagepool add', '/informix/storage', '0', '0', '20000', '1' );
```

Registering one or more push data event conditions using the sysadmin task command:

```
execute function informix.task('pushdata register',
 {table:"creditcardtxns",owner:"informix",database:"creditdb",query:"select uid, cardid, carddata from
 creditcardtxns where carddata.Amount::int >= 100",label:"card txn alert"})
```

Registering session-specific attributes, like timeout, using the pushdata register task command:

```
execute function informix.task('pushdata register', { timeout:"60",max_pending_ops:"0",maxrecs:"1"})
```

De-registering one or more registered event conditions using the pushdata deregister command:

```
To de-register one or more event conditions for the given table:
execute function informix.task('pushdata deregister', {table:"usertable",owner:"informix",database:"ycsb"})
```

To de-register all event conditions with the same label attribute tag:

```
execute function informix.task('pushdata deregister', { label:"card txns"})
```

Note: To deregister a specific event condition, either use the label attribute, or specify a query attribute, along with the table, owner and database attributes.

API to read event data:

The client must invoke the smartblob read API to read an event data. Input for the smartblob read API must include:

- The session ID returned from running the pushdata open task command.
- The input buffer pointer
- The input buffer size—this should be at least equal to the sum of the before image size, the after image size, and 1024 bytes. If multiple records are expected from one read call, then the input buffer size should be equal to the sum of the before image size, the after image size, and 1024, multiplied by the number of records.
- The error code pointer.

ESQLC READ API Example:

```
/*
         * Read data into the buffer
         */
        bytesread = ifx_lo_read(sessionid, databuf, bytes_per_read, &loreaderr);
```

Table 11.

Attribute name	Description
operation	Operation type: Insert/Delete/Update
table	Table name
owner	Table owner
database	Database name
label	Optional user-specified data for the event condition
txnid	8 byte unique ID: • higher order 4 bytes: commit work log ID • lower order 4 bytes: commit work log position.
commit_time	Transaction commit time for the event data.
op_num	Increasing sequence number for the event document within a given transaction. If the transaction generates 10 events, then each document returned will have an incremental op_num value, starting from 1 to 10.
rowdata	Row data in JSON document format. Data is returned using the column name as key and the column data as value.
before_rowdata	Before row data for an Update operation.

Table 11.

(continued)

Attribute name	Description
ifx_isTimeout	Document with this attribute is returned with its value set to true if no event gets triggered within the timeout interval that is specified by the client.
ifx_warn_total_skipcount	A warning document with this attribute is returned, containing the cumulative number of events that are discarded, from exceeding the max_pending_ops attribute threshold.

Sample output from the smartblob read API for an Insert operation:

```
{"operation":"insert","table":"creditcardtxns","owner":"informix","database":"creditdb","label":"card txn
 alert",
 "txnid":2250573177224,"commit_time":1488243530,"op_num":1,"rowdata":{"uid":22,"cardid":"6666-6666-6666-6666","
carddata":{"Merchant":"Sams Club","Amount":200,"Date":2017-05-01T10:35:10.000Z } }}
```

Sample output from the smartblob read API for Update operation:

```
{"opertion":"update",table:"creditcardtxns","owner":"informix","database":"creditdb","label":"card txn
 alert","txnid":2250573308360,"commit_time":1488243832,"op_num":1,"rowdata":{uid:21,cardid:"7777-7777-7777-7777
","carddata":{"Merchant":"Sams Club","Amount":200,"Date":"25-Jan-2017
 16:15"} },"before_rowdata":{"uid":21,"cardid":"6666-6666-6666-6666","carddata":{"Merchant":"Sams
 Club","Amount":200,"Date":2017-05-01T10:35:10.000Z  } }}
```

Sample output from the smartblob read API for Delete operation:

```
{"opertion":"delete","table":"creditcardtxns","owner":"informix","database":"creditdb","label":"card txn
 alert","txnid":2250573287760,"commit_time":1488243797,"op_num":1,"rowdata":{"uid":22,"cardid":"6666-6666-6666-
6666","carddata":{"Merchant":"Sams Club","Amount":200,"Date":2017-05-01T13:35:06.000Z } }}
```

Sample output from the smartblob read API for a multi-row buffer, when the maxrecs input attribute is set to greater than 1:

```
{[
{"operation":"Insert","table":"creditcardtxns","owner":"informix","database":"creditdb","label":"card txn
 alert",
 "txnid":2250573309999,"commit_time":1487781325,"op_num":1,"rowdata":{uid:"7","cardid":"6666-6666-6666-6666","c
arddata":{"Merchant":"Sams Club","Amount":200,"Date":2017-05-01T15:10:10.000Z } }},
{"operation":"insert",table:"creditcardtxns","owner":"informix","database":"creditdb","label":"card txn
 alert","txnid":2250573177224,"commit_time":1488243530,"op_num":1,"rowdata":{"uid":22,"cardid":"6666-6666-6666-
6666","carddata":{"Merchant":"Sams Club","Amount":200,"Date":2017-05-01T16:20:10.000Z } }}
]}
```

Using the sample *pushdata* ESQL/C program

You can run the *pushdata* ESQL/C program to safely preview the process of registering event triggers with your Informix server, to retrieve event data in JSON format.

The program file, **pushdata.ec**, can be found in the */demo/cdc* folder of your Informix installation folder.

Detach trigger

Using the detach trigger methods in IfmxThreadedSmartTrigger, you can declare a Smart Trigger to be 'detachable'. A detachable trigger has an unique identifier which allows you to reconnect to the session on the server.

```
/* Detach a trigger */
IfxSmartTrigger push = new IfxSmartTrigger("jdbc-url-here");
push.detachable(true); //Set the trigger as detachable
push.open();
String session1 = push.getDetachableSessionID(); //Get the session id
//Closes the JDBC connection and returns the session ID
//This is the same session id as you get from the call above
Session1 = push.detach();
```

On detaching from the session, you can create a new Smart Trigger object and pass in the session ID.

```
push = new IfxSmartTrigger(jdbc-url-here);
//Assign the session ID before you start the smart trigger
push.sessionID(sessionID);
TestPushCallback callback1 = new TestPushCallback();
push.registerCallback("test-label-pushtest", callback1);
push.start();
//You pick up where you left off, retrieving any messages you missed from the server
```

Tuning and troubleshooting

These topics provides tuning and troubleshooting information for HCL Informix® JDBC Driver.

Debug your JDBC API program

You can set the SQLIDEBUG connection property to generate binary protocol trace. You set the connection property SQLIDEBUG to specify a file. For example:

```
SQLIDEBUG=C:\\tmp\\ifxjdbctrace
```

A new trace file is generated for every connection and is suffixed with a timestamp. If you are using the **IfxDataSource** interface, you can use the IfxDataSource.setIfxSQLIDEBUG (String fname) method. Debug versions of the JDBC jar files are not included in HCL Informix® JDBC Driver, Version 3.00.JC1 and later.

Important: The binary SQLI protocol trace feature (SQLIDEBUG) should only be used when directed by the IBM® technical support representative.

Java Logging

You can enable tracing of the Informix JDBC driver using Java's built in logging mechanism.

Note: Enabling tracing can have a noticeable impact on driver performance. Enable only logging to assist in diagnosing a problem with the driver.

Following example shows logging properties you can set:

```
Set the default logging level for Informix JDBC
com.informix.jdbc.level=FINEST

#Set the logging level for the console output
java.util.logging.ConsoleHandler.level=FINEST
java.util.logging.ConsoleHandler.formatter=com.informix.util.JdbcLogFormatter
com.informix.jdbc.handlers=java.util.logging.ConsoleHandler

#Set the properties for a file based logger
java.util.logging.FileHandler.level     = FINEST
java.util.logging.FileHandler.formatter = com.informix.util.JdbcLegacyLogFormatter
java.util.logging.FileHandler.append    = false
java.util.logging.FileHandler.pattern   = /tmp/jdbc/jdbc.log
```

When choosing the formatter for the log output you can pick between *com.informix.util.JdbcLogFormatter* and *com.informix.util.JdbcLegacyLogFormatter*. The legacy formatter matches the historical format the JDBC driver outputted messages in. The JdbcLogFormatter is a more modern format, which includes machine readable timestamps, so the logs can be read in by logging systems. If you need to manually add the properties, you can use a call as shown below in Java to load the properties for the Java Logging.

```
LogManager.getLogManager().readConfiguration(new FileInputStream("logging.properties"));
```

Manage performance

This section describes issues that might affect the performance of your queries:

- The **FET_BUF_SIZE** and **BIG_FET_BUF_SIZE** environment variables
- Memory management of large objects
- Reducing network traffic
- Using bulk inserts
- Tuning the connection pool.

Manage the fetch buffer size

Use the **FET_BUF_SIZE** and **SRV_FET_BUF_SIZE** environment variables to set the size of the fetch buffer.

When a SELECT statement is sent from a Java™ program to the HCL Informix® database, the returned rows, or *tuples*, are stored in a tuple buffer in HCL Informix® JDBC Driver. The default size of the tuple buffer is the larger of the returned tuple size or 4096 bytes.

You can use the Informix® **FET_BUF_SIZE** environment variable to override the default size of the tuple buffer. **FET_BUF_SIZE** can be set to any positive integer less than or equal to 2 GiB (2147483648). If the **FET_BUF_SIZE** environment variable is set, and its value is larger than the default tuple buffer size, the tuple buffer size is set to the value of **FET_BUF_SIZE**.

Similarly, you can use the **SRV_FET_BUF_SIZE** environment variable to set the fetch buffer size for the local database server to use when it participates in cross-server distributed DML transactions. For 11.70.xC5 and newer versions, the maximum size to which **SRV_FET_BUF_SIZE** can be set is 1048576 (= 1 MiB).

Increasing the size of the tuple buffer can reduce network traffic between your Java™ program and the database, often resulting in better performance of queries. There are times, however, when increasing the size of the tuple buffer can actually degrade the performance of queries. This could happen if your Java™ program has many active connections to a database or if the swap space on your computer is limited. If this is true for your Java™ program or computer, you might not want to use the **FET_BUF_SIZE** or **SRV_FET_BUF_SIZE** environment variables to increase the size of the tuple buffer.

For more information about setting Informix® environment variables, see Connect to the database on page 14. For more information about increasing the fetch buffer size, see the *HCL® Informix® Guide to SQL: Reference*.

Related information

FET_BUF_SIZE environment variable on page

SRV_FET_BUF_SIZE environment variable on page

Manage memory for large objects

Whenever a large object (a BYTE, TEXT, BLOB, or CLOB data type) is fetched from the database server, the data is either cached into memory or stored in a temporary file (if it exceeds the memory buffer). A JDBC applet can cause a security violation if it tries to create a temporary file on the local computer. In this case, the entire large object must be stored in memory.

You can specify how large object data is stored by using an environment variable, **LOBCACHE**, that you include in the connection property list, as follows:

- To set the maximum number of bytes allocated in memory to hold the data, set the **LOBCACHE** value to that number of bytes.

  If the data size exceeds the **LOBCACHE** value, the data is stored in a temporary file. If a security violation occurs during creation of this file, the data is stored in memory.
- To always store the data in a file, set the **LOBCACHE** value to `0`.

  In this case, if a security violation occurs, HCL Informix® JDBC Driver makes no attempt to store the data in memory. This setting is not supported for unsigned applets. For more information, see Using the driver in an applet on page 12.
- To always store the data in memory, set the **LOBCACHE** value to a negative number.

  If the required amount of memory is not available, HCL Informix® JDBC Driver throws the **SQLException** message `Out of Memory`.

If the **LOBCACHE** size is invalid or not defined, the default size is 4096.

You can set the **LOBCACHE** value through the database URL, as follows:

```
URL = jdbc:informix-sqli://158.58.9.37:7110/test:user=guest;
password=iamaguest;informixserver=oltapshm;
lobcache=4096";
```

The preceding example stores the large object in memory if the size is 4096 bytes or fewer. If the large object exceeds 4096 bytes, HCL Informix® JDBC Driver tries to create a temporary file. If a security violation occurs, memory is allocated for the entire large object. If that fails, the driver throws an **SQLException** message.

Here is another example:

```
URL = "jdbc:informix-sqli://icarus:7110/testdb:
   user=guest:passwd=whoknows;informixserver=olserv01;lobcache=0";
```

The preceding example uses a temporary file for storing the fetched large object.

Here is a third example:

```
URL = "jdbc:informix-sqli://icarus:7110/testdb:user=guest:
   passwd=whoknows;informixserver=olserv01;lobcache=-1";
```

The preceding example always uses memory to store the fetched large object.

For programming information about how to use the TEXT and BYTE data types in a Java™ program, see BYTE and TEXT data types on page 98. For programming information about how to use the BLOB and CLOB data types in a Java™ program, see Smart large object data types on page 125.

Reduce network traffic

The two environment variables **OPTOFC** and **IFX_AUTOFREE** can be used to reduce network traffic when you close **Statement** and **ResultSet** objects.

Set **OPTOFC** to `1` to specify that the ResultSet.close() method does not require a network round trip if all the qualifying rows have already been retrieved in the clients tuple buffer. The database server automatically closes the cursor after all the rows have been retrieved.

HCL Informix® JDBC Driver might or might not have additional rows in the clients tuple buffer before the next ResultSet.next() method is called. Therefore, unless HCL Informix® JDBC Driver has received all rows from the database server, the ResultSet.close() method might still require a network round trip when **OPTOFC** is set to 1.

Set **IFX_AUTOFREE** to `1` to specify that the Statement.close() method does not require a network round trip to free the database server cursor resources if the cursor has already been closed in the database server.

You can also use the setAutoFree(boolean flag) and getAutoFree() methods to free database server cursor resources. For more information, see The Auto Free feature on page 87.

The database server automatically frees the cursor resources after the cursor is closed, either explicitly by the ResultSet.close() method or implicitly by the **OPTOFC** environment variable.

When the cursor resources have been freed, the cursor can no longer be referenced.

For examples of how to use the **OPTOFC** and **IFX_AUTOFREE** environment variables, see the `autofree.java` and `optofc.java` demonstration examples described in Sample code files on page 224. In these examples, the variables are set with the Properties.put() method.

For more information about setting Informix® environment variables, see Informix environment variables with the HCL Informix JDBC Driver on page 23.

Bulk inserts

The bulk insert feature improves the performance of single INSERT statements that are executed multiple times with multiple value settings. For more information, see Perform bulk inserts on page 63.

A connection pool

To improve the performance and scalability of your application, you can obtain your connection to the database server through a **DataSource** object that references a **ConnectionPoolDataSource** object. HCL Informix® JDBC Driver provides a Connection Pool Manager as a transparent component of the **ConnectionPoolDataSource** object. The Connection Pool Manager keeps a closed connection in a pool instead of returning the connection to the database server as closed. Whenever a user requests a new connection, the Connection Pool Manager gets the connection from the pool, avoiding the overhead of having the server close and re-open the connection.

Using the **ConnectionPoolDataSource** object can significantly improve performance in cases where your application receives frequent, periodic connection requests.

For complete information about how and why to use a **DataSource** or **ConnectionPoolDataSource** object, see the JDBC 3.0 API.

Important: This feature does not affect IfxXAConnectionPoolDataSource, which operates under the assumption that connection pooling is handled by the transaction manager.

Deploying a ConnectionPoolDataSource object

About this task

In the following steps:

- The variable **cpds** refers to a **ConnectionPoolDataSource** object.
- The JNDI logical name for the **ConnectionPoolDataSource** object is **myCPDS**.
- The variable **ds** refers to a **DataSource** object.
- The logical name for the **DataSource** object is **DS_Pool**.

To deploy a ConnectionPoolDataSource object:

1. Instantiate an **IfxConnectionPoolDataSource** object.
2. Set any desired tuning properties for the object:

```
cpds.setIfxCPMInitPoolSize(15);
cpds.setIfxCPMMinPoolSize(2);
cpds.setIfxCPMMaxPoolSize(20);
cpds.setIfxCPMServiceInterval(30);
```

3. Register the **ConnectionPoolDataSource** object using JNDI to map a logical name to the object:

```
Context ctx = new InitialContext();
ctx.bind("myCPDS",cpds);
```

4. Instantiate an **IfxDataSource** object.
5. Associate the **DataSource** object with the logical name you registered for the **ConnectionPoolDataSource** object:

```
ds.setDataSourceName("myCPDS",ds);
```

6. Register the **DataSource** object using JNDI:

```
Context ctx = new InitialContext();
ctx.bind("DS_Pool",ds);
```

Tune the Connection Pool Manager

During the deployment phase, you or your database administrator can control how connection pooling works in your applications by setting values for any of these Connection Pool Manager properties:

- IFMX_CPM_INIT_POOLSIZE lets you specify the initial number of connections to be allocated for the pool when the **ConnectionPoolDataSource** object is first instantiated and the pool is initialized. The default is `0`.

  Set this property if your application will need many connections when the **ConnectionPoolDataSource** object is first instantiated.

  To obtain the value, call getIfxCPMInitPoolSize().

  To set the value, call **setIfxCPMInitPoolSize (int *init*)**.

- IFMX_CPM_MAX_CONNECTIONS lets you specify the maximum number of simultaneous physical connections that the DataSource object can have with the server.

  `The value -1` specifies an unlimited number. The default is `-1`.

  To obtain the value, call getIfxCPMMaxConnections().

  To set the value, call **setIfxCPMMaxConnections(int *limit*)**.

- IFMX_CPM_MIN_POOLSIZE lets you specify the minimum number of connections to maintain in the pool. See the IFMX_CPM_MIN_AGELIMIT parameter for what to do when this minimum number of connections kept in the pool exceeds the age limit. The default is `0`.

  To obtain the value, call getIfxCPMMinPoolSize().

  To set the value, call **setIfxCPMMinPoolSize(int *min*)**.

- IFMX_CPM_MAX_POOLSIZE lets you specify the maximum number of connections to maintain in the pool. When the pool reaches this size, all connections return to the server. The default is `50`.

  To obtain the value, call getIfxCPMMaxPoolSize().

  To set the value, call **setIfxCPMMaxPoolSize(int *max*)**.

- IFMX_CPM_AGELIMIT lets you specify the time, in seconds, that a free connection is kept in the free connection pool.

  The default is `-1`, which means that the free connections are retained until the client terminates.

  To obtain the value, call getIfxCPMAgeLimit().

  To set the value, call **setIfxCPMAgeLimit(long *limit*)**.
- IFMX_CPM_MIN_AGELIMIT lets you specify the additional time, in seconds, that a connection in the free connection pool is retained when no connection requests have been received.

  Use this setting to reduce resources held in the pool when there are expected periods in which no connection requests will be made. A value of `0` indicates that no additional time is given to a connection in the minimum pool: the connection is released to the server whenever it exceeds IFMX_CPM_AGELIMIT.

  The default is `-1`, which means that a minimum number of free connections is retained until the client terminates.

  To obtain the value, call getIfxCPMMinAgeLimit().

  To set the value, call **setIfxCPMAgeMinLimit(long *limit*)**.
- IFMX_CPM_SERVICE_INTERVAL lets you specify the pool service frequency, in milliseconds.

  Pool service activity includes adding free connections (if the number of free connections falls below the minimum value) and removing free connections. The default is `50`.

  To obtain the value, call getIfxCPMServiceInterval().

  To set the value, call **setIfxCPMServiceInterval (long *interval*)**.
- IFMX_CPM_ENABLE_SWITCH_HDRPOOL lets you specify whether to allow automatic switching between the primary and secondary connection pools of an HDR database server pair.

  Set this property if your application relies on High-Availability Data Replication with connection pooling. The default is `false`.

  To obtain the value, call getIfxCPMSwitchHDRPool().

  To set the value, call **setIfxCPMSwitchHDRPool(boolean *flag*)**.

A demonstration program is available in the **connection-pool** directory within the `demo` directory where your JDBC driver is installed. For connection pooling with HDR, a demonstration program is available in the `hdr` directory within the `demo` directory. For details about the files, see Sample code files on page 224.

Some of these properties overlap Sun JDBC 3.0 properties. The following table lists the Sun JDBC 3.0 properties and their Informix® equivalents.

Sun JDBC property name	Informix® property name	Additional information
initialPoolSize	IFMX_CPM_INIT_POOLSIZE	

Sun JDBC property name	Informix® property name	Additional information
maxPoolSize	IFMX_CPM_MAX_POOLSIZE	For maxPoolSize, 0 indicates no maximum size. For IFMX_CPM_MAX_POOLSIZE, you must specify a value.
minPoolSize	IFMX_CPM_MIN_POOLSIZE	
maxIdleTime	IFMX_CPM_AGELIMIT	For maxIdleTime, 0 indicates no time limit. For IFMX_CPM_AGELIMIT, -1 indicates no time limit.

The following Sun JDBC 3.0 properties are not supported:

- maxStatements
- propertyCycle

High-Availability Data Replication with connection pooling

HCL Informix® JDBC Driver implementation of connection pooling provides the ability to pool connections with database servers in an HDR pair:

- The primary pool contains connections to the primary server in an HDR pair.
- The secondary pool contains connections to the secondary server in an HDR pair.

You do not have to change application code to take advantage of connection pooling with HDR. Set the IFMX_CPM_ENABLE_SWITCH_HDRPOOL property to `TRUE` to allow switching between the two pools. When switching is allowed, the Connection Pool Manager validates and activates the appropriate connection pool.

When the primary server fails, the Connection Pool Manager activates the secondary pool. When the secondary pool is active, the Connection Pool Manager validates the state of the pool to check if the primary server is running. If the primary server is running, the Connection Pool Manager switches new connections to the primary server and sets the active pool to the primary pool.

If IFMX_CPM_ENABLE_SWITCH_HDRPOOL is set to `FALSE`, you can force switching to the other connection pool by calling the activateHDRPool_Primary() or activateHDRPool_Secondary() methods:

```
public void activateHDRPool_Primary(void) throws SQLException
public void activateHDRPool_Secondary(void) throws SQLException
```

The activateHDRPool_Primary() method switches the primary connection pool to be the active connection pool. The activateHDRPool_Secondary() method switches the secondary connection pool to be the active pool.

You can use the isReadOnly(), isHDREnabled(), and getHDRtype() methods with connection pooling (see Checks for read-only status of high-availability secondary servers on page 41).

A demonstration program is available in the `hdr` directory within the `demo` directory where HCL Informix® JDBC Driver is installed. For details about the files, see Sample code files on page 224.

Clean pooled connections

You can alter connections from their original, default properties by setting database properties, such as AUTOCOMMIT and TRANSACTION ISOLATION. When a connection is closed, these properties revert to their default values. However, a *pooled* connection does not automatically revert to default properties when it is returned to the pool.

In HCL Informix® JDBC Driver, you can call the scrubConnection() method to:

- Reset the database properties and connection level properties to the default values.
- Close open cursors and transactions.
- Retain all statements.

This now enables the application server to cache the statements, and it can be used across applications and sessions to provide better performance for end-user applications.

The signature of the scrubConnection() method is:

```
public void scrubConnection() throws SQLException
```

The following example demonstrates how to call scrubConnection():

```
try
  {
  IfmxConnection conn = (IfmxConnection)myConn;
  conn.scrubConnection();
  }
catch (SQLException e)
  {
  e.printStackTrace();
  }
```

The following method verifies whether a call to scrubConnection() has released all statements:

```
public boolean scrubConnectionReleasesAllStatements()
```

Manage connections

The following table contrasts different implementations of the connection.close() and scrubConnection() methods when they are in connection pool setup or not.

Connection pooling status	**Behavior with connection.close() method**	**Behavior with scrubconnection() method**
Non-connection pool setup	Closes database connection, all associated statement objects, and their result sets Connection is no longer valid.	Returns connection to default state, keeps opened statements, but closes result sets Connection is still valid. Releases resources associated with result sets only.
Connection Pool with HCL Informix® Implementation	Closes connection to the database and reopens it to close any	Returns a connection to the default state and keeps all open statements,

Connection pooling status	Behavior with connection.close() method	Behavior with scrubconnection() method
	statements associated with the connection object and reset the connection to its original state Connection object is then returned to the connection pool and is available when requested by a new application connection.	but closes all result sets. Calling this method is not recommended here.
Connection Pool with AppServer Implementation	Defined by users connection pooling implementation	Returns connection to default state and retains opened statements, but closes result sets

Avoid application hanging problems (HP-UX only)

If your JDBC application hangs on your HP-UX server, check the setting for the PTHREAD_COMPAT_MODE environment variable on the HP-UX server. The **PTHREAD_COMPAT_MODE** environment variable should be set to 1. This variable tells the pthread library (libpthread) to run in 1 X 1 mode instead of MxN mode. 1 X 1 is the default mode now on HP-UX. Setting this environment variable should resolve the hang problem.

Appendixes

Sample code files

This section contains tables that list and briefly describe the code examples provided with the client-side version of HCL Informix® JDBC Driver.

Most of these examples can be adapted to work with server-side JDBC by changing the syntax of the connection URL. For more information, see Format of database URLs on page 19.

The examples in the `tools/udtudrmgr` directory and the `demo/xml` directory are for client-side JDBC only in the 2.2 release.

Summary of available examples

The examples are provided in two directories:

- The `demo` directory where your HCL Informix® JDBC Driver software is installed
- The `tools` directory beneath the `demo` directory

Examples in the demo directory

Each example has its own subdirectory. Most of the directories include a README file that describes the examples and how to run them.

Directory	Type of examples
`basic`	Examples that show common database operations
`bson`	Examples that show the usage of the IfxBSONObject extension class, which is used to access the Informix® BSON data type.
`clob-blob`	Examples that use smart large objects
`udt-distinct`	Examples that use opaque and DISTINCT data types (there are additional examples using opaque types in Examples in the udtudrmgr directory on page 233)
`complex-types`	Examples that use row and collection types
`rmi`	An example using Remote Method Invocation
`stores7`	The **stores7** demonstration database
`pickaseat`	An example using **DataSource** objects
`connection-pool`	Examples that illustrate using a connection pool
`proxy`	Examples that illustrate using an HTTP proxy server
`xml`	Examples that illustrate storing and retrieving XML documents
`hdr`	Examples that illustrate using High-Availability Data Replication

Examples in the basic directory

The following table lists the files in the `basic` directory.

Demo program name	Description
`autofree.java`	Shows how to use the **IFX_AUTOFREE** environment variable
`BatchUpdate.java`	Shows how to send batch updates to the server
`ByteType.java`	Shows how to insert into and select from a table that contains a column of data type BYTE
`CallOut1.java`	Executes a C function that has an OUT parameter using **CallableStatement** methods
`CallOut2.java`	Executes an SPL function that has an OUT parameter using **CallableStatement** methods

Demo program name	Description
`CallOut3.java`	Executes a C function that has a Boolean OUT parameter using the IfmxCallableStatement.IfxRegisterOut Parameter() method
`CallOut4.java`	Executes a C function that has a CLOB type OUT parameter and uses the IfmxCallableStatement.hasOutParameter() method
`CreateDB.java`	Creates a database called **testDB**
`DBCENTURYSelect.java`	Uses the getString() method to retrieve a date string representation in which the four-digit year expansion is based on the **DBCENTURY** property value
`DBCENTURYSelect2.java`	Retrieves a date string representation in which the four-digit year expansion is based on the **DBCENTURY** property value using string-to-binary conversion Uses the getDate() method to build a **java.sql.Date** object upon which the date string representation is based
`DBCENTURYSelect3.java`	Retrieves a date string representation in which the four-digit year expansion is based on the **DBCENTURY** property value using string-to-binary conversion Uses the getTimestamp() method to build a **java.sql.Timestamp** object upon which the date string representation is based
`DBCENTURYSelect4.java`	Retrieves a date string representation in which the four-digit year expansion is based on the **DBCENTURY** property value using binary-to-string conversion Uses the getDate() method to build a **java.sql.Date** object upon which the date string representation is based
`DBCENTURYSelect5.java`	Retrieves a date string representation in which the four-digit year expansion is based on the **DBCENTURY** property value using binary-to-string conversion Uses the getTimestamp() method to build a **java.sql.Timestamp** object upon which the date string representation is based
`DBConnection.java`	Creates connections to both a database and a database server
`DBDATESelect.java`	Shows how to retrieve a date object and a date string representation from the database based on the **DBDATE** property value from the URL string
`DBMetaData.java`	Shows how to retrieve information about a database with the **DatabaseMetaData** interface
`DropDB.java`	Drops a database called **testDB**
`ErrorHandling.java`	Shows how to retrieve RSAM error messages

Demo program name	Description
GLDATESelect.java	Shows how to retrieve a date object and a date string representation from the database based on the **GL_DATE** property value from the URL string
Intervaldemo.java	Shows how to insert and select HCL Informix® interval data
LOCALESelect.java	Shows how to retrieve a date object and a date string representation from the database based on the **CLIENT_LOCALE** property value from the URL string
locmsg.java	Shows how to use Informix® extension methods that support localized error messages
MultiRowCall.java	Shows how to return multiple rows in a stored procedure call
OptimizedSelect.java	Shows how to use the **FET_BUF_SIZE** environment variable to adjust the HCL Informix® JDBC Driver tuple buffer size
optofc.java	Shows how to use the **OPTOFC** environment variable
PropertyConnection.java	Shows how to specify connection environment variables via a property list
RSMetaData.java	Shows how to retrieve information about a result set with the **ResultSetMetaData** interface
ScrollCursor.java	Shows how to retrieve a result set with a scroll cursor
Serial.java	Shows how to insert and select Informix® SERIal and SERIal8 data
SimpleCall.java	Shows how to call a stored procedure
SimpleConnection.java	Shows how to connect to a database or database server
SimpleSelect.java	Shows how to send a simple SELECT query to the database server
TextConv.java	Shows how to convert a file from the client code set to Unicode and then from Unicode to the database code set
TextType.java	Shows how to insert into and select from a table that contains a column of data type TEXT
UpdateCursor1.java	Shows how to create an updatable scroll cursor using a ROWID column in the query
UpdateCursor2.java	Shows how to create an updatable scroll cursor using a SERIAL column in the query
UpdateCursor3.java	Shows how to create an updatable scroll cursor using a primary key column in the query

Examples in the bson directory

The following table lists the files in the `bson` directory.

Demo program name	Description
`IfxBSONObjectDemo.java`	Shows the usage of BSON and JSON data types.

Examples in the clob-blob directory

The following table lists the files in the `clob-blob` directory.

Demo program name	Description
`demo1.java`	Shows how to create two tables with BLOB and CLOB columns and compare the data
`demo2.java`	Shows how to create one table with BYTE and TEXT columns and a second table with BLOB and CLOB columns and how to compare the data
`demo3.java`	Shows how to create one table with BLOB and CLOB columns and a second table with BYTE and TEXT columns and how to compare the data
`demo4.java`	Shows how to create two tables with BYTE and TEXT columns and compare the data
`demo5.java`	Shows how to store data from a file into a BLOB table column
`demo6.java`	Shows how to read a portion of the data in a smart large object
`demo_11.java`	Shows how to read data from a file into a buffer and write the contents of the buffer into a smart large object
`demo_13.java`	Shows how to write data into a smart large object and then insert the smart large object into a table
`demo_14.java`	Shows how to fetch smart large object data from a table

Examples in the udt-distinct directory

The following table lists the files in the `udt-distinct` directory (there are additional examples using opaque types in Examples in the udtudrmgr directory on page 233.)

Demo program name	Description
`charattrUDT.java`	Shows how to implement an opaque fixed-length type using **SQLData**
`createDB.java`	Creates a database that the other `udt-distinct` demonstration files use
`createTypes.java`	Shows how to create opaque and distinct types in the database
`distinct_d1.java`	Shows how to create a distinct type without using **SQLData**

Demo program name	Description
distinct_d2.java	Shows how to create a second distinct type without using **SQLData**
dropDB.java	Drops the database that the other udt-distinct demonstration files use
largebinUDT.java	Shows how to implement an opaque type (smart large object embedded) using **SQLData**
manualUDT.java	Shows how to implement an opaque type that allows you to change the position in the input stream
myMoney.java	Shows how to implement a distinct type using **SQLData**
udt_d1.java	Shows how to create a fixed-length opaque type
udt_d2.java	Shows how to create an opaque type with an embedded smart large object
udt_d3.java	Shows how to create an opaque type that allows you to change the position in the input stream

Examples in the complex-types directory

The following table lists the files in the complex-types directory.

Demo program name	Description
createDB.java	Creates a database with named rows
list1.java	Inserts and selects a simple collection using both the **java.sql.Array** and **java.util.Collection** classes
list2.java	Inserts and selects a collection with a nested row element
	Uses both the **java.sql.Array** and **java.util.Collection** classes for the collection and both the **SQLData** and **Struct** interfaces for the nested row
r1_t.java	Defines the **SQLData** class for named row **r1_t**
r2_t.java	Defines the **SQLData** class for named row **r2_t**
GenericStruct.java	Instantiates a **java.sql.Struct** object for inserting into named or unnamed rows
row1.java	Inserts and selects a simple named row using both the **SQLData** and **Struct** interfaces
row2.java	Inserts and selects a named row with a nested collection using both the **SQLData** and **Struct** interfaces
	The **SQLData** interface uses the HCL Informix® IfmxComplexSQLOutput. writeObject() and IfmxComplexSQLOutput.readObject() extension methods to write and read the nested collection.
row3.java	Inserts and selects an unnamed row with a nested collection
fullname.java	Contains the **SQLData** class for the named row **fullname_t**

Demo program name	Description
	Used by the **demo1.java** and **demo2.java** files
person.java	Contains the **SQLData** class for the named row **person_t** Used by the demo1.java and demo2.java files
demo1.java	Fetches a named row into an **SQLData** object
demo2.java	Inserts an **SQLData** object into a named row column
demo3.java	Fetches an unnamed row column into a **Struct** object
demo4.java	Inserts a **Struct** object into a named row column
demo5.java	Fetches the Informix® SET column into a **java.util.HashSet** object
demo6.java	Fetches the Informix® SET column into a **java.util.TreeSet** object
	A customized type mapping is provided to override the default.
demo7.java	Inserts a **java.util.HashSet** object into the Informix® SET column
demo8.java	Fetches the Informix® SET column into a **java.sql.Array** object
dropDB.java	Drops the database

Examples in the proxy directory

The following table lists the files in the proxy directory. A README file in the directory contains setup information.

Demo program name	Description
ProxySelect.java	(application) Creates a sample database and connects to it using four scenarios: • Connection with a proxy server and no LDAP server • Connection with an LDAP server and no proxy server • Connection using an sqlhosts file • Direct connection (no proxy servlet, sqlhosts file, or LDAP server)
proxy.sh	(shell script) Launches ProxySelect.java. To run the script (and the demo), type: `proxy.sh -d ProxySelect -s 2`
proxy.java	(applet) Performs the same operations as ProxySelect.java from an applet. To run the applet, type: `appletviewer proxy.dital`

Demo program name	Description
proxy.dital	HTML file for proxy.java
ifmx.conf	Sample LDAP configuration file
ifmx.ldif	Sample LDAP ldif file

Examples in the connection-pool directory

The following table lists the files in the connection-pool directory. A README file in the directory contains setup information.

Demo program name	Description
AppSimulator.java	Simulates multiple client threads making DataSource connections
SetupDB.java	Creates and populates a sample database. See the comments at the beginning of the code for a sample run command
DS_Pool.prop	Lists properties for a connection-pooling application
myCPDS.prop	Lists properties for a connection-pooling application, with the optional tuning parameters included
DS_no_Pool.prop	Lists properties for an application without connection pooling
Register.java	Registers a **DataSource** object with a JNDI Name registry A sample run command is: `java Register DS_no_Pool /tmp`
runDemo	(Shell script) Creates and populates a sample database; registers the data sources DS_no_Pool and DS_Pool; and runs an application to simulate multiple client threads that connect to the sample database

Examples in the xml directory

The following table lists the files in the xml directory.

Demo program name	Description
`CreateDB.java`	Creates a sample database
`makefile`	Compiles the examples
`myHandler.java`	Sample class of callback routines for the SAX parser
`sample1.xml`	Simple XML slide
`sample2.xml`	Sample set of XML slides
`sample2.dtd`	Document-type definition for `sample1.xml`
`xmldemo1.java`	Uses XMLtoString(), getInputSource(), and `myHandler.java` to convert the XML in `sample1.xml` to an InputSource object and then parses it using the SAX parser

Examples in the hdr directory

The following table lists the files in the `hdr` directory. A README file in the directory contains setup information.

Demo program name	Description
`SetupDB.java`	Creates a sample database and table
`Register.java`	Registers the DS_no_Pool and DS_Pool **DataSource** objects with a JNDI Name registry. A sample run command is: `java Register DS_no_Pool /tmp`
`AppSimulator.java`	Simulates High-Availability Data Replication redirection for pooled and nonpooled connections made with the DataSource.getConnection() method
`HdrSimpleConnect.java`	Shows how to implement HDR redirection with the DriverManager.getConnection() method

Examples in the tools directory

The `tools` directory includes the following subdirectories:

- The `udtudrmgr` directory contains examples that use UDT and UDR Manager to create opaque types and UDRs.
- The `classgenerator` directory contains sample output files of the ClassGenerator utility.

Examples in the udtudrmgr directory

The following table lists the files in the `udtudrmgr` directory. A `README` file in the directory contains setup information.

Demo program name	Description
`createDB.java`	Creates a sample database
`dropDB.java`	Drops the sample database
`Circle.java`	(Demo application 1) Implements a Java™ class, using the default **Input** and **Output** functions, to be converted to a Java™ opaque type
`PlayWithCircle.java`	(Demo application 1) Uses the Circle opaque type in a client application
`Circle2.java`	(Demo application 2) Implements a Java™ class, with user-supplied **Input** and **Output** functions, to be converted to a Java™ opaque type
`PlayWithCircle2.java`	(Demo application 2) Uses the Circle2 opaque type in a client application
`MyCircle.java`	(Demo application 3) Creates a fixed-length opaque type without a preexisting Java™ class
`Group1.java`	(Demo application 4) Maps methods in an existing Java™ class to Java™ UDRs
`PlayWithGroup1.java`	(Demo application 4) Uses the UDRs from `Group1.java` in a client application

DataSource extensions

This section lists the HCL Informix® extensions to standard JDBC classes:

- The **IfxDataSource** class, which implements the DataSource interface
- The **IfxConnectionPoolDataSource** class, which implements the **ConnectionPoolDataSource** interface

For information about how and why to use a **DataSource** or **ConnectionPoolDataSource** object, see the JDBC 3.0 API.

HCL Informix® JDBC Driver provides extensions for the following purposes:

- Reading and writing properties
- Getting and setting standard properties
- Getting and setting Informix® connection properties
- Getting and setting Connection Pool DataSource properties

Read and write properties

The following methods are defined in the extended **DataSource** interface for reading and writing properties. These methods allow you to define a new **DataSource** object by editing the property list of an existing **DataSource** object.

```
public Properties getDsProperties();
```

Returns the **Property** object contained in the **DataSource** object

```
public void addProp(String key, Object value);
```

Adds a property to the property list

The *key* parameter specifies which property is to be added.

The *value* parameter is the value of the property.

```
public Object getProp(String key);
```

Gets the value of a property from the property list

The *key* parameter specifies which property is to be retrieved.

```
public void removeProperty(String key);
```

Removes a property from the property list

The *key* parameter specifies which property is to be removed.

```
public void readProperties(InputStream in) throws IOException;
```

Reads properties into a **DataSource** object from an **InputStream** object

The *in* parameter is the **InputStream** object from which the properties are to be read.

An exception occurs when an I/O error is encountered while reading from the input stream.

```
public void writeProperties(OutputStream out) throws IOException;
```

Writes the properties of the **DataSource** object to an **OutputStream** object

The *out* parameter is the **OutputStream** object to which the properties are to be written.

An exception occurs when an I/O error is encountered while writing to the output stream.

Get and set standard properties

The following methods are defined in the extended **DataSource** interface for getting and setting properties defined in the JDBC 3.0 API.

Property	getXXX() and setXXX() method signatures
portNumber	public int getPortNumber(); public void setPortNumber(int *value*);
databaseName	public String getDatabaseName(); public void setDatabaseName(String *value*);

Property	getXXX() and setXXX() method signatures
serverName	public String getServerName(); public void setServerName(String *value*);
user	public String getUser(); public void setUser(String *value*);
password	public String getPassword(); public void setPassword(String *value*);
description	public String getDescription(); public void setDescription(String *value*);
dataSourceName	public String getDataSourceName(); public void setDataSourceName(String *value*);

The **networkProtocol** and **roleName** properties are not supported by HCL Informix® JDBC Driver.

Get and set Informix® connection properties

The following methods are defined in the extended **DataSource** interface for getting and setting HCL Informix® environment variable values.

Note: Support for Communication Support Module (CSM) is removed starting Informix Server 14.10.xC9 . You should use Transport Layer Security (TLS)/Secure Sockets Layer (SSL) instead.

Environment variable	getIfxXXX() and setIfxXXX() method signatures
CLIENT_LOCALE	public String getIfxCLIENT_LOCALE() public void setIfxCLIENT_LOCALE(String value)
CSM	public String getIfxCSM() public void setIfxCSM(String csm)
DBANSIWARN	public boolean isIfxDBANSIWARN() public void setIfxDBANSIWARN(boolean value)
DBCENTURY	public String getIfxDBCENTURY() public void setIfxDBCENTURY(String value)

Environment variable	getIfxXXX() and setIfxXXX() method signatures
DBDATE	public String getIfxDBDATE() public void setIfxDBDATE(String value)
DB_LOCALE	public String getIfxDB_LOCALE() public void setIfxDB_LOCALE(String value)
DBSPACETEMP	public String getIfxDBSPACETEMP() public void setIfxDBSPACETEMP(String value)
DBTEMP	public String getIfxDBTEMP() public void setIfxDBTEMP(String value)
DBUPSPACE	public String getIfxDBUPSPACE() public void setIfxDBUPSPACE(String value)
DELIMIDENT	public boolean isIfxDELIMIDENT() public void setIfxDELIMIDENT(boolean value)
ENABLE_TYPE_CACHE	public boolean isIfxENABLE_TYPE_CACHE() public void setIfxENABLE_TYPE_CACHE(boolean value)
ENABLE_HDRSWITCH	public booleangetIfxENABLE_HDRSWITCH() public void setIfxENABLE_HDRSWITCH(boolean value)
FET_BUF_SIZE	public int getIfxFET_BUF_SIZE() public void setIfxFET_BUF_SIZE(int value)
GL_DATE	public String getIfxGL_DATE() public void setIfxGL_DATE(String value)
IFX_AUTOFREE	public boolean isIfxIFX_AUTOFREE() public void setIfxIFX_AUTOFREE(boolean value)
IFX_CODESETLOB	public int getIfxIFX_CODESETLOB() public void setIfxIFX_CODESETLOB(int codesetlobFlag)
IFX_DIRECTIVES	public String getIfxIFX_DIRECTIVES() public void setIfxIFX_DIRECTIVES(String value)

Environment variable	getIfxXXX() and setIfxXXX() method signatures
IFX_EXTDIRECTIVES	public String getIfxIFX_EXTDIRECTIVES() public void setIfxIFX_EXTDIRECTIVES(String value)
IFX_FLAT_UCSQ	public int getIfxIFX_FLAT_UCSQ() public void setIfxIFX_FLAT_UCSQ(int value)
IFX_GET_SMFLOAT_AS_FLOAT	public boolean getIfxIFX_GET_SMFLOAT_AS_FLOAT() public void setIfxIFX_IFX_GET_SMFLOAT_AS_FLOAT(boolean value)
IFX_ISOLATION_LEVEL	public String getIfxIFX_ISOLATION_LEVEL() public void setIfxIFX_ISOLATION_LEVEL (String iso_level)
IFX_LOCK_MODE_WAIT	public int getIfxIFX_LOCK_MODE_WAIT() public void setIfxIFX_LOCK_MODE_WAIT(int lock_time)
IFX_SET_FLOAT_AS_SMFLOAT	public boolean getIfxIFX_SET_FLOAT_AS_SMFLOAT() public void setIfxIFX_SET_FLOAT_AS_SMFLOAT(boolean value)
IFX_SOC_KEEPALIVE	public void setIfxIFX_SOC_KEEPALIVE(boolean keepalive) public boolean getIfxIFX_SOC_KEEPALIVE()
IFX_TRIMTRAILINGSPACES	public int getIfxIFX_TRIMTRAILINGSPACES() public void setIfxIFX_TRIMTRAILINGSPACES(int value)
IFXHOST	public String getIfxIFXHOST() public void setIfxIFXHOST(String value)
IFXHOST_SECONDARY	public String getIfxIFXHOST_SECONDARY() public void setIfxIFXHOST_SECONDARY(String value)
IFX_USEPUT	public boolean isIfxIFX_USEPUT() public void setIfxIFX_USEPUT(boolean value)
IFX_XASPEC	public String getIfxIFX_XASPEC() (returns `y` or `n`) public void IfxIFX_XASPEC(String XASPEC_flag) (only `y`, `Y`, `n`, or `N` are valid)
IFX_XASTDCOMPLIANCE_XAEND	public int getIfxIFX_XASTDCOMPLIANCE_XAEND() public void setIfxIFX_XASTDCOMPLIANCE_XAEND(int value)

Environment variable	getIfxXXX() and setIfxXXX() method signatures
INFORMIXCONRETRY	public int getIfxINFORMIXCONRETRY() public void setIfxINFORMIXCONRETRY(int value)
INFORMIXCONTIME	public int getIfxINFORMIXCONTIME() public void setIfxINFORMIXCONTIME(int value)
INFORMIXOPCACHE	public String getIfxINFORMIXOPCACHE() public void setIfxINFORMIXOPCACHE(String value)
INFORMIXSERVER_SECONDARY	public String getIfxINFORMIXSERVER_SECONDARY() public void setIfxINFORMIXSERVER_SECONDARY(String value)
INFORMIXSTACKSIZE	public int getIfxINFORMIXSTACKSIZE() public void setIfxINFORMIXSTACKSIZE(int value)
JDBCTEMP	public String getIfxJDBCTEMP() public void setIfxJDBCTEMP(String value)
LDAP_IFXBASE	public String getIfxLDAP_IFXBASE() public void setIfxLDAP_IFXBASE(String value)
LDAP_PASSWD	public String getIfxLDAP_PASSWD() public void setIfxLDAP_PASSWD(String value)
LDAP_URL	public String getIfxLDAP_URL() public void setIfxLDAP_URL(String value)
LDAP_USER	public String getIfxLDAP_USER() public void setIfxLDAP_USER(String value)
LOBCACHE	public int getIfxLOBCACHE() public void setIfxLOBCACHE(int value)
NEWCODESET	public String getIfxNEWCODESET() public void setIfxNEWCODESET(String value)
NEWLOCALE	public String getIfxNEWLOCALE() public void setIfxNEWLOCALE(String value)

Environment variable	getIfxXXX() and setIfxXXX() method signatures
NEWNLSMAP	public String getIfxNEWNLSMAP() public void setIfxNEWNLSMAP(String value)
NODEFDAC	public String getIfxNODEFDAC() public void setIfxNODEFDAC(String value)
OPT_GOAL	public String getIfxOPT_GOAL() public void setIfxOPT_GOAL(String value)
OPTCOMPIND	public String getIfxOPTCOMPIND() public void setIfxOPTCOMPIND(String value)
OPTOFC	public String getIfxOPTOFC() public void setIfxOPTOFC(String value)
PATH	public String getIfxPATH() public void setIfxPATH(String value)
PDQPRIORITY	public String getIfxPDQPRIORITY() public void setIfxPDQPRIORITY(String value)
PLCONFIG	public String getIfxPLCONFIG() public void setIfxPLCONFIG(String value)
PLOAD_LO_PATH	public String getIfxPLOAD_LO_PATH() public void setIfxPLOAD_LO_PATH(String value)
PORTNO_SECONDARY	public String getIfxPORTNO_SECONDARY public void setIfxPORTNO_SECONDARY(int value)
PROXY	public String getIfxPROXY() public void setIfxPROXY(String value)
PSORT_DBTEMP	public String getIfxPSORT_DBTEMP() public void setIfxPSORT_DBTEMP(String value)
PSORT_NPROCS	public String getIfxPSORT_NPROCS() public void setIfxPSORT_NPROCS(String value)

Environment variable	getIfxXXX() and setIfxXXX() method signatures
SECURITY	public String getIfxSECURITY() public void setIfxSECURITY(String value)
SQLH_FILE	public String getIfxSQLH_FILE() public void setIfxSQLH_FILE(String value)
SQLH_TYPE	public String getIfxSQLH_TYPE() public void setIfxSQLH_TYPE(String value)
SQLIDEBUG	public String getIfxSQLIDEBUG () public void setIfxSQLIDEBUG (String value)
STMT_CACHE	public String getIfxSTMT_CACHE() public void setIfxSTMT_CACHE(String value)
USEV5SERVER	public boolean isIfxUSEV5SERVER() public void setIfxUSEV5SERVER(boolean value)

Get and set connection pool DataSource properties

The code you write to use a **ConnectionPoolDataSource** object is the same as the code you write to use a **DataSource** object. Additional tuning parameters let you or your database administrator control some aspects of connection pool management with the Connection Pool Manager. These are more fully described in A connection pool on page 219. The following table summarizes them.

Property	getXXX() and setXXX() method signatures
IFMX_CPM_ENABLE_SWITCH_HDRPOOL	public void setIfxCPMSwitchHDRPool (boolean *flag*) public int getIfxCPMSwitchHDRPool()
IFMX_CPM_INIT_POOLSIZE	public void setIfxCPMInitPoolSize (int *init*) public int getIfxCPMInitPoolSize()
IFMX_CPM_MAX_CONNECTIONS	public void setIfxCPMMaxConnections (int *limit*) public int getIfxCPMMaxConnections()
IFMX_CPM_MIN_POOLSIZE	public void setIfxCPMMinPoolSize (int *min*) public int getIfxCPMMinPoolSize()

Property	getXXX() and setXXX() method signatures
IFMX_CPM_MAX_POOLSIZE	public void setIfxCPMMaxPoolSize (int *max*) public int getIfxCPMMaxPoolSize()
IFMX_CPM_MIN_AGELIMIT	public void setIfxCPMMinAgeLimit (long *limit*) public long getIfxCPMMinAgeLimit()
IFMX_CPM_MAX_AGELIMIT	public void setIfxCPMMaxAgeLimit (long *limit*) public long getIfxCPMMaxAgeLimit()
IFMX_CPM_SERVICE_INTERVAL	public void setIfxCPMServiceInterval (long *interval*) public long getIfxCPMServiceInterval()

Mapping data types

This section discusses mapping issues between data types defined in a Java™ program and the data types supported by the HCL Informix® database server.

Data type mapping between Informix® and JDBC data types

Because there are variations between the SQL data types supported by each database vendor, the JDBC API defines a set of generic SQL data types in the class **java.sql.Types**. Use these JDBC API data types to reference generic SQL types in your Java™ programs that use the JDBC API to connect to HCL Informix® databases.

The following table shows the Informix® data type to which each JDBC API data type maps.

JDBC API data type	Informix® data type
BIGINT	INT8, BIGINT, BIGSERIAL
BINARY	BYTE
BIT [1]	BOOLEAN
REF	Not supported
CHAR	CHAR(*n*)
DATE	DATE
DECIMAL	DECIMAL
DOUBLE	FLOAT
FLOAT	FLOAT[2]
INTEGER	INTEGER

JDBC API data type	Informix® data type
LONGVARBINARY	BYTE or BLOB
LONGVARCHAR	TEXT or CLOB
NUMERIC	DECIMAL
NUMERIC	MONEY
REAL	SMALLFLOAT
SMALLINT	SMALLINT
TIME	DATETIME HOUR TO SECOND[2]
TIMESTAMP	DATETIME YEAR TO FRACTION(5)[3]
TINYINT	SMALLINT
VARBINARY	BYTE
VARCHAR	VARCHAR(*m*,*r*)
BOOLEAN	BOOLEAN

[1] With Java™ 1.4 is , java.sql.Types.BOOLEAN maps to BOOLEAN.

[2] This mapping is JDBC compliant. You can map the JDBC FLOAT data type to the Informix® SMALLFLOAT data type for backward compatibility by setting the **IFX_SET_FLOAT_AS_SMFLOAT** environment variable to 1.

[3] Informix® DATETIME types are very restrictive and are not interchangeable. For more information, see Field lengths and date-time data on page 262.

Data type mapping between extended types and Java™ and JDBC types

The following table lists mappings between the extended data types supported in HCL Informix® and the corresponding Java™ and JDBC types.

JDBC type	Java™ object type	Informix® type
java.sql.Types.LONGVARCHAR	java.sql.String java.io.inputStream	LVARCHAR IfxTypes.IFX_TYPE_LVARCHAR
java.sql.Types.JAVA_OBJECT	java.sql.SQLData	Opaque type IfxTypes.IFX_TYPE_UDTFIXED IfxTypes.IFX_TYPE_UDTVAR

JDBC type	Java™ object type	Informix® type
java.sql.Types.LONGVARBINARY java.sql.Types.BLOB	java.sql.Blob java.io.inputStream byte[]	BLOB IfxTypes.IFX_TYPE_BLOB
java.sql.Types.LONGVARCHAR java.sql.Types.CLOB	java.sql.Clob java.io.inputStream java.lang.String	CLOB IfxTypes.IFX_TYPE_CLOB
java.sql.Types.LONGVARBINARY java.sql.Types.BLOB	java.io.inputStream java.sql.Blob byte[]	BYTE IfxTypes.IFX_TYPE_BYTE
java.sql.Types.LONGVARCHAR java.sql.Types.CLOB	java.io.InputStream java.sql.Clob java.sql.String	TEXT IfxTypes.IFX_TYPE_TEXT
java.sql.Types.JAVA_OBJECT java.sql.Types.STRUCT	java.sql.SQLData java.sql.Struct	Named row IfxTypes.IFX_TYPE_ROW
java.sql.Types.STRUCT	java.sql.Struct	Unnamed row IfxTypes.IFX_TYPE_ROW
java.sql.Types.ARRAY java.sql.Types.OTHER	java.sql.Array java.util.LinkedList java.util.HashSet java.util.TreeSet	set, multiset IfxTypes.IFX_TYPE_SET IfxTypes.IFX_TYPE_MULTISET
java.sql.Types.ARRAY java.sql.Types.OTHER	java.sql.Array java.util.ArrayList java.util.LinkedList	LIST IfxTypes.IFX_TYPE_LIST

A Java™ boolean object can map to the Informix® smallint data type or the Informix® boolean data type. HCL Informix® JDBC Driver attempts to map it according to the column type. However, in cases such as **PreparedStatement** host variables, HCL Informix® JDBC Driver cannot access the column types, so the mapping is somewhat limited. For more details on data type mapping, see Data type mapping for PreparedStatement.setXXX() extensions on page 244.

Data type mapping between C opaque types and Java™

To create an opaque type using Java™, you can use the UDT and UDR Manager facility. For more information, see Work with opaque types on page 154.

All opaque data is stored in the database server table in a C struct, which is made up of various DataBlade® API types, as defined in the opaque type. (For more information, see the *HCL® Informix® DataBlade® API Programmer's Guide*.)

The following table lists the mapping of DataBlade® API types to their corresponding Java™ types.

DataBlade® API type	Java™ type
MI_LO_HANDLE	BLOB or CLOB
gl_wchar_t	String
mi_boolean	boolean
mi_char	String
mi_char1	String
mi_date	Date
mi_datetime	TimeStamp
mi_decimal	BigDecimal
mi_double_precision	double
mi_int1	byte
mi_int8	long
mi_integer	int
mi_interval	Not supported
mi_money	BigDecimal
mi_numeric	BigDecimal
mi_real	float
mi_smallint	short
mi_string	String
mi_unsigned_char1	String
mi_unsigned_int8	long
mi_unsigned_integer	int
mi_unsigned_smallint	short
mi_wchar	String

The C struct may contain padding bytes. HCL Informix® JDBC Driver automatically skips these padding bytes to make sure the next data member is properly aligned. Therefore, your Java™ objects do not have to take care of alignment themselves.

Data type mapping for PreparedStatement.setXXX() extensions

HCL Informix® introduces many extended data types. As a result, there can be multiple mappings between a JDBC or Java™ data type and the corresponding Informix® data type.

For example, you can use PreparedStatement.setAsciiStream() to insert into either a text column or a CLOB column. Similarly, you can also use PreparedStatement.setBinaryStream() to insert into a byte column or a BLOB column. Because the actual column information is not available to HCL Informix® JDBC Driver at all times, there can be ambiguity for the driver when it maps data types.

Normally, with INSERT, SELECT, or DELETE statements, the column information is available to the driver, so the driver can determine how the data can be sent to the database server.

However, when the data is referenced in an UPDATE statement or inside a WHERE clause, HCL Informix® JDBC Driver does not have access to the column information. In those cases, unless you use the Informix® extensions, the driver maps those columns using the corresponding Informix® data types listed in the first table in Data type mapping between Informix and JDBC data types on page 241. For the PreparedStatement.setAsciiStream() method, the driver tries to map to a text data type, and for the PreparedStatement.setBinaryStream() method, it tries to map to a byte data type.

The mapping extensions

To direct the driver to map to a certain data type (so there is no ambiguity in UPDATE statements and WHERE clauses), you can use extensions to the PreparedStatement.setXXX() methods. The only data types that might have ambiguity are boolean, lvarchar, text, byte, BLOB, and CLOB.

To use these extended methods, you must cast your **PreparedStatement** references to **IfmxPreparedStatement**. For example, the following code casts the statement variable **p_stmt** to **IfmxPreparedStatement** to call the IfxSetObject() method and insert the contents of a file as a large object of type CLOB. IfxSetObject() is defined as **I**:

```
public void IfxSetObject(int i, Object x, int scale, int ifxType)
   throws SQLException
public void IfxSetObject(int i, Object x, int ifxType) throws
   SQLexception
```

The code is:

```
File file = new File("sblob_06.dat");
int fileLength = (int)file.length();
byte[] buffer = new byte[fileLength];
FileInputStream fin = new FileInputStream(file);
fin.read(buffer,0,fileLength);
String str = new String(buffer);

writeOutputFile("Prepare");
PreparedStatement p_stmt = myConn.prepareStatement
   ("insert into sblob_t20(c1) values(?)");

writeOutputFile("IfxSetObject");
((IfmxPreparedStatement)p_stmt).IfxSetObject(1,str,30,IfxTypes.IFX
   _TYPE_CLOB);
```

For the **IfmxPreparedStatement.IfxSetObject** extension, you cannot simply overload the method signature with an added **ifxType** parameter, because such overloading creates method ambiguity. You must name the method to **IfxSetObject** instead.

The extensions for opaque types

The extensions for processing opaque types allow your application to specify the return type to which the database server should cast the opaque type before returning it to the client. This is known as *prebinding* the return value. The methods are:

- setBindColType(), which allows applications to specify the output type of result-set values using standard JDBC data types from **java.sql.Types**
- setBindColIfxType(), which allows applications to specify the output type of result-set values using HCL Informix® data types from **com.informix.lang.IfxTypes**

  For more information about the available types, see The IfxTypes class on page 249.
- clearBindColType(), which resets values set through the previous two methods

In the following topics:

- The *colIndex* parameter specifies the column: 1 is the first column, 2 the second, and so forth
- The *sqltype* parameter is a value from **java.sql.Types**: for example, `Types.INTEGER`.
- The *ifxtype* parameter is a value from **IfxTypes**: for example, `IfxTypes.IFX_TYPE_DECIMAL`.

The setBindColType() methods

The methods are as follows:

```
public void setBindColType(int colIndex, int sqltype) throws SQLException;
public void setBindColType(int colIndex, int sqltype, int scale)
   throws SQLException;
public void setBindColType(int colIndex, int sqltype, String name)
   throws SQLException;
```

The first overloaded method allows applications to specify the output type to be `java.sql.DECIMAL` or `java.sql.NUMERIC`; the *scale* parameter specifies the number of digits to the right of the decimal point. The second overloaded method allows applications to specify the output type to be `java.sql.STRUCT`, `java.sql.ARRAY`, `java.sql.DISTINCT`, or `java.sql.JAVA_OBJECT` by assigning one of these values to the *name* parameter.

The setBindColIfxType() methods

The methods are as follows:

```
public void setBindColIfxType(int colIndex, int ifxtype) throws SQLException;
public void setBindColIfxType(int colIndex, int ifxtype, int scale)
   throws SQLException;
public void setBindColIfxType(int colIndex, int ifxtype, String name)
   throws SQLException;
```

The first overloaded method allows applications to specify the output type to be `IFX_TYPE_DECIMAL` or `IFX_TYPE_NUMERIC`; the *scale* parameter specifies the number of digits to the right of the decimal point. The second overloaded method allows applications to specify the output type to be `IFX_TYPE_LIST`, `IFX_TYPE_ROW`, `IFX_TYPE_MULTISET`, `IFX_TYPE_SET`, `IFX_TYPE_UDTVAR`, or `IFX_TYPE_UDTFIXED` by assigning one of these values to the *name* parameter.

The clearBindColType() method

The method is as follows:

```
public void clearBindColType() throws SQLException;
```

Prebinding example

The following code from the `udt_bindCol.java` sample program prebinds an opaque type to the HCL Informix® VARCHAR and then to a standard Java™ Integer type. The table used in this example has one **int** column and one opaque type column and is defined as follows:

```
create table charattr_tab (int_col int, charattr_col charattr_udt)
```

The code to select and prebind the opaque type in the **charattr_col** column is as follows:

```
String s = "select int_col, charattr_col as cast_udt_to_lvc, " +
    "charattr_col as cast_udt_to_int from charattr_tab order by 1";

pstmt = conn.prepareStatement(s);
    ((IfxPreparedStatement)pstmt).setBindColIfxType(2,IfxTypes.IFX_TYPE_LVARCHAR);
    ((IfxPreparedStatement)pstmt).setBindColType(3,Types.INTEGER);

ResultSet rs = pstmt.executeQuery();

System.out.println("Fetching data ...");
int curRow = 0;
while (rs.next())
{
     curRow++;
     int intret = rs.getInt("int_col");
     String strret = rs.getString("cast_udt_to_lvc");
     int intret2 = rs.getInt("cast_udt_to_int");
} // end while
```

Other mapping extensions

The remaining method signatures are listed next, along with any additional considerations that apply. In each case, the HCL Informix® type must be the last parameter to the standard JDBC PreparedStatement.setXXX() interface.

IfmxPreparedStatement.setArray()

```
public void setArray(int parameterIndex, Array x, int ifxType)
   throws SQLException
```

IfmxPreparedStatement.setAsciiStream()

```
public void setAsciiStream(int i, InputStream x, int length, int
 ifxType) throws SQLException
```

When your application is inserting a very large ASCII value into a LONGVARCHAR column, it is sometimes more efficient to send the ASCII value to the server using **java.io.InputStream**.

IfmxPreparedStatement.setBigDecimal()

```
public void setBigDecimal(int i, BigDecimal x, int ifxType)
   throws SQLException
```

IfmxPreparedStatement.setBinaryStream()

```
public void setBinaryStream(int i, InputStream x, int length, int
 ifxType) throws SQLException
```

When your application is inserting a very large binary value into a LONGVARbinary column, it is sometimes more efficient to send the binary value to the server using **java.io.InputStream**.

IfmxPreparedStatement.setBlob()

```
public void setBlob(int parameterIndex, Blob x, int ifxType)
   throws SQLException
```

IfmxPreparedStatement.setBoolean()

```
public void setBoolean(int i, boolean x, int ifxType) throws
   SQLException
```

IfmxPreparedStatement.setByte()

```
public void setByte(int i, byte x, int ifxType) throws
   SQLException
```

IfmxPreparedStatement.setBytes()

```
public void setBytes(int i, byte x[], int ifxType) throws
   SQLException
```

IfmxPreparedStatement.setCharacterStream()

```
public void setCharacterStream(int parameterIndex, Reader reader,
   int length, int ifxType) throws SQLException
```

When your application is setting a LONGVARCHAR parameter to a very large UNICODE value, it is sometimes more efficient to send the UNICODE value to the server using **java.io.Reader**.

IfmxPreparedStatement.setClob()

```
public void setClob(int parameterIndex, Clob x, int ifxType)
   throws SQLException
```

IfmxPreparedStatement.setDate()

```
public void setDate(int i, Date x, int ifxType) throws
   SQLException
public void setDate(int parameterIndex, Date x, Calendar Cal,
   int ifxType) throws SQLException
```

IfmxPreparedStatement.setDouble()

```
public void setDouble(int i, double x, int ifxType) throws SQ
   LException
```

IfmxPreparedStatement.setFloat()

```
public void setFloat(int i, float x, int ifxType) throws
   SQLException
```

IfmxPreparedStatement.setInt()

```
public void setInt(int i, int x, int ifxType) throws SQLException
```

IfmxPreparedStatement.setLong()

```
public void setLong(int i, long x, int ifxType) throws
   SQLException
```

IfmxPreparedStatement.setNull()

```
public void setNull(int i, int sqlType, int ifxType) throws
   SQLException
```

IfmxPreparedStatement.setShort()

```
public void setShort(int i, short x, int ifxType) throws
SQLException
```

IfmxPreparedStatement.setString()

```
public void setString(int i, String x, int ifxType) throws
   SQLException
```

IfmxPreparedStatement.setTime()

```
public void setTime(int i, Time x, int ifxType) throws
   SQLException
public void setTime(int parameterIndex, Time time, Calendar Cal,
   int ifxType) throws SQLException
```

IfmxPreparedStatement.setTimestamp()

```
public void setTimestamp(int i, Timestamp x, int ifxType) throws
   SQLException
public void setTimestamp(int parameterIndex, Timestamp x, Calendar
 Cal) throws SQLException
```

The IfxTypes class

The extended **IfmxPreparedStatement** methods require you to pass in the HCL Informix® data type to which you want to map. These types are part of the **com.informix.lang.IfxTypes** class.

The following table shows the **IfxTypes** constants and the corresponding Informix® data types.

IfxTypes constant	Informix® data type
IfxTypes.IFX_TYPE_BIGINT	BIGINT
IfxTypes.IFX_TYPE_BIGSERIAL	BIGSERIAL
IfxTypes.IFX_TYPE_CHAR	CHAR

IfxTypes constant	Informix® data type
IfxTypes.IFX_TYPE_SMALLINT	SMALLINT
IfxTypes.IFX_TYPE_INT	INT
IfxTypes.IFX_TYPE_FLOAT	FLOAT
IfxTypes.IFX_TYPE_SMFLOAT	SMALLFLOAT
IfxTypes.IFX_TYPE_DECIMAL	DECIMAL
IfxTypes.IFX_TYPE_SERIAL	SERIAL
IfxTypes.IFX_TYPE_DATE	DATE
IfxTypes.IFX_TYPE_MONEY	MONEY
IfxTypes.IFX_TYPE_NULL	NULL
IfxTypes.IFX_TYPE_DATETIME	DATETIME
IfxTypes.IFX_TYPE_BYTE	BYTE
IfxTypes.IFX_TYPE_TEXT	TEXT
IfxTypes.IFX_TYPE_VARCHAR	VARCHAR
IfxTypes.IFX_TYPE_INTERVAL	INTERVAL
IfxTypes.IFX_TYPE_NCHAR	NCHAR
IfxTypes.IFX_TYPE_NVARCHAR	NVARCHAR
IfxTypes.IFX_TYPE_INT8	INT8
IfxTypes.IFX_TYPE_SERIAL8	SERIAL8
IfxTypes.IFX_TYPE_SET	SQLSET
IfxTypes.IFX_TYPE_MULTISET	SQLMULTISET
IfxTypes.IFX_TYPE_LIST	SQLLIST
IfxTypes.IFX_TYPE_ROW	SQLROW
IfxTypes.IFX_TYPE_COLLECTION	COLLECTION
IfxTypes.IFX_TYPE_UDTVAR	UDTVAR
IfxTypes.IFX_TYPE_UDTFIXED	UDTFIXED
IfxTypes.IFX_TYPE_REFSER8	REFSER8
IfxTypes.IFX_TYPE_LVARCHAR	LVARCHAR
IfxTypes.IFX_TYPE_SENDRECV	SENDRECV

IfxTypes constant	Informix® data type
IfxTypes.IFX_TYPE_BOOL	BOOLEAN
IfxTypes.IFX_TYPE_IMPEXP	IMPEXP
IfxTypes.IFX_TYPE_IMPEXPBIN	IMPEXPBIN
IfxTypes.IFX_TYPE_CLOB	CLOB
IfxTypes.IFX_TYPE_BLOB	BLOB

Extension summary

The tables in this section list the PreparedStatement.setXXX() methods that HCL Informix® JDBC Driver supports for nonextended data types and Informix® extended data types.

Nonextended data types

The following tables list the PreparedStatement.setXXX() methods that HCL Informix® JDBC Driver supports for nonextended data types. The top heading lists the standard JDBC API data types defined in the **java.sql.Types** class. These translate to specific Informix® data types, as shown in the table in Data type mapping between extended types and Java and JDBC types on page 242. The tables below list the setXXX() methods you can use to write data of a particular JDBC API data type. An uppercase and bold **X** indicates the setXXX() method that it is recommended you use with HCL Informix® JDBC Driver; a lowercase `x` indicates other setXXX() methods that HCL Informix® JDBC Driver supports.

Numeric JDBC API data types

Table 12. Numeric JDBC API data types from java.sql.Types

setXXX() method	TINYINT	SMALLINT	INTEGER	BIGINT
setByte()	**X**	x	x	x
setShort()	x	**X**	x	x
setInt()	x	x	**X**	x
setLong()	x	x	x	**X**
setFloat()	x	x	x	x
setDouble()	x	x	x	x
setBigDecimal()	x	x	x	x
setBoolean()	x	x	x	x
setString()	x	x	x	x
setObject()	x	x	x	x

Table 13. Numeric JDBC API data types from java.sql.Types (continued)

setXXX() method	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC
setByte()	x	x	x	x	x
setShort()	x	x	x	x	x
setInt()	x	x	x	x	x
setLong()	x	x	x	x	x
setFloat()	**X**	x	x	x	x
setDouble()	x	**X**	**X**	x	x
setBigDecimal()	x	x	x	**X**	**X**
setBoolean()	x	x	x	x	x
setString()	x	x	x	x	x
setObject()	x	x	x	x	x

Character and chronological JDBC API data types

Table 14. Character and chronological JDBC API data types from java.sql.Types

setXXX() method	CHAR	VARCHAR	LONGVARCHAR	BINARY
setByte()	x[1 on page 253]	x[1 on page 253]		
setShort()	x[1 on page 253]	x[1 on page 253]		
setInt()	x[1 on page 253]	x[1 on page 253]		
setLong()	x[1 on page 253]	x[1 on page 253]		
setFloat()	x[1 on page 253]	x[1 on page 253]		
setDouble()	x[1 on page 253]	x[1 on page 253]		
setBigDecimal()	x	x		
setBoolean()	x	x		
setString()	**X**	**X**	x	x
setBytes()			x	**X**
setDate()	x	x		
setTime()	x	x		
setTimestamp()	x	x		

Table 14. Character and chronological JDBC API data types from java.sql.Types (continued)

setXXX() method	CHAR	VARCHAR	LONGVARCHAR	BINARY
setAsciiStream()			**X**	x
setCharacterStream()			**X**	x
setBinaryStream()			x	x
setObject()	x	x	x[2 on page 253]	x

Notes:

1. The column value must match the type of setXXX() exactly, or an **SQLException** is raised. If the column value is not within the allowed value range, the setXXX() method raises an exception instead of converting the data type. For example, setByte(1) raises an **SQLException** if the value being written is `1000`.
2. A byte array is written.

Table 15. Character and chronological JDBC API data types from java.sql.Types (continued)

setXXX() method	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP
setString()	x	x	x	x	x
setBytes()	**X**	x			
setDate()			**X**		x
setTime()				**X**	x
setTimestamp()			x		**X**
setAsciiStream()	x	x			
setCharacterStream()	x	x			
setBinaryStream()	x	**X**			
setObject()	x	x[1 on page 253]	x	x[2 on page 253]	x

Notes:

1. A byte array is written.
2. A **Timestamp** object is written instead of a **Time** object.

The setMaxRows() method writes an SQL null value.

Informix® extended data types

The following table lists the PreparedStatement.setXXX() methods that HCL Informix® JDBC Driver supports for the Informix® extended data types, the mappings for which are shown in the table Data type mapping between extended types and Java and JDBC types on page 242. The table lists the setXXX() methods you can use to write data of a particular extended data type.

An uppercase and bold **X** indicates the recommended setXXX() method to use; a lowercase `x` indicates other setXXX() methods supported by HCL Informix® JDBC Driver. The table does not include setXXX() methods that you cannot use with any of the Informix® extended data types.

Table 16. Informix® extended data types

setXXX() method	BOOLEAN	LVARCHAR	Opaque types	BLOB	CLOB	BYTE	TEXT
setByte()	x	x					
setShort()	x						
setInt()	x						
setBoolean()	**X**						
setString()		**X**			x		x
setBytes()				x		x	
setAsciiStream()		x			x		**X**
setCharacterStream()		x			x		**X**
setBinaryStream()	x			x		**X**	
setObject()	x	x	**X**	x	x	x	x
setArray()							
setBlob()				**X**			
setClob()					**X**		

Table 17. Informix® extended data types (continued)

setXXX() method	NAMED ROW	UNNAMED ROW	SET or MULTISET	LIST
setObject()	**X**	**X**	x	x
setArray()			x	x

The setMaxRows() method writes an SQL null value.

Data type mapping for ResultSet.getXXX() methods

Use the ResultSet.getXXX() methods to transfer data from the HCL Informix® database to a Java™ program that uses the JDBC API to connect to the Informix® database. For example, use the ResultSet.getString() method to get the data stored in a column of data type LVARCHAR.

Important: If you use an expression within an SQL statement—for example, `SELECT mytype::LVARCHAR FROM mytab`—you might not be able to use ResultSet.getXXX(columnName) to retrieve the value. Use ResultSet.getXXX(columnIndex) to retrieve the value instead.

The getXXX() methods return a null value if the retrieved column value is an SQL null value.

The tables in this section list the ResultSet.getXXX() methods that HCL Informix® JDBC Driver supports for nonextended data types and Informix® extended data types.

Nonextended data types

The following tables list the ResultSet.getXXX() methods that HCL Informix® JDBC Driver supports for nonextended data types. The top heading lists the standard JDBC API data types defined in the **java.sql.Types** class. These translate to specific Informix® data types, as shown in the first table in Data type mapping between Informix and JDBC data types on page 241. The tables list the getXXX() methods you can use to retrieve data of a particular JDBC API data type.

An uppercase and bold **X** indicates the recommended getXXX() method to use; a lowercase `x` indicates other getXXX() methods supported by HCL Informix® JDBC Driver.

Numeric JDBC API data types

Table 18. Numeric JDBC API data types from java.sql.Types

getXXX() method	TINYINT	SMALLINT	INTEGER	BIGINT
getByte()	**X**	x	x	x
getShort()	x	**X**	x	x
getInt()	x	x	**X**	x
getLong()	x	x	x	**X**
getFloat()	x	x	x	x
getDouble()	x	x	x	x
getBigDecimal()	x	x	x	x
getBoolean()	x	x	x	x
getString()	x	x	x	x
getObject()	x	x	x	x

Table 19. Numeric JDBC API data types from java.sql.Types (continued)

getXXX() method	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC
getByte()	x	x	x	x	x
getShort()	x	x	x	x	x
getInt()	x	x	x	x	x
getLong()	x	x	x	x	x
getFloat()	**X**	x	x	x	x
getDouble()	x	**X**	**X**	x	x
getBigDecimal()	x	x	x	**X**	**X**
getBoolean()	x	x	x	x	x
getString()	x	x	x	x	x
getObject()	x	x	x	x	x

Character and chronological JDBC API data types

Table 20. Character and chronological JDBC API data types from java.sql.Types

getXXX() method	CHAR	VARCHAR	LONGVARCHAR	BINARY
getByte()	x[1 on page 257]	x[1 on page 257]		
getShort()	x[1 on page 257]	x[1 on page 257]		
getInt()	x[1 on page 257]	x[1 on page 257]		
getLong()	x[1 on page 257]	x[1 on page 257]		
getFloat()	x[1 on page 257]	x[1 on page 257]		
getDouble()	x[1 on page 257]	x[1 on page 257]		
getBigDecimal()	x	x		
getBoolean()	x	x		
getString()	**X**	**X**	x	x
getBytes()	x	x	x	**X**
getDate()	x	x		
getTime()	x	x		
getTimestamp()	x	x		

Table 20. Character and chronological JDBC API data types from java.sql.Types (continued)

getXXX() method	CHAR	VARCHAR	LONGVARCHAR	BINARY
getAsciiStream()			**X**	x
getCharacterStream()			**X**	x
getBinaryStream()			x	x
getObject()	x	x	x[2 on page 257]	x

Notes:

1. The column value must match the type of getXXX() exactly, or an **SQLException** is raised. If the column value is not within the allowed value range, the getXXX() method raises an exception instead of converting the data type. For example, getByte(1) raises an **SQLException** if the column value is `1000`.
2. A byte array is returned.

Table 21. Character and chronological JDBC API data types from java.sql.Types (continued)

getXXX() method	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP
getString()	x	x	x	x	x
getBytes()	**X**	x			
getDate()			**X**		x
getTime()				**X**	x
getTimestamp()			x		**X**
getAsciiStream()	x	x			
getCharacterStream()	x	x			
getBinaryStream()	x	**X**			
getObject()	x	x[1 on page 258]	x	x[2 on page 258]	x

Notes:

1. A byte array is returned.
2. A **Timestamp** object is returned instead of a **Time** object.

Informix® extended data types

The following table lists the ResultSet.getXXX() methods that HCL Informix® JDBC Driver supports for the Informix® extended data types, the mappings for which are shown in the table Data type mapping between extended types and Java and JDBC types on page 242. The table lists the getXXX() methods you can use to retrieve data of a particular extended data type.

An uppercase and bold **X** indicates the recommended getXXX() method to use; a lowercase `x` indicates other getXXX() methods supported by HCL Informix® JDBC Driver. The table does not include getXXX() methods that you cannot use with any of the Informix® extended data types.

Table 22. Informix® extended data types

getXXX() method	BOOLEAN	LVARCHAR	Opaque types	BLOB	CLOB	BYTE
getByte()	x	x				
getShort()	x					
getInt()	x					
getBoolean()	**X**					
getString()		**X**			x	
getBytes()				x		x
getAsciiStream()		x			x	
getCharacterStream()		x			x	
getBinaryStream()	x			x		**X**
getObject()	x	x	**X**	x	x	x
getBlob()				**X**		
getClob()					**X**	

Table 23. Informix® extended data types (continued)

getXXX() method	TEXT	NAMED ROW	UNNAMED ROW	SET or MULTISET	LIST
getString()	x				
getBytes()					

Table 23. Informix® extended data types (continued) (continued)

getXXX() method	TEXT	NAMED ROW	UNNAMED ROW	SET or MULTISET	LIST
getAsciiStream()	**X**				
getCharacterStream()	**X**				
getBinaryStream()					
getObject()	x	**X**	**X**	x	x
getArray()				x	x
getBlob()					
getClob()					

Data type mapping for UDT manager and UDR manager

When you use the **UDTManager** and **UDRManager** classes to create opaque types and Java™ UDRs in the database server, the driver maps Java™ method arguments and return types to SQL data types according to the tables in this section. Any data type not shown in these tables is not supported.

If the Java™ method has arguments of any of the following Java™ types, the arguments and return type are mapped to SQL types in the server as shown in the following table. The table shows the HCL Informix® data type to which each Java™ data type maps.

Java™ data type	SQL data type
boolean, java.lang.Boolean	BOOLEAN
char	CHAR(1)
byte	CHAR(1)
short, java.lang.Short	SMALLINT
int, java.lang.Integer	INT
long, java.lang.Long	INT8
float, java.lang.Float	SMALLFLOAT
double, java.lang.Double	FLOAT[1]
java.lang.String	LVARCHAR
java.math.BigDecimal	DECIMAL

Java™ data type	SQL data type
	Default precision is set by the server to be: DECIMAL(16,0) for an ANSI database decimal (16,255) for a non-ANSI database
java.sql.Date	DATE
java.sql.Time	DATETIME HOUR TO SECOND
java.sql.Timestamp	DATETIME YEAR TO FRACTION(5)
com.informix.lang.IntervalYM	INTERVAL YEAR TO MONTH
com.informix.lang.IntervalDF	INTERVAL DAY TO FRACTION(5)
java.sql.Blob	BLOB
java.sql.Clob	CLOB

[1] This mapping is JDBC compliant. You can map the Java™ double data type (via the JDBC FLOAT data type) to the Informix® SMALLFLOAT data type for backward compatibility by setting the **IFX_GET_SMFLOAT_AS_FLOAT** environment variable to 1.

Mapping for casts

The following table shows the mapping supported between the type defined for the *ifxtype* parameter in the UDTMetaData.setXXXCast() methods and SQL data types in the server.

ifxtype parameter type from com.informix.lang.IfxTypes	Informix® data type
IFX_TYPE_CHAR	CHAR
IFX_TYPE_SMALLINT	SMALLINT
IFX_TYPE_INT	INT
IFX_TYPE_FLOAT	FLOAT
IFX_TYPE_SMFLOAT	SMALLFLOAT
IFX_TYPE_DECIMAL	DECIMAL
IFX_TYPE_SERIAL	SERIAL
IFX_TYPE_DATE	DATE
IFX_TYPE_MONEY	MONEY
IFX_TYPE_DATETIME	DATETIME
IFX_TYPE_BYTE	BYTE

ifxtype parameter type from com.informix.lang.IfxTypes	Informix® data type
IFX_TYPE_TEXT	TEXT
IFX_TYPE_VARCHAR	VARCHAR
IFX_TYPE_INTERVAL	INTERVAL
IFX_TYPE_NCHAR	NCHAR
IFX_TYPE_NVARCHAR	NVARCHAR
IFX_TYPE_INT8	INT8
IFX_TYPE_SERIAL8	SERIAL8
IFX_TYPE_LVARCHAR	LVARCHAR
IFX_TYPE_SENDRECV	SENDRECV
IFX_TYPE_BOOL	BOOLEAN
IFX_TYPE_IMPEXP	IMPEXP
IFX_TYPE_IMPEXPBIN	IMPEXPBIN
IFX_TYPE_CLOB	CLOB
IFX_TYPE_BLOB	BLOB

Mapping for field types

The following table shows the mapping supported between the types defined for the *ifxtype* parameter in the UDTMetaData.setFieldType() method and the Java™ data types as they appear in the Java™ class file. Data types not shown in this table are not supported within the opaque type.

ifxtype parameter type from com.informix.lang.IfxTypes	Java™ data type
IFX_TYPE_BIGINT	long
IFX_TYPE_BIGSERIAL	long
IFX_TYPE_CHAR	java.lang.String
IFX_TYPE_SMALLINT	short
IFX_TYPE_INT	int
IFX_TYPE_FLOAT	double
IFX_TYPE_SMFLOAT	float[1]
IFX_TYPE_DECIMAL	java.lang.BigDecimal
IFX_TYPE_SERIAL	int

ifxtype parameter type from com.informix.lang.IfxTypes	Java™ data type
IFX_TYPE_DATE	Date
IFX_TYPE_MONEY	java.lang.BigDecimal
IFX_TYPE_DATETIME	java.lang.Timestamp if starting qualifier is Year, Month, or Day; otherwise, java.lang.Time (see Field lengths and date-time data on page 262).
IFX_TYPE_INTERVAL	com.informix.lang.IfxIntervalYM if starting qualifier is Year or Month; otherwise, com.informix.lang.IfxIntervalDF (see Field lengths and date-time data on page 262).
IFX_TYPE_NCHAR	java.lang.String
IFX_TYPE_INT8	long
IFX_TYPE_SERIAL8	long
IFX_TYPE_BOOL	boolean
IFX_TYPE_CLOB	java.sql.Clob
IFX_TYPE_BLOB	java.sql.Blob

[1] This mapping is JDBC compliant. You can map IFX_TYPE_SMFLOAT data type (via the JDBC FLOAT data type) to the Java™ double data type for backward compatibility by setting the IFX_GET_SMFLOAT_AS_FLOAT environment variable to 1.

Field lengths and date-time data

When you set a field type to a date-time or interval data type by calling setFieldType(IFX_TYPE_DATETIME) or setFieldType(IFX_TYPE_INTERVAL), the driver maps the date-time field to either **java.sql.Timestamp** or **java.sql.Time**, depending on the encoded length you set by calling setFieldLength().

For example, given that the standard format for a date-time field is YYYY-MM-DD HH:MM:SS, the driver uses the following mapping algorithm:

- If the encoded length has the start code from *hour* or less, it is mapped to **java.sql.Time**.
- If the encoded length has the start code from *year* or less, it is mapped to **java.sql.TimeStamp**.

For intervals, the standards are either YYYY-MM or DD HH:MM:SS.*frac*. The mapping is as follows:

- If the encoded length has the start code from *day* or less, it is mapped to **com.informix.jdbc.IfxIntervalDF**.
- If the encoded length has the start code from *year* or less, it is mapped to **com.informix.jdbc.IfxIntervalYM**.

Convert internal Informix® data types

For your Java™ application to work with the internal server representation of HCL Informix® data types, use the **IfxToJavaType** class. For example, if your application is using the HCL Informix® Change Data Capture API, you can use the **IfxToJavaType** class to interpret the byte stream.

The IfxToJavaType class

The **IfxToJavaType** class handles all the HCL Informix® to Java™ data type conversions. Separate methods are provided for converting each Informix® data type.

The primitive data types of Java™ are boolean, char, byte, short, int, long, float, double. When ever possible, the conversion returns the primitive data type rather than the Object.

The following table shows the data types that can be converted between the Informix® data types to Java™ data types.

Table 24. Conversion between Informix® and Java™ data types

Informix® data types	Java™ data types
BIGINT	long
BYTE	int (as a large object ID, without an input stream)
CHAR (n) / CHARACTER (n)	string
DATE	java.sql.Date
DATETIME	java.sql.Timestamp
DATETIME	interval
DATETIME	string
DEC/DECIMAL (p,s)	java.lang.Bignum
DOUBLE PRECISION (n)	double
FLOAT	Same as DOUBLE PRECISION
INT8	long
INT/INTEGER	int
INTERVAL	interval
MONEY (p,s)	Same as DECIMAL
NUMERIC (p,s)	Same as DECIMAL
REAL	real
SERIAL (n)	int
SMALLFLOAT	Same as REAL
SMALLINT	short
TEXT	int (as a large object ID, without an input stream)

Table 24. Conversion between Informix® and Java™ data types (continued)

Informix® data types	Java™ data types
VARCHAR (m,r)	string

In addition to the conversion methods, the follow methods are also provided

- convertDateToDays()
- convertDaysToDate()
- rleapyear()
- widenByte()

The convertDateToDays() method

The convertDateToDays() method converts java.sql.Date to an **int** data type that encodes the number of days since January 1, 1900 as 1. Dates earlier than January 1, 1900 are encoded as negative numbers.

Method signature

```
public static int convertDateToDays (java.sql.Date dt)
```

Input parameter

dt

The java.sql date.

The convertDaysToDate() method

The convertDaysToDate() method converts days to year, date, or month. The convertDaysToDate() method handles negative days, interpreted as backwards from December 31, 1899 as 0. The convertDaysToDate() method interprets January 1, 1900 as 1. No dates before January 1, 0000 are allowed. The method relies on HCL Informix® to generate valid dates.

Method signature

```
public static java.sql.Date convertDaysToDate (int dt)
```

Input parameter

dt

The number of days since January 1, 1900 (as 1).

The IfxToJavaChar() method

The IfxToJavaChar() method converts the HCL Informix® CHAR (n) and CHARACTER (n) data types to the Java™ **string** data type. The conversion is achieved by creating a string from given bytes.

Method signature

```
public String IfxToJavaChar (byte b [], short prec,boolean encoption)
public String IfxToJavaChar (byte b [], boolean encoption)
```

```
public String IfxToJavaChar (byte b [], int offset, int length,
       short prec, boolean encoption)
public String IfxToJavaChar (byte b [], int offset, int length,
       boolean encoption)
public String IfxToJavaChar (byte b [], short prec, String dbEnc,
       boolean encoption)
public String IfxToJavaChar (byte b [], String dbEnc, boolean encoption)
       throws IOException
public String IfxToJavaChar (byte b [], int offset, int length,
       short prec,
       String dbEnc, boolean encoption)
public String IfxToJavaChar (byte b [], int offset, int length,
       String dbEnc, boolean encoption)
```

Input parameters

b

The bytes encoding data

dbEnc

The JDK encoding.

offset

The offset into byte array.

prec

The precision as received from Informix®.

length

The data length.

The IfxToJavaDate() method

The IfxToJavaDate() method converts the HCL Informix® DATE data type to the Java™ **java.sql.Date** data type.

Method signature

```
public static java.sql.Date IfxToJavaDate (byte b [], short prec)
public static java.sql.Date IfxToJavaDate (byte b [])
public static java.sql.Date IfxToJavaDate (byte b [], int offset,
       int length, short prec)
public static java.sql.Date IfxToJavaDate (byte b [], int offset)
```

Input parameters

b

The bytes encoding data

offset

The offset into byte array.

prec

The precision as received from Informix®.

length

The data length.

The IfxToJavaDateTime() method

The IfxToJavaDateTime() method converts the HCL Informix® DATETIME data type to the Java™ **java.sql.Timestamp** data type. The conversion path is Informix® to decimal to timestamp.

Method signature

```
public static java.sql.Timestamp IfxToJavaDateTime (byte b [], short prec)
public static java.sql.Timestamp IfxToJavaDateTime (byte b [], int offset,
    int length, short prec)
public static java.sql.Timestamp IfxToJavaDateTime (byte b [], int offset,
    int length, short prec, Calendar cal)
```

Input parameters

b

The bytes encoding data

offset

The offset into byte array.

prec

The precision as received from Informix®.

length

The data length.

The IfxToDateTimeUnloadString() method

The IfxToDateTimeUnloadString() method converts the HCL Informix® DATETIME data type to the Java™ **string** data type, which is in format compatible with SQL LOAD/UNLOAD format. The conversion path is Informix® to decimal to string.

Method signature

```
public static String IfxToDateTimeUnloadString (byte b [], int offset,
    int length, short prec)
```

Input parameters

b

The bytes encoding data

offset

The offset into byte array.

prec

The precision as received from Informix®.

length

The data length.

The IfxToJavaDecimal() method

The IfxToJavaDecimal() method converts the HCL Informix® DECIMAL data type to the Java™ **java.lang.Bignum** data type.

Method signature

```
public static java.math.BigDecimal IfxToJavaDecimal (byte b [], short prec)
public static java.math.BigDecimal IfxToJavaDecimal (byte b [], int offset,
    int length, short prec)
```

Input parameters

b

The bytes encoding data

offset

The offset into byte array.

prec

The precision as received from Informix®.

length

The data length.

The IfxToJavaDouble() method

The IfxToJavaDouble() method converts the HCL Informix® DOUBLE PRECISION data type to the Java™ **double** data type.

Method signature

```
public static double IfxToJavaDouble (byte b [], short prec)
public static double IfxToJavaDouble (byte b [])
public static double IfxToJavaDouble (byte b [], int offset, int length,
       short prec)
public static double IfxToJavaDouble (byte b [], int offset)
```

Input parameters

b

The bytes encoding data

offset

The offset into byte array.

prec

The precision as received from Informix®.

length

The data length.

The IfxToJavaInt() method

The IfxToJavaInt() method converts the HCL Informix® INTEGER data type to the Java™ **int** data type.

Method signature

```
public static int IfxToJavaInt (byte b [], short prec)
public static int IfxToJavaInt (byte b [])
public static int IfxToJavaInt (byte b [], int offset, int length,
       short prec)
public static int IfxToJavaInt (byte b [], int offset)
```

Input parameters

b

The bytes encoding data

offset

The offset into byte array.

prec

The precision as received from Informix®.

length

The data length.

The IfxToJavaInterval() method

The IfxToJavaInterval() method converts the HCL Informix® DATETIME data type to the Java™ **interval** data type. The conversion path is Informix® to decimal to interval.

Method signature

```
public static Interval IfxToJavaInterval (byte b [], short prec)
public static Interval IfxToJavaInterval (byte b [], int offset, int length,
    short prec)
```

Input parameters

b

The bytes encoding data

offset

The offset into byte array.

prec

The precision as received from Informix®.

length

The data length.

The IfxToJavaLongBigInt() method

The IfxToJavaLongBigInt() method converts the HCL Informix® BIGINT data type to the Java™ **long** data type.

Method signature

```
public static long IfxToJavaLongBigInt(byte b [], short prec)
public static long IfxToJavaLongBigInt(byte b [])
public static long IfxToJavaLongBigInt(byte buf [], int offset,
       int length, short prec)
public static long IfxToJavaLongBigInt(byte b[], int offset)
```

Input parameters

b* and *buff

The bytes encoding data

offset

The offset into byte array.

prec

The precision as received from Informix®.

length

The data length.

The IfxToJavaLongInt() method

The IfxToJavaLongInt() method converts the HCL Informix® INT8 data type to the Java™ **long** data type.

Method signature

```
public  static long IfxToJavaLongInt(byte b [], short prec)
public  static long IfxToJavaLongInt(byte b [])
public  static long IfxToJavaLongInt(byte buf [], int offset, int length,
        short prec)
public  static long IfxToJavaLongInt(byte buf [], int offset)
```

Input parameters

b* and *buf

The bytes encoding data

offset

The offset into byte array.

prec

The precision as received from Informix®.

length

The data length.

The IfxToJavaReal() method

The IfxToJavaReal() method converts the HCL Informix® REAL data type to the Java™ **real** data type.

Method signature

```
public static float IfxToJavaReal (byte b [], short prec)
public static float IfxToJavaReal (byte b [])
public static float IfxToJavaReal (byte b [], int offset,
       int length, short prec)
public static float IfxToJavaReal (byte b [], int offset)
```

Input parameters

b

The bytes encoding data

offset

The offset into byte array.

prec

The precision as received from Informix®.

length

The data length.

The IfxToJavaSmallInt() method

The IfxToJavaSmallInt() method converts the HCL Informix® SMALLINT data type to the Java™ **short** data type.

Method signature

```
public static short IfxToJavaSmallInt (byte b [], short prec)
public static short IfxToJavaSmallInt (byte b [])
public static short IfxToJavaSmallInt (byte b [], int offset,
       int length, short prec)
public static short IfxToJavaSmallInt (byte b [], int offset)
```

Input parameters

b

The bytes encoding data

offset

The offset into byte array.

prec

The precision as received from Informix®.

length

The data length.

The rleapyear() method

The rleapyear() method determines if the year is a leap year.

Method signature

```
public static final boolean rleapyear(int yr)
```

The widenByte() method

The widenByte() method moves BYTE into the **short** data type in such a way that the high bit is not propagated.

Method signature

```
protected static final short widenByte(byte b)
```

Index

J

S

W

X