IBM Informix
JDBC Driver

Programmer's Guide

UNIX and Microsoft Windows Environments

IBM Informix Dynamic Server, Version 7.x
IBM Informix Dynamic Server, Workgroup and Developer Editions, Version 7.x
IBM Informix Dynamic Server with Advanced Decision Support and Extended Parallel Options , Version 8.x
IBM Informix Extended Parallel Server, Version 8.3
IBM Informix Extended Parallel Server, Version 8.4
IBM Informix Dynamic Server with Universal Data Option, Version 9.x
IBM Informix Dynamic Server, Version 9.2x and later
IBM Informix OnLine Dynamic Server, Version 5.x
IBM Informix SE, Versions 5.x and 7.2x

Note:
Before using this information and the product it supports, read the information in the appendix entitled "Notices."

Table of Contents

Chapter 2   **Connecting to the Database**

Introduction

In This Introduction

This introduction provides:

- An overview of IBM Informix Java™ documentation
- An overview of the information in this manual
- A description of the conventions used in this manual
- A list of new features in JDBC 2.21 JC4

IBM Informix Java Documentation

The following table presents common Java programming tasks and tells where to find their documentation.

To Do This	Consult This Document
Set up your environment to run a Java application	
Install the JDK	*J/Foundation Developer's Guide*
	Sun Microsystems Web site also has documentation.
Install a Java-enabled server	*J/Foundation Developer's Guide*
	Informix by Example Web site has a step-by-step article: http://examples.informix.com
Configure your environment	*J/Foundation Developer's Guide*
	Informix by Example Web site
Install a JDBC client	*IBM Informix JDBC Driver Programmer's Guide* (this manual)

(1 of 3)

To Do This	Consult This Document
Make sure a client on a different computer can communicate with the database server (connectivity)	*IBM Informix Dynamic Server Administrator's Guide*
Perform basic database operations	
From a client, using JDBC API	*IBM Informix JDBC Driver Programmer's Guide* (this manual)
From a client, using embedded SQL	*IBM Informix Embedded SQLJ User's Guide*
In the database server, using JDBC and SQL	*J/Foundation Developer's Guide*
Create opaque and distinct types	
Understand concepts	*IBM Informix User-Defined Routines and Data Types Developer's Guide*
Create using the client JDBC driver	*IBM Informix JDBC Driver Programmer's Guide* (this manual)
	For differences between server and client JDBC drivers, see the JDBC Driver chapter in *J/Foundation Developer's Guide* (for example, the UDT/UDR Manager classes are available only from a client in the 9.3 and earlier server releases).
Create in the database server (using the built-in server JDBC driver)	*DataBlade Developer's Kit User's Guide*
	J/Foundation Developer's Guide
Work with smart large objects	*IBM Informix JDBC Driver Programmer's Guide* (this manual)
	The 2.2 version includes concepts and usage information for both client-side and server-side operations on smart large objects.
Store and retrieve XML documents	*IBM Informix JDBC Driver Programmer's Guide* (this manual) (supported only on the client in the 9.3 and earlier server releases)

(2 of 3)

To Do This	Consult This Document
Use IBM Informix **GLS for internationalization**	*IBM Informix JDBC Driver Programmer's Guide* (this manual)
	J/Foundation Developer's Guide for differences between server and client JDBC driver
Debug a Java application	*IBM Informix JDBC Driver Programmer's Guide* (this manual)
Access nonrelational data	*Java Virtual-Table Interface Programmer's Manual*

(3 of 3)

About This Manual

This guide describes how to install, load, and use IBM Informix JDBC Driver to connect to an Informix database from within a Java application or applet. You can also use IBM Informix JDBC Driver for writing user-defined routines that are executed in the server.

This section discusses the organization of the manual, the intended audience, and the associated software products you must have to use IBM Informix JDBC Driver.

Organization of This Manual

This manual includes the following chapters:

- This introduction discusses manual organization and conventions, explains what is and what is not documented in this manual, and, in addition, lists the new features in JDBC 2.21.JC4.

- Chapter 1, "Getting Started," describes IBM Informix JDBC Driver and the JDBC application programming interface (API) in general. It provides information essential for programmers to start using the product immediately, including how to install and load the driver.

- Chapter 2, "Connecting to the Database," explains in more detail the Informix-specific information needed to use IBM Informix JDBC Driver to connect to an Informix database. This information includes how to create a connection to an Informix database, how to set connection properties, and how to use various connection methods supported by IBM Informix JDBC Driver.

- Chapter 3, "Performing Database Operations," explains how to query the database, handle errors and transactions, and use XML documents.

- Chapter 4, "Working With Informix Types," explains how to use the Informix-specific data types supported in IBM Informix JDBC Driver.

- Chapter 5, "Working with Opaque Types," explains how to use the Informix extensions for opaque types and user-defined routines.

- Chapter 6, "Internationalization and Date Formats," explains how IBM Informix JDBC Driver extends the Java JDK 1.2 international-ization features by providing access to Informix databases that are based on different locales and code sets.

- Chapter 7, "Tuning and Troubleshooting," provides troubleshooting tips to solve programming errors and problems with the driver. It also describes browser security issues when you use IBM Informix JDBC Driver in a Java applet.

- Appendix A, "Sample Code Files," provides an overview of the files installed with IBM Informix JDBC Driver that contain the examples referred to in the guide.

- Appendix B, "DataSource Extensions," lists the Informix DataSource interface extensions.

- Appendix C, "Mapping Data Types," explains how to map between data types defined in a Java program and the data types supported by the Informix database server.

A Notices appendix that contains information about IBM products, services, and features is included. A glossary of relevant terms and a list of error messages follow the chapters, and an index directs you to areas of particular interest.

Material Not Covered

The JDBC manual will not duplicate information about new features documented in the IBM Informix Dynamic Server doc set, but will document JDBC-specific info and reference the IDS manuals that describe the feature in detail.

In addition, this manual will not discuss SQL features implemented in the IDS and XPS servers and implicitly supported by JDBC.

This guide does not describe all the interfaces, classes, and methods of the JDBC API and does not provide detailed descriptions of how to use the JDBC API to write Java applications that connect to Informix databases. The examples in the guide provide enough information to show how to use IBM Informix JDBC Driver but do not provide an extensive description of the JDBC API.

For more information about the JDBC API, visit the Java Web site at http://java.sun.com/products/jdbc/index.html.

This manual describes 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 IBM Informix JDBC Driver.

This manual does not describe interfaces and limitations that are unique to the server-side version of the IBM Informix JDBC Driver; that information is covered in the *J/Foundation Developer's Guide*. For more information, see "Client- and Server-Side JDBC Drivers" on page 1-14.

Types of Users

This guide is for Java programmers who use the JDBC API to connect to Informix databases using IBM Informix JDBC Driver. To use this guide, you should know how to program in Java and, in particular, understand the classes and methods of the JDBC API.

Software Dependencies

To use IBM Informix JDBC Driver to connect to an Informix database, you must use one of the following Informix database servers:

- IBM Informix Dynamic Server, Version 7.x
- IBM Informix Dynamic Server, Workgroup and Developer Editions, Version 7.x
- IBM Informix Dynamic Server with Advanced Decision Support and Extended Parallel Options , Version 8.x
- IBM Informix Extended Parallel Server, Version 8.3 or later
- IBM Informix Dynamic Server with Universal Data Option, Version 9.x
- IBM Informix Dynamic Server, Version 9.2x or later
- IBM Informix OnLine Dynamic Server, Version 5.x
- IBM Informix SE, Versions 5.x and 7.2x

You must also use Java Development Kit (JDK) Version 1.2 or later.

Assumptions About Your Locale

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

The examples in this manual are written with the assumption that you are using the default locale, **en_us.8859-1**. This locale supports U.S. English format conventions for date, time, and currency. In addition, this locale supports the ISO 8859-1 code set, which includes the ASCII code set plus many 8-bit characters such as é, è, and ñ.

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

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

Documentation Conventions

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

- ■ Typographical conventions
- ■ Icon conventions

Typographical Conventions

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

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

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

Tip: *The text and many of the examples in this manual show routine and data type names in mixed lettercasing (uppercase and lowercase). Because IBM Informix Dynamic Server is case insensitive, you do not need to enter routine names exactly as shown: you can use uppercase letters, lowercase letters, or any combination of the two.*

Icon Conventions

Throughout the documentation, you will find text that is identified by several different types of icons. This section describes these icons.

Comment Icons

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

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

Platform Icons

Platform icons identify paragraphs that contain platform-specific information.

Icon	Description
UNIX	Identifies information that is specific to the UNIX environment.
Windows	Identifies information that is specific to the Windows environment.

These icons can apply to a row in a table, one or more paragraphs, or an entire section. A ♦ symbol indicates the end of the platform-specific information.

New Features in This Release

The IBM Informix JDBC Driver, Version 2.21 JC4, introduces the following features, supported by IBM Informix Dynamic Server, Version 9.40.

■ JDBC Driver support for 32K LVARCHAR feature

IBM Informix Dynamic Server, Version 9.40 supports an optional parameter for LVARCHAR to specify the desired maximum length of 32739. The following is an example of acceptable syntax:

```
CREATE TABLE   TAB1 (COL1 LVARCHAR(32739))
```

Users can specify any value, from 1 to 32739 bytes. For compatibility with existing SQL, if the optional maximum size parameter is omitted, then a size of 2KB will be used.

■   *Describe Input* Parameter
(JDBC support for java.sql.ParameterMetaData interface)

The IBM Informix JDBC Driver, Version 2.21.JC4, implements a new JDBC 3.0 interface called **java.sql.ParameterMetaData** to describe input parameters in prepared statements. The **getParameterMeta-Data()** method retrieves the metadata for a particular statement. Besides supporting the JDBC 3.0 ParameterMetaData interface, the IBM Informix JDBC Driver implements additional methods to this interface to extend its functionality.

For additional information about this feature, see "JDBC Support for Describe Input" on page 3-21.

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

■   Support for Multiple UDR OUT Parameters

In JDBC, a CallableStatement object provides a standard way to call or execute user-defined procedures and functions and stored procedures (hereafter referred to as UDRs) for all relational databases. Results are returned as a resultset or OUT parameter.

The IBM Informix Dynamic Server prior to Version 9.40 supports a single OUT parameter in user-defined functions, but not an OUT parameter in user-defined procedures. Version 9.40 adds support for multiple OUT parameters in user-defined functions and user-defined procedures.

For additional information about this feature, see "Using Call-ableStatement OUT Parameters" on page 3-8.

■ Enhancement to the JDBC **ReadOnly** method for CTS Compliance

IBM Informix Dynamic Server, Version 9.40, does not support read-only connections.

Previous versions of the JDBC driver threw an unsupported method exception when **setReadOnly()** was called. The new implementation of the **setReadOnly()** method from the **java.sql.Connection** interface now accepts the value passed to it by the calling process as *no op* (returns to the calling process without any interaction to the server) instead of throwing an exception.

This change has been made to synchronize the functionality present in the IBM DB2 JDBC driver to that of the IBM Informix JDBC driver and also to achieve a higher level of compliance to the Sun Conformance Test (CTS).

For additional information about this feature, see the *Important* note after "Unsupported Methods" on page 3-19.

■ New connection properties: **IFX_LOCK_MODE_WAIT** and **IFX_ISOLATION_LEVEL**

Application servers can use these properties for obtaining connections using IBM Informix JDBC Driver.

An application can use the **IFX_LOCK_MODE_WAIT** connection property to override the default that defines how the database server tries to access a locked row or table.

An application can use the **IFX_ISOLATION_LEVEL** connection property to define the degree of concurrency among processes that attempt to access the same rows simultaneously.

For additional information about this feature, see "Using a Data-Source Object" on page 2-5 and "Format of Database URLs" on page 2-10. In addition, see "Getting and Setting Informix Connection Properties" on page B-4.

Additional Documentation

This section describes the following parts of the documentation set:

- Online manuals
- Other online files

Online Manuals

The following related IBM Informix documents complement the information in this manual. You can find these manuals at http://www-3.ibm.com/software/data/informix/pubs/library/.

- IBM Informix Java products are documented in the following additional manuals:

  □ *IBM Informix User-Defined Routines and Data Types Developer's Guide* contains introductory information about writing opaque types and user-defined routines in Java.

  □ *J/Foundation Developer's Guide* contains detailed information about writing user-defined routines in Java.

  □ *IBM Informix Embedded SQLJ User's Guide* contains information about using IBM Informix Embedded SQLJ to embed SQL statements in your Java programs.

- If you have never used Structured Query Language (SQL), read the *IBM Informix Guide to SQL: Tutorial*. It provides a tutorial on SQL as it is implemented by IBM Informix products. It also describes the fundamental ideas and terminology for planning and implementing a relational database.

- A companion volume to the *Tutorial*, the *IBM Informix Guide to SQL: Reference*, includes details of the IBM Informix system catalog tables, describes Informix and common environment variables that you should set, and describes the column data types that Informix database servers support.

- The *IBM Informix Guide to SQL: Syntax* provides information about SQL syntax as it is implemented by IBM Informix products.

- *IBM Informix Error Messages* is useful if you do not want to look up your error messages online.

Other Online Files

The following sections describe the online files that supplement the information in this manual. Please examine these files before you begin using your database server.

This section describes two sources of online information:

- Documentation notes and release notes
- Javadoc pages

Documentation notes, which contain additions and corrections to the manuals, and release notes are located in the directory where the product is installed. The following online files supplement the information in this manual.

Online File	Purpose
jdbcrel.htm	The release notes describe any special actions required to configure and use IBM Informix JDBC Driver on your computer. Additionally, this file contains information about any known problems and their workarounds.
jdbcdoc.htm	The documentation notes describe features not covered in the manuals or modified since publication.

After installation, these files are located in the following directories:

UNIX

- $*JDBCLOCATION*/**doc/release**, where $*JDBCLOCATION* refers to the directory where you installed IBM Informix JDBC Driver. ♦

Windows

- **%***JDBCLOCATION*%**doc\\release**, where **%***JDBCLOCATION*% refers to the directory where you installed IBM Informix JDBC Driver. ♦

Please examine these files because they contain vital information about application and performance issues.

The javadoc pages describe the Informix extension classes, interfaces, and methods in detail.

UNIX

Javadoc pages are located in $*JDBCLOCATION*/**doc/javadoc**, where $*JDBCLOCATION* refers to the directory where you installed IBM Informix JDBC Driver. ♦

Windows

Javadoc pages are located in **%*JDBCLOCATION*%\\doc\\javadoc**, where **%*JDBCLOCATION*%** refers to the directory where you installed IBM Informix JDBC Driver. ♦

For more information about the JDBC API, visit the Sun Microsystems site at http://java.sun.com/j2se/1.4/docs/guide/jdbc/.

Compliance with Industry Standards

The American National Standards Institute (ANSI) has established a set of industry standards for SQL. IBM Informix SQL-based products are fully compliant with SQL-92 Entry Level (published as ANSI X3.135-1992), which is identical to ISO 9075:1992. In addition, many features of Informix database servers comply with the SQL-92 Intermediate and Full Level and X/Open SQL CAE (common applications environment) standards.

IBM Welcomes Your Comments

To help us with future versions of our manuals, let us know about any corrections or clarifications that you would find useful. Include the following information:

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

Send electronic mail to us at the following address:

docinf@us.ibm.com

This address is reserved for reporting errors and omissions in our documentation. For immediate help with a technical problem, contact Customer Services.

Getting Started

In This Chapter

This chapter provides an overview of IBM Informix JDBC Driver and the JDBC API. It includes the following sections:

- What Is JDBC?
- What Is a JDBC Driver?
- Overview of IBM Informix JDBC Driver
- Installing the Driver
- Using the Driver in an Application
- Using the Driver in an Applet
- Installing the Driver

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.

The JDBC API is consistent with the style of the core Java interfaces and classes, such as **java.lang** and **java.awt**. The following table describes the interfaces, classes, and exceptions that make up the JDBC API.

Interface, Class, or Exception	Description
java.sql.CallableStatement	Interface used to execute stored procedures
java.sql.Connection	Interface used to establish a connection to a database
	SQL statements run within the context of a connection.
java.sql.DatabaseMetaData	Interface used to return information about the database
java.sql.Driver	Interface used to locate the driver for a particular database management system
java.sql.PreparedStatement	Interface used to send precompiled SQL statements to the database server and obtain results
java.sql.ResultSet	Interface used to process the results returned from executing an SQL statement
java.sql.ResultSetMetaData	Interface used to return information about the columns in a **ResultSet** object
java.sql.Statement	Interface used to send static SQL statements to the database server and obtain results
java.sql.Date	Subclass of **java.util.Date** used for the SQL DATE data type
java.sql.DriverManager	Class used to manage a set of JDBC drivers
java.sql.DriverPropertyInfo	Class used to discover and supply properties to a connection
java.sql.Time	Subclass of **java.util.Date** used for the SQL TIME data type
java.sql.TimeStamp	Subclass of **java.util.Date** used for the SQL TIMESTAMP data type

(1 of 3)

Interface, Class, or Exception	Description
java.sql.Types	Class used to define constants that are used to identify standard SQL data types, such as VARCHAR, INTEGER, and DECIMAL
java.sql.Array	Class used to identify SET, MULTISET, and LIST data types These types can also be identified as collections.
java.sql.Blob	Class used to identify BLOB data types
java.sql.Clob	Class used to identify CLOB data types
java.sql.String	Class used to identify long text data types such as LVARCHAR
java.sql.Struct	Class used to identify named and unnamed row data types
java.sql.DataTruncation	Exception thrown or warning reported when data has been truncated
java.sql.SQLData	Class used to support opaque types, distinct types, and named and unnamed row data types
java.sql.SQLInput	Class used to support retrieval of opaque types and named and unnamed row data types
java.sql.SQLOutput	Class used to support writing of opaque types and named and unnamed row data types
java.sql.SQLException	Exception that provides information about a database error
java.sql.SQLWarning	Warning that provides information about a database warning
javax.sql.DataSource	Interface used as a factory for **Connection** objects

(2 of 3)

Interface, Class, or Exception	Description
javax.sql.ConnectionPoolData-Source	Interface used as a factory for **Connection** objects that provide hooks for connection pool management
javax.sql.PooledConnection	Interface whose instances are **Connection** objects that provide hooks for connection pool management
javax.sql.ConnectionEvent	Class that provides information about the source of a connection-related event
javax.sql.ConnectionEventListener	Interface whose instances are objects that register to receive events generated by a **PooledConnection** object
javax.sql.XADataSource	Interface used as a factory for **XAConnection** objects that provide support for distributed transactions
javax.sql.XAConnection	Interface whose instances are **XAConnection** objects that provide support for distributed transactions

(3 of 3)

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.

For more information about the JDBC API, visit the Sun Microsystems Web site at http://java.sun.com/products/jdk/1.2/docs/guide/jdbc/.

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 be 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 allows the client applications to connect directly to the database server.

Overview of IBM Informix JDBC Driver

IBM Informix JDBC Driver is a native-protocol, pure-Java driver (Type 4). This means that when you use IBM Informix JDBC Driver in a Java program that uses the JDBC API to connect to an Informix database, your session connects directly to the database or database server, without a middle tier.

IBM Informix JDBC Driver is based on Version 2.0 of the JDBC API.

Classes Implemented in IBM Informix JDBC Driver

To support **DataSource** objects, connection pooling, and distributed transactions, IBM Informix JDBC Driver provides classes that implement interfaces and classes described in the JDBC 2.0 *Standard Extension Specification* from Sun Microsystems.

Informix Classes That Implement Java Interfaces

The following table lists the Java interfaces and classes and the 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

Informix Classes that Extend the Java Specification

To support the Informix implementation of SQL statements and data types, IBM Informix JDBC Driver provides classes that extend the JDBC 2.0 *Standard Extension Specification*. 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.sql.Connection	com.informix.jdbc.IfmxConnection	Opaque, distinct, and complex types
java.sql.Statement	com.informix.jdbc.IfmxStatement	Single result sets, autofree mode, statement types, and SERIAL data type processing
java.lang.Object	com.informix.lang.IfxTypes	Representing data types
java.lang.Object	com.informix.jdbc.IfxStatementTypes	Representing SQL statements
java.sql.CallableStatement	com.informix.jdbc.IfmxCallableStatement	Parameter processing with Informix types
java.sql.PreparedStatement	com.informix.jdbc.IfmxPreparedStatement	Parameter processing with Informix types
java.sql.ResultSet	com.informix.jdbc.IfmxResultSet	Informix interval data types
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.SQLOutput	com.informix.jdbc.IfmxComplexSQLOutput	Opaque, distinct, and complex types
java.lang.Object	com.informix.jdbc.Interval	Interval qualifiers and some common methods for the next two classes (base class for the next two)
java.lang.Object	com.informix.jdbc.IntervalYM	Interval year-to-month

JDBC Interface or Class	Informix Class	Adds Methods or Constants for...
java.lang.Object	com.informix.jdbc.IntervalDF	Interval day-to-fraction
java.lang.Object	com.informix.jdbc.IfxSmartBlob	Access methods for smart large objects
java.sql.Blob	com.informix.jdbc.IfxBblob	Binary large objects
java.sql.Clob	com.informix.jdbc.IfxCblob	Character 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.SQLInput	com.informix.jdbc.IfmxUDTSQLInput	Opaque, distinct, and complex types
java.sql.SQLOutput	com.informix.jdbc.IfmxUDTSQLOutput	Opaque, distinct, and complex types

(2 of 2)

Informix Classes That Provide Support Beyond the Java Specification

A number of 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

Using UDTManager and UDRManager Classes with JDK 1.4

In previous releases, the **UDTManager** and **UDRManager** helper classes included in **ifxtools.jar** were not accessible from a packaged class. As of IBM Informix JDBC Driver 2.21.JC3, all these classes are in the **udtudrmgr** package. For backwards compatibility, unpackaged versions of these classes are also included.

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

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

Files in IBM Informix JDBC Driver

IBM Informix JDBC Driver is released in a new program file, **setup.jar**. For instructions on how to install the driver, refer to "Installing the Driver" on page 1-14.

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/ifxjdbc-g.jar**

Debug version of **ifxjdbc.jar**

The file is compiled with the **-g** 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/ifxtools-g.jar**

Debug version of **ifxtools.jar**. The file is compiled with the **-g** 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.

■ **lib/ifxjdbcx.jar**

Includes the implementation of DataSource-, connection pooling-, and XA-related class files

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

■ **lib/ifxjdbcx-g.jar**

Debug version of **ifxjdbcx.jar**

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

■ **lib/ifxsqlj.jar**

Includes the classes for runtime support of SQLJ programs

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

■ **lib/ifxsqlj-g.jar**

Debug version of **ifxsqlj.jar**

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

■ **lib/xerces.jar**

Includes Java API classes for XML parsing.

■ **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 Appendix A, "Sample Code Files."

■ **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 IBM Informix JDBC Driver.

Client- and Server-Side JDBC Drivers

The IBM Informix JDBC Driver exists in two versions: The client-side driver is intended for client Java applications accessing an Informix database server. The server-side driver is intended for creating and executing user-defined routines (UDRs) and applications that are written in Java to run directly in the database server.

The server-side driver derives from the client-side driver; thus the two drivers share most features.

This manual describes the public JDBC interfaces and subprotocols that are supported in both the client- and server-side versions of the driver. The *J/Foundation Developer's Guide* describes the interfaces and subprotocols that the IBM Informix JDBC Driver provides specifically for server-side JDBC applications, as well as restrictions that apply to server-side JDBC applications.

Installing the Driver

You can obtain IBM Informix JDBC Driver, Version 2.21 from the product CD or from www.ibm.com/software/data/developer/informix. The contents of the CD or web download are as follows:

- **setup.jar**
- **doc/jdbcdoc.htm**
- **doc/jdbcrel.htm**
- **doc/install.txt**

The documentation directory **/doc** contains documentation notes and release notes in **.html** format and install notes in text format. Refer to these documents for any new information not available in this manual. You can install IBM Informix JDBC Driver in the following modes:

- **Graphical mode.** The graphical mode launches an install program in a graphical user interface.
- **Console mode.** The console mode sends messages and information to the console instead of displaying them in a GUI.
- **Silent mode.** The silent mode requires no interaction to run the install.

Tip: *The following sections describe the three installation modes for all platforms from the product CD-ROM. If you have obtained IBM Informix JDBC Driver as a .zip file from www.ibm.com/software/data/developer/informix instead of from the CD-ROM, unzip the file to a directory on your computer and substitute the name of that directory for* `<CD-ROM dir>` *in the procedure below.*

Important: *You can enable logging of install events by specifying the* ***-log*** *option followed by arguments for file type, event type, and file location. For more information about logging, see* *"Logging Install Events" on page 1-17.*

Installing in Graphical Mode

This section describes how to install IBM Informix JDBC Driver in graphical mode.

To install IBM Informix JDBC Driver in graphical mode

1. Load the CD into the CD-ROM drive.

2. At the command prompt, execute the following command to launch the GUI:

   ```
   java -cp <CD-ROM dir>/setup.jar run
   ```

3. To continue through the copyright statement, click **Next**.

4. Choose the option:

   ```
   I accept the terms in the license agreement.
   ```
   Click **Next**.

5. Browse to specify a directory in which to install the IBM Informix JDBC Driver or accept the default directory.

   On a Windows platform, the default directory will be similar to:

   ```
   C:\Program Files\IBM\Informix JDBC Driver 2.21
   ```

6. You will see the following message:

   ```
   InstallShield Wizard has successfully installed IBM Informix JDBC
   Driver.
   ```

   Click **Finish** to exit the wizard.

Installing in Console Mode

This section describes how to install IBM Informix JDBC Driver in console mode.

To install IBM Informix JDBC Driver in console mode

1. Load the CD into the CD-ROM drive.

2. At the command prompt, execute the following command to launch the console mode installation:

   ```
   java -cp <CD-ROM dir>/setup.jar run -console
   ```

   You will see the copyright statement in the console screen.

3. At the following prompt, enter your selection:

   ```
   Press Enter to continue viewing the license agreement, or, Enter
   "1" to accept the agreement, "2" to decline it or "99" to go back
   to the previous screen.
   ```

4. Specify a directory in which to install the IBM Informix JDBC Driver or accept the default directory.

   ```
   IBM Informix JDBC Driver will be installed in the following
   location.
   ```

   The console shows the install directory.

   The screen notifies you that the uninstaller is being added to the directory.

5. When you see:

   ```
   The InstallShield Wizard has successfully installed IBM Informix
   JDBC Driver
   ```

   Press ENTER to close the wizard.

Installing in Silent Mode

This section describes how to install IBM Informix JDBC Driver in silent mode.

To install Informix JDBC Driver in silent mode

1. Load the CD into the CD-ROM drive.

2. At the command prompt, execute the following command:

```
java -cp setup.jar run -silent -P
product.installLocation=<destination-dir>
```

Where `<destination-dir>` is where you want to install the JDBC Driver.

The installation is complete once the command has finished executing.

Logging Install Events

For each of the installation modes, you can enable logging by specifying the **-log** option when you execute the command to install the driver. Add the -log option followed by arguments for file type, event type, and file location. For instance, to install the IBM Informix JDBC Driver in graphical mode and retain a log of the event, you would execute the following:

```
java -cp setup.jar run -log #![filename] @ [event type];[event
type]
```

Where # echoes the display to standard output, `![filename]` is your name for the log file, and `@` precedes the event type. You can omit the `[filename]` argument to save the log information to the default file name.

A table of common event types follows.

Argument	Event Type
err	Errors
wrn	Warning
msg1	Primary events
msg2	Secondary events

(1 of 2)

Argument	Event Type
dbg	Debug events
ALL	All events
NONE	Disables logging and clears the log file

(2 of 2)

For example, the following of the command installs IBM Informix JDBC Driver in the graphical mode and logs all events to **/tmp/jdbcinstall.log**:

```
java -cp setup.jar run -log  !/tmp/jdbcinstall.log  @ ALL
```

The following command installs IBM Informix JDBC Driver in silent mode and logs error events to **/tmp/jdbcinstall.log**:

```
java -cp setup.jar run -silent -P product.installLocation=< > -log
!"/tmp/jdbcinstall.log" @err
```

Using the Driver in an Application

To use IBM 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 pathname of **ifxjdbc.jar** to **CLASSPATH:**

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

To use the version of the driver that supports debugging, specify **ifxjdbc-g.jar** instead of **ifxjdbc.jar**.

To add localized message support, specify **ifxlang.jar** as well:

```
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 use the version of the driver that supports debugging, specify
ifxjdbc-g.jar instead of **ifxjdbc.jar**.

To add localized message support, specify **ifxlang.jar** as well:

```
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 pathname of **ifxjdbc.jar** to **CLASSPATH**:

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

To use the version of the driver that supports debugging, specify
ifxjdbc-g.jar instead of **ifxjdbc.jar**.

To add localized message support, specify **ifxlang.jar** as well:

```
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 use the version of the driver that supports debugging, specify
ifxjdbc-g.jar instead of **ifxjdbc.jar**.

To add localized message support, specify **ifxlang.jar** as well:

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

For more information on the **jar** utility, refer to the Java documentation at
http://java.sun.com/j2se/1.3/docs/guide/jar/.

Using the Driver in an Applet

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

To use IBM Informix JDBC Driver in an applet

1. Install **ifxjdbc.jar** in the same directory as your applet class file.

   To use the version of the driver that supports debugging, install the **ifxjdbc-g.jar** instead of **ifxjdbc.jar**.

2. Specify **ifxjdbc.jar** in the ARCHIVE attribute of the APPLET tag in your HTML file, as shown in the following example:

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

   To use the version of the driver that supports debugging, specify **ifxjdbc-g.jar** instead of **ifxjdbc.jar**.

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 as well.*

See your Web server documentation for information on installing files in the root directory.

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

- **sqlhosts file and LDAP server access.** The host name and port number properties 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 are always required, unless your applet is using the HTTP proxy server. For more information on the HTTP proxy server, see "Using an HTTP Proxy Server" on page 2-32.

■ **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 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 .

Uninstalling the Driver

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

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

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

To uninstall IBM Informix JDBC Driver in graphical mode

Execute the following command to launch the uninstall program in GUI mode:

```
java -cp <destination-dir>/_uninst/uninstall.jar run
```

Where <destination-dir> is the directory in which you installed the IBM Informix JDBC Driver.

The **Uninstall** program guides you through the uninstallation.

To uninstall IBM Informix JDBC Driver in console mode

Execute the following command to launch the uninstall program in console mode:

```
java -cp <destination-dir>/_uninst/uninstall.jar run -console
```

Where <destination-dir> is the directory in which you installed the IBM Informix JDBC Driver.

The messages in the console screen guide you through the uninstallation.

To uninstall IBM Informix JDBC Driver in silent mode

Execute the following command to launch the uninstall program in silent mode:

```
java -cp <destination-dir>/_uninst/uninstall.jar run -silent
```

Where <destination-dir> is the directory in which you installed the IBM Informix JDBC Driver.

The **Uninstall** program does not send you any messages but uninstalls the driver.

Connecting to the Database

In This Chapter

This chapter explains the information you need to use IBM Informix JDBC Driver to connect to an Informix database. The chapter includes the following sections:

You must first establish a connection to an 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 IBM 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 impacting the application code.

A **DataSource** object can also provide support for connection pooling and distributed transactions. In addition, Enterprise Java Beans 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

Loading IBM Informix JDBC Driver

To load IBM 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.

Once you have loaded IBM Informix JDBC Driver, you are ready to connect to an Informix database or database server.

Windows

If you are writing an applet to be viewed with Microsoft Internet Explorer, you might need to explicitly register IBM 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 IBM Informix JDBC Driver twice, which does not cause a problem. ♦

Using a DataSource Object

For information about how and why to use a **DataSource** object, see the JDBC 2.0 *Standard Extension Specification* provided by Sun Microsystems, available on the Web at http://java.sun.com/products/jdk/1.2.

IBM 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 Appendix B 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.

(1 of 3)

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 name of the user who wants to connect to the Informix database or database server You must specify both the user and the password.
PASSWORD	password	String	Yes	The password of the user You must specify both the user and the password.
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

(2 of 3)

Informix Connection Property	DataSource Property	Data Type	Required?	Description
IFX_LOCK_MODE _WAIT	None	int	No	Application can use this property to override the default server process for accessing a locked row or table
				For more information, see "Getting and Setting Informix Connection Properties" on page B-4.
IFX_ISOLATION_L EVEL	None	String	No	Defines the degree of concurrency among processes that attempt to access the same rows simultaneously
				For more information, see "Getting and Setting Informix Connection Properties" on page B-4.

(3 of 3)

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

If an LDAP (Lightweight Directory Access Protocol) 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 using the standard **DataSource** properties. For more information, see "Dynamically Reading the Informix sqlhosts File" on page 2-23.

For a list of supported environment variables (properties), see "Using Informix Environment Variables" on page 2-16. For a list of Informix **DataSource** extensions, which allow you to define environment variable values and connection pool tuning parameters, see Appendix B, "DataSource Extensions." The driver does not consult the user's environment to determine environment variable values.

For information about the **ConnectionPoolDataSource** object, see "Using a Connection Pool" on page 7-10.

You can use a DataSource object with High-Availability Data Replication. For more information, see "Using High-Availability Data Replication" on page 2-28.

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());
}
```

The following are examples of the **IFX_LOCK_MODE_WAIT** connection property using 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 ();
```

Using the DriverManager.getConnection() Method

To create a connection to an 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 an Informix database, and process the results.

The **DriverManager** class keeps track of 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 "Specifying Properties" on page 2-14 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** 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 IBM Informix JDBC Driver is **tcp**. Shared memory and other connection types are not supported. For more information about connection types, see the Administrator's Guide for your database server.*

> **Important:** *Not all methods of the **Connection** interface are supported by IBM Informix JDBC Driver. For a list of unsupported methods, see "Unsupported Methods" on page 3-19.*

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

For connections from a client, use the following format to specify database URLs:

```
jdbc:informix-sqli://[{ip-address|host-name}:port-number][dbname]:
    INFORMIXSERVER=server-name;[user=user;password=password]
    [;name=value[;name=value]...]
```

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

```
jdbc:informix-direct://[/dbname:;[user=user;password=password] ]
    [;name=value[;name=value]...]
```

In the preceding syntax:

- Curly brackets ({}) together with vertical lines (|) denote more than one choice of variable.

- *Italics* denote a variable value.

- Brackets ([]) denote an optional value.

- Words or symbols not enclosed in brackets are required (INFORMIXSERVER=, for example).

Blank spaces are not allowed in the database URL.

For example, on the client you might use:

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

On the server, you might use:

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

Important: *Connections using server-side JDBC have different syntax. For details, see the J/Foundation Developer's Guide or the release notes for your version of the database server.*

The following table describes the variable parts of the database URL and the equivalent Informix connection properties.

Informix Connection Property	Database URL Variable	Required?	Description
IFXHOST	*ip-address* *host-name*	Yes for client-side JDBC, unless **SQLH_TYPE** is defined or **IFXHOST** is used; no for server-side JDBC	The IP address or the host name of the computer running the Informix database server
PORTNO	*port-number*	Yes for client-side JDBC, unless **SQLH_TYPE** is defined or **PORTNO** is used; no for server-side JDBC	The port number of the Informix database server. The port number is listed in the **/etc/services** file.
DATABASE	*dbname*	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	*server-name*	Yes	The name of the Informix database server to which you want to connect
USER	*user*	Yes	The name of the user who wants to connect to the Informix database or database server. You must specify both the user and the password or neither. If you specify neither, the driver calls **System.getProperty()** to obtain the name of the user currently running the application, and the client is assumed to be trusted.

(1 of 2)

Informix Connection Property	Database URL Variable	Required?	Description
PASSWORD	*password*	Yes	The password of the user
			You must specify both the user and the password or neither. If you specify neither, the driver calls **System.getProperty()** to obtain the name of the user currently running the application, and the client is assumed to be trusted.
none	*name=value*	No	A name-value pair that specifies a value for the Informix environment variable contained in the *name* variable, recognized by either IBM Informix JDBC Driver or Informix database servers
			The *name* variable is case insensitive.
			See "Specifying Properties" on page 2-14 and "Using Informix Environment Variables" on page 2-16 for more information.

(2 of 2)

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 2-23.

Database Versus Database Server Connections

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

To create a connection to an 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 an Informix database server, you can execute an SQL statement that connects to a database in your Java program.

All connections to both databases and database servers must include the name of an Informix database server via the **INFORMIXSERVER** environment variable.

Important: *If you are connecting to a 5.x database server (either IBM Informix OnLine or IBM Informix SE) you must specify* USEV5SERVER=1.

The example given in "Using the DriverManager.getConnection() Method" on page 2-9 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** 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());
        }
```

Specifying Properties

When you use the **DriverManager.getConnection()** method to create a connection, IBM 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 user's environment for any environment variables.

To specify Informix environment variables in the name-value pairs of the connection database URL, refer to "Format of Database URLs" on page 2-10.

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 "Using the DriverManager.getConnection() Method" on page 2-9.

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!");
    }
```

Using Informix Environment Variables

The following table lists most of the Informix environment variables supported by the client JDBC driver. For server-side JDBC, you should 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 "Specifying Properties" on page 2-14.

For a list of environment variables that provide internationalization features, see Chapter 6. For a list of environment variables useful for troubleshooting, see Chapter 7.

Supported Informix Environment Variables	Description
BIG_FET_BUF_SIZE	In IBM Informix Extended Parallel Server, Version 8.4, overrides the default size of the tuple buffer and allows it to be increased up to 2GB.
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 pathname of the directory into which you want IBM Informix Enterprise 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 system disk space that the UPDATE STATISTICS statement can use when it simultaneously constructs multiple-column distributions
DELIMIDENT	When set to Y, specifies that strings set off by double quotes are delimited identifiers
ENABLE_CACHE_TYPE	When set to 1, caches the data type information for opaque, distinct, or row data 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.

(1 of 7)

Supported Informix Environment Variables	Description
ENABLE_HDRSWITCH	When set to `true`, secondary server properties are used to connect to the secondary server in an HDR pair, 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 "Using the Auto Free Feature" on page 3-38.
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 "Converting Using the IFX_CODESETLOB Environment Variable" on page 6-21.
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.

Supported Informix Environment Variables	Description
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 backward compatibility with older versions of IBM 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 Informix-specific variable IFX_ISOLATION_LEVEL. The default value is 1 (Committed Read). If the value has been set explicitly, it returns the set value.
	Returns: integer
	Sets the value of Informix specific variable IFX_ISOLATION_LEVEL. Possible values:
	■ `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`)
	Specifying `U` after the mode means `retain update locks`.
	(See *Important* note following table.)
	For example, a value could be: `2U` (equivalent to `SET ISOLATION TO COMMITTED READ RETAIN UPDATE LOCKS`.

(3 of 7)

Supported Informix Environment Variables	Description
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 Informix-specific variable IFX_LOCK_MODE_WAIT. 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 Informix-specific variable IFX_LOCK_MODE_WAIT. Possible values:
	■ `-1` WAIT till 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_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 SMALL-FLOAT data type. This setting enables backward compatibility with older versions of IBM Informix JDBC Driver.
IFX_USEPUT	When set to `1`, enables bulk inserts
	For more information, see "Performing Bulk Inserts" on page 3-5.
IFXHOST	Sets the host name or host IP address
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 default value of **INFORMIXCONTIME** (15 seconds)
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 15 seconds.
INFORMIXOPCACHE	Specifies the size of the memory cache for the staging-area blobspace of the client application

(4 of 7)

Supported Informix Environment Variables	Description
INFORMIXSERVER	Specifies the default database server to which an explicit or implicit connection is made by a client application
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 pathname.
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 (**0**).** The data is always stored in a file. If a security violation occurs, the driver makes no attempt to store the data in memory. ■ **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.
NODEFDAC	When set to YES, prevents default table and routine privileges from being granted to the PUBLIC user when a new table or routine is created in a database that is not ANSI compliant Default is NO.
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

(5 of 7)

Supported Informix Environment Variables	Description
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 client's tuple buffer. The database server automatically closes the cursor after all the rows have been retrieved.
	IBM Informix JDBC Driver might not have additional rows in the client's tuple buffer before the next **ResultSet.next()** method is called. Therefore, unless IBM 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 should 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
PLOAD_LO_PATH	Specifies the pathname 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 "Using an HTTP Proxy Server" on page 2-32.
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 "Using Password Encryption" on page 2-39.

(6 of 7)

Supported Informix Environment Variables	Description
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 2-23.
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.
USEV5SERVER	When set to 1, specifies that the Java program is connecting to an IBM Informix OnLine or IBM Informix SE 5.x database server
	This environment variable is mandatory if you are connecting to an IBM Informix OnLine or IBM Informix SE 5.x database server.

(7 of 7)

Important: *RETAIN UPDATE LOCKS is not supported in IBM* Informix Dynamic Server, *Version 5.x. The* U *option will be ignored when connecting to a 5.x server.*

The following are code examples of the IFX_LOCK_MODE_WAIT and **IFX_ISOLATION_LEVEL** environment variables:

- IFX_LOCK_MODE_WAIT

```
Connection conn = DriverManager.getConnection ( "jdbc:Informix-
sqli://cleo:1550:
INFORMIXSERVER=cleo_921;IFXHOST=cleo;PORTNO=1550;user=rdtest;
password=my_passwd; IFX_LOCK_MODE_WAIT=1";);
```

- **IFX_ISOLATION_LEVEL**

```
Connection conn = DriverManager.getConnection( "jdbc:Informix-
sqli://cleo:1550:
INFORMIXSERVER=cleo_921;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.*

For a detailed description of a particular environment variable, refer to *IBM Informix Guide to SQL: Reference*. You can find the online version of this guide at http://www-3.ibm.com/software/data/informix/pubs/library/.

Dynamically Reading the Informix sqlhosts File

IBM 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 an Informix database server anywhere on the network. For more information about this file, see the *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 an 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 *Administrator's Guide for IBM Informix Dynamic Server*. For more information on HDR, see "Using High-Availability Data Replication" on page 2-28.

An unsigned applet cannot access the **sqlhosts** file or an LDAP server. For more information, see "Using the Driver in an Applet" on page 1-20.

Connection Property Syntax

You can let IBM 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, IBM 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 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.ius`

   The *host-name* and *port-number* 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 using a **DataSource** object, see Appendix B, "DataSource Extensions."

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 "Using an HTTP Proxy Server" on page 2-32 for more information.

Administration Requirements

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:

http://java.sun.com/products/jndi/

http://www.openldap.org

■ If you want to use the 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 server's port number. For more information, see "Utilities to Update the LDAP Server with sqlhosts Data," 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

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 IBM Informix JDBC Driver). Make sure that the **CLASSPATH** variable also points to the JNDI JAR files and LDAP SPI JAR files.

SqlhUpload

This utility loads the **sqlhosts** entries from a flat ASCII file to the LDAP server in the prescribed format. 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 Informix base in LDAP IS *sqlhostsRdn*. The default name is **sqlhosts**.

The utility prompts for other required information, such as the Informix base DN (distinguished name) 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 *Administrator's Guide* for your database server.

SqlhDelete

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 **SqlhUpload** on .

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

Using High-Availability Data Replication

High-Availability Data Replication (HDR) provides synchronous data repli-
cation for IBM Informix Dynamic Server by maintaining a backup copy of the
entire database server that applications can access quickly in the event of a
catastrophic failure. If one of the database servers in the replication pair fails,
clients can be redirected to connect to the alternate database server. For more
information on HDR, see the *Administrator's Guide* for your database server.

HDR server pairs are composed of a primary and a secondary server. The
primary server is the default server. The secondary server is read-only;
update operations are not allowed.

To write application code to support HDR, follow these guidelines, which are
explained in the sections below:

- Set the secondary server connection properties and enable HDR.
- Check if the server is read-only (a secondary server) and take appro-
  priate action.
- If a connection fails, retry the connection to the alternate server and
  rerun the query.

You can use HDR with connection pooling. For more information, see "Using
High-Availability Data Replication with Connection Pooling" on page 7-14.

Demonstration programs are available in the **hdr** directory within the **demo**
directory where IBM Informix JDBC Driver is installed. For details about the
files, see Appendix A.

Secondary Server Connection Properties

Specify the secondary server and enable HDR using the following connection
properties in the connection URL:

- INFORMIXSERVER_SECONDARY = *secondary_server*;
- PORTNO_SECONDARY = *secondary_portnumber*;
- IFXHOST_SECONDARY = *secondary_hostmachine*;
- ENABLE_HDRSWITCH = true;

The following example shows a connection URL for an HDR server pair named **hdr1** and **hdr2**:

```
jdbc:informix-sqli://123.45.67.89:1533/testDB:
   INFORMIXSERVER=hdr1;IFXHOST=host1;PORTNO=1500;
   user=rdtest;password=test;INFORMIXSERVER_SECONDARY=hdr2;
   IFXHOST_SECONDARY=host2;PORTNO_SECONDARY=1600;
   ENABLE_HDRSWITCH=true;
```

When 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 the section "Getting and Setting Informix Connection Properties" on page B-4.

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

Checking for Read-Only Status

Update operations fail if the connection is to a secondary server, because secondary servers are read-only. Therefore, you should write applications to check for read-only connections before starting update operations.

Checking for Read-Only Status

Use the methods in the following table to check the server type and whether HDR is enabled.

Information Obtained	Method Signature	Notes
Whether the server is read-only (a secondary server)	`public boolean isReadOnly() throws SQLException`	Returns `true` if the active server is a secondary server
		Returns an exception if a database access error occurs
		If ENABLE_HDRSWITCH is set to false, **isReadOnly()** returns the value initially set after the last successful HDR connection was obtained.
Whether HDR is enabled	`public boolean isHDREnabled()`	Returns `true` if both servers in the HDR pair are available
		Returns `false` is 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 establishing a connection and then set a flag, like READ_ONLY, and perform operations 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, resulting in connections and uncommitted transactions being lost.

Retrying Connections

Write applications so that if a connection is lost during query operations,
IBM Informix JDBC Driver returns a new connection to the secondary
database server and the application reruns the queries.

The following code 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;
        }
    }
```

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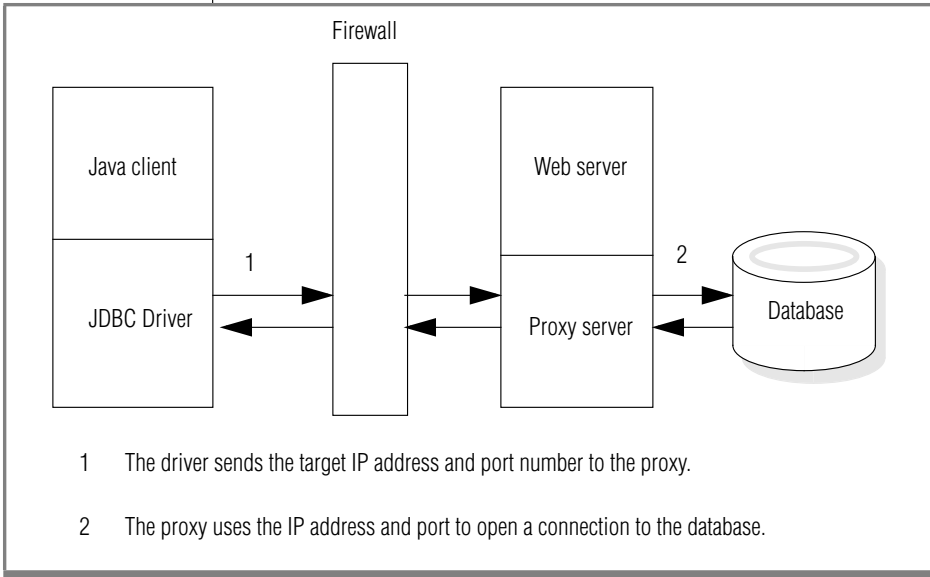
The Informix HTTP proxy handles both of these problems. The proxy is a servlet that runs in the middle tier between a JDBC client and an Informix database server. The proxy extracts SQL requests from the JDBC client and transmits them to the database server. The client (the end user) is unaware of this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

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

Figure 2-1 illustrates how the proxy enables a connection to a database that is behind 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

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 IBM Informix JDBC Driver after the product bundle is installed.

3. Add the IBM 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 server's 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 application's URL:

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

   For example:

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

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 Web sites offer more information about proxy servlets:

- http://java.sun.com/products/servlet/
- http://java.sun.com/
- http://www.sun.com/java
- http://java.apache.org

Specifying 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 should send 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 client's session expiration equal to 2 x PROXYTIMEOUT. For example, if PROXY-TIMEOUT is set to 60 seconds, the proxy sets the client's expiration time to 120 seconds. When the expiration time is reached, the proxy removes the client's 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.

Using the Proxy with an LDAP Server

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 IBM Informix JDBC Driver, see the sections beginning with "Connection Property Syntax" on page 2-24.

Figure 2-2 on page 2-37 illustrates how the proxy works with an LDAP server. The figure also shows lookup from an **sqlhosts** file; for more information, see "Specifying sqlhosts File Lookup" on page 2-38.

Figure 2-2
*Lookup by the
Proxy*
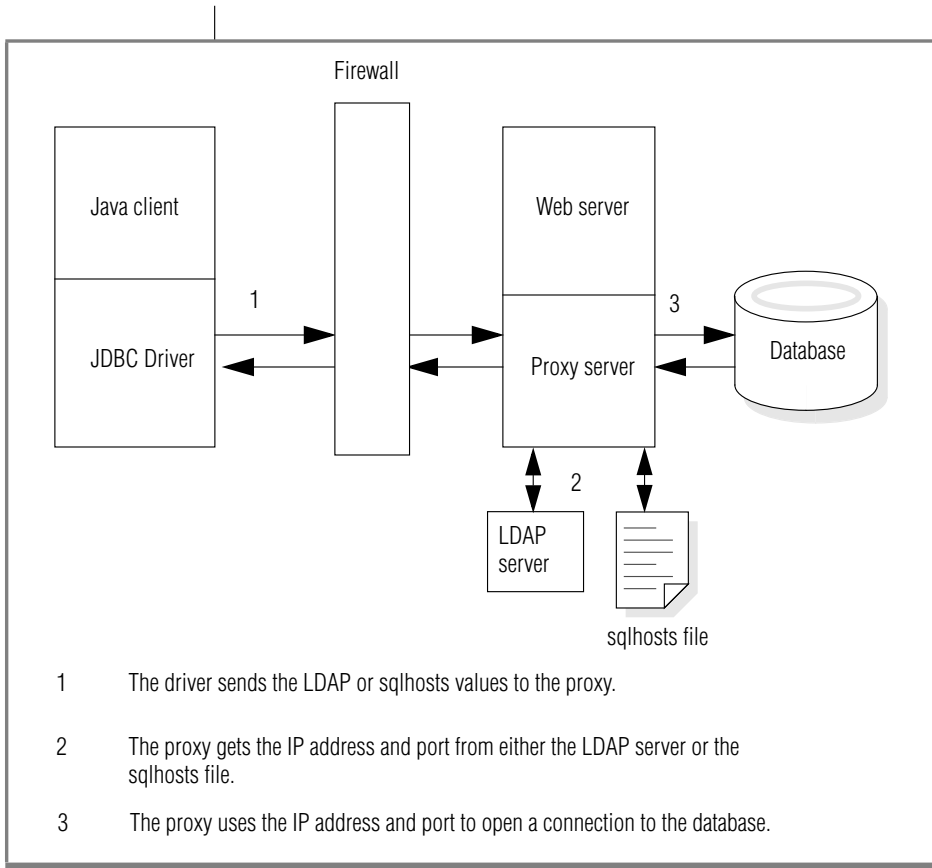
The proxy LDAP feature requires the JNDI class libraries and LDAP service provider files (**jndi.jar**, **ldap.jar**, and **providerutil.jar**). These JAR files can be downloaded from the following location: http://java.sun.com/products/jndi/index.html#download.

After downloading and installing the files, add their full pathnames to the CLASSPATH setting on your Web server. The files are in the **lib** directory under the installation directory.

Specifying Where LDAP Lookup Occurs

When used in conjunction with other LDAP keywords, the SQLH_LOC keyword indicates where an LDAP lookup should occur.

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 hostname 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.

Specifying 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
```

Using Other Multitier Solutions

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

- **Remote Method Invocation (RMI)**. IBM 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 IBM Informix JDBC Driver; see Appendix A, "Sample Code Files," for details.

- **Other communication protocols, such as CORBA.** IBM Informix JDBC Driver resides on an application server that is a middle tier between the Java applet or application and Informix database computers.

Using 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 IBM 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 case insensitive. You can type it in upper or lowercase letters.

Configuring the Database Server

The SECURITY=PASSWORD setting is supported in the 7.31, 8.3, 9.1x, 9.2x, and later 9.x versions of the Informix database server. The connection is rejected if used with any other versions of the server.

If the SECURITY=PASSWORD setting is specified in the IBM 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 server's **sqlhosts** SERVERNAME option. After this option is set on the server, only clients using the SECURITY=PASSWORD setting can connect to that server name.

To see if the SPWDCSM CSM option is supported for your version of Informix database server, check the database server release notes. See the *Administrator's Guide* for your database server for general details on how to configure the CSM options.

JCE Security Package

To use the SECURITY=PASSWORD option, you must install a JDK Java cryptography extension (JCE) compliant security package on the JDBC client and include the installation directory of the security package in the **CLASSPATH** variable.

You can download the free Sun JCE 1.2 security package from the following Web site: http://java.sun.com/products/jce.

Sun JCE is available only in the U.S. or Canada. If your site does not comply with this or other Sun JCE licensing restrictions, you can try using IBM Informix JDBC Driver with other JCE-certified security package providers. However, these packages have not been tested and certified to work correctly with Informix database servers configured to use the SPWDCSM CSM option.

To install the Sun JCE package, download the Sun JCE distribution, extract the JAR file containing the Sun JCE provider packages, and make sure the **CLASSPATH** environment variable includes the extracted JAR filename.

Edit the **jdk1.2/lib/security/java.security** file to add the following two lines:

```
security.provider.1=sun.security.provider.Sun
security.provider.2=com.sun.crypto.provider.SunJCE
```

Closing 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 on deallocating resources, see "Deallocating Resources" on page 3-16. For more information on the **scrubConnection()** method, see "Cleaning Pooled Connections" on page 7-16.

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 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 server's connection-pool manager can call **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.*

Performing Database Operations

In This Chapter

This chapter explains what you need to use IBM Informix JDBC Driver to perform operations against an Informix database. This chapter includes the following sections:

- Querying the Database
- Handling Errors
- Handling Transactions
- Storing and Retrieving XML Documents
- Other Informix Extensions to the JDBC API
- Accessing Database Metadata

Querying the Database

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

The following sections describe Informix differences from and extensions to the JDBC 2.0 specification from Sun Microsystems:

- Performing Batch Updates
- Performing Bulk Inserts
- Using Scroll Cursors
- Using Hold Cursors
- Using CallableStatement OUT Parameters
- Using Escape Syntax

Performing Batch Updates

The batch update feature is similar to Informix multiple 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 IBM Informix JDBC Driver follows the Sun Microsystems JDBC 2.0 specification, with these exceptions:

■   SQL statements

■   Return value from **Statement.executeBatch()**

The following sections give details.

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, refer to *IBM Informix Guide to SQL: Syntax*.

Return Value from Statement.executeBatch() Method

The return value differs from the Sun Microsystems JDBC 2.0 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 "Using Informix Environment Variables" on page 2-16.

■ 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.

Performing Bulk Inserts

A bulk insert is an Informix extension to Sun's JDBC 2.0 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.

Using Scroll Cursors

The scroll cursors feature in IBM Informix JDBC Driver follows the Sun Microsystems JDBC 2.0 specification, with these exceptions:

- Scroll sensitivity
- Client-side scrolling
- Result set updatability

Scroll Sensitivity

The 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 2.0 specification. IBM 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. IBM Informix JDBC Driver supports the scrolling of the result set only according to how the database server supports scrolling.

Result Set Updatability

The JDBC 2.0 API from Sun Microsystems does not provide exact specifications for SQL queries that yield updatable result sets. Generally, queries that meet the following criteria can produce updatable result sets:

- 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.

IBM 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 or SERIAL8 column in the table.
3. The driver looks for the table's primary key in the system catalogs.
4. If none of these is provided, the driver returns an error.
5. When you delete a row in a result set, the **ResultSet.absolute()** method is affected, because the positions of the rows after the deleted row change.

   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 an updatable scrollable cursor, see the **UpdateCursor1.java**, **UpdateCursor2.java**, and **UpdateCursor3.java** files.

Using Hold Cursors

To create a hold cursor, use the **executeQuery()** method defined in one of the following Informix extensions:

- **IfmxStatement** interface

  ```
  public java.sql.ResultSet executeQuery(String sql, boolean
      withHold) throws SQLException;
  ```
- **IfmxPreparedStatement** interface

  ```
  public ResultSet executeQuery(boolean withHold) throws
      SQLException;
  ```

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

Using CallableStatement OUT Parameters

CallableStatement methods handle OUT parameters in SPL and C functions and Java user-defined routines (UDRs). Two **registerOutParameter()** methods specify the data type of OUT parameters to the driver. A series of **get*XXX*()** methods retrieve OUT parameters.

IBM Informix Dynamic Server, Version 9.2x and 9.3x, considers OUT parameters to be statement local variables (SLVs). SLVs are valid only for the life of a single statement and cannot be returned directly upon executing the routine. The JDBC **CallableStatement** interface provides a method for retrieving OUT parameters.

For background information, refer to the following documentation:

- *IBM 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 an Informix database.

- *J/Foundation Developer's Guide* describes how to write Java UDRs for use in the database server.

- The *IBM Informix Guide to SQL: Tutorial* describes how to write stored procedure language (SPL) routines.

- The *IBM Informix DataBlade API Programmer's Guide* describes how to write external C routines.

Only Informix database servers versions 9.2 and higher return an OUT parameter to IBM Informix JDBC Driver. IBM Informix Dynamic Server, Version 9.4, supports multiple OUT parameters.

For examples of how to use OUT parameters, see the **CallOut1.java**, **CallOut2.java**, **CallOut3.java**, and **CallOut4.java** example programs.

Server Restrictions and Limitations

Versions 9.2x and 9.3x of IBM Informix Dynamic Server 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.

- 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 IBM Informix Dynamic Server, Version 9.4 allows:

- Any and all 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 on UDRs, see *IBM Informix User-Defined Routines and Data Types Developer's Guide* and *J/Foundation Developer's Guide*.

Driver Enhancement

In JDBC a 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 resultset or as an OUT parameter. The IBM Informix JDBC Driver 2.21.JC4 has been enhanced to support the new functionality in the 9.40 server.

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**:

```java
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
```

The above example requires server-side support for multiple OUT parameters; hence it will only work for IBM Informix Dynamic Server, Version 9.4 or above. For more information on UDRs, see *IBM Informix User-Defined Routines and Data Types Developer's Guide* and *J/Foundation Developer's Guide*.

Driver Restrictions and Limitations

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

■ IBM Informix Dynamic Server, Version 9.2, 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 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 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 **setNull()** 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 **setNull()** and **registerOutParameter()** fix these problems. See "IN and OUT Parameter Type Mapping," next.

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 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
Date	IFX_TYPE_DATE
Decimal	IFX_TYPE_DECIMAL
Distinct*	Depends on base type

(1 of 3)

java.sql.Types	com.informix.lang.IfxTypes
Double	IFX_TYPE_FLOAT
Float	IFX_TYPE_FLOAT[1]
Integer	IFX_TYPE_INT
Java_Object*	IFX_TYPE_UDTVAR
	IFX_TYPE_UDTFIX
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

(2 of 3)

java.sql.Types	com.informix.lang.IfxTypes
Varchar	IFX_TYPE_VCHAR (*n*)
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 backward compatibility by setting the IFX_SET_FLOAT_AS_SMFLOAT connection property to 1.

<div align="right">(3 of 3)</div>

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 "Using the IfxTypes Class" on page C-13.

Using Escape Syntax

Escape syntax indicates information that must be translated from JDBC format to 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*'}

<div align="right">(1 of 2)</div>

Type of Statement	Escape Syntax
Time	{t '*hh:mm:ss*'}
Timestamp (Datetime)	{ts '*yyyy-mm-dd hh:mm:ss*[.*fffff*]'}
Function call	{fn *func*[(*args*)]}
Escape character	{escape '*escape-char*'}
Outer join	{oj *outer-join-statement*}

(2 of 2)

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.

Using Result Sets

The IBM 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.

Deallocating 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 IBM 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 make a call to **Connection.close()**, which will take care of releasing these resources.

Executing 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.

Example of Sending a Query to an 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 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 on which **get*XXX*()** methods retrieve individual column values, refer to "Data Type Mapping for ResultSet.getXXX() Methods" on page C-18.

Unsupported Methods

The following JDBC API methods are not supported by IBM Informix JDBC Driver and cannot be used in a Java program that connects to an Informix database:

- **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.cancel()**
- **Statement.setMaxFieldSize()**
- **Statement.setQueryTimeout()**

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

- **CallableStatement.execute()**

  Returns a single result set

- **DatabaseMetaData.getProcedureColumns()**

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

- **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 *Important* note below)

Important: *IBM Informix servers do not currently support read-only connections. For the IBM 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 IBM Informix JDBC Driver to the IBM DB2 JDBC driver and also to achieve a higher level of compliance in the Sun Conformance Test (CTS).*

JDBC Support for Describe Input

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

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

The IBM Informix JDBC Driver, Version 2.21.JC4, 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 meta data 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 IBM Informix JDBC Driver has implemented additional methods to the **ParameterMetaData** interface to extend its functionality.

Return Type	Method	Description
int	**getParameterLength** (int param)	Retrieves parameter's length
int	**getParameterExtendedId** (int param)	Retrieves parameter's extended id
java.lang.String	**getParameterExtendedName** (int param)	Retrieves parameter's extended name
java.lang.String	**getParameterExtendedOwnerName** (int param)	Retrieves parameter's extended type's owner name
int	**getParameterSourceType** (int param)	Retrieves parameter's SourceType
int	**getParameterAlignment** (int param)	Retrieves parameter's alignment

Below is an example of using the **ParameterMetaData** interface in the IBM 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));
    }
. . .
```

Handling Errors

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

Using the SQLException Class

Whenever an error occurs from either IBM 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 IBM Informix JDBC Driver error code

- **getSQLState()**

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

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

All IBM Informix JDBC Driver errors have error codes of the form -79*XXX*, such as `-79708 Method can't take null parameter`.

For a list of Informix database server errors, refer to *IBM Informix Error Messages*. You can find the online version of this guide at http://www-3.ibm.com/software/data/informix/pubs/library/. For a list of IBM Informix JDBC Driver errors, see "Error Messages" near the end of this book.

The following example from the **SimpleSelect.java** program shows how to use the **SQLException** class to catch IBM Informix JDBC Driver or database server errors 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());
    }
```

Retrieving Informix Error Message Text

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 an 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.

Retrieving the Syntax Error Offset

To determine the exact location of a syntax error, use the **getSQLStatement-Offset()** 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() );
}
```

Catching 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 IBM Informix JDBC Driver.

Handling 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 autocommit mode off, explicitly call **Connection.setAutoCommit(false)**.

When autocommit mode is off, IBM 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 **setAuto-Commit(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.

If the Java program sets autocommit mode on during a transaction, IBM Informix JDBC Driver commits the current transaction if the JDK is version 1.4 and later, otherwise the driver rolls back the current transaction before turning on autocommit.

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 should avoid performing these operations directly.

Storing and Retrieving 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 increasingly popular method of representing data in an open, platform-independent format.

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

- **SAX (Simple API for XML)** is an event-driven protocol, with the programmer providing the callback methods that the XML parser invokes when it analyzes a document.

- **DOM (Document Object Model)** is a random-access protocol, which converts an XML document into a collection of objects in memory that can be manipulated at the programmer's 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.

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 **Input-Source**, 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.

Setting 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.

Setting Your CLASSPATH

To use the XML methods, add the pathnames 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 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 using code from a JAR file that supports the SAX, DOM, and JAXP methods. Informix includes the xml4j JAXP JAR file, **xerces.jar**, with the JDBC driver. To use **ifxtools.jar**, you must add these JAR files to your CLASSPATH setting.

JDK version 1.4 or later uses the Sun Microsystems 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 on how to set the properties see .

Specifying 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 quotes:

```
% 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 quotes:

```
% 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**.

Inserting 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 pathname (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 non-validating XML parser. To change the default, see the previous section, "Specifying a Parser Factory" on page 3-30.

- The *nsa* parameter tells the SAX parser factory whether it should use a parser that can handle namespaces.

The following methods parse a file 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 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 "Inserting Data Examples" on page 3-34.

If no value is specified, or if *handler* is set to NULL, the driver uses the default 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.*

Retrieving 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 "Inserting Data" on page 3-31.

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 "Retrieving Data Examples" on page 3-35.)

```
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 "Retrieving Data Examples" on page 3-35.

Inserting Data Examples

The examples in this section illustrate converting XML documents to formats acceptable for insertion into Informix database columns.

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 using 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
    }
```

XMLtoInputStream() Example

The following example inserts an XML file into a text column. In this example, table **tab2** is created using 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
    }
```

Retrieving Data Examples

The following examples illustrate retrieving data from Informix database columns and converting the data to formats acceptable to XML parsers.

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'
    }
```

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
    }
```

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 Sun's 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 using 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 **Input-Source** object for parsing. This 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
    }
```

Other Informix Extensions to the JDBC API

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

Another Informix extension, the **com.informix.jdbc.Message** class, is fully described in "Handling Errors" on page 3-24.

Using the Auto Free Feature

If you enable the 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

There are two ways to obtain version information about IBM 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 IBM Informix JDBC Driver.

   An example of a version of IBM 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.

Important: *For version X.Y of IBM 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 IBM 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.

Accessing Database Metadata

To access information about an Informix database, use the JDBC API
DatabaseMetaData interface.

IBM Informix JDBC Driver is completely compatible with the JDBC API speci-
fication for accessing database metadata. The driver supports all the methods
of the **DatabaseMetaData** interface.

IBM 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. For example, IBM Informix SE does
not have a sysmaster database, therefore you cannot use the
DatabaseMetaData interface with it.

IBM 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, IBM Informix JDBC Driver interprets the JDBC API term *catalogs* to
mean the names of Informix databases. The **DatabaseMetaData.getCat-
alogs()** 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 Appendix A for more information about this example.

Working With Informix Types

In This Chapter

This chapter explains the Informix-specific data types (other than opaque types) supported in IBM Informix JDBC Driver. For information on opaque types, see Chapter 5, "Working with Opaque Types." The chapter includes the following sections:

- Distinct Data Types
- BYTE and TEXT Data Types
- SERIAL and SERIAL8 Data Types
- INTERVAL Data Type
- Collections and Arrays
- Named and Unnamed Rows
- Smart Large Object Data Types

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 Appendix C, "Mapping Data Types," to map this Java object to the corresponding SQL type name.

Inserting 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);
```

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, IBM 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.

Retrieving 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 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 kind of 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 kind of 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, refer to *IBM Informix Guide to SQL: Reference* and *IBM Informix Guide to SQL: Syntax*. You can find the online version of both of these guides at http://www-3.ibm.com/software/data/informix/pubs/library/.

Caching 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 "Managing Memory for Large Objects" on page 7-7.

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, IBM 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()** and **ResultSet.getAsciiStream()** methods to retrieve a stream of binary or ASCII data from columns of type BYTE and TEXT, respectively. 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 very similar to selecting from a column of type BYTE, except the **getAsciiStream()** method is used instead of **getBinaryStream()**.

SERIAL and SERIAL8 Data Types

IBM 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 Informix-specific classes 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()** and **getSerial8()** methods after an INSERT statement to return the serial value that was automatically inserted into the SERIAL or SERIAL8 column of a table, respectively. 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 Informix-specific implementation of the **Statement** interface. 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, refer to *IBM Informix Guide to SQL: Reference* and *IBM Informix Guide to SQL: Syntax*. You can find both of these guides online at http://www-3.ibm.com/software/data/informix/pubs/library/.

INTERVAL Data Type

The 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*, refer to the following manuals:

- *IBM Informix Guide to SQL: Tutorial*
- *IBM Informix Guide to SQL: Reference*
- *IBM Informix Guide to SQL: Syntax*

You can find these guides online at
http://www-3.ibm.com/software/data/informix/pubs/library/.

The Interval Class

The **com.informix.lang.Interval** class is an Informix-specific extension to the JDBC 2.0 specification from Sun Microsystems. **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 Localized Error Messages" on page 6-25.*

For information about the string INTERVAL formats in this section, refer to the *IBM 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 IBM Informix JDBC Driver.)

Using 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.

Variable	Description
TU_YEAR	Time unit for the YEAR qualifier field
TU_MONTH	Time unit for the MONTH qualifier field

(1 of 2)

Variable	Description
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

(2 of 2)

Using 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.

Using 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 "Using Interval Methods" on page 4-14. 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 "Using Interval Methods" on page 4-14. 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.

Using 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 using **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.

Using 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 "Using Interval Methods" on page 4-14. The second ver-
sion 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, refer to the *IBM 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 "Using Interval Methods" on page 4-14. 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.

Using 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 IBM Informix JDBC Driver, shows how to insert into and select from the two types of INTERVAL data types.

Collections and Arrays

Sun's JDBC 2.0 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, IBM 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 2.0 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 an Informix MULTISET data type. An instance of the **java.util.List** interface is mapped to an Informix LIST data type. You can override these defaults by creating a customized type mapping.

For information about customized type mappings, see Appendix C.

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 Appendix A, "Sample Code Files."

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,
IBM 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, IBM 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.

For more information about the uses of **HashSet** and **TreeSet** objects, refer to the class definitions in the documentation from Sun Microsystems.

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

Sun's JDBC 2.0 specification refers to an SQL type called a *structured type* or *struct*, which is equivalent to an 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 IBM 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 Appendix C.

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***)**

Using 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 "Using the ClassGenerator Utility" on page 4-39.

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 Appendix C.

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 Appendix A.

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**. IBM 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 infor-
mation 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.

IBM 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. IBM 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.write*XXX*() 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.

Using the Struct Interface

The JDBC 2.0 documentation does not specify that **Struct** objects can be parameters to the **PreparedStatement.setObject()** method. However, IBM 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. IBM 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 manuals: *J/Foundation Developer's Guide* (for information specific to Java); *IBM Informix User-Defined Routines and Data Types Developer's Guide* and *IBM Informix Guide to SQL: Reference* (both for general information about user-defined routines); and *IBM 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 Appendix A.

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.

```java
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, IBM 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.

Using the ClassGenerator Utility

The **ClassGenerator** utility generates a Java class for a named row type defined in the system catalog. The utility is an Informix extension to Sun's JDBC 2.0 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);
   }
}
```

Caching Type Information

When objects of some data types insert data into columns of certain other data types, IBM 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

You can set an environment variable, ENABLE_CACHE_TYPE=1, in the database URL to have the driver cache the 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 4 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:

- You can use the standard JDBC API methods described in the JDBC 2.0 specification from Sun Microsystems. This is the simpler approach.

- You can use Informix extensions that are based on smart large object support within IBM Informix Dynamic Server, which are described in this section. This is the approach that offers more options.

This section contains the following subsections:

- Smart Large Objects in the Database Server
- Smart Large Objects in a Client Application
- Using the Informix Extensions: Steps for Creating Smart Large Objects
- Steps for Accessing Smart Large Objects
- Performing Operations on Smart Large Objects
- Working with Storage Characteristics
- Working with Status Characteristics
- Working with Locks
- Caching Large Objects
- Smart Large Object Examples

Smart Large Objects in the Database Server

In the 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. Figure 4-1 shows how the LO handle contains information about the location of the actual employee picture in the **sbspace1_100** sbspace.
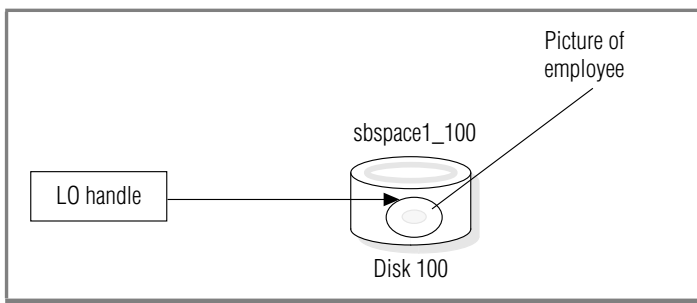
Figure 4-1
Smart Large Object in the Database Server

In Figure 4-1, 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 *IBM Informix Dynamic Server 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

This option is portable to other platforms.

Alternatively, you can use the Informix extensions to access the data. This option is not portable but offers more flexibility.

Overview of the Informix Extensions

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 Figure 4-2. The Informix extensions allow your application to create a smart large object independently and then insert 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.
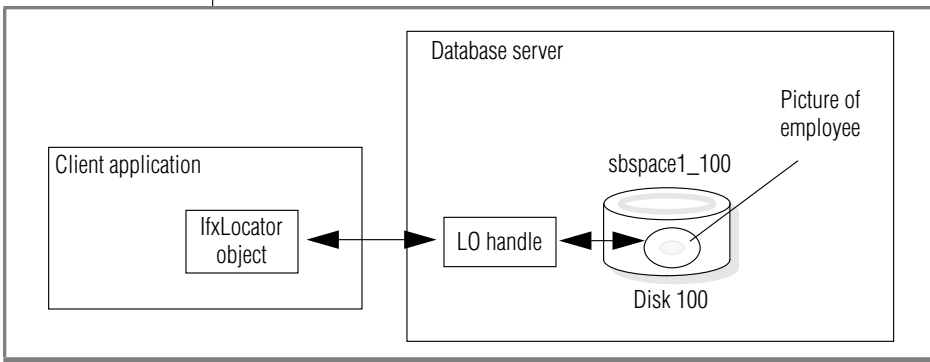
Figure 4-2
Locating a Smart Large Object In a Client Application

Support for Informix smart large object data types is available only with 9.x versions of the database server.

Using the Informix Extensions: Steps for Creating Smart Large Objects

The Informix smart large object implementation is based on five 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 Sun's JDBC 2.0 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 IBM 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 on the **onspaces** utility that creates sbspaces, see the *IBM Informix Dynamic Server Administrator's Guide*. For an example of creating an sbspace, see "Example of Setting sbspace Characteristics" on page 4-64.

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 "Working with Storage Characteristics" on page 4-61.

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 4-52).

11. Execute **IfxSmartBlob.IfxLoRelease()** to release the locator pointer.

Creating 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 "Working with Storage Characteristics" on page 4-61.

Creating 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 4-2 on page 4-45; 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 Localized Error Messages" on page 6-25.

Creating 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 "Positioning Within a Smart Large Object" on page 4-54).

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	LO_APPEND
	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.	
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 on locking, see .

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

(1 of 2)

Open-Mode Flag	Description
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
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.

(2 of 2)

Inserting a Smart Large Object into a Column

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.

Important: *The sbspace for the smart large object must exist in the database server before the insertion executes.*

Steps for Accessing Smart Large Objects

Follow these steps to use the 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.

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.

Performing 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** class implements the **java.sql.Blob** interface, and the **IfxCblob** class implements the **java.sql.Clob** interface. In addition to the methods defined by Sun, each class supports the Informix extensions for binary smart large objects with constructor and locator methods.

The **IfxBblob** and **IfxCblob** classes are a bridge between the way of handling smart large object data described in Sun's JDBC 2.0 specification and the Informix extensions. 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 2.0 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
```

Opening 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 "Creating an IfxSmartBlob Object" on page 4-48.

Positioning 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
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.

Reading 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 **IfxLoCreate()** 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 "Positioning Within a Smart Large Object" on page 4-54.

Writing 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.

Truncating 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.

Measuring 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
```

Closing and Releasing 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.

Converting 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	Notes
Converts a byte array to a hexidecimal character string	`public static String toHexString(` `byte[] byteBuf);`	Works on data other than IfxLocator Provided in the **com.informix.util.stringUtil** class

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 hexidecimal 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)
  {}
```

Working 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 Figure 4-3 shows, to obtain the storage characteristics for a new smart large object.
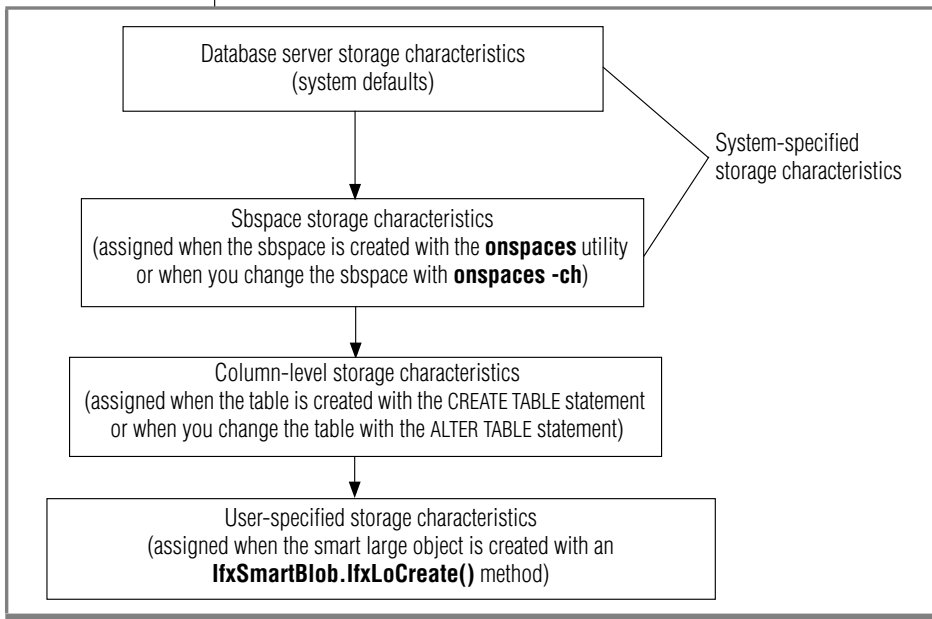
Figure 4-3
Storage-Characteristics Hierarchy

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	"Using System-Specified Storage Characteristics" on page 4-62
		IBM Informix Dynamic Server Administrator's Guide
When a database table is created	Keywords in PUT clause of CREATE TABLE statement	*IBM Informix Guide to SQL: Syntax*
When a smart large object is created	Create flags and methods in the **ifxLobDescriptor** class	"Setting Create Flags" on page 4-74

Using System-Specified Storage Characteristics

The database administrator establishes system-specified storage character-istics when he or she initializes 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 Figure 4-5 on page 4-68 and Figure 4-6 on page 4-69.

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 "Setting Create Flags" on page 4-74.

- 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 4-73.

- 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).

Obtaining 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"
```

Figure 4-4 shows the resulting system-specified storage characteristics for all smart large objects in 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
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

Working 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 IBM 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 on extents, see the chapter on disk structure and storage in the *IBM Informix Dynamic Server 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.

Figure 4-5 summarizes the ways to specify disk-storage information for a smart large object.

Figure 4-5
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 an IBM 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

Working 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

Figure 4-6 summarizes how you can alter these attributes at the system, column, and application levels.

Figure 4-6
Specifying Attribute Information

Attribute 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 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

The following sections provide more information about these attributes.

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 "Setting Create Flags" on page 4-74.

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 over-flow 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 record-ing, 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. (Figure 4-6 on page 4-69 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).

ANSI

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 on smart large object locks, see .

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 "Setting Create Flags" on page 4-74.

The database server also tracks the last-modification time and the last change in status for a smart large object. For more information, see "Working with Status Characteristics" on page 4-75.

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 "Setting Create Flags" on page 4-74.

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 on the structure of sbspace pages, see the *IBM Informix Dynamic Server Administrator's Guide.*

Changing the Storage Characteristics

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. 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 "Setting Create Flags," next.

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
```

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.

Setting 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 4-70.

- 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 4-71.

- 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 4-72.

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 on all of the characteristics, see the section describing the PUT clause for the CREATE TABLE statement, in the *IBM Informix Guide to SQL: Syntax*.

Working 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. Figure 4-7 shows the status information that you can obtain.

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 4-71.
Last-change time	The time, in seconds, of the last change in status for the smart large object
	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.

(1 of 2)

Status Information	Description
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 "Working with Storage Characteristics" on page 4-61.

(2 of 2)

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 Figure 4-8.

Figure 4-8
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()

Working 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.

Figure 4-9 shows the lock modes that a smart large object can support.

Figure 4-9
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.

Using 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 "Positioning Within a Smart Large Object" on page 4-54.

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.

Caching 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 "Managing Memory for Large Objects" on page 7-7.

Smart Large Object Examples

The examples on the following pages illustrate some of the tasks discussed in this section.

Creating a Smart Large Object

This example illustrates the steps shown in "Using the Informix Extensions: Steps for Creating Smart Large Objects" on page 4-45.

```
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.

Inserting 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.

Retrieving Data from a Smart Large Object

The example in this section illustrates the steps in "Steps for Accessing Smart Large Objects" on page 4-52.

The following code example shows how to access the smart large object data using 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. IBM 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.

Working with Opaque Types

In This Chapter

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 2.0 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 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 *J/Foundation Developer's Guide*.

For detailed information about opaque types and UDRs, see the following manuals:

- *IBM 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 *J/Foundation Developer's Guide* discusses information specific to writing UDRs in Java.

You can find the online versions of both these guides at http://www-3.ibm.com/software/data/informix/pubs/library/.

This chapter includes the following topics:

- Using the IfmxUDTSQLInput Interface
- Using the IfmxUDTSQLOutput Interface
- Mapping Opaque Data Types
- Caching Type Information
- Creating Opaque Types and UDRs
- Examples

Using 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.

Reading 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 IBM 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.

Positioning 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()**, IBM Informix JDBC Driver generates an **SQLException** if the new position specified is after the end of the input stream.

Setting or Obtaining 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.

Using 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, IBM 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, IBM Informix JDBC Driver pads the array or string with zeros. In both methods, you must supply a length for IBM Informix JDBC Driver to write the next attribute properly, because the opaque type is unknown to the driver.

Mapping Opaque Data Types

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. IBM Informix JDBC Driver cannot perform any type conversion because the type structure is unknown to it.

IBM 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 IBM Informix Dynamic Server 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 .

Caching Type Information

When objects of some data types insert data into columns of certain other data types, IBM 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

You can set an environment variable, ENABLE_CACHE_TYPE=1, in the database URL, to have the driver cache the 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

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 manuals:

- For information about configuring your system to support Java UDRs, see the *J/Foundation Developer's Guide*.

- For detailed information about developing opaque types, see *IBM 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, IBM 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, IBM 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 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.

Steps to Creating Opaque Types

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.

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 5-16.

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 *IBM 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 *Administrator's Guide* for your database server and the *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 Chapter 2.

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 5-17.

    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 pathname 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 "Specifying a JAR File Temporary Path" on page 5-25.

8. Create the opaque type:

```
udtmgr.createUDT(mdata, "Circle2.jar", "Circle2", 0);
```

For additional information on creating an opaque type from existing code, see "Creating an Opaque Type from Existing Code" on page 5-25.

For a complete code example of using the preceding steps to create an opaque type, see "Creating an Opaque Type from an Existing Java Class with UDTManager" on page 5-40.

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 *Administrator's Guide* for your database server and the *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 Chapter 2, "Connecting to the Database."

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 on setting characteristics for opaque types, see "Specifying Characteristics for an Opaque Type" on page 5-18.

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 5-22.

6. If desired, specify a pathname 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 "Specifying a JAR File Temporary Path" on page 5-25.

7. Send the class definition to the database server:

```
udtmgr.createUDT(mdata, jarfilename, classname, 0);
```

For more information, see "Sending the Class Definition to the Database Server" on page 5-24.

For a complete code example of using the preceding steps to create an opaque type, see "Creating an Opaque Type Without an Existing Java Class" on page 5-50.

Steps to Creating UDRs

The following section tells 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 5-16.

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 *Administrator's Guide* for your database server and the *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 Chapter 2.

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. 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 "Creating UDRs" on page 5-28.

7. Specify the JAR file SQL name:

   ```
   mdata.setJarFileSQLName("group1_jar");
   ```

8. If desired, specify a pathname 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 "Specifying a JAR File Temporary Path" on page 5-25.

9. Install the UDRs in the database server:

```
udrmgr.createUDRs(mdata, "Group1.jar", "Group1", 0);
```

For more information, see "Creating UDRs" on page 5-28.

For complete code examples of creating UDRs, see "Creating UDRs with UDRManager" on page 5-53.

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 5-34.

- 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 4-3.

- 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 "Creating an Opaque Type Using Default Support Functions" on page 5-40.

- 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 nongraphical Java packages that are included in the JDK, 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 C-21.
- 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 C-21.

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 filename 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.*

Specifying Characteristics for an Opaque Type

The following sections 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 5-25.

Using the methods in the **UDTMetaData** class, you can specify characteristics for a new opaque type. The characteristics you can specify are described on the following pages. These settings apply for new opaque types; for opaque types created from existing files, see "Creating an Opaque Type from Existing Code" on page 5-25.

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

Specifying 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
```

Specifying 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 C-24. 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);
```

Specifying 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.

Specifying Alignment

The **setAlignment()** method specifies the opaque type's 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

Specifying 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 filenames of the Java class and JAR files generated when you call the **UDTManager.createUDTClass()** and **UDTManager.createJar()** methods. For example, if you called setSQLName("circle") and then called **createUDTClass()** and **createJar()**, the class filename generated would be **circle.class** and the JAR filename would be **circle.jar**. You can specify a Java class filename 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.*

Specifying 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 filename of the **.class** file generated by the **createUDTClass()** 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

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.

Creating 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.

Creating 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 filename to your application.

Sending 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 pathname 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 pathnames 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 filename must be included in your CLASSPATH setting if you use an absolute pathname.

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* "Executing in a Transaction" on page 5-33.

Specifying 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 *J/Foundation Developer's Guide* and the SQLJ specification (Section 4.21 of the document *SQLJ-Part 1: SQL Routines Using the Java Programming Language*, available on the Web at http://www.sqlj.org).

Specifying 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 pathname 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 pathname, and you must ensure that the path exists on the server file system.

Creating an Opaque Type from Existing Code

The preceding pages 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 5-17.) 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
```

Using 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 C-22. 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");
```

Using setSupportUDR() and setUDR()

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 *IBM 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 5-17.

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 on creating an opaque type from existing code, see "To create an opaque type from an existing Java class" on page 5-12.

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 "Creating UDRs" on page 5-28.

Removing 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 "Executing in a Transaction" on page 5-33.*

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 "Specifying Deployment Descriptor Actions" on page 5-24.

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.*

Creating 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 *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 *Administrator's Guide* for your database server and the *J/Foundation Developer's Guide*.

■ Calling methods in the **UDRMetaData** class to specify the information necessary for IBM Informix JDBC Driver to register the UDRs in the database server

■ If desired, specifying a pathname 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 pathname of the client-side JAR file that contains the Java method definitions. If you use the absolute pathname, the JAR filename 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 page 5-15.

For the *deploy* parameter, see "Specifying Deployment Descriptor Actions" on page 5-24.

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* *.*

Removing 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 .

Removing 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	Corresponding 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} );
```

Obtaining 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.

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 5-17.

Information Obtained	Method Signature	Notes
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

(1 of 2)

Information Obtained	Method Signature	Notes
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 5-21.
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 5-25. 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

(2 of 2)

getXXX() Methods in the UDRMetaData Class

To obtain information about UDRs, use the methods in the following table.

Information Obtained	Method Signature	Notes
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

Executing 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 chapter contains examples for creating and using opaque types and UDRs. The following examples are included:

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"));
```

Inserting 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 **set*XXX*()** 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);
...
```

Retrieving Data

To retrieve Informix opaque types, you must use **ResultSet.getObject()**. IBM 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();
```

Using 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 Informix extension classes. For more information about smart large objects, see "Smart Large Object Data Types" on page 4-42.

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, "Hong Kong");
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 4-42 for details.

Creating 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.

Creating 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;
    }

}
```

Using 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());
                            }
        }
```

Creating 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;
        }

    }
```

Using 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. Applica-
tions 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();
```

Creating 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("------------------");

    }
```

Creating 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'");

...

}
```

Internationalization and Date Formats

In This Chapter

This chapter explains how IBM Informix JDBC Driver extends the JDK 1.2 internationalization features by providing access to Informix databases that are based on different locales and code sets. This chapter includes the following sections:

- Support for JDK 1.1 and 1.2 Internationalization
- Support for IBM Informix GLS Variables
- Support for DATE End-User Formats
- Precedence Rules for End-User Formats
- Support for Code-Set Conversion
- User-Defined Locales
- Support for Localized Error Messages

Internationalization 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), refer to the *IBM Informix GLS User's Guide*.

Support for JDK 1.1 and 1.2 Internationalization

Versions 1.1 and 1.2 of the JDK provide a rich set of APIs for developing global applications. These internationalization APIs are based on the Unicode 2.0 code set and can adapt text, numbers, dates, currency, and user-defined objects to any country's conventions.

The internationalization 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 localization support classes, and new classes for date and time handling.

For more information about JDK internationalization support, see http://java.sun.com/products/jdk/1.2/docs/guide/internat/index.html

*Warning: There is no connection between JDK locales and JDK code sets; you must keep these 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 IBM Informix GLS Variables

Internationalization adds several environment variables to IBM Informix JDBC Driver, which are summarized in the following table. All international-ization properties are available on and optional for servers that support GLS.

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 backward compatibility; **GL_DATE** is preferred.
DB_LOCALE	Specifies the locale of the database
	IBM 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.

(1 of 2)

Supported Informix Environment Variables	Description
GL_DATE	Specifies the end-user formats of values in DATE columns
	This variable is supported in Informix database server versions 7.2x, 8.x, and 9.x.
NEWCODESET	Allows new code sets to be defined between releases of IBM Informix JDBC Driver
NEWLOCALE	Allows new locales to be defined between releases of IBM Informix JDBC Driver

(2 of 2)

Important: *The **DB_LOCALE**, **CLIENT_LOCALE**, and **GL_DATE** variables are supported only if the database server supports the IBM Informix GLS feature. If these environment variables are set and your application connects to a non-GLS server (server versions earlier than 7.2), a connection exception occurs. If you connect to a non-GLS server and do not set these variables, the behavior is the same as for IBM Informix JDBC Driver Version 1.22.*

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: *IBM 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 on **GL_DATE**, see *IBM Informix GLS User's Guide*.

GL_DATE Variable

The **GL_DATE** environment variable specifies the end-user formats of values in DATE columns. This variable is supported in Informix database servers 7.2x, 8.x, and 9.x. A **GL_DATE** format string can contain the following characters:

- One or more white-space characters
- An ordinary character (other than the % symbol or a white-space character)
- A formatting directive, which is composed of the % 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.
%h	Same as the %b formatting directive
%iy	The year as a two-digit decade (00 through 99) It is the Informix-specific formatting directive for %y.

(1 of 2)

Directive	Replaced By
%iY	The year as a four-digit decade (0000 through 9999)
	It is the Informix-specific formatting directive 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 through 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 through 99)
%Y	The year as a four-digit decade (0000 through 9999)
%%	% (to allow % in the format string)

(2 of 2)

Important: *GL_DATE optional date format qualifiers for field specifications are not supported.*

For example, 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 an Informix-specific format

   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:

```
stmt = conn.createStatement();
cmd = "create table tablename (col1 date, col2 char(100));";
rc = stmt.executeUpdate(cmd);
...
String[] dateVals = {
   "'* Oct 08, 1998 this day is: (Thursday), a fine day *'",
   "{d '1998-09-29'}"
   };

String[] charVals = {
   "'* Oct 08, 1998 this day is: (Thursday), a fine day *'",
   "'* Sep 29, 1998 this day is: (Tuesday), a fine day *'"
   };
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 (char) = " + charVals[i]);
   }
```

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();
```

DBDATE Variable

Support for the **DBDATE** environment variable provides backward compati-bility for client applications that are based on Informix database server versions prior to 7.2x, 8.x, or 9.x. You should 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, IBM 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, IBM 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 an Informix-specific format. 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();
```

DBCENTURY Variable

If a **String** value represents a DATE value that has less than a three-digit year and **DBCENTURY** is set, IBM 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.
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 *IBM 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.

IBM 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 IBM Informix JDBC Driver.

For examples of how to use **DBCENTURY** with IBM 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 in conjunction 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 in conjunction with the **GL_DATE** format string to display DATE values. If the **DB_LOCALE** value is not valid, the database locale is used in conjunction with the **GL_DATE** format string.

  - ❑ If neither **CLIENT_LOCALE** nor **DB_LOCALE** values are specified, the database locale is used in conjunction 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 neither **CLIENT_LOCALE** nor **DB_LOCALE** values are 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 *IBM 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

IBM 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 IBM 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 this **DB_LOCALE** value is incorrect, the database locale (stored in the database **systables** catalog) is used in the connection and in the code-set conversion.

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 JDK 1.2 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
ASCII	364	ASCII
sjis-s	932	SJIS
sjis	57350	SJIS
utf8	57372	UTF8
big5	57352	Big5
CP1250	1250	Cp1250

(1 of 2)

Informix Code Set Name	Informix Code Set Number	JDK Code Set
CP1251	1251	Cp1251
CP1252	1252	Cp1252
CP1253	1253	Cp1253
CP1254	1254	Cp1254
CP1255	1255	Cp1255
CP1256	1256	Cp1256
CP1257	1257	Cp1257
cp949	57356	Cp949
KS5601	57356	Cp949
ksc	57356	Cp949
ujis	57351	EUC_JP
gb	57357	ISO2022CN_GB
GB2312-80	57357	ISO2022CN_GB
cp936	57357	ISO2022CN_GB

(2 of 2)

You cannot use an Informix locale with a code set for which there is no JDK-supported encoding. This incorrect usage results in an `Encoding not supported` error message.

If the connection is made but the database server returns a warning of a mismatch between the **DB_LOCALE** value sent and the real value in the database **systables** catalog, the correct database locale is automatically extracted from the **systables** catalog, and the client uses the correct JDK encoding for the connection.

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
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 using the platform's 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 **InputStream-Reader()** 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.

Connecting 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.

Code-Set Conversion for TEXT Data Types

IBM 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.

Converting Using 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 values listed in the following table.

Value	Result
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.

Converting Using JDK Methods

The **getBytes()**, **getString()**, **InputStreamReader()**, and **OutputStream-Writer()** methods take a code-set parameter that converts to and from Unicode and the specified code set. These methods are covered in detail in Sun's JDK documentation.

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.

User-Defined Locales

IBM Informix JDBC Driver uses the JDK internationalization API to manipulate international data. The classes and methods in this API take a JDK 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 JDK names. These mappings are kept in internal tables, which are updated periodically.

For example, the Informix and JDK names for the ASCII code set are 8859-1 and 8859_1, respectively. IBM Informix JDBC Driver maps 8859-1 to 8859_1 in its internal tables and uses the appropriate JDK name in the JDK classes and methods.

Because new locales may be created between updates of these tables, two new 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.

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.

Support for Localized 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 localized 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 localized versions of all message text supported by IBM 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 IBM Informix GLS Variables" on page 6-5.

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 Chapter 4, "Working With Informix Types."

For an example of how to use the localized error message support feature, see the **locmsg.java** program, which is included with IBM Informix JDBC Driver.

Tuning and Troubleshooting

In This Chapter

This chapter provides tuning and troubleshooting information for IBM Informix JDBC Driver. It covers the following topics:

- Debugging Your JDBC API Program
- Managing Performance

Debugging Your JDBC API Program

If your Java program contains JDBC API programming errors, you might want to use the debug version of IBM Informix JDBC Driver instead of the optimized version to try to find where the errors occur in your program.

Using the Debug Version of the Driver

The debug version of IBM Informix JDBC Driver, called **ifxjdbc-g.jar**, is exactly the same as the optimized version (called **ifxjdbc.jar**), except that it has been compiled with the **-g** option instead of the **-O** option.

The difference in compilation options also means that the debug version of IBM Informix JDBC Driver has been embedded with tracing information. When you use the debug version of the driver, you can turn on tracing and get a better idea of what the JDBC API portion of your Java program is actually doing.

For instructions on how to use the debug version of IBM Informix JDBC Driver in a Java application or Java applet, refer to "Using the Driver in an Application" on page 1-18 or "Using the Driver in an Applet" on page 1-20, respectively.

Turning On Tracing

Trace output consists of the following two kinds of information:

- General information from IBM Informix JDBC Driver
- Informix native SQLI protocol messages sent between your Java program and the Informix database server

To turn on tracing, specify the environment variables **TRACE**, **TRACEFILE**, **PROTOCOLTRACE**, and **PROTOCOLTRACEFILE** in the connection property list when you establish a connection to an Informix database or database server. The following table describes the tracing environment variables.

Environment Variable	Description
TRACE	Traces general information from IBM Informix JDBC Driver.
	Can be set to one of the following levels:
	■ `0`. Tracing not enabled This is the default value.
	■ `1`. Traces the entry and exit points of methods
	■ `2`. Same as Level 1, except generic error messages are also traced
	■ `3`. Same as Level 2, except data variables are also traced
TRACEFILE	Specifies the full pathname of the operating system file on the client computer to which the **TRACE** messages are written

(1 of 2)

Environment Variable	Description
PROTOCOLTRACE	Traces the SQLI protocol messages sent between your Java program and the Informix database server
	Can be set to the following levels:
	■ `0`. Protocol tracing not enabled This is the default value.
	■ `1`. Traces message IDs
	■ `2`. Same as Level 1, except the data in the message packets is also traced
PROTOCOLTRACFILE	Specifies the full pathname of the operating system file on the client computer to which the **PROTOCOLTRACE** messages are written

(2 of 2)

The following example of a database URL specifies the highest level of protocol tracing and sets tracing output to the operating system file **/tmp/trace.out**:

```
String url = "jdbc:informix-sqli://123.45.67.89:1533:
    informixserver=myserver;user=rdtest;password=test;
PROTOCOLTRACE=2;PROTOCOLTRACEFILE=/tmp/trace.out";
```

For more information on establishing a connection to an Informix database or database server and using a property list, see Chapter 2, "Connecting to the Database." For information about how to set these connection properties using a **DataSource** object instead of a database URL, see Appendix B, "DataSource Extensions."

Managing 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.

The FET_BUF_SIZE and BIG_FET_BUF_SIZE Environment Variables

When a SELECT statement is sent from a Java program to an Informix database, the returned rows, or *tuples*, are stored in a tuple buffer in IBM 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 32,767. 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**.

XPS

In IBM Informix Extended Parallel Server, Version 8.4, you can use the **BIG_FET_BUF_SIZE** connection property to override the default size of the tuple buffer. The XPS server allows the fetch buffer size to be increased up to 2 GB.

BIG_FET_BUF_SIZE can be set to any positive integer less than or equal to 2 GB. If the **BIG_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 **BIG_FET_BUF_SIZE**. This could help increase the insert cursor performance for tables fragmented on multiple coservers in IBM Informix Extended Parallel Server, Version 8.4. ♦

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 **BIG_FET_BUF_SIZE** environment variable to increase the size of the tuple buffer.

For more information on setting Informix environment variables, see Chapter 2, "Connecting to the Database." For more information on increasing the fetch buffer size, see the *IBM Informix Guide to SQL: Reference*.

Managing 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, IBM 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 1-20.

■ To always store the data in memory, set the **LOBCACHE** value to a negative number.

If the required amount of memory is not available, IBM 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:158.58.9.37:711test: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, IBM 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 on how to use the TEXT and BYTE data types in a Java program, refer to "BYTE and TEXT Data Types" on page 4-7. For programming information on how to use the BLOB and CLOB data types in a Java program, refer to "Smart Large Object Data Types" on page 4-42.

Reducing 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 client's tuple buffer. The database server automatically closes the cursor after all the rows have been retrieved.

IBM Informix JDBC Driver might or might not have additional rows in the client's tuple buffer before the next **ResultSet.next()** method is called. Therefore, unless IBM 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 "Using the Auto Free Feature" on page 3-38.

The database server automatically frees the cursor resources right 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 Appendix A, "Sample Code Files." In these examples, the variables are set with the **Properties.put()** method.

For more information on setting Informix environment variables, refer to "Using Informix Environment Variables" on page 2-16.

Using 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 "Performing Bulk Inserts" on page 3-5.

Using 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. IBM 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 2.0 *Standard Extension Specification* provided by Sun Microsystems, available from the following Web site: http://java.sun.com/products/jdk/1.2/docs/guide/jdbc/index.html

Important: *This feature does not affect IfxXAConnectionPoolDataSource, which operates under the assumption that connection pooling is handled by the transaction manager.*

The following sections discuss how to use connection pooling with IBM Informix JDBC Driver:

- "Deploying a ConnectionPoolDataSource Object," next
- "Tuning the Connection Pool Manager" on page 7-12
- "Using High-Availability Data Replication with Connection Pooling" on page 7-14
- "Cleaning Pooled Connections" on page 7-16

Deploying a ConnectionPoolDataSource Object

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);
   ```

Tuning 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 **ConnectionPool-DataSource** 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 physical connections that the DataSource object can obtain from the server, excluding those returned to 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 Appendix A.

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	Notes
initialPoolSize	IFMX_CPM_INIT_POOLSIZE	
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

Using High-Availability Data Replication with Connection Pooling

IBM 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 "Checking for Read-Only Status" on page 2-29).

A demonstration program is available in the **hdr** directory within the **demo** directory where IBM Informix JDBC Driver is installed. For details about the files, see Appendix A.

Cleaning 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.

The former behavior of the **scrubConnection()** method reset the database properties and closed all the statements and result sets associated with the connection by reopening the database. This required the cleanup of the statement cache maintained by the application servers.

In this implementation of IBM Informix JDBC Driver, Version 2.21, you can call the **scrubConnection()** method to:

- Retain all statements.
- Reset the database properties and connection level properties to the default values.
- Close open cursors and transactions.

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()
```

You can use the public boolean **scrubConnectionReleasesAllStatements()** method to determine whether a call to **scrubConnection()** results in the old behavior or the new behavior. For prior versions of IBM Informix JDBC Driver, the **scrubConnectionReleasesAllStatements()** method will return a `true`. With this release of IBM Informix JDBC Driver, Version 2.21, **scrubConnectionReleasesAllStatements()** will return `false`.

Managing 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 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 the default state and keeps all open statements, but closes all result sets. Calling this method is not recommended here.
Connection Pool with AppServer Implementation	Defined by user's connection pooling implementation	Returns connection to default state and retains opened statements, but closes result sets

Sample Code Files

This appendix contains tables that list and briefly describe the code examples provided with the client-side version of IBM 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 2-10.

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 IBM 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
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 A-13)
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
CallOut3.java	Executes a C function that has a Boolean OUT parameter using the **IfmxCallableStatement.IfxRegisterOutParameter()** 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

(1 of 4)

Demo Program Name	Description
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
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 Informix INTERVAL data

(2 of 4)

Demo Program Name	Description
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 IBM 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

(3 of 4)

Demo Program Name	Description
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

(4 of 4)

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 .)

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**
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 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**
	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

(1 of 2)

Demo Program Name	Description
demo4.java	Inserts a **Struct** object into a named row column
demo5.java	Fetches an Informix SET column into a **java.util.HashSet** object
demo6.java	Fetches an 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 an Informix SET column
demo8.java	Fetches an Informix SET column into a **java.sql.Array** object
dropDB.java	Drops the database

(2 of 2)

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.html`
proxy.html	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

(1 of 2)

Demo Program Name	Description
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

(2 of 2)

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 **s**ample 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 appendix lists the 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 2.0 *Standard Extension Specification* provided by Sun Microsystems, available from the following Web site: http://java.sun.com/products/jdk/1.2/docs/guide/jdbc/index.html.

IBM 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

Reading and Writing 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.

Getting and Setting Standard Properties

The following methods are defined in the extended **DataSource** interface for getting and setting properties defined in Sun's JDBC 2.0 *Standard Extension Specification*.

Property	get*XXX*() and set*XXX*() Method Signatures
portNumber	`public int getPortNumber();`
	`public void setPortNumber(int value);`
databaseName	`public String getDatabaseName();`
	`public void setDatabaseName(String value);`
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 IBM Informix JDBC Driver.

Getting and Setting Informix Connection Properties

The following methods are defined in the extended **DataSource** interface for getting and setting Informix environment variable values.

Environment Variable	getIfx*XXX*() and setIfx*XXX*() Method Signatures
CLIENT_LOCALE	`public String getIfxCLIENT_LOCALE();`
	`public void setIfxCLIENT_LOCALE(String value);`
DBANSIWARN	`public boolean isIfxDBANSIWARN();`
	`public void setIfxDBANSIWARN(boolean value);`
DBCENTURY	`public String getIfxDBCENTURY();`
	`public void setIfxDBCENTURY(String value);`
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_CACHE_TYPE	`public boolean isIfxENABLE_CACHE_TYPE();`
	`public void setIfxENABLE_CACHE_TYPE(boolean value);`
ENABLE_HDRSWITCH	`public boolean getIfxENABLE_HDRSWITCH();`
	`public void setIfxENABLE_HDRSWITCH(boolean value);`

(1 of 5)

Environment Variable	getIfx*XXX*() and setIfx*XXX*() Method Signatures
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_BATCHUPDATE_PER_ SPEC	`public boolean getIfxBATCHUPDATE_PER_SPEC();` `public void setIfxBATCHUPDATE_PER_SPEC(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);`
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);`
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);`

(2 of 5)

Environment Variable	getIfx*XXX*() and setIfx*XXX*() 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);`

(3 of 5)

Environment Variable	getIfx*XXX*() and setIfx*XXX*() Method Signatures
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);`
PROTOCOLTRACE	`public int getIfxPROTOCOLTRACE();` `public void setIfxPROTOCOLTRACE(int value);`
PROTOCOLTRACEFILE	`public String getIfxPROTOCOLTRACEFILE();` `public void setIfxPROTOCOLTRACEFILE(String value);`
PROXY	`public String getIfxPROXY();` `public void setIfxPROXY(String value);`
PSORT_DBTEMP	`public String getIfxPSORT_DBTEMP();` `public void setIfxPSORT_DBTEMP(String value);`

(4 of 5)

Environment Variable	getIfx*XXX*() and setIfx*XXX*() Method Signatures
PSORT_NPROCS	```public String getIfxPSORT_NPROCS();``` ```public void setIfxPSORT_NPROCS(String value);```
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);```
STMT_CACHE	```public String getIfxSTMT_CACHE();``` ```public void setIfxSTMT_CACHE(String value);```
TRACE	```public int getIfxTRACE();``` ```public void setIfxTRACE(int value);```
TRACEFILE	```public String getIfxTRACEFILE();``` ```public void setIfxTRACEFILE(String value);```
USEV5SERVER	```public boolean isIfxUSEV5SERVER();``` ```public void setIfxUSEV5SERVER(boolean value);```

(5 of 5)

Getting and Setting 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 "Using a Connection Pool" on page 7-10. The following table summarizes them.

Property	get*XXX*() and set*XXX*() 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()`
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 appendix discusses mapping issues between data types defined in a Java program and the data types supported by the Informix database server. It covers the following topics:

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 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
BINARY	BYTE
BIT	Not supported
REF	Not supported
CHAR	CHAR(n)
DATE	DATE
DECIMAL	DECIMAL
DOUBLE	FLOAT
FLOAT	FLOAT[1]
INTEGER	INTEGER
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)[2]
TINYINT	SMALLINT

(1 of 2)

JDBC API Data Type	Informix Data Type
VARBINARY	BYTE
VARCHAR	VARCHAR(*m,r*)

[1] 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.

[2] Informix DATETIME types are very restrictive and are not interchangeable. For more information, see "Field Lengths and Date-Time Data" on page C-25.

Data Type Mapping Between Extended Types and Java and JDBC Types

The following table lists mappings between the extended data types supported in IBM Informix Dynamic Server and the corresponding Java and JDBC types.

JDBC Type	Java Object Type	Informix Type
java.sql.Types.OTHER java.sql.Types.SMALLINT	boolean smallint	BOOLEAN IfxTypes.IFX_TYPE_BOOL
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
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

JDBC Type	Java Object Type	Informix Type
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

(2 of 2)

A Java **boolean** object can map to a **smallint** object or an Informix BOOLEAN data type. IBM Informix JDBC Driver attempts to map it according to the column type. However, in cases such as **PreparedStatement** host variables, IBM Informix JDBC Driver cannot access the column types, so the mapping is somewhat limited. For more details on data type mapping, refer to "Data Type Mapping for PreparedStatement.setXXX() Extensions."

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 Chapter 5, "Working with Opaque Types."

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 *IBM 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

(1 of 2)

DataBlade API Type	Java Type
mi_string	String
mi_unsigned_char1	String
mi_unsigned_int8	long
mi_unsigned_integer	int
mi_unsigned_smallint	short
mi_wchar	String

(2 of 2)

The C struct may contain padding bytes. IBM 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

IBM Informix Dynamic Server 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 IBM 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, IBM 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 table on page C-2. 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.

Using 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.set*XXX*()** 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.

Using 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 Informix data types from **com.informix.lang.IfxTypes**

  For more information about the available types, see "Using the Ifx-Types Class" on page C-13.

- **clearBindColType()**, which resets values set through the previous two methods

In the following sections:

- 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`.

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.

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.

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 an 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_LVARCHA
R);
((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
```

Using Other Mapping Extensions

The remaining method signatures are listed next, along with any additional considerations that apply. In each case, the Informix type must be the last parameter to the standard JDBC **PreparedStatement.set*XXX*()** 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
```

Using the IfxTypes Class

The extended **IfmxPreparedStatement** methods require you to pass in the 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_CHAR	CHAR
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_NVCHAR	NVCHAR
IfxTypes.IFX_TYPE_INT8	INT8
IfxTypes.IFX_TYPE_SERIAL8	SERIAL8

(1 of 2)

IfxTypes Constant	Informix Data Type
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.IFX_TYPE_BOOL	BOOLEAN
IfxTypes.IFX_TYPE_IMPEXP	IMPEXP
IfxTypes.IFX_TYPE_IMPEXPBIN	IMPEXPBIN
IfxTypes.IFX_TYPE_CLOB	CLOB
IfxTypes.IFX_TYPE_BLOB	BLOB

(2 of 2)

Extension Summary

The following table lists the **PreparedStatement.set*XXX*()** methods that
IBM 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 on
page C-2. The table lists the **set*XXX*()** methods you can use to write data of a
particular JDBC API data type.

An uppercase and bold **X** indicates the recommended **setXXX()** method to use; a lowercase x indicates other **setXXX()** methods supported by IBM Informix JDBC Driver.

setXXX() Method	\\	TINYINT	SMALLINT	INTEGER	BIGINT	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC	BIT	CHAR	VARCHAR	LONGVARCHAR	BINARY	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP
	JDBC API Data Types from java.sql.Types																			
setByte()		**X**	x	x	x	x	x	x	x	x		x^1	x^1							
setShort()		x	**X**	x	x	x	x	x	x	x		x^1	x^1							
setInt()		x	x	**X**	x	x	x	x	x	x		x^1	x^1							
setLong()		x	x	x	**X**	x	x	x	x	x		x^1	x^1							
setFloat()		x	x	x	x	**X**	x	x	x	x		x^1	x^1							
setDouble()		x	x	x	x	x	**X**	**X**	x	x		x^1	x^1							
setBigDecimal()		x	x	x	x	x	x	x	**X**	**X**		x	x							
setBoolean()		x	x	x	x	x	x	x	x	x		x	x							
setString()		x	x	x	x	x	x	x	x	x		**X**	**X**	x	x	x	x	x	x	x
setBytes()														x	**X**	**X**	x			
setDate()												x	x					**X**		x
setTime()												x	x						**X**	x
setTimestamp()												x	x					x		**X**
setAsciiStream()														**X**	x	x	x			
setCharacterStream()														**X**	x	x	x			
setUnicodeStream()																				

(1 of 2)

setXXX() Method	JDBC API Data Types from java.sql.Types																		
	TINYINT	SMALLINT	INTEGER	BIGINT	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC	BIT	CHAR	VARCHAR	LONGVARCHAR	BINARY	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP
setBinaryStream()													x	x	x	**X**			
setObject()	x	x	x	x	x	x	x	x	x		x	x	x^2	x	x	x^2	x	x^3	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.

[3] A **Timestamp** object is written instead of a **Time** object.

(2 of 2)

The **setNull()** method writes an SQL null value.

The following table lists the **PreparedStatement.setXXX()** methods that IBM Informix JDBC Driver supports for the Informix extended data types, the mappings for which are shown in the table on . 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 **set*XXX*()** method to use; a lowercase x indicates other **set*XXX*()** methods supported by IBM Informix JDBC Driver. The table does not include **set*XXX*()** methods that you cannot use with any of the Informix extended data types.

set*XXX*() Method	Informix Extended Data Types										
	BOOLEAN	LVARCHAR	Opaque types	BLOB	CLOB	BYTE	TEXT	NAMED ROW	UNNAMED ROW	SET or MULTISET	LIST
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	X	X	x	x
setArray()										x	x
setBlob()				X							
setClob()					X						

The **setNull()** method writes an SQL null value.

Data Type Mapping for ResultSet.getXXX() Methods

Use the **ResultSet.get*XXX*()** methods to transfer data from an Informix database to a Java program that uses the JDBC API to connect to an 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 following table lists the **ResultSet.get*XXX*()** methods that IBM 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 on . The table lists the **get*XXX*()** methods you can use to retrieve data of a particular JDBC API data type.

An uppercase and bold **X** indicates the recommended **get*XXX*()** method to use; a lowercase x indicates other **get*XXX*()** methods supported by IBM Informix JDBC Driver.

get*XXX*() Method	JDBC API Data Types from java.sql.Types																		
	TINYINT	SMALLINT	INTEGER	BIGINT	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC	BIT	CHAR	VARCHAR	LONGVARCHAR	BINARY	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP
getByte()	X	x	x	x	x	x	x	x	x		x^1	x^1							
getShort()	x	X	x	x	x	x	x	x	x		x^1	x^1							
getInt()	x	x	X	x	x	x	x	x	x		x^1	x^1							
getLong()	x	x	x	X	x	x	x	x	x		x^1	x^1							
getFloat()	x	x	x	x	X	x	x	x	x		x^1	x^1							

(1 of 2)

getXXX() Method	TINYINT	SMALLINT	INTEGER	BIGINT	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC	BIT	CHAR	VARCHAR	LONGVARCHAR	BINARY	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP
	JDBC API Data Types from java.sql.Types																		
getDouble()	x	x	x	x	x	**X**	**X**	x	x		x^1	x^1							
getBigDecimal()	x	x	x	x	x	x	x	**X**	**X**		x	x							
getBoolean()	x	x	x	x	x	x	x	x	x		x	x							
getString()	x	x	x	x	x	x	x	x	x		**X**	**X**	x	x	x	x	x	x	x
getBytes()													x	**X**	**X**	x			
getDate()											x	x					**X**		x
getTime()											x	x						**X**	x
getTimestamp()											x	x					x		**X**
getAsciiStream()													**X**	x	x	x			
getCharacterStream()													**X**	x	x	x			
getUnicodeStream()																			
getBinaryStream()														x	x	**X**			
getObject()	x	x	x	x	x	x	x	x	x		x	x	x^2	x	x	x^2	x	x^3	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.

[3] A **Timestamp** object is returned instead of a **Time** object.

(2 of 2)

The **getXXX()** methods return a null value if the retrieved column value is an SQL null value.

The following table lists the **ResultSet.get*XXX*()** methods that IBM Informix JDBC Driver supports for the Informix extended data types, the mappings for which are shown in the table on . The table lists the **get*XXX*()** methods you can use to retrieve data of a particular extended data type.

An uppercase and bold **X** indicates the recommended **get*XXX*()** method to use; a lowercase x indicates other **get*XXX*()** methods supported by IBM Informix JDBC Driver. The table does not include **get*XXX*()** methods that you cannot use with any of the Informix extended data types.

get*XXX*() Method	Informix Extended Data Types										
	BOOLEAN	LVARCHAR	Opaque types	BLOB	CLOB	BYTE	TEXT	NAMED ROW	UNNAMED ROW	SET or MULTISET	LIST
getByte()	x	x									
getShort()	x										
getInt()	x										
getBoolean()	**X**										
getString()		**X**			x		x				
getBytes()				x		x					
getAsciiStream()		x			x		**X**				
getCharacterStream()		x			x		**X**				
getBinaryStream()	x			x		**X**					
getObject()	x	x	**X**	x	x	x	x	**X**	**X**	x	x
getArray()										x	x
getBlob()				**X**							
getClob()					**X**						

The **get*XXX*()** methods return a null value if the retrieved column value is an SQL null value.

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 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 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)

(1 of 2)

Java Data Type	SQL Data Type
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.

(2 of 2)

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

(1 of 2)

ifxtype Parameter Type from com.informix.lang.IfxTypes	Informix Data Type
IFX_TYPE_BYTE	BYTE
IFX_TYPE_TEXT	TEXT
IFX_TYPE_VARCHAR	VARCHAR
IFX_TYPE_INTERVAL	INTERVAL
IFX_TYPE_NCHAR	NCHAR
IFX_TYPE_NVCHAR	NVCHAR
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

(2 of 2)

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_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
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 C-25).
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 C-25).
IFX_TYPE_NCHAR	java.lang.String
IFX_TYPE_INT8	long
IFX_TYPE_SERIAL8	long
IFX_TYPE_BOOL	boolean

(1 of 2)

ifxtype Parameter Type from com.informix.lang.IfxTypes	Java Data Type
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.

(2 of 2)

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**.

Notices

IBM may not offer the products, services, or features discussed in this document in all countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

> IBM Director of Licensing
> IBM Corporation
> North Castle Drive
> Armonk, NY 10504-1785
> U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

> IBM World Trade Asia Corporation
> Licensing
> 2-31 Roppongi 3-chome, Minato-ku
> Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

> IBM Corporation
> J46A/G4
> 555 Bailey Avenue
> San Jose, CA 95141-1003
> U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:
This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

> © (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. (enter the year or years). All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

AIX; DB2; DB2 Universal Database; Distributed Relational Database Architecture; NUMA-Q; OS/2, OS/390, and OS/400; IBM Informix®; C-ISAM®; Foundation.2000™; IBM Informix® 4GL; IBM Informix® DataBlade® Module; Client SDK™; Cloudscape™; Cloudsync™; IBM Informix® Connect; IBM Informix® Driver for JDBC; Dynamic Connect™; IBM Informix® Dynamic Scalable Architecture™ (DSA); IBM Informix® Dynamic Server™; IBM Informix® Enterprise Gateway Manager (Enterprise Gateway Manager); IBM Informix® Extended Parallel Server™; i.Financial Services™; J/Foundation™; MaxConnect™; Object Translator™; Red Brick Decision Server™; IBM Informix® SE; IBM Informix® SQL; InformiXML™; RedBack®; SystemBuilder™; U2™; UniData®; UniVerse®; wintegrate® are trademarks or registered trademarks of International Business Machines Corporation.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Windows, Windows NT, and Excel are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

Other company, product, and service names used in this publication may be trademarks or service marks of others.

Glossary

applet
A program created with Java classes, that is not intended to be run on its own but rather to be embedded in another application, such as a browser.

autocommit mode
A mode in which a COMMIT statement is automatically executed after each statement sent to the database server.

BLOB
A smart large object data type that stores any kind of binary data, including images. The database server performs no interpretation on the contents of a BLOB column.

See also *smart large object*.

blobpage
The unit of disk allocation within a blobspace. The size of a blobpage is determined by the DBA and can vary from blobspace to blobspace.

blobspace
A logical collection of chunks that is used to store TEXT and BYTE data.

See also *dbspace*.

built-in data type
A fundamental data type defined by the database server; for example, INTEGER, CHAR, or SERIAL8.

BYTE
A built-in data type for a simple large object that stores any type of binary data. The object can be as large as 2^{31} bytes.

cast	A mechanism that the database server uses to convert data from one data type to another. The server provides built-in casts that it performs automatically. Users can create both implicit and explicit casts.
	See also *cast support function*, *explicit cast*, *implicit cast*, *system-defined cast*.
cast support function	A function that is used to implement an implicit or explicit cast by performing the necessary operations for conversion between two data types. A cast support function is optional unless the internal storage representations of the two data types are not equivalent.
CLASSPATH	An environment variable that tells the Java virtual machine (JVM) and other applications where to find the Java class libraries used in a Java program.
CLOB	A data type for a smart large object that stores text items, such as PostScript or HTML files.
	See also *smart large object*.
code set	A set of unique bit patterns that are mapped to the characters contained in a specific natural language, which include the alphabet, digits, punctuation, and diacritical marks. There can be more than one code set for a language; for example, the code sets for the English language include ASCII, ISO8895-1, and Microsoft 1252. You specify the code set that your database server uses when you set the GLS locale.
	See also *locale*.
collection	An instance of a collection data type; a group of elements of the same data type stored in a SET, MULTISET, or LIST object.
	See also *collection data type*.
collection data type	A complex data type that groups values, called elements, of a single data type in a column. Collection data types consist of the SET, MULTISET, or LIST type constructor and an element type, which can be any data type, including a complex data type.
complex data type	A data type that is built from a combination of other data types using an SQL type constructor or the CREATE ROW TYPE statement and whose components can be accessed through SQL statements. Complex data types include collection data types and row data types.
concurrency	The ability of two or more processes to access the same database simultaneously.

connection	An association between an application and a database environment, created by a CONNECT or DATABASE statement. Database servers can also have connections to one another.
	See also *explicit connection*, *implicit connection*.
constructed data type	A complex data type created with a type constructor; for example, a collection data type or an unnamed row data type.
CORBA	(Common Object Request Broker Architecture) The CORBA 2.0 specification describes a convention called Object Request Broker (ORB), the infrastructure for distributed-object computing. CORBA enables client applications to communicate with remote objects and invoke operations statically or dynamically.
cursor	An SQL object that points to a row in the results table returned by a SELECT statement. A cursor enables an application to process data from multiple data sets simultaneously rather than sequentially.
cursor function	A user-defined function that returns one or more rows of data and requires a cursor to execute. An SPL function is a cursor function when its RETURN statement contains the WITH RESUME keywords. An external function is a cursor function when it is defined as an iterator function.
database URL	A URL passed to the **DriverManager.getConnection()** method that specifies the subprotocol (the database connectivity mechanism), the database or database server identifier, and a list of properties that can include Informix environment variables.
data type	See *built-in data type*, *extended data type*.
DataBlade API	The C application programming interface (API) for IBM Informix Dynamic Server. The DataBlade API is used for the development of DataBlade modules. The DataBlade API contains routines to process data in the database server and return the results to the calling application.
DataBlade API data types	A set of Informix C data types that correspond to some of the Informix SQL data types, including extended data types. You should use these data types instead of the standard C data types to ensure portable applications.

dbspace
A logical collection of one or more chunks within which you store databases and tables. Because chunks represent specific regions of disk space, the creators of databases and tables can control where their data is physically located by placing databases or tables in specific dbspaces.

See also *BLOB*.

delimiter
The boundary of an input field or the terminator for a database column or row. Some files and prepared objects require a semicolon (;), comma (,), pipe (|), space, or tab delimiter between statements.

distinct data type
A data type based on an existing opaque, built-in, distinct, or named row data type, known as its source type. The distinct data type has the same internal storage representation as its source type, but it has a different name. To compare a distinct data type with its source type requires an explicit cast. A distinct data type inherits all routines that are defined on its source type.

DOM
(Document Object Model) A tree of objects with interfaces for traversing the tree and writing an XML version of it, as defined by the Document Object Model Level 1 Specification (available at http://www.w3.org/DOM/). A DOM object has the data type **Document**.

See also *SAX*, *JAXP*, *XML.*

explicit cast
A cast that requires a user to specify the CAST AS keyword or cast operator (::) to convert data from one data type to another.

See also *cast*, *cast support function*.

explicit connection
A connection made to a database environment that uses the CONNECT statement.

See also *implicit connection*.

extended data type
A data type that is not built-in; namely, a collection data type, row data type, opaque data type, or distinct data type.

fundamental data type
A data type that cannot be broken into smaller pieces by the database server using SQL statements; for example, built-in data types and opaque data types.

Global Language Support (GLS)	An application environment that allows Informix application programming interfaces (APIs) and database servers to handle different languages, cultural conventions, and code sets. Developers use the GLS libraries to manage all string, currency, date, and time data types in their code. Using GLS, you can add support for a new language, character set, and encoding by editing resource files, without access to the original source code and without rebuilding the client software.
host variable	A C or COBOL program variable that is referenced in an embedded statement. A host variable is identified by the dollar sign ($) or colon (:) that precedes it.
implicit cast	A cast that the database server automatically performs to convert data from one data type to another. See also *cast*, *cast support function*.
implicit connection	A connection made using a database statement (DATABASE, CREATE DATABASE, START DATABASE, DROP DATABASE). See also *explicit connection*.
IP address	The unique ID of each computer on the Internet. The format consists of four numerical strings separated by dots, such as `123.45.67.89`.
jar utility	A JavaSoft utility that creates Java archive, or JAR, files. JAR is a platform-independent file format that aggregates many files into one.
JAXP	(Java API for XML Parsing) An API for parsing XML documents, using two main parsing methods, Simple API for XML (SAX) and Document Object Model (DOM.) JAXP provides a "plugability layer" around the SAX and DOM APIs, which standardizes access to different implementations of SAX and DOM. The plugability layer is a set of methods for instantiating and configuring SAX parsers and creating DOM objects. For more information, see http://java.sun.com/xml. See also *SAX*, *DOM*, *XML*
keyword	A word that has meaning to a programming language. In Informix SQL, keywords are shown in syntax diagrams in all uppercase letters. They must be used in SQL statements exactly as shown in the syntax, although they can be in either uppercase or lowercase letters.

large object   A data object that exceeds 255 bytes in length. A large object is logically stored in a table column but physically stored independently of the column, because of its size. Large objects can contain non-ASCII data. IBM Informix Dynamic Server recognizes two kinds of large objects; simple large objects (TEXT, BYTE) and smart large objects (CLOB and BLOB).

See also *simple large object*, *smart large object*.

LIST data type   A collection data type in which elements are ordered and duplicates are allowed.

See also *collection data type*.

locale   A set of files that define the native-language behavior of the program at run-time. The rules are usually based on the linguistic customs of the region or the territory. The locale can be set through an environment variable that dictates output formats for numbers, currency symbols, dates, and time, as well as collation order for character strings and regular expressions.

See also *Global Language Support (GLS)*.

LVARCHAR   A built-in data type that stores varying-length character data greater than 256 bytes. It is used for input and output casts for opaque data types. LVARCHAR supports code-set order for comparisons of character data.

metadata   Data about data. Metadata provides information about data in the database or used in the application. Metadata can be data attributes, such as name, size, and data type, or descriptive information about data.

MULTISET data type   A collection data type in which elements are not ordered and duplicates are allowed.

See also *collection data type*.

named row data type A row data type that is created with the CREATE ROW TYPE statement and has a name. A named row data type can be used to construct a typed table and can be part of a type or table hierarchy.

See also *row data type*, *unnamed row data type*.

opaque data type   An extended data type that contains one or more members but whose internal structure is interpreted by the database server using user-defined support routines.

RMI	(Remote Method Invocation) A method for creating distributed Java-to-Java applications, in which the methods of remote Java objects can be invoked from other Java virtual machines, possibly on different hosts.

row data type	A complex data type consisting of a group of ordered data elements (fields) of the same or different data types. The fields of a row type can be of any supported built-in or extended data type, including complex data types, except SERIAL and SERIAL8 and, in certain situations, TEXT and BYTE.

There are two kinds of row data types:

- Named row types, created using the CREATE ROW TYPE statement
- Unnamed row types, created using the ROW constructor

See also *named row data type*, *unnamed row data type*.

SAX	(Simple API for XML) An event-driven interface for processing XML documents in which the parser invokes one of several methods supplied by the caller when a "parsing event" occurs. Events include recognizing an XML tag, finding an error, encountering a reference to an external entity, or processing am Document Type Definition (DTD) specification.

See also *DOM*, *XML*, *JAXP*.

scroll cursor	A cursor that can fetch the next row or any prior row, thereby allowing it to read rows multiple times.

servlet	An extension method for many common protocols, especially HTTP. Servlets are modules that run inside request/response-oriented servers. Servlets are similar to applets in that their classes might be dynamically loaded, either across the network or from local storage. However, servlets differ from applets in that they lack a graphical interface.

SET data type	A collection data type in which elements are not ordered and duplicates are not allowed.

See also *collection data type*.

simple large object	A large object that is stored in a blobspace, is not recoverable, and does not obey transaction isolation modes. Simple large objects include TEXT and BYTE data types.

See also *TEXT*, *BYTE*.

smart large object	A large object that:
	■ Is stored in an sbspace, a logical storage area that contains one or more chunks
	■ Has read, write, and seek properties similar to a UNIX file
	■ Is recoverable
	■ Obeys transaction isolation modes
	■ Can be retrieved in segments by an application
	Smart large objects include CLOB and BLOB data types.
sqlhosts file	An Informix file containing information that lets a client application find and connect to an Informix database server anywhere on the network.
SQLSTATE	A variable that contains status values about the outcome of SQL statements.
support routines	The internal routines that the database server automatically invokes to process a data type, cast, aggregate, or access method.
	The database server uses user-defined support routines to perform operations (such as converting to and from the internal, external, and binary representations of the type) on opaque data types.
	A secondary access method uses a support routine in an operator class to perform operations (such as building or searching) on an index.
sysmaster database	A master database created and maintained by every Informix database server. The **sysmaster** database contains the ON-Archive catalog tables and system monitoring interface (SMI) tables. Do not modify this database.
system catalog	A group of database tables that contain information about the database itself, such as the names of tables or columns in the database, the number of rows in a table, the information about indexes and database privileges, and so on.
system-defined cast	A cast that is built into the database server. A system-defined cast performs automatic conversions between different built-in data types.
TEXT	A built-in data type for a simple large object that stores text data and can be as large as 2^{31} bytes.
tuple buffer	The section of IBM Informix JDBC Driver memory that stores the retrieved rows from a SELECT statement.

unnamed row data type   A row type created with the ROW constructor that has no defined name and no inheritance properties. Two unnamed row types are equivalent if they have the same number of fields and if corresponding fields have the same data type, even if the fields have different names.

XML   (Extensible Markup Language) A markup language defined by the World Wide Web Consortium (W3C) that provides rules, guidelines, and conventions for describing structured data in a plain text, editable file. XML uses tags only to delimit pieces of data, leaving the interpretation of the data to the application that uses it.

See also *DOM*, *SAX*, *JAXP*.

Error Messages

-79700   Method not supported

IBM Informix JDBC Driver does not support this JDBC method.

-79702   Can't create new object

The software could not allocate memory for a new **String** object.

-79703   Row/column index out of range

The row or column index is out of range. Compare the index to the number of rows and columns expected from the query to ensure that it is within range.

-79704   Can't load driver

IBM Informix JDBC Driver could not create an instance of itself and register it in the **DriverManager** class. The rest of the **SQLException** text describes what failed.

-79705   Incorrect URL format

The database URL you have submitted is invalid. IBM Informix JDBC Driver does not recognize the syntax. Check the syntax and try again.

-79706   Incomplete input

An invalid character was found during conversion of a **String** value to an **IntervalDF** or **IntervalYM** object. Check "INTERVAL Data Type" on page 4-12 for correct values.

-79707   Invalid qualifier

An error was found during construction of an **Interval** qualifier from atomic elements: length, start, or end values. Check the length, start, and end values to verify that they are correct. See "INTERVAL Data Type" on page 4-12 for correct values.

-79708   Can't take null input

The string you have provided is null. IBM Informix JDBC Driver does not understand null input in this case. Check the input string to ensure that it has the proper value.

-79709   Error in date format

The expected input is a valid date string in the following format: *yyyy-mm-dd*. Check the date and verify that it has a four-digit year, followed by a valid two-digit month and two-digit day. The delimiter must be a hyphen (-).

-79710   Syntax error in SQL escape clause

Invalid syntax was passed to a JDBC escape clause. Valid JDBC escape clause syntax is demarcated by curly braces and a keyword: for example, {*keyword syntax*}. Check the JDBC 2.0 documentation from Sun Microsystems for a list of valid escape clause keywords and syntax.

-79711   Error in time format

An invalid time format was passed to a JDBC escape clause. The escape clause syntax for time literals has the following format: {t '*hh:mm:ss*'}.

-79712   Error in timestamp format

An invalid time stamp format was passed to a JDBC escape clause. The escape clause syntax for time stamp literals has the following format: {ts '*yyyy-mm-dd hh:mm:ss.f*...'}.

-79713   Incorrect number of arguments

An incorrect number of arguments was passed to the scalar function escape syntax. The correct syntax is {fn *function*(*arguments*)}. Verify that the correct number of arguments was passed to the function.

-79714   `Type not supported`

You have specified a data type that is not supported by IBM Informix JDBC Driver. Check your program to make sure the data type used is among those supported by the driver.

-79715   `Syntax error`

Invalid syntax was passed to a JDBC escape clause. Valid JDBC escape clause syntax is demarcated by curly braces and a keyword: {*keyword syntax*}. Check the JDBC 2.0 documentation from Sun Microsystems for a list of valid escape clause keywords and syntax.

-79716   `System or internal error`

An operating or runtime system error or a driver internal error occurred. The accompanying message describes the problem.

-79717   `Invalid qualifier length`

The length value for an **Interval** object is incorrect. See "INTERVAL Data Type" on page 4-12 for correct values.

-79718   `Invalid qualifier start code`

The start value for an **Interval** object is incorrect. See "INTERVAL Data Type" on page 4-12 for correct values.

-79719   `Invalid qualifier end code`

The end value for an **Interval** object is incorrect. See "INTERVAL Data Type" on page 4-12 for correct values.

-79720   `Invalid qualifier start or end code`

The start or end value for an **Interval** object is incorrect. See "INTERVAL Data Type" on page 4-12 for correct values.

-79721   `Invalid interval string`

An error occurred during conversion of a **String** value to an **IntervalDF** or **IntervalYM** object. Check "INTERVAL Data Type" on page 4-12 for the correct format.

-79722   `Numeric character(s) expected`

An error occurred during conversion of a **String** value to an **IntervalDF** or **IntervalYM** object. A numeric value was expected and not found. Check "INTERVAL Data Type" on page 4-12 for the correct format.

-79723   `Delimiter character(s) expected`

An error occurred during conversion of a **String** value to an **IntervalDF** or **IntervalYM** object. A delimiter was expected and not found. Check the "INTERVAL Data Type" on page 4-12 for the correct format.

-79724   `Character(s) expected`

An error occurred during conversion of a **String** value to an **IntervalDF** or **IntervalYM** object. `End of string` was encountered before conversion was complete. Check "INTERVAL Data Type" on page 4-12 for the correct format.

-79725   `Extra character(s) found`

An error occurred during conversion of a **String** value to an **IntervalDF** or **IntervalYM** object. `End of string` was expected, but there were more characters in the string. Check "INTERVAL Data Type" on page 4-12 for the correct format.

-79726   `Null SQL statement`

The SQL statement passed in was null. Check the SQL statement string of your program to make sure it contains a valid statement.

-79727   `Statement was not prepared`

The SQL statement was not prepared properly. If you use host variables (for example, `insert into mytab values (?, ?);`) in your SQL statement, you must use **connection.prepareStatement()** to prepare the SQL statement before you can execute it.

-79728   `Unknown object type`

If this is a null opaque type, the type is unknown and cannot be processed. If this is a complex type, the data in the collection or array is of an unknown type and cannot be mapped to an Informix type. If this is a row, one of the elements in the row cannot be mapped to an Informix type. Verify the customized type mapping or data type of the object.

-79729   `Method cannot take argument`

The method does not take an argument. Refer to your Java API specification or the appropriate section of this guide to make sure you are using the method properly.

-79730   `Connection not established`

A connection was not established. You must obtain the connection by calling the **DriverManager.getConnection()** or **DataSource.getConnection()** method first.

-79731   `MaxRows out of range`

You have specified an out-of-range **maxRow** value. Make sure you specify a value between 0 and **Integer.MAX_VALUE**.

-79732   `Illegal cursor name`

The cursor name specified is not valid. Make sure the string passed in is not null or empty.

-79733   `No active result`

The statement does not contain an active result. Check your program logic to make sure you have called the **executeXXX()** method before you attempt to refer to the result.

-79734   `INFORMIXSERVER has to be specified`

INFORMIXSERVER is a property required for connecting to an Informix database. You can specify it in the database URL or as part of a **Properties** object that is passed to the **connect()** method.

-79735   `Can't instantiate protocol`

An internal error occurred during a connection attempt. Call technical support.

-79736   `No connection/statement establish yet`

There is no current connection or statement. Check your program to make sure a connection was properly established or a statement was created.

-79737   No meta data

There is no metadata available for this SQL statement. Make sure the statement generates a result set before you attempt to use it.

-79738   No such column name

The column name specified does not exist. Make sure the column name is correct.

-79739   No current row

The cursor is not properly positioned. You must first position the cursor within the result set by using a method such as **ResultSet.next()**, **ResultSet.beforeFirst()**, **ResultSet.first()**, or **ResultSet.absolute()**.

-79740   No statement created

There is no current statement. Make sure the statement was properly created.

-79741   Can't convert to

There is no data conversion possible from the column data type to the one specified. The actual data type is appended to the end of this message. Review your program logic to make sure that the conversion you have asked for is supported. Refer to Appendix C for the data mapping matrix.

-79742   Can't convert from

No data conversion is possible from the data type you specified to the column data type. The actual data type is appended to the end of this message. Check your program logic to make sure that the conversion you have asked for is supported. Refer to Appendix C for the data mapping matrix.

-79744   Transactions not supported

The user has tried to call **commit()** or **rollback()** on a database that does not support transactions or has tried to set **autoCommit** to False on a nonlogging database. Verify that the current database has the correct logging mode and review the program logic.

-79745   Read only mode not supported

Informix does not support read-only mode.

-79746   `No Transaction Isolation on non-logging db's`

Informix does not support setting the transaction isolation level on nonlogging databases.

-79747   `Invalid transaction isolation level`

If the database server could not complete the rollback, this error occurs. See the rest of the **SQLException** message for more details about why the rollback failed.

This error also occurs if an invalid transaction level is passed to **setTransactionIsolation()**. The valid values are:

- ■   `TRANSACTION_READ_UNCOMMITTED`
- ■   `TRANSACTION_READ_COMMITTED`
- ■   `TRANSACTION_REPEATABLE_READ`
- ■   `TRANSACTION_SERIALIZABLE`

-79748   `Can't lock the connection`

IBM Informix JDBC Driver normally locks the connection object just before beginning the data exchange with the database server. The driver could not obtain the lock. Only one thread at a time should use the connection object.

-79749   `Number of input values does not match number of question marks`

The number of variables that you set using the **PreparedStatement.setXXX()** methods in this statement does not match the number of ? placeholders that you wrote into the statement. Locate the text of the statement and verify the number of placeholders and then check the calls to **PreparedStatement.setXXX()**.

-79750   `Method only for queries`

The **Statement.executeQuery(String)** and **PreparedStatement.execute-Query()** methods should only be used if the statement is a SELECT statement. For other statements, use the **Statement.execute(String)**, **Statement.executeBatch()**, **Statement.executeUpdate(String)**, **Statement.getUpdateCount()**, **Statement.getResultSet()**, or **PreparedStatement.executeUpdate()** method.

-79751   `Forward fetch only [in JDBC 1.2]`

The result set is not set to FETCH_FORWARD. Call
Resultset.setF**etchDirection(ResultSet.FETCH_FORWARD)** to reset the
direction.

-79755   `Object is null`

The object passed in is null. Check your program logic to make sure your
object reference is valid.

-79756   `Must start with 'jdbc'`

The first token of the database URL must be the keyword JDBC (case insen-
sitive), as in the following example:

```
jdbc:informix-sqli://mymachine:1234/mydatabase:user=me:
    password=secret
```

-79757   `Invalid subprotocol`

The current valid subprotocol is **informix-sqli**.

-79758   `Invalid IP address`

When you connect to an Informix database server via an IP address, the IP
address must be valid. A valid IP address is a set of four numbers between 0
and 255, separated by dots (.): for example, 127.0.0.1.

-79759   `Invalid port number`

The port number must be a valid four-digit number, as follows:

```
jdbc:informix-sqli://mymachine:1234/mydatabase:user=me:
    password=secret
```

In this example, `1234` is the port number.

-79760   `Invalid database name`

This statement contains the name of a database in some invalid format.

The maximum length for database names and cursor names depends on the
version of the database server. In 7.x, 8.x, and 9.1x versions of the Informix
database server, the maximum length is 18 characters.

For IBM Informix SE, database names should be no longer than 10 characters
(fewer in some host operating systems).

Both database and cursor names must begin with a letter and contain only letters, numbers, and underscore characters. In the 6.0 and later versions of the database server, database and cursor names can begin with an underscore.

In MS-DOS systems, filenames can be a maximum of eight characters plus a three-character extension.

-79761   `Invalid Property format`

The database URL accepts property values in key=value pairs. For example, `user=informix:password=informix` adds the key=value pairs to the list of properties that are passed to the connection object. Check the syntax of the key=value pair for syntax errors. Make sure there is only one = sign; that there are no spaces separating the key, value, or =; and that key=value pairs are separated by one colon(:), again with no spaces.

-79762   `Attempt to connect to a non 5.x server`

When connecting to a Version 5.x database server, the user must set the database URL property USE5SERVER to any non-null value. If a connection is then made to a Version 6.0 or later database server, this exception is thrown. Verify that the version of the database server is correct and modify the database URL as needed.

-79764   `Invalid Fetch Direction value`

An invalid fetch direction was passed as an argument to the **Statement.setFetchDirection()** or **ResultSet.setFetchDirection()** method. Valid values are FETCH_FORWARD, FETCH_REVERSE, and FETCH_UNKNOWN.

-79765   `ResultSet Type is TYPE_FETCH_FORWARD, direction can only be FETCH_FORWARD`

The result set type has been set to TYPE_FORWARD_ONLY, but the **setFetchDirection()** method has been called with a value other than FETCH_FORWARD. The direction specified must be consistent with the result type specified.

-79766   Incorrect Fetch Size value

The **Statement.setFetchSize()** method has been called with an invalid value. Verify that the value passed in is greater than 0. If the **setMaxRows()** method has been called, the fetch size must not exceed that value.

-79767   ResultSet Type is TYPE_FORWARD_ONLY

A method such as **ResultSet.beforeFirst()**, **ResultSet.afterLast()**, **ResultSet.first()**, **ResultSet.last()**, **ResultSet.absolute()**, **ResultSet.relative()**, **ResultSet.current()**, or **ResultSet.previous()** has been called, but the result set type is TYPE_FORWARD_ONLY. Call only the **ResultSet.next()** method if the result set type is TYPE_FORWARD_ONLY.

-79768   Incorrect row value

The **ResultSet.absolute(int)** method has been called with a value of 0. The parameter must be greater than *0*.

-79769   A customized type map is required for this data type

You must register a customized type map to use any opaque types.

-79770   Cannot find the SQLTypeName specified in the SQLData or Struct

The **SQLTypename** object you specified in the **SQLData** or **Struct** class does not exist in the database. Make sure that the type name is valid.

-79771   Input value is not valid

The input value is not accepted for this data type. Make sure this is a valid input for this data type.

-79772   No more data to read or write. Verify your SQLData class or getSQLTypeName()

This error occurs when a Java user-defined routine attempts to read or set a position beyond the end of the opaque type data available from a data input stream. Check the length and structure of the opaque type carefully against the data-input UDR code. The **SQLTypeName** object that was returned by the **getSQLTypeName()** method might also be incorrect.

-79774   `Unable to create local file`

Large object data read from the database server can be stored either in memory or in a local file. If the **LOBCACHE** value is `0` or the large object size is greater than the **LOBCACHE** value, the large object data from the database server is always stored in a file. In this case, if a security exception occurs, IBM Informix JDBC Driver makes no attempt to store the large object into memory and throws this exception.

-79775   `Only TYPE_SCROLL_INSENSITIVE and TYPE_FORWARD_ONLY are supported`

IBM Informix JDBC Driver only supports a result set type of TYPE_SCROLL_INSENSITIVE and TYPE_FORWARD_ONLY. Only these values should be used.

-79776   `Type requested (%s) does not match row type information (%s) type`

Row type information was acquired either through the system catalogs or through the supplied row definition. The row data provided does not match this row element type. The type information must be modified, or the data must be provided.

-79777   `readObject/writeObject() only supports UDTs, Distincts and complex types`

The **SQLData.writeObject()** method was called for an object that is not a user-defined, distinct, or complex type. Verify that you have provided customized type-mapping information.

-79778   `Type mapping class must be a java.util.Collection implementation`

You provided a type mapping to override the default for a SET, LIST, or MULTISET data type, but the class does not implement the **java.util.Collection** interface.

-79780   `Data within a collection must all be the same Java class and length.`

Verify that all the objects in the collection are of the same class.

-79781   `Index/Count out of range`

Array.getArray() or **Array.getResultSet()** was called with index and count values. Either the index is out of range or the count is too big. Verify that the number of elements in the array is sufficient for the index and count values.

-79782   `Method can be called only once`

Make sure methods such as **Statement.getUpdateCount()** and **Statement.getResultSet()** are called only once per result.

-79783   `Encoding or code set not supported`

The encoding or code set entered in the **DB_LOCALE** or **CLIENT_LOCALE** variable is not valid. Check "Support for Code-Set Conversion" on page 6-16 for valid code sets.

-79784   `Locale not supported`

The locale entered in the **DB_LOCALE** or **CLIENT_LOCALE** variable is not valid. Check "Support for Code-Set Conversion" on page 6-16 for valid locales.

-79785   `Unable to convert JDBC escape format date string to localized`
               `date string`

The JDBC escape format for date values must be specified in the format {d *'yyyy-mm-dd'*}. Verify that the JDBC escape date format specified is correct. Verify the **DBDATE** and **GL_DATE** settings for the correct date string format if either of these was set to a value in the connection database URL string or property list.

-79786   `Unable to build a Date object based on localized date string`
               `representation`

The localized date string representation specified in a CHAR, VARCHAR, or LVARCHAR column is not correct, and a date object cannot be built based on the year, month, and day values. Verify that the date string representation conforms to the **DBDATE** or **GL_DATE** date formats if either one of these is specified in a connection database URL string or property list. If neither **DBDATE** or **GL_DATE** is specified but a **CLIENT_LOCALE** or **DB_LOCALE** is explicitly set in a connection database URL string or property list, verify that the date string representation conforms to the JDK short default format (**DateFormat.SHORT**).

-79788   `User name must be specified`

The user name is required to establish a connection with IBM Informix JDBC Driver. Make sure you pass in `user=your_user_name` as part of the database URL or one of the properties.

-79789   `Server does not support GLS variables DB_LOCALE, CLIENT_LOCALE or GL_DATE`

These variables can only be used if the database server supports GLS. Check the documentation for your database server version and omit these variables if they are not supported.

-79790   `Invalid complex type definition string`

The value returned by the **getSQLTypeName()** method is either null or invalid. Check the string to verify that it is either a valid named-row name or a valid row type definition.

-79792   `Row must contain data`

The **Array.getAttributes()** or **Array.getAttributes(Map)** method has returned 0 elements. These methods must return a nonzero number.

-79793   `Data in array does not match getBaseType() value`

The **Array.getArray()** or **Array.getArray(Map)** method has returned an array where the element type does not match the JDBC base type.

-79794   `Row length provided (%s) doesn't match row type information (%s)`

Data in the row does not match the length in the row type information. You do not have to pad string lengths to match what is in the row definition, but lengths for other data types should match.

-79795   `Row extended id provided (%s) doesn't match row type information (%s)`

The extended ID of the object in the row does not match the extended ID as defined in row type information. Either update the row information (if you are providing the row definition) or check the type mapping information.

-79796   Cannot find UDT, distinct or named row (%s) in database

The **getSQLTypeName()** method has returned a name that can not be found in the database. Verify that the **Struct** or **SQLData** object returns the correct information.

-79797   DBDATE setting must be at least 4 characters and no longer than 6 characters

This error occurs because the **DBDATE** format string that is passed to the database server either has too few characters or too many. To fix the problem, verify the **DBDATE** format string with the user documentation and make sure that the correct year, month, day, and possibly era parts of the **DBDATE** format string are correctly identified.

-79798   A numerical year expansion is required after 'Y' character in DBDATE string

This error occurs because the **DBDATE** format string has a year designation (specified by the character Y), but there is no character following the year designation to denote the numerical year expansion (2 or 4). To fix the problem, modify the **DBDATE** format string to include the numerical year expansion value after the Y character.

-79799   An invalid character is found in the DBDATE string after the 'Y' character

This error occurs because the **DBDATE** format string has a year designation (specified by the character Y), but the character following the year designation is not a 2 or 4 (for two-digit years and four-digit years, respectively). To fix the problem, modify the **DBDATE** format string to include the required numerical year expansion value after the Y character. Only a 2 or 4 character should immediately follow the Y character designation.

-79800   No 'Y' character is specified before the numerical year expansion value

This error occurs because the **DBDATE** format string has a numerical year expansion (2 or 4 to denote two-digit years or four-digit years, respectively), but the year designation character (Y) was not found immediately before the numerical year expansion character specified. To fix the problem, modify the **DBDATE** format string to include the required Y character immediately before the numerical year expansion value requested.

-79801   `An invalid character is found in DBDATE format string`

This error occurs because the **DBDATE** format string has a character that is not allowed. To fix the problem, modify the **DBDATE** format string to only include the correct date part designations: year (`Y`), numerical year expansion value (`2` or `4`), month (`M`), and day (`D`). Optionally, you can include an era designation (`E`) and a default separator character (hyphen, dot, or slash), which is specified at the end of the **DBDATE** format string. Refer to the user documentation for further information on correct **DBDATE** format string character designations.

-79802   `Not enough tokens are specified in the string representation of a date value`

This error occurs because the date string specified does not have the minimum number of tokens or separators needed to form a valid date value (composed of year, month, and day numerical parts). For example, 12/15/98 is a valid date string representation with the slash character as the separator or token. But 12/1598 is not a valid date string representation, because there are not enough separators or tokens. To fix the problem, modify the date string representation to include a valid format for separating the day, month, and year parts of a date value.

-79803   `Date string index out of bounds during date format parsing to build Date object`

This error occurs because there is not a one-to-one correspondence between the date string format required by **DBDATE** or **GL_DATE** and the actual date string representation you defined. For example, if **GL_DATE** is set to `%b %D %y` and you specify a character string of `Oct`, there is a definite mismatch between the format required by **GL_DATE** and the actual date string. To fix the problem, modify the date string representation of the **DBDATE** or **GL_DATE** setting so that the date format specified matches one-to-one with the required date string representation.

-79804   `No more tokens are found in DBDATE string representation of a`
`date value`

This error occurs because the date string specified does not have any more tokens or separators needed to form a valid date value (composed of year, month, and day numerical parts) based on the **DBDATE** format string. For example, 12/15/98 is a valid date string representation when **DBDATE** is set to `MDY2/`. But 12/1598 is not a valid date string representation, because there are not enough separators or tokens. To fix the problem, modify the date string representation to include a valid format for separating the day, month, and year parts of a date value based on the **DBDATE** format string setting.

-79805   `No era designation found in DBDATE/GL_DATE string represen-`
`tation of date value`

This error occurs because the date string specified does not have a valid era designation, as required by the **DBDATE** or **GL_DATE** format string setting. For example, if **DBDATE** is set to `Y2MDE-`, but the date string representation specified by the user is `98-12-15`, this is an error because there is no era designation at the end of the date string value. To fix the problem, modify the date string representation to include a valid era designation based on the **DBDATE** or **GL_DATE** format string setting. In this example, a date string representation of `98-12-15 AD` would probably suffice, depending on the locale.

-79806   `Numerical day value can not be determined from date string based`
`on DBDATE`

This error occurs because the date string specified does not have a valid numerical day designation as required by the **DBDATE** format string setting. For example, if **DBDATE** is set to `Y2MD-`, but the date string representation you specify is `98-12-blah`, this is an error, because `blah` is not a valid numerical day representation. To fix the problem, modify the date string representation to include a valid numerical day designation (from `1` to `31`) based on the **DBDATE** format string setting.

-79807   `Numerical month value can not be determined from date string based on DBDATE`

This error occurs because the date string specified does not have a valid numerical month designation as required by the **DBDATE** format string setting. For example, if **DBDATE** is set to `Y2MD-`, but the date string representation you specify is `98-blah-15`, this is an error, because `blah` is not a valid numerical month representation. To fix the problem, modify the date string representation to include a valid numerical month designation (from 1 to 12) based on the **DBDATE** format string setting.

-79808   `Not enough tokens specified in %D directive representation of date string`

This error occurs because the date string specified does not have the correct number of tokens or separators needed to form a valid date value based on the **GL_DATE** %D directive (*mm/dd/yy* format). For example, 12/15/98 is a valid date string representation based on the **GL_DATE** %D directive, but 12/1598 is not a valid date string representation, because there are not enough separators or tokens. To fix the problem, modify the date string representation to include a valid format for the **GL_DATE** %D directive.

-79809   `Not enough tokens specified in %x directive representation of date string`

This error occurs because the date string specified does not have the correct number of tokens or separators needed to form a valid date value based on the **GL_DATE** %x directive (format required is based on day, month, and year parts, and the ordering of these parts is determined by the specified locale). For example, 12/15/98 is a valid date string representation based on the **GL_DATE** %x directive for the U.S. English locale, but 12/1598 is not a valid date string representation because there are not enough separators or tokens. To fix the problem, modify the date string representation to include a valid format for the **GL_DATE** %x directive based on the locale.

-79810   `This release of JDBC requires to be run with JDK 1.2+`

IBM Informix JDBC Driver Version 2.x requires a JDK version of 1.2 or greater.

-79811   `Connection without user/password not supported`

You called the **getConnection()** method for the **DataSource** object, and the user name or the password is null. Use the user name and password arguments of the **getConnection()** method or set these values in the **DataSource** object.

-79812   `User/Password does not match with datasource`

You called the **getConnection(user, passwd)** method for the **DataSource** object, and the values you supplied did not match the values already found in the data source.

-79814   `Blob/Clob object is either closed or invalid`

If you retrieve a smart large object using the **ResultSet.getBlob()** or **ResultSet.getClob()** method or create one using the **IfxBlob()** or **IfxCblob()** constructor, a smart large object is opened. You can then read from or write to the smart large object. After you execute the **IfxBlob.close()** method, do not use the smart large object handle for further read/write operations, or this exception is thrown.

-79815   `Not in Insert mode. Need to call moveToInsertRow() first`

You tried to use the **insertRow()** method, but the mode is not set to Insert. Call the **moveToInsertRow()** method before calling **insertRow()**.

-79816   `Cannot determine the table name`

The table name in the query is either incorrect or refers to a table that does not exist.

-79817   `No serial, rowid, or primary key specified in the statement`

The updatable scrollable feature works only for tables that have a SERIAL column, a primary key, or a row ID specified in the query. If the table does not have any of the above, an updatable scrollable cursor cannot be created.

-79818   `Statement concurrency type is not set to CONCUR_UPDATABLE`

You tried to call the **insertRow()**, **updateRow()**, or **deleteRow()** method for a statement that has not been created with the CONCUR_UPDATABLE concurrency type. Re-create the statement with this type set for the concurrency attribute.

-79819   `Still in Insert mode. Call moveToCurrentRow() first`

You cannot call the **updateRow()** or **deleteRow()** method while still in Insert mode. Call the **moveToCurrentRow()** method first.

-79820   `Function contains an output parameter`

You have passed in a statement that contains an OUT parameter, but you have not used the driver's **CallableStatement.registerOutParameter()** and **get*XXX*()** methods to process the OUT parameter.

-79821   `Name unneccessary for this data type`

If you have a data type that requires a name (an opaque type or complex type) you must call a method that has a parameter for the name, such as the following methods:

```
public void IfxSetNull(int i, int ifxType, String name)
public void registerOutParameter(int parameterIndex,
    int sqlType, java.lang.String name);
public void IfxRegisterOutParameter(int parameterIndex,
    int ifxType, java.lang.String name);
```

The data type you have specified *does not* require a name. Use another method that does not have a type parameter.

-79822   `OUT parameter has not been registered`

The function specified using the **CallableStatement** interface has an OUT parameter that has not been registered. Call one of the **registerOutParameter()** or **IfxRegisterOutParameter()** methods to register the OUT parameter type before calling the **executeQuery()** method.

-79823   `IN parameter has not been set`

The function specified using the **CallableStatement** interface has an IN parameter that has not been set. Call the **setNull()** or **IfxSetNull()** method if you want to set a null IN parameter. Otherwise, call one of the set methods inherited from the **PreparedStatement** interface.

-79824   `OUT parameter has not been set`

The function specified using the **CallableStatement** interface has an OUT parameter that has not been set. Call the **setNull()** or **IfxSetNull()** method if you want to set a null OUT parameter. Otherwise, call one of the set methods inherited from the **PreparedStatement** interface.

-79825   `Type name is required for this data type`

This data type is an opaque type, distinct type, or complex type, and it requires a name. Use set methods for IN parameters and register methods for OUT parameters that take a type name as a parameter.

-79826   `Ambiguous java.sql.Type, use IfxRegisterOutParameter()`

The SQL type specified either has no mapping to an Informix data type or has more than one mapping. Use one of the **IfxRegisterOutParameter()** methods to specify the Informix data type.

-79827   `Function doesn't have an output parameter`

This function does not have an OUT parameter, or this function has an OUT parameter whose value the server version does not return. None of the methods in the **CallableStatement** interface apply. Use the inherited methods from the **PreparedStatement** interface.

-79828   `Function parameter specified isn't an OUT parameter`

Informix functions can have only one OUT parameter, and it is always the last parameter.

-79829   `Invalid directive used for the GL_DATE environment variable`

One or more of the directives specified by the **GL_DATE** environment variable is not allowed. Refer to "GL_DATE Variable" on page 6-7 for a list of the valid directives for a **GL_DATE** format.

-79830   `Insufficient information given for building a Time or Timestamp Java object.`

To perform string-to-binary conversions correctly for building a **java.sql.Timestamp** or **java.sql.Time** object, all the DATETIME fields must be specified for the chosen date string representation. For **java.sql.Timestamp** objects, the year, month, day, hour, minute, and second parts must be specified in the string representation. For **java.sql.Time** objects, the hour, minute, and second parts must be specified in the string representation.

-79831   Exceeded maximum no. of connections configured for Connection
          Pool Manager

If you repeatedly connect to a database using a **DataSource** object without closing the connection, connections accumulate. When the total number of connections for the **DataSource** object exceeds the maximum limit (100), this error is thrown.

-79834   Distributed transactions (XA) are not supported by this database
          server.

This error occurs when the user calls the method **XAConnection.getConnection()** against an XPS server.

-79836   Proxy Error: No database connection

This error is thrown by the Informix HTTP Proxy if you try to communicate with the database on an invalid or bad database connection. Make sure your application has opened a connection to the database, check your Web server and database error logs.

-79837   Proxy Error: Input/output error while communicating with
          database

This error is thrown by the Informix HTTP Proxy if an error is detected while the proxy is communicating with the database. This error can occur if your database server is not accessible. Make sure your database is accessible, check your database and Web server error logs.

-79838   Cannot execute change permission command (chmod/attrib).

The driver is unable to change the permissions on the client JAR file. This could happen if your client platform does not support the **chmod** or **attrib** command, or if the user running the JDBC application does not have the authority to change access permissions on the client JAR file.

Make sure that the **chmod** or **attrib** command is available for your platform and that the user running the application has the authority to change access permissions on the client JAR file.

-79839   Same Jar SQL name already exists in the system catalog.

The JAR filename specified when your application called
UDTManager.createJar() has already been registered in the database server.
Use **UDTMetaData.setJarFileSQLName()** to specify a different SQL name for
the JAR file.

-79840   Unable to copy jar file from client to server.

This error occurs when the pathname set using **setJarTmpPath()** is not
writable by user **informix** or the user specified in the JDBC connection. Make
sure the pathname is readable and writable by any user.

-79842   No UDR information was set in UDRMetaData.

Your application called the **UDRManager.createUDRs()** method without
specifying any UDRs for the database server to register.

Specify UDRs for the database server to register by calling the
UDRMetaData.setUDR() method before calling the
UDRManager.createUDRs() method.

-79843   SQL name of the jar file was not set in UDR/UDT MetaData.

Your application called either the **UDTManager.createUDT()** or the
UDRManager.createUDRs() method without specifying an SQL name for the
JAR file containing the opaque types or UDRs for the database server to
register.

Specify an SQL name for a JAR file by calling the
UDTMetaData.setJarFileSQLName() or **UDRMetaData.setJarFileSQLName()**
method before calling the **UDTManager.createUDT()** or
UDRManager.createUDRs() method.

-79844   Can't create/remove UDT/UDR as no database is specified in the
         connection.

Your application created a connection without specifying a database. The
following example establishes a connection and opens a database named
test:

```
url = "jdbc:informix-sqli:myhost:1533/test:" +
"informixserver=myserver;user=rdtest;password=test";
conn = DriverManager.getConnection(url);
```

The following example establishes a connection with no database open:

```
url = "jdbc:informix-sqli:myhost:1533:" +
"informixserver=myserver;user=rdtest;password=test";
conn = DriverManager.getConnection(url);
```

To resolve this problem, use the following SQL statements after the connection is established and before calling the **createUDT()** or **createUDRs()** methods:

```
Statement stmt = conn.createStatement();
stmt.executeUpdate("create database test ...");
```

Alternatively, use the following code:

```
stmt.executeUpdate("database test");
```

-79845   JAR file on the client does not exist or can't be read.

This error occurs for one of the following reasons:

■   You failed to create a client JAR file.

■   You specified an incorrect pathname for the client JAR file.

■   The user running the JDBC application or the user specified in the connection does not have permission to open or read the client JAR file.

-79846   Invalid JAR file name.

The client JAR file your application specified as the second parameter to **UDTManager.createUDT()** or **UDRManager.createUDRs()** must end with the **.jar** extension.

-79847   The 'javac' or 'jar' command failed.

The driver encountered compilation errors in one of the following cases:

■   Compiling **.class** files into **.jar** files, using the **jar** command, in response to a **createJar()** command from the JDBC application

■   Compiling **.java** files into **.class** files and **.jar** files, using the **javac** and **jar** commands, in response to a **UDTManager.createUDTClass()** method call from the JDBC application.

Error Messages   **23**

-79848   `Same UDT SQL name already exists in the system catalog.`

Your application called **UDTMetaData.setSQLName()** and specified a name that is already in the database server.

-79849   `UDT SQL name was not set in UDTMetaData.`

Your application failed to call **UDTMetaData.setSQLName()** to specify an SQL name for the opaque type.

-79850   `UDT field count was not set in UDTMetaData.`

Your application called **UDTManager.createUDTClass()** without first specifying the number of fields in the internal data structure that defines the opaque type. Specify the number of fields by calling **UDTMetaData.setFieldCount()**.

-79851   `UDT length was not set in UDTMetaData.`

Your application called **UDTManager.createUDTClass()** without first specifying a length for the opaque type. Specify the total length for the opaque type by calling **UDTMetaData.setLength()**.

-79852   `UDT field name or field type was not set in UDTMetaData.`

Your application called **UDTManager.createUDTClass()** without first specifying a field name and data type for each field in the data structure that defines the opaque type. Specify the field name by calling **UDTMetaData.setFieldName()**; specify a data type by calling **UDTMetaData.setFieldType().**

-79853   `No class files to be put into the jar.`

Your application called the **createJar()** method and passed a zero-length string for the classnames parameter. The method signature is as follows:

```
createJar(UDTMetaData mdata, String[] classnames)
```

-79854   `UDT java class must implement java.sql.SQLData interface.`

Your application called **UDTManager.createUDT()** to create an opaque type whose class definition does not implement the **java.sql.SQLData** interface. UDTManager cannot create an opaque type from a class that does not implement this interface.

-79855   Specified UDT java class is not found.

Your application called the **UDTManager.createUDT()** method but the driver could not find a class with the name you specified for the third parameter.

-79856   Specified UDT does not exists in the database.

Your application called **UDTManager.removeUDT(String** *sqlname***)** to remove an opaque type named *sqlname* from the database, but the opaque type with that name does not exist in the database.

-79857   Invalid support function type.

This error occurs only if your application called the **UDTMetaData.setSupportUDR()** method and passed an integer other than 0 through 7 for the *type* parameter.

Use the constants defined for the support UDR types. For more information, see .

-79858   The command to remove file on the client failed.

If **UDTMetaData.keepJavaFile()** is not called or is set to FALSE, the driver removes the generated **.java** file when the **UDTManager.createUDTClass()** method executes. This error results if the driver was unable to remove the **.java** file.

-79859   Invalid UDT field number.

Your application called a **UDTMetaData.setXXX()** or **UDTMetaData.getXXX()** method and specified a field number that was less than 0 or greater than the value set through the **UDTMetaData.setFieldCount()** method.

-79860   Ambiguous java type(s) - can't use Object/SQLData as method
          argument(s).

One or more parameters of the method to be registered as a UDR is of type **java.lang.Object** or **java.sql.SQLData**. These Java data types can be mapped to more than one Informix data type, so the driver is unable to choose a type.

Avoid using **java.lang.Object** or **java.sql.SQLData** as method arguments.

-79861   `Specified UDT field type has no Java type match.`

Your application called **UDTMetaData.setFieldType()** and specified a data type that has no 100 percent match in Java. The following data types are in this category:

```
IfxTypes.IFX_TYPE_BYTE
IfxTypes.IFX_TYPE_TEXT
IfxTypes.IFX_TYPE_VARCHAR
IfxTypes.IFX_TYPE_NVCHAR
IfxTypes.IFX_TYPE_LVARCHAR
```

Use IFX_TYPE_CHAR or IFX_TYPE_NCHAR instead; these data types map to **java.lang.String**.

-79862   `Invalid UDT field type.`

Your application called **UDTMetaData.setFieldType()** and specified an unsupported data type for the opaque type. For supported data types, see "Mapping for Field Types" on page C-24.

-79863   `UDT field length was not set in UDTMetaData.`

Your application specified a field of character, date-time, or interval type by calling **UDTMetaData.setFieldType()**, but failed to specify a field length. Call **UDTMetaData.setFieldLength()** to set a field length.

-79864   `Statement length exceeds the maximum`

Your application issued an SQL PREPARE, DECLARE, or EXECUTE IMMEDIATE statement that is longer than the database server can handle. The limit differs with different implementations, but in most cases is up to 32,000 characters. Review the program logic to ensure that an error has not caused your application to present a string that is longer than intended. If the text has the intended length, revise the application to present fewer statements at a time.

This is the same as error -460 returned by the database server.

-79865   `Statement already closed`

This error occurs when an application attempts to access a statement method after the **stmt.close()** method.

-79868   `ResultSet not open, operation not permitted`

This error occurs when an application attempts to access a **ResultSet** method after the **ResultSet.close()** method.

-79877   `Invalid parameter value for setting maximum field size to a`
          `value less than zero`

          This error occurs when an application attempts to set the maximum field size
          to a value less than zero.

-79878   `ResultSet not open, operation next not permitted. Verify that`
          `autocommit is OFF`

          This error occurs when an application attempts to access the **ResultSet.next()**
          method without executing a result set query.

-79879   `An unexpected exception was thrown. See next exception for`
          `details`

          This error occurs when a non-SQL exception occurs; for example, an IO
          exception.

-79880   `Unable to set JDK Version for the Driver.`

          This error occurs when the driver cannot obtain the JDK version from the
          Java virtual machine.

-79881   `Already in local transaction, so cannot start XA transaction.`

          This error occurs when the application attempts to start an XA transaction
          while a local transaction is still in progress.

Index

N

Q

Qualifiers, binary, for INTERVAL
   data types 4-13
Querying the database 3-3

R

read() method 4-11
readArray() method 4-6
readAsciiStream() method 5-8
readBinaryStream() method 5-8
readByte() method 4-27
readBytes() method 5-5, 5-8
readCharacterStream()
   method 4-6, 4-27, 5-8
readObject() method 4-27, 5-8
Read-only connections 3-21
readProperties() method B-2
readRef() method 4-6, 4-27, 5-8
readSQL() method 4-28, 4-31, 4-39,
   5-7
readString() method 5-5, 5-8
Ref type C-2
refreshRow() method 3-19
registerDriver() method 2-5
Registering IBM Informix JDBC
   Driver 2-5
registerOutParameter()
   method 3-8, Error-19
 type mappings for 3-13
Relative distinguished name
   (RDN) 2-27
relative() method Error-10
Remote database access 2-32
Remote method invocation
   (RMI) 2-39
removeJar() method 5-28, 5-30
removeProperty() method B-2
removeUDR() method 5-30
removeUDT() method 5-28
Restrictions, driver 3-12
Restrictions,server 3-9
ResultSet class 6-9, 6-12
ResultSet interface 1-4, 3-3, 3-16,
   7-9, C-18 to C-20
ResultSetMetaData interface 1-4,
   3-3

Retrieving
   database names 3-40
   date values 6-9, 6-12
   Informix error message text 3-25
   syntax error offset 3-26
   user names 3-40
   version information 3-39
   XML data 3-32
RMI 2-39
ROLLBACK WORK statement 4-79
row3.java example program A-8
rowDeleted() method 3-19
rowInserted() method 3-19
rowUpdated() method 3-19
RSMetaData.java example
   program A-5

S

SAX (Simple API for XML) 3-28
Sbspace
   metadata area 4-70
   name of 4-67, 4-68
   user-data area 4-70
SBSPACENAME configuration
   parameter 4-63, 4-68
Schemas, IBM Informix JDBC
   Driver interpretation 3-40
Scroll cursors 3-6
ScrollCursor.java example
   program 3-7, A-5
scrubConnection() method 2-42,
   7-16
Search, anonymous, of sqlhosts
   information 2-24
SECURITY environment
   variable 2-21, 2-39
Security, JCE package for 2-40
Selecting smart large objects 4-52
SERIAL columns and scroll
   cursors 3-7
SERIAL data type 4-11
SERIAL8 data type 4-11
Serial.java example program A-5
Server restrictions,limitations 3-9
SERVERNAME option 2-40
Service provider interface
   (SPI) 2-23

Servlets 2-32
SessionMgr class 2-34
SessionMgr.class file 1-13, 2-34
Set interface 4-21
set() method 4-17, 4-20
setAlignment() method 5-20
setArray() method 4-21, C-10
setAsciiStream() method 4-8, 4-9,
   C-7, C-10
setAutoAlignment() method 5-6
setAutoCommit() method 3-27
setAutoFree() method 3-38, 7-9
setBigDecimal() method 4-5, 4-6,
   C-10
setBinaryStream() method 4-8, 4-9,
   C-7, C-11
setBlob() method C-11
setBoolean() method C-11
setByte() method C-11
setBytes() method C-11
setCatalog() method 3-19
setCharacterStream() method C-11
setClassName() method 5-21
setClob() method C-11
setCurrentPosition() method 5-5
setDatabaseName() method B-3
setDataSourceName() method B-3
setDate() method C-11
setDescription() method B-3
setDouble() method C-12
setExplicitCast() method 5-26
setFetchDirection()
   method Error-8, Error-9
setFetchSize() method 3-19,
   Error-10
setFieldCount() method 5-19
setFieldLength() method 5-19
setFieldName method 5-19
setFieldType() method 5-19
setFieldTypeName() method 5-19
setFloat() method C-12
setIfxCLIENT_LOCALE()
   method B-4
setIfxCPMInitPoolSize()
   method B-9
setIfxCPMMaxAgeLimit()
   method B-9
setIfxCPMMaxConnections()
   method B-9

Symbols