

Informix JDBC Driver

Programmer's Guide

UNIX and Windows Environments

Informix Dynamic Server, Version 7.x

Informix Dynamic Server, Workgroup and Developer Editions, Version 7.x

Informix Dynamic Server with Advanced Decision Support and Extended Parallel Options, Version 8.x

Informix Dynamic Server with Universal Data Option, Version 9.x

INFORMIX-OnLine Dynamic Server, Version 5.x

INFORMIX-SE, Versions 5.x and 7.2x

Version 2.0

April 1999

Part No. 000-5292

Published by INFORMIX® Press

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

© 1999 Informix Corporation. All rights reserved. The following are trademarks of Informix Corporation or its affiliates:

Answers OnLine™; CBT Store™; C-ISAM®; Client SDK™; ContentBase™; Cyber Planet™; DataBlade®; Data Director™; Decision Frontier™; Dynamic Scalable Architecture™; Dynamic Server™; Dynamic Server™, Developer Edition™; Dynamic Server™ with Advanced Decision Support Option™; Dynamic Server™ with Extended Parallel Option™; Dynamic Server™ with MetaCube® ROLAP Option; Dynamic Server™ with Universal Data Option™; Dynamic Server™ with Web Integration Option™; Dynamic Server™, Workgroup Edition™; FastStart™; 4GL for ToolBus™; If you can imagine it, you can manage itSM; Illustra®; INFORMIX®; Informix Data Warehouse Solutions... Turning Data Into Business Advantage™; INFORMIX®-Enterprise Gateway with DRDA®; Informix Enterprise Merchant™; INFORMIX®-4GL; Informix-JWorks™; InformixLink®; Informix Session Proxy™; InfoShelf™; Interforum™; I-SPY™; Mediazation™; MetaCube®; NewEra™; ON-Bar™; OnLine Dynamic Server™; OnLine for NetWare®; OnLine/Secure Dynamic Server™; OpenCase®; ORCA™; Regency Support®; Solution Design LabsSM; Solution Design ProgramSM; SuperView®; Universal Database Components™; Universal Web Connect™; ViewPoint®; Visionary™; Web Integration Suite™. The Informix logo is registered with the United States Patent and Trademark Office. The DataBlade logo is registered with the United States Patent and Trademark Office.

Documentation Team: June Smith, Juliet Shackell, Oakland Editing and Production

GOVERNMENT LICENSE RIGHTS

Software and documentation acquired by or for the US Government are provided with rights as follows:

(1) if for civilian agency use, with rights as restricted by vendor's standard license, as prescribed in FAR 12.212; (2) if for Dept. of Defense use, with rights as restricted by vendor's standard license, unless superseded by a negotiated vendor license, as prescribed in DFARS 227.7202. Any whole or partial reproduction of software or documentation marked with this legend must reproduce this legend.

Table of Contents

Introduction

In This Introduction	3
About This Manual	3
Organization of This Manual	3
Material Not Covered	4
Types of Users	4
Software Dependencies	4
Assumptions About Your Locale.	5
Documentation Conventions	5
Typographical Conventions	6
Icon Conventions	7
Additional Documentation	8
Printed Documentation	8
On-Line Documentation	9
Vendor-Specific Documentation	9
Compliance with Industry Standards	10
Informix Welcomes Your Comments	10

Chapter 1

Getting Started

In This Chapter	1-3
What Is JDBC?	1-3
What Is a JDBC Driver?	1-6
Overview of Informix JDBC Driver	1-6
Installing the Driver	1-8
Interactive Installation	1-8
Silent Installation	1-10
Uninstalling the Driver	1-13
Using the Driver in an Application	1-14
Using the Driver in an Applet	1-15

Chapter 2	Programming with Informix JDBC Driver	
	In This Chapter	2-3
	Establishing a Connection	2-3
	Loading Informix JDBC Driver	2-4
	Creating a Connection	2-4
	Dynamically Reading the Informix sqlhosts File	2-16
	Password Encryption	2-20
	Accessing Database Metadata	2-21
	Querying the Database	2-22
	Batch Updates	2-22
	Scroll Cursors	2-23
	Informix-Specific Information About Querying a Database	2-24
	Example of Sending a Query to an Informix Database	2-25
	Escape Syntax	2-26
	Unsupported Methods	2-27
	Handling Errors	2-28
	Using the SQLException Class	2-29
	Retrieving Informix Error Message Text	2-30
	Internationalization	2-31
	JDK 1.1 and 1.2 Internationalization Support	2-31
	Support for Informix GLS Variables	2-31
	Support for Date End-User Formats	2-33
	Precedence Rules Regarding DATE Value End-User Formats	2-39
	Support for Code-Set Conversion	2-40
	Handling Transactions	2-46
	Other Informix Extensions to the JDBC API	2-47
	The Auto Free Feature	2-47
	Obtaining Driver Version Information	2-48
	Using an HTTP Proxy Server	2-49
Chapter 3	Manipulating Informix Data Types	
	In This Chapter	3-5
	Manipulating Informix Opaque Types	3-5
	IfmxUdtSQLInput Interface	3-6
	IfmxUdtSQLOutput Interface	3-7
	Mapping Opaque Types	3-7
	Caching Type Information	3-9
	Inserting Data Examples	3-10
	Retrieving Data Example	3-13
	Using Smart Large Objects Examples	3-13

Unsupported Methods	3-16
Manipulating Informix Distinct Types	3-16
Caching Type Information	3-17
Inserting Data Examples.	3-17
Retrieving Data Example	3-19
Unsupported Methods	3-20
Manipulating Informix BYTE and TEXT Data Types	3-20
Caching Large Objects	3-20
Inserting or Updating Data Example	3-21
Selecting Data Example	3-23
Manipulating Informix BLOB and CLOB Data Types	3-25
IfxLobDescriptor	3-26
IfxLocator.	3-26
IfxSmartBlob.	3-27
IfxBlob and IfxCblob Classes.	3-31
Caching Large Objects	3-33
Creating a Smart Large Object Example	3-33
Inserting Data Example	3-34
Retrieving Data Example	3-35
Manipulating Informix SERIAL and SERIAL8 Data Types	3-37
Manipulating Informix INTERVAL Data Types	3-38
The Interval Class	3-39
The IntervalYM Class.	3-40
The IntervalDF Class	3-43
Interval Example	3-45
Manipulating Informix Collections and Arrays	3-45
Collection Examples	3-46
Array Example	3-49
Manipulating Informix Named and Unnamed Rows	3-50
Using the SQLData Interface	3-51
Using the Struct Interface	3-52
Interval and Collection Support	3-53
Caching Type Information	3-53
SQLData Examples	3-54
Struct Examples.	3-58
The ClassGenerator Utility	3-63
Unsupported Methods	3-65
Mapping Data Types	3-66
Mapping Between Informix and JDBC Data Types.	3-66
PreparedStatement.setXXX() Extensions	3-69

	Supported ResultSet.getXXX() Methods	3-78
Chapter 4	Troubleshooting	
	In This Chapter	4-3
	Debugging Your JDBC API Program	4-3
	Using the Debug Version of the Driver	4-3
	Turning on Tracing	4-4
	Performance Issues	4-5
	Using the FET_BUF_SIZE Environment Variable	4-5
	Memory Management of Large Objects	4-6
	Reducing Network Traffic.	4-7
Appendix A	Sample Code Files	
	Glossary	
	Error Messages	
	Index	

Introduction

In This Introduction	3
About This Manual.	3
Organization of This Manual	3
Material Not Covered	4
Types of Users	4
Software Dependencies	4
Assumptions About Your Locale.	5
Documentation Conventions	5
Typographical Conventions	6
Icon Conventions	7
Comment Icons	7
Platform Icons	7
Additional Documentation	8
Printed Documentation	8
On-Line Documentation.	9
Vendor-Specific Documentation	9
Compliance with Industry Standards	10
Informix Welcomes Your Comments.	10

In This Introduction

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

About This Manual

This guide describes how to install, load, and use Informix JDBC Driver to connect to an Informix database from within a Java application or applet. You can also use 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 Informix JDBC Driver.

Organization of This Manual

This manual includes the following chapters:

- [Chapter 1, “Getting Started,”](#) describes Informix JDBC Driver and the JDBC application programming interface (API) in general. It provides essential information for programmers to immediately start using the product, such as instructions on how to install and load the driver.
- [Chapter 2, “Programming with Informix JDBC Driver,”](#) explains in more detail the Informix-specific information needed to use Informix JDBC Driver to connect to an Informix database. This information includes how to create a connection to an Informix database, query tables, and handle errors.

- [Chapter 3, “Manipulating Informix Data Types,”](#) explains the Informix-specific data types supported in Informix JDBC Driver. This information includes how to map data types.
- [Chapter 4, “Troubleshooting,”](#) provides troubleshooting tips to solve programming errors and problems with the driver. It also describes browser security issues when you use Informix JDBC Driver in a Java applet.
- [Appendix A, “Sample Code Files,”](#) lists examples referred to in the guide.

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

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 Informix JDBC Driver but do not provide an extensive description of the JDBC API.

For more information about the JDBC API, visit the JavaSoft Web site at:

<http://www.javasoft.com/products/jdk/1.2/docs/guide/jdbc/index.html>

Types of Users

This guide is for Java programmers who use the JDBC API to connect to Informix databases using 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 Informix JDBC Driver to connect to an Informix database, you must use one of the following Informix database servers:

- Informix Dynamic Server, Version 7.x

- Informix Dynamic Server, Workgroup and Developer Editions, Version 7.x
- Informix Dynamic Server with Advanced Decision Support and Extended Parallel Options, Version 8.x
- Informix Dynamic Server with Universal Data Option, Version 9.x
- INFORMIX-OnLine Dynamic Server, Version 5.x
- 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 dates, times, 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 [Informix Guide to GLS Functionality](#).

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.
<i>italics</i> italics <i>italics</i>	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 <i>monospace</i>	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 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.


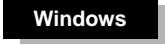
Comment Icons

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

Icon	Label	Description
	<i>Warning:</i>	Identifies paragraphs that contain vital instructions, cautions, or critical information
	<i>Important:</i>	Identifies paragraphs that contain significant information about the feature or operation that is being described
	<i>Tip:</i>	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
	Identifies information that is specific to the UNIX environment.
	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.

Additional Documentation

This section describes the following parts of the documentation set:

- Printed documentation
- On-line documentation
- Vendor-specific documentation

Printed Documentation

The following related Informix documents complement the information in this manual:

- If you have never used Structured Query Language (SQL), read the [Informix Guide to SQL: Tutorial](#). It provides a tutorial on SQL as it is implemented by Informix products. It also describes the fundamental ideas and terminology for planning and implementing a relational database.
- A companion volume to the *Tutorial*, the [Informix Guide to SQL: Reference](#), includes details of the 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 [Informix Guide to SQL: Syntax](#) provides information about SQL syntax as it is implemented by Informix products.
- [Informix Error Messages](#) is useful if you do not want to look up your error messages on-line.

On-Line Documentation

The Informix Answers OnLine CD allows you to print chapters or entire books and perform full-text searches for information in specific books or throughout the documentation set. You can install the documentation or access it directly from the CD. For information about how to install, read, and print on-line manuals, see the installation insert that accompanies Answers OnLine. You can also obtain the same information on the Web at <http://www.informix.com/answers>.

In addition to the Informix set of manuals, the following on-line files supplement the information in this manual.

On-Line File	Purpose
JDBCREL.TXT	The release notes describe any special actions required to configure and use Informix JDBC Driver on your computer. Additionally, this file contains information about any known problems and their workarounds.
JBCDOC.TXT	The documentation notes describe features not covered in the manuals or modified since publication.

UNIX

On-line files are located in `$JDBCLOCATION/doc/release`, where `$JDBCLOCATION` refers to the directory where you installed Informix JDBC Driver. ♦

Windows

On-line files are located in `%JDBCLOCATION%\doc\release`, where `%JDBCLOCATION%` refers to the directory where you installed Informix JDBC Driver. ♦

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

Vendor-Specific Documentation

For more information about the JDBC API, visit the JavaSoft Web site at:

<http://www.javasoft.com/products/jdk/1.2/docs/guide/jdbc/index.html>

Compliance with Industry Standards

The American National Standards Institute (ANSI) has established a set of industry standards for SQL. 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.

Informix Welcomes Your Comments

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

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

Write to us at the following address:

Informix Software, Inc.
Technical Publications
300 Lakeside Dr., Suite 2700
Oakland, CA 94612

If you prefer to send electronic mail, our address is:

`doc@informix.com`

We appreciate your suggestions.

Getting Started

In This Chapter	1-3
What Is JDBC?	1-3
What Is a JDBC Driver?	1-6
Overview of Informix JDBC Driver	1-6
Installing the Driver	1-8
Interactive Installation	1-8
Silent Installation	1-10
Uninstalling the Driver	1-13
Using the Driver in an Application	1-14
Using the Driver in an Applet	1-15

In This Chapter

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

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

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.
java.sql.Types	Class used to define constants that are used to identify standard SQL data types, such as VARCHAR, INTEGER, and DECIMAL.

(1 of 2)

Interface, Class, or Exception	Description
java.sql.Array	Class used to identify SET, MULTISSET, 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.

(2 of 2)

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 JavaSoft Web site at:

<http://www.javasoft.com/products/jdk/1.2/docs/guide/jdbc/index.html>

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 **Driver-Manager** 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.
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.
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.
Sends JDBC API calls to a middle-tier net 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.
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 Informix JDBC Driver

Informix JDBC Driver is a native-protocol, pure-Java driver (Type 4). This means that when you use 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.

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

Informix JDBC Driver is released as a Java class file called **setup.class**. For instructions on how to install the driver, refer to [“Installing the Driver” on page 1-8](#).

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

- **lib/ifxjdbc.jar**

JAR file that contains the 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**

JAR file that contains the **ClassGenerator** utility, the lightweight directory access protocol (LDAP) loader, and other utilities.

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.

- **demo/basic/***

- **demo/rmi/***

- **demo/stores7/***

- **demo/clob-blob/***

- **demo/udt-distinct/**

- **demo/complex-types/***

Directories that contain the sample Java 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.
- **doc/release/***
Directory that contains the on-line release and documentation notes, as well as the HTML and PDF versions of this programmer's guide.

Installing the Driver

Informix JDBC Driver is released as a Java class file called **setup.class**.

There are two ways to install the driver: using a **Setup** program or using the command line. The following sections describe the two ways for both UNIX and Windows.

Interactive Installation

This section describes how to interactively install Informix JDBC Driver with the **Setup** program.

UNIX

To interactively install Informix JDBC Driver on UNIX

1. If you are installing Informix JDBC Driver from a CD-ROM, load the disc into the CD-ROM drive.
On Hewlett-Packard platforms, you must use the **-o cdcase** option of the **mount** command to read the CD in case-sensitive mode.
2. Copy the **ifxjdbc.tar** file from the Web or the CD into a temporary directory (*not* the directory into which you are installing Informix JDBC Driver).

Warning: *If you copy the tar file to the same directory into which you attempt to install the driver, the installation fails.*

3. Execute the following command:

```
tar xvf ifxjdbc.tar
```

The **setup.class** and **install.txt** files appear in the temporary directory.



4. Be sure your **CLASSPATH** environment variable points to Version 1.2 or later of the Java Development Kit (JDK).
5. At the UNIX shell prompt, create a directory to hold the contents of the driver.

For example, to create the directory **/work/jdbcdriver_home**, execute the following command:

```
mkdir /work/jdbcdriver_home
```

6. Change directory to the temporary directory that contains the **setup.class** file.
7. Launch the **Setup** program with the **java** command at the UNIX shell prompt:

```
java setup
```

8. The **Setup** program guides you through the installation of Informix JDBC Driver.

The following warning message might appear:

```
Font specified in font.properties not found [-b&h-lucida sans
typewriter-bold-r-normal-sans-*-%d-*-*-*-*-*iso8859-1]
```

This condition does not affect the installation.

After the **Welcome** window, the program asks you for your serial number and key. It then asks you to accept a licensing agreement. The program then asks you for the name of the directory that will hold the contents of the driver. In this example, this directory is called **/work/jdbcdriver_home** and was created in Step 5 of these instructions.

The installation is complete when you get to the **Installation Complete** window. ♦

Windows

To interactively install Informix JDBC Driver on Windows

1. If you are installing Informix JDBC Driver from a CD-ROM, load the disc into the CD-ROM drive.
2. Copy the **ifxjdbc.tar** file from the Web or the CD into a temporary directory (*not* the directory into which you are installing Informix JDBC Driver).

Warning: If you copy the tar file to the same directory into which you attempt to install the driver, the installation fails.



3. Use WinZip or a similar utility to unpack the tar file. The **setup.class** and **install.txt** files appear in the temporary directory.
4. Be sure your **CLASSPATH** environment variable points to Version 1.2 or later of the Java Development Kit (JDK).
5. Using Windows Explorer, create a directory to hold the contents of the driver.

Assume, for this example, that the new directory is called **c:\work\jdbcdriver_home**.

6. Change directory to the temporary directory that contains the **setup.class** file.
7. Launch the **Setup** program with the **java** command at the Windows command prompt:

```
java setup
```

8. The **Setup** program guides you through the installation of Informix JDBC Driver.

The following warning message might appear:

```
Font specified in font.properties not found [-b&h-lucida sans  
typewriter-bold-r-normal-sans-*-%d-*-*m-*iso8859-1]
```

This condition does not affect the installation.

After the **Welcome** window, the program asks you for your serial number and key. It then asks you to accept a licensing agreement. The program then asks you for the name of the directory that will hold the contents of the driver. In this example, this directory is called **c:\work\dbcdriver_home** and was created in Step 5 of these instructions.

The installation is complete when you get to the **Installation Complete** window. ♦

Silent Installation

This section describes how to silently install Informix JDBC Driver from the UNIX shell prompt or Windows command line.

UNIX



To silently install Informix JDBC Driver on UNIX

1. If you are installing Informix JDBC Driver from a CD-ROM, load the disc into the CD-ROM drive.
On Hewlett-Packard platforms, you must use the **-o cdcase** option of the **mount** command to read the CD in case-sensitive mode.
2. Copy the **ifxjdbc.tar** file from the Web or the CD into a temporary directory (*not* the directory into which you are installing Informix JDBC Driver).

Warning: *If you copy the tar file to the same directory into which you attempt to install the driver, the installation fails.*

3. Execute the following command:

```
tar xvf ifxjdbc.tar
```

The **setup.class** and **install.txt** files appear in the temporary directory.

4. Be sure your **CLASSPATH** environment variable points to Version 1.2 or later of the Java Development Kit (JDK).
5. At the UNIX shell prompt, create a directory to hold the contents of the driver.

For example, to create the directory **/work/jdbcdriver_home**, execute the following command:

```
mkdir /work/jdbcdriver_home
```

6. Change directory to the temporary directory that contains the **setup.class** file.

7. Execute the following command at the UNIX shell prompt:

```
java setup -o <directory> serialNo=<serial_no> key=<key>
```

In this command, *directory* refers to the directory that will hold the contents of the driver (created in Step 5 of these instructions), and *serial_no* and *key* refer to the installation serial number and key.

The keywords **serialNo** and **key** are case sensitive. You can also use the keywords **SERIALNO**, **serialno**, and **KEY**.

For example, to install Informix JDBC Driver in the directory **/work/jdbcdriver_home** using a serial number of **INF#J123456** and a key of **ABCDEF**, execute the following command:

```
java setup -o /work/jdbcdriver_home serialNo=INF#J123456 key=ABCDEF
```

If the specified directory already contains Informix JDBC Driver files, the command asks you if you want to overwrite them.

The installation is complete after the command has finished executing. ♦

Windows

To silently install Informix JDBC Driver on Windows

1. If you are installing Informix JDBC Driver from a CD-ROM, load the disc into the CD-ROM drive.
2. Copy the **ifxjdbc.tar** file from the Web or the CD into a temporary directory (*not* the directory into which you are installing Informix JDBC Driver).

Warning: *If you copy the tar file to the same directory into which you attempt to install the driver, the installation fails.*

3. Use WinZip or a similar utility to unpack the tar file. The **setup.class** and **install.txt** files appear in the temporary directory.
4. Be sure your **CLASSPATH** environment variable points to Version 1.2 or later of the Java Development Kit (JDK).
5. Using Windows Explorer, create a directory to hold the contents of the driver.
Assume, for this example, that the new directory is called **c:\work\jdbcdriver_home**.
6. Change directory to the temporary directory that contains the **setup.class** file.



- Execute the following command at the Windows command prompt:

```
java setup -o <directory> serialNo=<serial_no> key=<key>
```

In this command, *directory* refers to the directory that will hold the contents of the driver (created in Step 5 of these instructions), and *serial_no* and *key* refer to the installation serial number and key.

The keywords **serialNo** and **key** are case sensitive. You can also use the keywords **SERIALNO**, **serialno**, and **KEY**.

For example, to install Informix JDBC Driver in the directory **c:\work\jdbcdriver_home** using a serial number of **INF#J123456** and a key of **ABCDEF**, execute the following command:

```
java setup -o c:\work\jdbcdriver_home serialNo=INF#J123456 key=ABCDEF
```

If the directory already contains Informix JDBC Driver files, the command asks you if you want to overwrite them.

The installation is complete once the command has finished executing. ♦

Uninstalling the Driver

Uninstalling Informix JDBC Driver completely removes the driver and all of its components from your computer. The following sections describe how to uninstall Informix JDBC Driver on UNIX and Windows.

To uninstall Informix JDBC Driver on UNIX

- Change to the directory in which you installed Informix JDBC Driver.

For example, if you installed the driver in the directory **/work/jdbcdriver_home**, execute the following command at the UNIX shell prompt:

```
cd /work/jdbcdriver_home
```

- Launch the **Uninstall** program with the **java** command:

```
java uninstall
```

- The **Uninstall** program guides you through the uninstallation of Informix JDBC Driver. ♦

Windows

To uninstall Informix JDBC Driver on Windows

1. Change to the directory in which you installed Informix JDBC Driver. For example, if you installed the driver in the directory `c:\work\jdbcdriver_home`, execute the following command at the command prompt:

```
cd c:\work\jdbcdriver_home
```

2. Launch the **Uninstall** program with the **java** command:

```
java uninstall
```

3. The **Uninstall** program guides you through the uninstallation of Informix JDBC Driver. ♦

***Important:** When you uninstall Informix JDBC Driver, you always get a message that says the `ifxjdbc.jar` and `ifxjdbc-g.jar` files have changed, even if you have never used the driver. This is because the files are automatically written to during the installation of the driver.*



Using the Driver in an Application

To use 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.

There are two ways of setting your **CLASSPATH** environment variable:

- Add the full pathname of the `ifxjdbc.jar` file to the **CLASSPATH** environment variable, as shown in the following example:

```
setenv CLASSPATH /work/jdbcdriver_home/lib/ifxjdbc.jar:$CLASSPATH
```

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

- Unpack the `ifxjdbc.jar` file and add its directory to the **CLASSPATH** environment variable, as shown in the following example:

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

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

UNIX

There are two ways of setting your **CLASSPATH** environment variable:

- Add the full pathname of the **ifxjdbc.jar** file to the **CLASSPATH** environment variable, as shown in the following example:

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

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

- Unpack the **ifxjdbc.jar** file and add its directory to the **CLASSPATH** environment variable, as shown in the following example:

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

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

For more information on the **jar** utility, refer to the JavaSoft documentation at <http://www.javasoft.com>.

Using the Driver in an Applet

You can use Informix JDBC Driver in an applet to connect to an Informix database from a browser such as Netscape Navigator or Microsoft Internet Explorer. The following steps show how to specify Informix JDBC Driver in the applet and how to ensure that the driver is correctly downloaded from the Web server.

To use Informix JDBC Driver in an applet

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

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

2. Specify the **ifxjdbc.jar** file 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 the file **ifxjdbc-g.jar** instead of **ifxjdbc.jar**.

Important: A few browsers do not support the ARCHIVE attribute of the APPLET tag. If you think this is true of the browsers that are going to download your applet, you must unpack and install the **ifxjdbc.jar** file in the root directory of your Web server.

If the browsers also do 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 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.
- **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 the Microsoft Internet Explorer browser, because this browser provides an option to configure the applet permissions.



Programming with Informix JDBC Driver

In This Chapter	2-3
Establishing a Connection	2-3
Loading Informix JDBC Driver	2-4
Creating a Connection	2-4
Format of Database URLs	2-6
Database Versus Database Server Connections	2-8
Specifying Environment Variables with the Properties Class	2-10
Supported Informix Environment Variables	2-12
Dynamically Reading the Informix sqlhosts File	2-16
Database URL Syntax	2-17
Administration Requirements	2-18
Utilities to Update the LDAP Server with sqlhosts Data	2-18
Password Encryption.	2-20
Configuring the Database Server	2-20
JCE Security Package	2-20
Accessing Database Metadata	2-21
Querying the Database	2-22
Batch Updates	2-22
Scroll Cursors	2-23
Informix-Specific Information About Querying a Database.	2-24
Example of Sending a Query to an Informix Database	2-25
Escape Syntax	2-26
Unsupported Methods	2-27
Handling Errors	2-28
Using the SQLException Class	2-29
Retrieving Informix Error Message Text	2-30

Internationalization	2-31
JDK 1.1 and 1.2 Internationalization Support	2-31
Support for Informix GLS Variables	2-31
Support for Date End-User Formats	2-33
GL_DATE Variable	2-33
DBDATE Variable	2-36
DBCENTURY Variable	2-39
Precedence Rules Regarding DATE Value End-User Formats	2-39
Support for Code-Set Conversion	2-40
Unicode to Database Code Set	2-41
Unicode to Client Code Set	2-44
Connecting to a Database with Non-ASCII Characters	2-44
Code Set Conversion for TEXT Data Types	2-45
Handling Transactions	2-46
Other Informix Extensions to the JDBC API	2-47
The Auto Free Feature	2-47
Obtaining Driver Version Information	2-48
Using an HTTP Proxy Server	2-49

In This Chapter

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

- “Establishing a Connection”
- “Accessing Database Metadata”
- “Querying the Database”
- “Handling Errors”
- “Internationalization”
- “Handling Transactions”
- “Other Informix Extensions to the JDBC API”
- “Using an HTTP Proxy Server”

Establishing a Connection

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 Informix JDBC Driver.
2. Create a connection to either a database server or a specific database.

You have the following connection options:

- Dynamically reading the Informix `sqlhosts` file
- Encrypting the password

Loading Informix JDBC Driver

To load Informix JDBC Driver, use the **Class.forName()** method, passing it the value `com.informix.jdbc.IfxDriver`, as shown in the following code example from the **CreateDB.java** program:

```
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**. The **IfxDriver** class then creates an instance of the driver and registers it with the **DriverManager** class.

Once you have loaded 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 Informix JDBC Driver to avoid platform incompatibilities.

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

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

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

Creating a Connection

To create a connection to an Informix database or database server, use the **DriverManager.getConnection()** method. This method creates a **Connection** object, which is later 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 Environment Variables with the Properties Class” on page 2-10](#) for an example of how to specify a property list.

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

```
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 code example from the **CreateDB.java** program shows how to connect to database **testDB** using Informix JDBC Driver. 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 connection type supported by Informix JDBC Driver is **tcp**. Shared memory and other connections types are not supported. The connection type is specified in the **NETTYPE** parameter of the **sqlhosts** or **ONCONFIG** file. For more information, see the [Administrator’s Guide](#) for your database server.



Important: Not all methods of the **Connection** interface are supported by Informix JDBC Driver. For a list of unsupported methods, see [“Unsupported Methods” on page 2-25](#).



Tip: You do not have to explicitly close a database connection. The database server closes the connection automatically.

Format of Database URLs

Informix JDBC Driver supports database URLs of the following format:

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

Important: *Blank spaces are not allowed in the database URL.*



The following table describes the variable parts of the database URL.

Database URL Variable	Required?	Description
<i>ip-address</i> <i>host-name</i>	Yes, unless SQLH_TYPE is defined	<p>The IP address or the host name of the computer running the Informix database server.</p> <p>An example of an IP address is 123.45.67.89.</p> <p>An example of a host name is myhost.com or myhost.informix.com.</p> <p>If an LDAP server or sqlhosts file provides the IP address or host name 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 2-16.</p>
<i>port-number</i>	Yes, unless SQLH_TYPE is defined	<p>The port number of the Informix database server.</p> <p>If an LDAP server or sqlhosts file provides the port number through the SQLH_TYPE property, you do not have to specify it in the database URL. For more information, see “Dynamically Reading the Informix sqlhosts File” on page 2-16.</p>
<i>dbname</i>	No	<p>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.</p>

(1 of 2)

Database URL Variable	Required?	Description
<i>server-name</i>	Yes	The name of the Informix database server to which you want to connect. This is the value of the INFORMIXSERVER environment variable. The INFORMIXSERVER environment variable is required in the database URL, unless it is included in the property list.
<i>user</i> <i>password</i>	Yes	The name of the user that wants to connect to the Informix database or database server, and the password of that user. You must specify both the user and the password.
<i>name=value</i>	No	A name-value pair that specifies a <i>value</i> for the Informix environment variable contained in the <i>name</i> variable, recognized by either Informix JDBC Driver or Informix database servers. The <i>name</i> variable is case insensitive. Informix JDBC Driver reads Informix environment variables from either the database URL or from a connection property list, described in “Specifying Environment Variables with the Properties Class” on page 2-10 . The user’s environment is not consulted. Refer to “Supported Informix Environment Variables” on page 2-12 for a list of Informix environment variables supported by Informix JDBC Driver.

(2 of 2)

Database Versus Database Server Connections

Using the **DriverManager.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 later 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 *INFORMIX-OnLine Dynamic Server* or *INFORMIX-SE*), you must specify the **USEV5SERVER** environment variable in the database URL or property list. Its value must be 1, such as **USEV5SERVER=1**.

The example given in “[Creating a Connection](#)” on page 2-4 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** later in the Java program 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
```

Here is the example code:

```
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!");
}

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 Environment Variables with the Properties Class

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. Refer to [“Supported Informix Environment Variables” on page 2-12](#) for a list of supported Informix 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 2-6](#).

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 [“Creating a Connection” on page 2-4](#).

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
```

Here is the example program:

```
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!");
}
```

Supported Informix Environment Variables

The following table lists the Informix environment variables supported by Informix JDBC Driver.

Supported Informix Environment Variables	Description
CLIENT_LOCALE	Specifies the locale of the client that is accessing the database. Together with the DB_LOCALE variable, the database server uses this variable to establish the server processing locale. This variable is available on and optional for servers that support GLS.
DBANSIWARN	Checks for Informix extensions to ANSI standard syntax.
DBCENTURY	Enables you to specify the appropriate expansion for one- or two-digit year DATE and DATETIME values.
DBDATE	Specifies the end-user formats of values in DATE columns.
DB_LOCALE	Specifies the locale of the database. Together with the CLIENT_LOCALE variable, the database server uses this variable to establish the server processing locale. This variable is available on and optional for servers that support GLS.
DBSPACETEMP	Specifies the dbspaces in which temporary tables are built.
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 instead of asking the database server whenever a Struct or SQLData object inserts data into a column and getSQLTypeName() returns the type name information.

(1 of 5)

Supported Informix Environment Variables	Description
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.
GL_DATE	Specifies the end-user formats of values in DATE columns. This variable is supported in Informix database server versions 7.2x and beyond.
IFX_AUTOFREE	<p>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.</p> <p>The database server automatically frees the cursor resources after the cursor is closed, either explicitly by the ResultSet.close() method or implicitly by the OPTOFC environment variable.</p> <p>After the cursor resources have been freed, the cursor can no longer be referenced.</p> <p>For more information, see “The Auto Free Feature” on page 2-46.</p>
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 the INFORMIXCONTIME environment variable (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 blob space of the client application.
INFORMIXSERVER	Specifies the default database server to which an explicit or implicit connection is made by a client application.
INFORMIXSTACKSIZE	Specifies the stack size, in kilobytes, that the database server uses for a particular client session.

(2 of 5)

Supported Informix Environment Variables	Description
JDBCTEMP	Specifies where temporary files for handling smart large objects are created. You must supply an absolute pathname.
LOBCACHE	<p>Determines the buffer size for large object data that is fetched from the database server:</p> <ul style="list-style-type: none">■ If LOBCACHE > 0, the maximum LOBCACHE 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.■ If LOBCACHE = 0, the data is always stored in a file. In this case, if a security violation occurs, Informix JDBC Driver makes no attempt to store the data in memory.■ If LOBCACHE < 0, the data is always stored in memory. If the required amount of memory is not available, an error occurs. <p>If the LOBCACHE value is not specified, the default is 4096 bytes.</p>
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.
OPTCOMPIND	Specifies the join method that the query optimizer uses.

(3 of 5)

Supported Informix Environment Variables	Description
OPTOFC	<p>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.</p> <p>Informix JDBC Driver might not have additional rows in the client's tuple buffer before the next ResultSet.next() method is called. Therefore, unless 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.</p>
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.
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.

(4 of 5)

Supported Informix Environment Variables	Description
SECURITY=PASSWORD	Protects the user-provided password using 56-bit encryption when it is passed from the client to the database server.
SQLH_TYPE	When set to FILE, specifies that database URL information (such as the <i>host-name</i> , <i>port-number</i> , <i>user</i> , and <i>password</i>) 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 2-16.
USEV5SERVER	When set to 1, specifies that the Java program is connecting to an INFORMIX-OnLine or INFORMIX-SE 5.x database server. This environment variable is mandatory if you are connecting to an INFORMIX-OnLine or INFORMIX-SE 5.x database server.

(5 of 5)

For a detailed description of a particular environment variable, refer to [Informix Guide to SQL: Reference](#). You can find the on-line version of this guide at <http://www.informix.com/answers>.

Dynamically Reading the Informix sqlhosts File

Informix JDBC Driver now 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 (lightweight directory access protocol) server. The system administrator must load the **sqlhosts** data into the LDAP server using an Informix-supplied 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**.

An unsigned applet cannot access the **sqlhosts** file or an LDAP server. For more information, see [“Using the Driver in an Applet” on page 1-15](#).

Database URL Syntax

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

- **SQLH_TYPE=LDAP**
- **LDAP_URL=ldap://*host-name:port-number***
- **LDAP_IFXBASE=*Informix-base-DN***
- **LDAP_USER=*user***
- **LDAP_PASSWD=*password***

If **LDAP_USER** and **LDAP_PASSWD** are not specified, 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.

Here is an example database URL:

```
jdbc:informix-sqli[:/dbname]: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. 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**
- **SQLH_FILE=file://D:/local/myown/sqlhosts.ius**
- **SQLH_FILE=/u/local/sqlhosts.ius**

Here is an example database URL:

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

If the database URL references the LDAP server or **sqlhosts** file but also directly specifies the IP address, host name, and port number, the IP address, host name, and port number specified in the database URL take precedence.

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.
Refer to <http://www.netscape.com> for more information about LDAP directory servers.
- If you want to use the **SqlhUpload** and **SqlhDelete** utilities provided by Informix, which can load or delete the **sqlhosts** entries from a flat ASCII file, the **servicename** field in the **sqlhosts** file must specify the 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 Informix JDBC Driver). Make sure that the **CLASSPATH** variable also points to the JNDI JAR files and LDAP SPI JAR files.

Class 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 of this command have the following meanings:

- **sqlhfile.txt** is the **sqlhosts** file to be uploaded.
- *host-name:port-number* is the host name and port number of the LDAP server.
- *sqlhostsRdn* is the RDN (relative distinguished name) of the **sqlhosts** node under the Informix base in LDAP. 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** for the port number. For more information about the **servicename** field, see the [Administrator's Guide](#) for your database server.

Class 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 [page 2-18](#).

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

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 **SECURITY** option supported in Informix JDBC Driver is **PASSWORD**. By default, no **SECURITY** option is set. If the **PASSWORD** option is specified, the user-provided password is encrypted using 56-bit encryption when it is passed from the client to the database server.

Both the **SECURITY** and **PASSWORD** keywords are case insensitive.

Here is an example of the database URL syntax for the **PASSWORD** option:

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

Configuring the Database Server

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

If the **SECURITY=PASSWORD** option is enabled in the Informix JDBC client, the Informix database server must also be configured to support the corresponding **SECURITY** options requested by the client. Otherwise, an error is returned during connection.

To use the **SPWDCSM** CSM 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** option 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.

Informix has certified the SunJCE 1.2 security package, which you can download free from the following Web site:

```
http://java.sun.com/products/jdk/1.2/jce
```

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

To install the SunJCE package, download the SunJCE distribution, extract the JAR file containing the SunJCE 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
```

Accessing Database Metadata

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

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

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.

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, Informix JDBC Driver interprets the JDBC API term *catalogs* to mean the names of Informix databases. The method **DatabaseMetaData.getCatalogs()** 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, “Sample Code Files,”](#) for more information about this example.

Querying the Database

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.

Batch Updates

The batch update feature is similar to Informix multiple SQL PREPARE statements. For example, Informix supports the following statement:

```
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 Informix JDBC Driver follows the Sun Microsystems JDBC 2.0 specification, with these exceptions:

- SQL statements

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 [Informix Guide to SQL: Syntax](#).

- **Statement.executeBatch()** return value

The return value differs from the Sun Microsystems JDBC 2.0 specification in the following ways:

- Only the update count of the first statement executed in the batch is returned.
- 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.

Scroll Cursors

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

- Scroll sensitivity

The Informix database server implementation of scroll cursors places the rows fetched in a temporary table. If another process changes a row (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. 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. The 2.0 release of Informix JDBC Driver supports the scrolling of the result set only according to how the database server supports scrolling.

- Concurrency types

The JDBC 2.0 specification states that the result set can have two types of concurrency: read-only and updatable. A result set that uses read-only concurrency does not allow updates of its contents. A result set that is updatable allows updates and may use database write locks to mediate access to the same data item by different transactions.

The Informix database server does not support updatable scroll cursors. If a cursor is declared as SCROLL, you cannot issue an UPDATE WHERE CURRENT OF statement.

- Updates

The JDBC specifications provide a series of methods that an application can use to update a fetched result set if its concurrency type is CONCUR_UPDATABLE. According to this specification, rows can be updated, inserted, and deleted from the result set. However, Informix JDBC Driver supports only a read-only result set.

The **ScrollCursor.java** example file shows how to retrieve a result set with a scroll cursor.

Informix-Specific Information About Querying a Database

This section describes the Informix-specific information you need to know to use Informix JDBC Driver to query an Informix database and process the results.

The Informix JDBC Driver implementation of the **Statement.execute()** method returns a single **ResultSet** object. This implementation differs from the JDBC API specification, which states that the method can return multiple **ResultSet** objects.

Be sure to always explicitly 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.



Important: For best results, always call **ResultSet.close()** and **Statement.close()** methods to indicate to Informix JDBC Driver that you are done with the statement or result set. Otherwise, your program might not release all its resources on the database server.

Important: 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 **getXXX()** methods retrieve individual column values, refer to [“Supported ResultSet.getXXX\(\) Methods” on page 3-78.](#)

Escape Syntax

Valid escape syntax for SQL statements is as follows:

Type of Statement	Escape Syntax
Stored procedure	{call <i>procedure</i> }
Stored procedure	{var = call <i>procedure</i> }
Date	{d 'yyyy-mm-dd' }
Time	{t 'hh:mm:ss' }
Timestamp (Datetime)	{ts 'yyyy-mm-dd hh:mm:ss[.ffff]' }
Function call	{fn <i>func</i> (<i>args</i>)}
Escape character	{escape ' <i>escape-char</i> ' }
Outer join	{oj <i>outer-join-statement</i> }

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

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

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

Unsupported Methods

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

- **CallableStatement.registerOutParameter()**
- **Connection.isReadOnly()**
- **Connection.setCatalog()**
- **Connection.setReadOnly()**
- **PreparedStatement.setRef(int, Ref)**
- **PreparedStatement.setUnicodeStream()**
- **ResultSet.cancelRowUpdate()**
- **ResultSet.deleteRow()**
- **ResultSet.getFetchSize()**
- **ResultSet.getRef(int)**
- **ResultSet.getRef(String)**
- **ResultSet.getUnicodeStream()**
- **ResultSet.insertRow()**
- **ResultSet.moveToCurrentRow()**
- **ResultSet.moveToInsertRow()**
- **ResultSet.refreshRow()**
- **ResultSet.rowDeleted()**
- **ResultSet.rowInserted()**
- **ResultSet.rowUpdated()**
- **ResultSet.setFetchSize()**
- **ResultSet.updateRow()**
- **ResultSet.updateXXX()**
- **Statement.cancel()**
- **Statement.setMaxFieldSize()**
- **Statement.setQueryTimeout()**

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

- **CallableStatement.execute()**
Method returns a single result set.
- **PreparedStatement.execute()**
Method returns a single result set.
- **ResultSetMetaData.getCatalogName()**
Method always returns the value "".
- **ResultSetMetaData.getTableName()**
Method always returns the value "".
- **ResultSetMetaData.getSchemaName()**
Method always returns the value "".
- **ResultSetMetaData.isDefinitelyWritable()**
Method always returns `TRUE`.
- **ResultSetMetaData.isReadOnly()**
Method always returns `FALSE`.
- **ResultSetMetaData.isWritable()**
Method always returns `TRUE`.
- **Statement.execute()**
Method returns a single result set.

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 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 Informix JDBC Driver error code.
- **getSQLState()**
Returns a string that describes the **SQLSTATE** value. The string follows the X/Open **SQLSTATE** conventions.

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

For a list of Informix database server errors, refer to *Informix Error Messages*. You can find the on-line version of this guide at <http://www.informix.com/answers>.

The following example from the **SimpleSelect.java** program shows how to use the **SQLException** class to catch 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 the 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.

Internationalization

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. Informix JDBC Driver extends the Java JDK 1.2 internationalization features by providing access to Informix databases that are based on different locales and code sets.

JDK 1.1 and 1.2 Internationalization Support

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, consult this Web site:

<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 Informix GLS Variables

For general information about setting up Informix global language support (GLS), refer to the [Informix Guide to GLS Functionality](#).

Internationalization adds several new environment variables to Informix JDBC Driver:

- **DB_LOCALE**
- **CLIENT_LOCALE**
- **GL_DATE**
- **DBDATE**
- **DBCENTURY**

The locale environment variables are available on and optional for Informix servers that support GLS:

- The **DB_LOCALE** variable specifies the locale of the database. Informix JDBC Driver uses this variable to perform code set conversion between Unicode and the database locale. The database server uses **DB_LOCALE** with **CLIENT_LOCALE** to establish the server processing locale.
- The **CLIENT_LOCALE** variable specifies the locale of the client that is accessing the database. Informix JDBC Driver uses this variable only to provide defaults for user-defined formats and to display error messages.

If set, the **CLIENT_LOCALE** value establishes the database server processing locale to provide defaults for user-defined formats like the **GL_DATE** format, and user-defined types can use it for code-set conversion.

The **DB_LOCALE** and **CLIENT_LOCALE** values must be the same, or their code sets must be convertible. For more information, see the [Informix Guide to GLS Functionality](#).

The **GL_DATE**, **DBDATE**, and **DBCENTURY** variables are described in the following section.



Important: The **DB_LOCALE**, **CLIENT_LOCALE**, and **GL_DATE** variables are supported only if the database server supports the 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, the behavior is the same as for Informix JDBC Driver Version 1.22.

Support for Date End-User Formats

The date end-user format is the format in which a date 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.

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 and beyond. 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.
%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); 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)



Important: *GL_DATE* optional date format qualifiers for field specifications (for example `%4m` to display a month as a decimal number with a maximum field width of 4) are not supported.

Informix JDBC Driver does not support *ALS 6.0, 5.0, or 4.0* era-based formats in the *DBDATE* and *GL_DATE* environment variables.

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**, 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 compatibility for client applications that are based on Informix database server versions prior to 7.2x. Informix recommends that you 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, 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, 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**, in which SQL statements are sent to the database server unchanged. Enter the **DATE** value exactly as expected by the **DBDATE** 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 **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**, 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();
```



Important: Informix JDBC Driver does not support ALS 6.0, 5.0, or 4.0 era-based formats in the **DBDATE** and **GL_DATE** environment variables.

DBCENTURY Variable

The **DBCENTURY** environment variable enables you to choose the appropriate four-digit year expansion for one- or two-digit year DATE and DATETIME values. See the [Informix Guide to SQL: Reference](#) for detailed information about this environment variable.

Informix JDBC Driver accepts the **DBCENTURY** value but does not use it to expand a two-digit year to a four-digit year. When you specify a two-digit year, Informix JDBC Driver supplies the first two digits of the current year. For example, in 1999, the driver supplies 19; in 2000, the driver supplies 20.

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 and DATETIME values to Informix JDBC Driver.

Informix JDBC Driver also does not use the **DBCENTURY** value for string-to-date value conversion. Informix JDBC Driver performs string-to-DATE, string-to-DATETIME, and string-to-TIMESTAMP conversions under these circumstances:

- If you pass a Java **String** object to the **setObject()** method to insert data into a DATE or DATETIME column.
- If you use the **getDate()** or **getTimeStamp()** method to retrieve data from a string-type database column, such as CHAR.

Precedence Rules Regarding DATE Value 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 (the source code set) to another (the target code set). 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 [Informix Guide to GLS Functionality](#).

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

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 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
utf8	57372	UTF8
big5	57352	Big5
CP1250	1250	Cp1250
CP1251	1251	Cp1251
CP1252	1252	Cp1252
CP1253	1253	Cp1253
CP1254	1254	Cp1254
CP1255	1255	Cp1255
CP1256	1256	Cp1256
CP1257	1257	Cp1257

(1 of 2)

Informix Code Set Name	Informix Code Set Number	JDK Code Set
cp949	57356	Cp949
KS5601	57356	Cp949
ksc	57356	Cp949
ujjis	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 where 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

(1 of 2)

Supported Locales

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

(2 of 2)

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 **InputStreamReader()** or **OutputStreamWriter()** methods to manipulate text files. By default, these methods use the local encoding, but you can specify an encoding in the constructor of the class, as follows:

```
InputStreamReader = new InputStreamReader (in, "SJIS");
```

You and the JVM are responsible for getting external input into the correct Java Unicode string. Thereafter, the database locale encoding is used to send the data to and from the database server.

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, so 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

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 data by using the `getBytes()`, `getString()`, `InputStreamReader()`, and `OutputStreamWriter()` methods. These 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. 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.

```

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();
}
```

Handling Transactions

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

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

If the Java program sets autocommit mode on while inside a transaction, Informix JDBC Driver rolls back the current transaction before it actually turns autocommit mode on.

In a database that has been created with logging, if a COMMIT statement is sent to the database server (either with the **Connection.commit()** method or directly with an SQL statement) 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.

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.

Other Informix Extensions to the JDBC API

This section describes the Informix-specific extensions to the JDBC API not already covered 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 2-27](#).

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 above
- Version 9.0 and above

Obtaining Driver Version Information

There are two ways to obtain version information about Informix JDBC Driver: from your Java program, or from the UNIX or Windows command line. The command line method also allows you to obtain the serial number you provided when you installed the driver on your computer.

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 Informix JDBC Driver.

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

The `IfxDriver.getJDBCVersion()` method does not return the serial number you provided during installation of the driver.

Important: For Version X.Y of 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 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. An example of a serial number is INF#J000000.

Using an HTTP Proxy Server

You might need to use an HTTP proxy server because of these features:

- **Applets.** Because of security restrictions in Web browsers, if an applet is using Informix JDBC Driver, it can only connect to a database running on the same host as the Web server.



- **Firewalls.** Informix JDBC Driver cannot connect to a database from behind a firewall. The firewall prevents the browser from connecting to the database.

The solution to both these problems is to install Informix JDBC Driver on the same computer as the Java applet and install the HTTP proxy as a middle tier between the Java applet and Informix database machines.

The HTTP proxy feature is not part of the JDBC 2.0 specification. The HTTP proxy is a lightweight servlet that extracts SQL requests from the JDBC client and transmits them to a database server. The client (the end user) is unaware of this extra layer.

To specify a proxy, amend the database URL statement:

```
CURRENT_JDBC_URL;proxy=web-server-host-name:port-number
```

The Web server must support servlets. (For example, the Sun and Apache Java Web servers do.)

To add a proxy servlet to a Sun Java Web server

1. Copy two class files, **IfxJDBCProxy.class** and **SessionMgr.class**, to the servlet directory. These two class files reside in directory **proxy**, which is under the installation directory for Informix JDBC Driver after the product bundle is installed.
2. Go to the URL `http://server-host-name:port-number` using a Web browser. The Java Web server administrator page appears.
3. Enter `admin` as the user, followed by the administrator password, to log on to the Web server administrator management menu.
4. Choose **Web Service→Servlets→Add**.
5. Type a JDBC servlet name and class. Name the servlet for the JDBC proxy **IfxJDBCProxy**.

This procedure makes the Web server aware of the proxy servlet.

The steps for adding a proxy server to other Java Web servers are similar. Consult the documentation specific to your Web server for instructions on how to configure the servlet.

Depending on your Web server, the proxy servlet may 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://jserv.javasoft.com/index.html>
- <http://www.javasoft.com>
- <http://www.sun.com/java>
- <http://java.apache.org>

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

- **Remote method invocation (RMI).** 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 Informix JDBC Driver; see [Appendix A, “Sample Code Files,”](#) for details.
- **Other communication protocols, such as CORBA.** Informix JDBC Driver resides on an application server that is a middle tier between the Java applet or application and Informix database machines.

Manipulating Informix Data Types

In This Chapter	3-5
Manipulating Informix Opaque Types	3-5
IfmxUdtSQLInput Interface	3-6
IfmxUdtSQLOutput Interface	3-7
Mapping Opaque Types	3-7
Caching Type Information	3-9
Inserting Data Examples	3-10
Retrieving Data Example	3-13
Using Smart Large Objects Examples	3-13
Unsupported Methods	3-16
Manipulating Informix Distinct Types	3-16
Caching Type Information	3-17
Inserting Data Examples	3-17
Retrieving Data Example	3-19
Unsupported Methods	3-20
Manipulating Informix BYTE and TEXT Data Types	3-20
Caching Large Objects	3-20
Inserting or Updating Data Example	3-21
Selecting Data Example	3-23
Manipulating Informix BLOB and CLOB Data Types	3-25
IfxLobDescriptor	3-26
IfxLocator.	3-26
IfxSmartBlob.	3-27
IfxSmartBlob Methods	3-27
IfxSmartBlob Flag Values	3-30
IfxSmartBlob Whence Values.	3-31

IfxBlob and IfxCblob Classes	3-31
IfxBlob Class	3-32
IfxCblob Class	3-32
Caching Large Objects	3-33
Creating a Smart Large Object Example	3-33
Inserting Data Example	3-34
Retrieving Data Example	3-35
Manipulating Informix SERIAL and SERIAL8 Data Types	3-37
Manipulating Informix INTERVAL Data Types	3-38
The Interval Class	3-39
Variables for Binary Qualifiers	3-39
Interval Methods	3-40
The IntervalYM Class.	3-40
IntervalYM Constructors	3-41
IntervalYM Methods.	3-42
The IntervalDF Class	3-43
IntervalDF Constructors	3-43
IntervalDF Methods	3-44
Interval Example	3-45
Manipulating Informix Collections and Arrays	3-45
Collection Examples	3-46
Array Example	3-49
Manipulating Informix Named and Unnamed Rows	3-50
Using the SQLData Interface	3-51
Using the Struct Interface	3-52
Interval and Collection Support	3-53
Caching Type Information	3-53
SQLData Examples	3-54
Struct Examples.	3-58
The ClassGenerator Utility	3-63
Simple Named Row	3-63
Nested Named Row	3-64
Unsupported Methods	3-65
Mapping Data Types	3-66
Mapping Between Informix and JDBC Data Types.	3-66
PreparedStatement.setXXX() Extensions	3-69

The Mapping Extensions	3-70
The IfxTypes Class	3-73
Extension Summary	3-75
Supported ResultSet.getXXX() Methods	3-78

In This Chapter

This chapter explains the Informix-specific data types supported in Informix JDBC Driver. The chapter includes the following sections:

- “Manipulating Informix Opaque Types”
- “Manipulating Informix Distinct Types”
- “Manipulating Informix BYTE and TEXT Data Types”
- “Manipulating Informix BLOB and CLOB Data Types”
- “Manipulating Informix SERIAL and SERIAL8 Data Types”
- “Manipulating Informix INTERVAL Data Types”
- “Manipulating Informix Collections and Arrays”
- “Manipulating Informix Named and Unnamed Rows”
- “Mapping Data Types”

Manipulating Informix Opaque Types

Informix has extended the JDBC 2.0 definition of the `java.sql.SQLInput` and `java.sql.SQLOutput` methods to fully support Informix fixed binary and variable binary opaque types. This extension includes the following interfaces:

- `IfmxUdtSQLInput`
- `IfmxUdtSQLOutput`

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**. Here are the signatures and descriptions of each added method:

```
public int length();
```

This method returns the total length of the entire data stream.

```
public String readString(int maxlen) throws SQLException;
```

This method reads the next attribute in the stream as a Java string. This method is similar to the **SQLInput.readString()** method except that a fixed length of data is read in. Since the opaque type is unknown to the driver, you *must* supply a maximum length for the driver to read in the next attribute properly.

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

This method reads the next attribute in the stream as Java byte array. This method is similar to the **SQLInput.readBytes()** method except that a fixed length of data is read in. Since the opaque type is unknown to the driver, you *must* supply a maximum length for the driver to read in the next attribute properly.

IfmxUdtSQLOutput Interface

The `com.informix.jdbc.IfmxUdtSQLOutput` interface extends `java.sql.SQLOutput` with several added methods. To use these methods, you must cast the `SQLOutput` references to `IfmxUdtSQLOutput`. Here are the signatures and descriptions of each added method:

```
public void writeString(String x, int length) throws SQLException;
```

This method writes the next attribute to the stream as a Java String. This method is similar to the `SQLOutput.writeString()` method except that a fixed length of data is written to the stream. If the string passed in is shorter than the specified length, the driver pads the string with zeros. Since the opaque type is unknown to the driver, you *must* supply a length for the driver to write the next attribute properly.

```
public void writeBytes(byte[] b, int length) throws SQLException;
```

This method writes the next attribute to the stream as a Java byte array. This method is similar to the `SQLOutput.writeBytes()` method except that a fixed length of data is written to the stream. If the array passed in is shorter than the specified length, the driver pads the array with zeros. Since the opaque type is unknown to the driver, you *must* supply a length for the driver to write the next attribute properly.

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

Furthermore, the driver also requires that all opaque data be transported as Informix DataBlade API data types, as defined in **incl/public/mitypes.h** (this file is included in all Informix Dynamic Server with Universal Data Option 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. For more information, see the [DataBlade API Programmer's Manual](#).

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. The 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.

In addition, you must provide a custom type map as described in [“Mapping Data Types” on page 3-65](#) to map this Java object to the corresponding SQL type name.

Caching Type Information

When an `SQLData` object inserts data into an opaque type column and `getSQLTypeName()` returns the name of the opaque type, Informix JDBC Driver uses the type information to verify that the data provided matches the data the database server expects. The driver asks the database server for the type information each time.

However, you can set an environment variable in the database URL, `ENABLE_CACHE_TYPE=1`, so the driver caches the type information the first time it is retrieved. In this case, Informix JDBC Driver asks the cache for the type information before requesting the data from the database server.

Inserting Data Examples

You can insert an opaque type as either its original type or its cast type. Here is an example of how to insert opaque data using the original type:

```
String s = "insert into charattr_tab (int_col, charattr_col) values (?, ?)";
System.out.println(s);
pstmt = conn.prepareStatement(s);
...
charattrUDT charattr = new charattrUDT();
charattr.chr1 = "a";
charattr.bold = true;
charattr.fontsize = (short)1;

pstmt.setInt(1, 1);
System.out.println("setInt...ok");

pstmt.setObject(2, charattr);
System.out.println("setObject(charattrUDT)...ok");

pstmt.executeUpdate();
```

If a casting function is defined and you would like to insert data as the casting type instead of the original type, you must call the **setXXX()** method that corresponds to the casting type. For example, if you have defined a function casting CHAR or LVARCHAR to a **charattrUDT** column, you can use the **setString()** method to insert data, as follows:

```
// Insert into UDT column using setString(int,String) and Java String object.
String s =
    "insert into charattr_tab " +
    "(decimal_col, date_col, charattr_col, float_col) " +
    "values (?,?,?,?)";
writeOutputFile(s);
PreparedStatement pstmt = myConn.prepareStatement(s);
...
String strObj = "(A, f, 18)";
pstmt.setString(3, strObj);
...
```

The **charattrUDT** class must implement the **SQLData** interface, as in the following example:

```
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);
    }
    // overrides 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 type name `charattr_udt` to the `charattrUDT` class, as 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("charattr_udt", Class.forName("charattrUDT"));
```

Retrieving Data Example

To retrieve Informix opaque types, you must use **ResultSet.getObject()**. 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("\tint_col      " + intret);

    charattrUDT charattrret = (charattrUDT)rs.getObject("charattr_col");
    System.out.print("\tcharattr_col ");
    if (curRow == 2 || curRow == 6)
    {
        if (rs.isNull())
            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 Examples

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 BLOB and CLOB extension classes.

A large object is stored as a MI_LO_HANDLE object within the opaque type. The MI_LO_HANDLE object is created using the methods provided in **com.informix.jdbc.IfxSmartBlob**, 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 MI_LO_HANDLE, as follows:

```
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 [“Manipulating Informix BLOB and CLOB Data Types”](#) on page 3-25 for details.

Unsupported Methods

The following methods 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)**

Manipulating Informix Distinct Types

A distinct type can map to the underlying base type or to a user-defined Java object. For example, a distinct type of INT can map to INT or to a Java object that encapsulates the data representation. This Java object must implement the **java.sql.SQLData** interface. You must provide a custom type map as described in “[Mapping Data Types](#)” on page 3-65 to map this Java object to the corresponding SQL type name.

Caching Type Information

When an **SQLData** object inserts data into a distinct type column and **getSQLTypeName()** returns the name of the distinct type, Informix JDBC Driver uses the type information to verify that the data provided matches the data the database server expects. The driver asks the database server for the type information each time.

However, you can set an environment variable in the database URL, **ENABLE_CACHE_TYPE=1**, so the driver caches the type information the first time it is retrieved. In this case, Informix JDBC Driver asks the cache for the type information before requesting the data from the database server.

Inserting Data Examples

A distinct type can map to either the underlying base type or to a user-defined Java object that implements the **SQLData** interface. Here is the SQL statement that defines the distinct type:

```
CREATE DISTINCT TYPE mymoney AS NUMERIC(10, 2);
CREATE TABLE distinct_tab (mymoney_col mymoney);
```

Here 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, Informix JDBC Driver maps to the underlying type 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. Here is the SQL statement that defines the distinct type:

```
CREATE DISTINCT TYPE mymoney AS NUMERIC(10, 2);
```

Here is the rest of the example:

```
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);
    }
    // overrides 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 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
{
    System.out.println("Select (i)");
    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++;
        myMoney mymoneyret = (myMoney)rs.getObject("mymoney_col");
        System.out.println("currentrow=" + curRow + " : " + mymoneyret);
    }

    System.out.println("total rows expected: " + curRow);
    stmt.close();
}
catch (SQLException e)
{
    System.out.println("***ERROR: " + e.getErrorCode() + " " +
e.getMessage());
    e.printStackTrace();
}

```

Unsupported Methods

The following methods are not supported for distinct types:

- **java.sql.SQLInput**
 - **readCharacterStream()**
 - **readRef()**
- **java.sql.SQLOutput**
 - **writeCharacterStream(Reader x)**
 - **writeRef(Ref x)**

Manipulating Informix BYTE and TEXT Data Types

The 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 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 [Informix Guide to SQL: Reference](#) and [Informix Guide to SQL: Syntax](#). You can find the on-line version of both of these guides at <http://www.informix.com/answers>.

Caching Large Objects

Whenever a BLOB, CLOB, TEXT, or BYTE object is fetched from the database server, the data is cached into 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 “[Memory Management of Large Objects](#)” on page 4-6.

Inserting or Updating Data Example

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, 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 tabl(col1 byte)");
}
catch (SQLException e)
{
    System.out.println("Failed to create table ..." + e.getMessage());
}

System.out.println("Trying to insert data using Prepare Statement ...");
try
{
    pstmt = conn.prepareStatement("insert into tabl 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 **File** object. The **FileInputStream** object is cast to its superclass **InputStream**, which is the expected data type of the second parameter to the **PreparedStatement.setBinaryStream()** method. The **setBinaryStream()** method is executed on the already prepared INSERT statement, which sets the input stream parameter. Finally, the **PreparedStatement.executeUpdate()** method is executed, which actually inserts the contents of the **data.dat** operating system file into the BYTE column of the table.

The **TextType.java** program shows how to insert data into a TEXT column. It is very similar to inserting into a BYTE column, except the method **setAsciiStream()** is used to set the input parameter instead of **setBinaryStream()**.

Selecting Data Example

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 BYTE and TEXT columns, 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 to how select data from a **BYTE** column and print out the data to the standard output device:

```
try
{
    stmt = conn.createStatement();
    rs = stmt.executeQuery("Select * from tabl");
    while( rs.next() )
    {
        row++;
        value = rs.getBinaryStream(1);
        System.out.println("\nResult of row #" + row + ", size = "
            + value.available() + " from getAsciiStream(1) ..\n");
        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();
        }
        System.out.println(new String(ary).trim());
    }
    catch (Exception e)
    {
        System.out.println("Error occur during 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 actually print out the contents of the column to standard output device. The **dispValue()** method uses byte arrays and the **InputStream.read()** method to systematically read the contents of the **BYTE** column.

The **TextType.java** program shows how to select data from a TEXT column. It is very similar to selecting from a BYTE column, except the **getAsciiStream()** method is used instead of **getBinaryStream()**.

Manipulating Informix BLOB and CLOB Data Types

You can access BLOB and CLOB data types in two ways:

- You can use the standard JDBC API methods described in the JDBC 2.0 specification from Sun Microsystems.
- If you are familiar with Informix BLOB and CLOB data types, you can use Informix extensions that are based on smart large object support within Informix Dynamic Server with Universal Data Option, which are described in this section.

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

The Informix extensions allow a JDBC 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.

Columns of either data type have a theoretical limit of 4 terabytes and a practical limit determined by your disk capacity.

The Informix smart large object implementation is based on the following classes:

- **IfxLobDescriptor**
- **IfxLocator**
- **IfxSmartBlob**
- **IfxBblob** and **IfxCblob**

These classes allow you to create, insert, and fetch large objects.

To create a smart large object

1. Create an **IfxLobDescriptor** object.
2. Create an **IfxLocator** object.
3. Create an **IfxSmartBlob** object that includes all the methods necessary to create, open, read, and write to a smart large object.
4. Execute the **IfxSmartBlob.IfLoCreate()** method to create a large object on the database server. This method returns a locator handle.
5. Execute **IfxSmartBlob.IfLoWrite()** to write the data to the smart large object.
6. Execute **IfxSmartBlob.IfLoClose()** to close the large object.
7. Execute **IfxSmartBlob.IfLoRelease()** to release the locator.

IfxLobDescriptor

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 have to create an **IfxLobDescriptor** object, as follows:

```
IfxLobDescriptor loDesc = new IfxLobDescriptor(conn);
```

The *conn* is a **java.sql.Connection** object. The **IfxLobDescriptor()** constructor sets all the default values for the object.

IfxLocator

The **IfxLocator** object (usually known as the *locator pointer* or *large object locator*) 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, as follows:

```
IfxLocator loPtr = new IfxLocator();
```

IfxSmartBlob

The **IfxSmartBlob** class provides all the methods necessary to create, open, read, and write to a smart large object. You can create an **IfxSmartBlob** object as follows:

```
IfxSmartBlob smb = new IfxSmartBlob(conn);
```

IfxSmartBlob Methods

You can use the following methods to create, open, read, and write to a smart large object. Here are the signatures and descriptions of each method:

```
public int IfxLoCreate(IfxLobDescriptor lo_desc, int flag, IfxLocator loPtr)
    throws SQLException
```

This method creates a new smart large object on the server and opens it for access within a Java application.

The *lo_desc* is an **IfxLobDescriptor** object.

The *flag* is an integer value that specifies the mode in which the new smart large object is opened on the server. A table of flag values appears in [“IfxSmartBlob Flag Values” on page 3-30](#).

The *loPtr* is an **IfxLocator** object.

The return value is of type integer; this integer is a large object locator handle that you can use in subsequent read, write, seek, and close methods. This is similar to a file handle in a file management system.

```
public int IfxLoOpen(IfxLocator loPtr, int flag) throws SQLException
```

This method opens an existing smart large object in the database server.

The *loPtr* is an **IfxLocator** object.

The *flag* is an integer value that specifies the mode in which the new smart large object is opened on the server. A table of flag values appears in [“IfxSmartBlob Flag Values” on page 3-30](#).

The return value is an integer locator handle that you can use in subsequent read, write, seek, and close methods. This is similar to a file handle in a file management system.

```
public void IfxLoRelease(IfxLocator loPtr) throws SQLException
```

This method releases an **IfxLocator** object after a smart large object is closed using the **IfxLoClose()** method. This frees the resources on the server.

The *loPtr* is an **IfxLocator** object.

```
public void IfxLoClose(int lofd) throws SQLException
```

This method closes the large object on the database server side. For any further access to the same large object, you must reopen it with the **IfxLoOpen()** method.

The *lofd* is a locator handle obtained by the **IfxLoCreate()** or **IfxLoOpen()** method.

```
public long IfxLoSeek(int lofd, long offset, int whence) throws SQLException
```

This method sets the read or write position within an already opened large object. The absolute position depends on the value of the second parameter, *offset*, and the value of the third parameter, *whence*.

The *lofd* is a locator handle obtained by the **IfxLoCreate()** or **IfxLoOpen()** method.

The *offset* is an offset from the starting seek position.

The *whence* identifies the starting seek position. A table of whence values appears in [“IfxSmartBlob Whence Values” on page 3-31](#).

The return value is a long integer representing the absolute position within the smart large object.

```
public void IfxLoTruncate(int lofd, long offset) throws SQLException
```

This method truncates a large object at an offset defined by the second parameter.

The *lofd* is a locator handle obtained by the **IfxLoCreate()** or **IfxLoOpen()** method.

The *offset* is the absolute position at which the smart large object is truncated.

```
public long IfxLoSize(int lofd) throws SQLException
```

This method returns a long integer representing the size of the large object.

The *lofd* is a locator handle obtained by the **IfxLoCreate()** or **IfxLoOpen()** method.

```
public byte[] IfxLoRead(int lofd, int nbytes) throws SQLException
```

This method returns *nbytes* bytes of data as a `byte[]` from the smart large object residing on the database server. This method allocates memory.

The *lofd* is a locator handle obtained by the **IfxLoCreate()** or **IfxLoOpen()** method.

The *nbytes* is the number of bytes read.

```
public int IfxLoRead(int lofd, byte[] buffer, int nbytes) throws SQLException
```

This method returns *nbytes* bytes of data in an already allocated buffer.

The *lofd* is a locator handle obtained by the **IfxLoCreate()** or **IfxLoOpen()** method.

The *buffer* is the `byte[]` buffer where the data is read.

The *nbytes* is the number of bytes read.

```
public int IfxLoRead(int lofd, FileOutputStream fout, int nbytes) throws
SQLException
```

This method reads *nbytes* bytes of data and stores it in a **FileOutputStream** object.

The *lofd* is a locator handle obtained by the **IfxLoCreate()** or **IfxLoOpen()** method.

The *fout* is the **FileOutputStream** object in which the data is stored.

The *nbytes* is the number of bytes read.

The return value is the number of bytes read.

```
public int IfxLoWrite(int lofd, byte[] buffer) throws SQLException
```

This method writes *buffer.length* bytes of data from the buffer into the smart large object.

The *lofd* is a locator handle obtained by the **IfxLoCreate()** or **IfxLoOpen()** method.

The *buffer* is the `byte[]` buffer where the data is read.

The return value is the number of bytes written.

```
public int IfxLoWrite(int loFd, InputStream fin, int length) throws SQLException
```

This method writes *length* bytes of data from an **InputStream** object into a smart large object.

The *loFd* is a locator handle obtained by the **IfxLoCreate()** or **IfxLoOpen()** method.

The *fin* is the **InputStream** object from which data is written into the smart large object.

The *length* is the number of bytes written into the smart large object.

The return value is the number of bytes written.

IfxSmartBlob Flag Values

Use the 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
Read only	Allows read operations only.	IfxSmartBlob.LO_RDONLY
Write only	Allows write operations only.	IfxSmartBlob.LO_WRONLY
Write/Append	Appends data you write to the end of the smart large object. Equivalent to write-only mode followed by a seek to the end of the object.	IfxSmartBlob.LO_APPEND
Read/Write	Allows read and write operations.	IfxSmartBlob.LO_RDWR

Here is an example of 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.

IfxSmartBlob Whence Values

Use the *whence* values in the following table with the **IfxLoSeek()** methods 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

Here is an example of 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);
System.out.println("A Smart blob is created ..");
int n = smb.IfxLoWrite(loFd, fin, fileLength);
System.out.println("Wrote data from FileInputStream ..");
smb.IfxLoClose(loFd);
System.out.println("Closed the Smart Blob ..");
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.

IfxBlob and IfxCblob Classes

The **IfxBlob** 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 a smart large object.

As described in Sun's JDBC 2.0 specification, when you query a table containing a BLOB or CLOB column, a **Blob** or **Clob** object is returned, depending upon the column type. You can then use the JDBC 2.0 supporting methods for **Blob** and **Clob** objects to access the smart large object.

IfxBlob Class

The **IfxBlob** class implements the **java.sql.Blob** interface. In addition to the methods defined by Sun, this class has the following Informix-specific methods to support the Informix extensions for smart binary large objects. Here are the signatures and descriptions of each extension:

```
public IfxBlob(IfxLocator loPtr)
```

This constructor creates an **IfxBlob** object from the **IfxLocator** object *loPtr*.

```
public IfxLocator getLocator() throws SQLException
```

This method returns an **IfxLocator** object from an **IfxBlob** object. You can then open, read, and write to the smart large object using the **IfxSmartBlob.IfxLoOpen()**, **IfxLoRead()**, and **IfxLoWrite()** methods.

IfxCblob Class

The **IfxCblob** class implements the **java.sql.Clob** interface. In addition to the methods defined by Sun, this class has the following Informix-specific methods to support the Informix extensions for smart binary large objects. Here are the signatures and descriptions of each extension:

```
public IfxCblob(IfxLocator loPtr)
```

This constructor creates an **IfxCblob** object from the **IfxLocator** object *loPtr*.

```
public IfxLocator getLocator() throws SQLException
```

This method returns an **IfxLocator** object from an **IfxCblob** object. You can then open, read, and write to the smart large object using the **IfxSmartBlob.IfxLoOpen()**, **IfxLoRead()**, and **IfxLoWrite()** methods.

Caching Large Objects

Whenever a BLOB, CLOB, TEXT, or BYTE object is fetched from the database server, the data is cached into 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 [“Memory Management of Large Objects” on page 4-6](#).

Creating a Smart Large Object Example

The example in this section illustrates the following steps.

To create a smart large object

1. Create an **IfxLobDescriptor** object.
2. Create an **IfxLocator** object.
3. Create an **IfxSmartBlob** object.
4. Use the **IfxSmartBlob.IfxLoCreate()** method to create the smart large object.
5. Use the **IfxSmartBlob.IfxLoWrite()** method to write the data into the smart large object.
6. Close the smart large object using the **IfxSmartBlob.IfxLoClose()** method.
7. Release the **IfxLocator** object on the server using the **IfxSmartBlob.IfxLoRelease()** method.

The following code demonstrates these steps:

```
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 ");
```

Inserting Data Example

After creating a smart large object, you must insert it into a BLOB or CLOB column. 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* is an **IfxLocator** object obtained from one of the previous sets of steps.

2. Use the **PreparedStatement.setBlob()** or **setClob()** method to insert the **Blob** or **Clob** object into the table.

The following code demonstrates these steps:

```
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 ");
```

Retrieving Data Example

The example in this section illustrates the following steps.

To use the Informix extensions 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 **IfxLocator** object on the server using 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.

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 `rs.getBlob()` method gets a **Blob** object. The casting is required to convert the returned object to an **IfxBlob** object. Next, the `IfxBlob.getLocator()` method gets an **IfxLocator** object from the **IfxBlob** object. After the **IfxLocator** object is available, you can instantiate an **IfxSmartBlob** object and use the `IfxSmartBlob.open()` and `IfxSmartBlob.read()` methods to read the smart large object data. Fetching **Clob** data is similar, but it uses the methods `rs.getClob`, `IfxClob.getLocator`, and so on.

If you use `getBlob()` or `getClob()` to fetch a BLOB column, 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. Informix JDBC Driver implicitly gets the content from the database server for you, using basically the same steps as the sample code.

Manipulating Informix SERIAL and SERIAL8 Data Types

Informix JDBC Driver provides support for the Informix SERIAL and SERIAL8 data types via the methods `getSerial()` and `getSerial8()`, which are part of the implementation of the `java.sql.Statement` interface.

Since 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 be able to handle SERIAL and SERIAL8 table 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 [Informix Guide to SQL: Reference](#) and [Informix Guide to SQL: Syntax](#). You can find the on-line version of both of these guides at <http://www.informix.com/answers>.

Manipulating Informix INTERVAL Data Types

The Informix INTERVAL data type stores a value that represents a span of time. INTERVAL data types are divided into 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:

- [Informix Guide to SQL: Tutorial](#)
- [Informix Guide to SQL: Reference](#)
- [Informix Guide to SQL: Syntax](#)

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 actually create and manipulate INTERVAL data types.

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.

Variable	Description
TU_YEAR	Time unit for the YEAR qualifier field
TU_MONTH	Time unit for the MONTH qualifier field
TU_DAY	Time unit for the DAY qualifier field
TU_HOUR	Time unit for the HOUR qualifier field
TU_MINUTE	Time unit for the MINUTE qualifier field
TU_SECOND	Time unit for the SECOND qualifier field
TU_FRAC	Time unit for the leading FRACTION qualifier field
TU_F1	Time unit for the ending field of the first position of FRACTION
TU_F2	Time unit for the ending field of the second position of FRACTION
TU_F3	Time unit for the ending field of the third position of FRACTION
TU_F4	Time unit for the ending field of the fourth position of FRACTION
TU_F5	Time unit for the ending field of the fifth position of FRACTION

The **IntervalYM** and **IntervalDF** classes inherit these variables from the **Interval** base class.

Interval Methods

You can use the following methods to extract information about binary qualifiers. Here are the signatures and descriptions of each method:

```
public static byte getEndCode(short qualifier)
```

This method extracts the ending field code (one of the TU_XXX variables) from a qualifier.

```
public static java.lang.String getFieldName(byte code)
```

This method takes the TU_XXX value of part of an interval and returns the string value. For example, `getFieldName(TU_YEAR)` returns the string `YEAR`.

```
public static java.lang.String getIfxTypeName(int type, short qual)
```

This method takes a qualifier and returns the entire name of the interval in string form.

```
public static byte getLength(short qualifier)
```

This method extracts the length of a qualifier.

```
public static short getQualifier(byte length, byte startCode, byte endCode) throws  
java.sql.SQLException
```

This method creates a binary qualifier from a length, start code (TU_XXX), and end code (TU_XXX). For example, `getQualifier(4, TU_YEAR, TU_MONTH)` creates a binary representation of the `YEAR TO MONTH` qualifier.

```
public static byte getScale(short qualifier)
```

This method returns the number of digits in the FRACTION part of a day-time qualifier.

```
public static byte getStartCode(short qualifier)
```

This method extracts the starting field code (one of the TU_XXX variables) from a qualifier.

The **IntervalYM** and **IntervalDF** classes inherit these methods from **Interval**.

The IntervalYM Class

The **com.informix.lang.IntervalYM** class allows you to manipulate year-month intervals.

IntervalYM Constructors

The following constructors allow you to create year-month intervals. Here are the signatures and descriptions of each constructor:

```
public IntervalYM() throws java.sql.SQLException
```

This is the default constructor.

```
public IntervalYM(java.sql.Timestamp t1, java.sql.Timestamp t2) throws  
java.sql.SQLException
```

This constructor creates an interval from two timestamps.

```
public IntervalYM(int years, int months) throws java.sql.SQLException
```

This constructor creates a year-month interval from year and month values. Large month values are converted to years.

```
public IntervalYM(java.sql.String string) throws java.sql.SQLException
```

This constructor creates a year-month interval from a string. For information about string INTERVAL formats, refer to the [Informix Guide to SQL: Syntax](#).

```
public IntervalYM(java.sql.String string, int length, byte startCode, byte  
endCode) throws java.sql.SQLException
```

This constructor creates a year-month interval from a string and qualifier information. For information about string INTERVAL formats, refer to the [Informix Guide to SQL: Syntax](#).

IntervalYM Methods

The following methods allow you to manipulate year-month intervals. Here are the signatures and descriptions of each method:

```
boolean equals(java.lang.Object other)
```

This method compares two intervals for equality.

```
void fromString(java.lang.String other)
```

This method sets the values for the interval from a string.

```
long getMonths()
```

This method returns the number of months in the interval.

```
boolean greaterThan(IntervalYM other)
```

This method compares the first interval to the second to see if the first is longer.

```
boolean lessThan(IntervalYM other)
```

This method compares the first interval to the second to see if the second is longer.

```
void set(int years, int months)
```

This method sets the values for the interval from year and month values. Large month values are converted to years.

```
void set(java.lang.String string)
```

This method sets the values for the interval from a string.

```
void set(java.sql.Timestamp t1, java.sql.Timestamp t2)
```

This method sets the values for the interval from two timestamps.

```
void setQualifier(int length, byte startCode, byte endCode)
```

This method sets the qualifier from the length, start code, and end code.

```
void setQualifier(short qualifier)
```

This method sets the qualifier using an existing qualifier.

```
java.lang.String toString()
```

This method creates a string representation of the interval in the format *yyyy-mm*. The fields present depend on the qualifier. Blanks replace leading zeros.

The IntervalDF Class

The `com.informix.lang.IntervalDF` class allows you to manipulate day-time intervals.

IntervalDF Constructors

The following constructors allow you to create day-time intervals. Here are the signatures and descriptions of each constructor:

```
public IntervalDF() throws java.sql.SQLException
```

This is the default constructor.

```
public IntervalDF(java.sql.Timestamp t1, java.sql.Timestamp t2) throws
java.sql.SQLException
```

This constructor creates an interval from two timestamps.

```
public IntervalDF(long seconds, long nanos) throws java.sql.SQLException
```

This constructor creates a day-time interval from second and nanosecond values. Large second values are converted to minutes, hours, or days.

```
public IntervalDF(java.sql.String string) throws java.sql.SQLException
```

This constructor creates a day-time interval from a string. For information about string INTERVAL formats, refer to the [Informix Guide to SQL: Syntax](#).

```
public IntervalDF(java.sql.String string, int length, byte startCode, byte
endCode) throws java.sql.SQLException
```

This constructor creates a day-time interval from a string and qualifier information. For information about string INTERVAL formats, refer to the [Informix Guide to SQL: Syntax](#).

IntervalDF Methods

The following methods allow you to manipulate day-time intervals. Here are the signatures and descriptions of each method:

```
boolean equals(java.lang.Object other)
```

This method compares two intervals for equality.

```
void fromString(java.lang.String other)
```

This method sets the values for the interval from a string.

```
long getNanoSeconds()
```

This method returns the number of nanoseconds in the interval.

```
long getSeconds()
```

This method returns the number of seconds in the interval.

```
boolean greaterThan(IntervalDF other)
```

This method compares the first interval to the second to see if the first is longer.

```
boolean lessThan(IntervalDF other)
```

This method compares the first interval to the second to see if the second is longer.

```
void set(long seconds, long nanos)
```

This method sets the values for the interval from second and nanosecond values. Large second values are converted to minutes, hours, or days.

```
void set(java.lang.String string)
```

This method sets the values for the interval from a string.

```
void set(java.sql.Timestamp t1, java.sql.Timestamp t2)
```

This method sets the values for the interval from two timestamps.

```
void setQualifier(int length, byte startCode, byte endCode)
```

This method sets the qualifier from the length, start code, and end code.

```
java.lang.String toString()
```

This method creates a string representation of the interval in the format *dddd hh:mm:ss.nano*. The fields present depend on the qualifier. Blanks replace leading zeros.

Interval Example

The **Intervaldemo.java** program, which is included in Informix JDBC Driver, shows how to insert into and select from the two types of INTERVAL data types.

Manipulating Informix 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, 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.

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 [“Mapping Data Types” on page 3-65](#).



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.

Collection Examples

Here is the database schema:

```
create table tab ( a set(integer not null), b integer);
insert into tab values ("set{1, 2, 3}", 10);
```

Here is a fetch example using a **java.util.HashSet** object. The complete demonstration is in the **demo5.java** file in the **complex-types** directory. For more information, see [Appendix A, "Sample Code Files."](#)

```
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, 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.**

Here is a fetch example using a `java.util.TreeSet` object. The complete demonstration is in the `demo6.java` file in the `complex-types` directory. For more information, see [Appendix A, "Sample Code Files."](#)

```

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, 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.

Here is an example of an insert. The complete demonstration is in the **demo7.java** file in the **complex-types** directory. For more information, see [Appendix A, “Sample Code Files.”](#)

```

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)` method 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

Here is the database schema:

```

create table tab ( a set(integer not null), b integer);
insert into tab values ("set{1, 2, 3}", 10);

```

Here is a fetch example using a `java.sql.Array` object. The complete demonstration is in the `demo8.java` file in the `complex-types` directory. For more information, see [Appendix A, “Sample Code Files.”](#)

```
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 (primitive `int` types, not `Integer` objects). Because the `getArray()` method has been called with index and count values, only a subset of data is returned.

Manipulating Informix 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 Informix JDBC Driver instantiates a Java object or a **Struct** object for a fetched named row depends on whether there is a customized type-mapping entry or not, as follows:

- If there is an entry for a named row in the `Connection.getTypeMap()` map, or if you provided a type mapping using the `getObject()` method, a single Java object is instantiated.
- If there is no entry for a named row in the `Connection.getTypeMap()` map, and if you have not provided a type mapping using the `getObject()` method, a **Struct** object is instantiated.

Unnamed rows are always fetched into **Struct** objects.



***Important:** Regardless of whether you use the `SQLData` or `Struct` interface, if a named or unnamed row contains an opaque data type column, there must be a type-mapping entry for it. If you are using the `Struct` interface to access a row that contains an opaque data type column, you need a customized type map for the opaque data type column, but not for the row as a whole.*

For more information about custom type mapping, see [“Mapping Data Types” on page 3-65](#).

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. The class can have other members in addition to these. The members can be in any order and need not be public. The class must implement the `writeSQL()`, `readSQL()`, and `getSQLTypeName()` methods as defined in the `SQLData` interface, and can implement additional methods. You can use the `ClassGenerator` utility to create the class; for more information, see [“The ClassGenerator Utility” on page 3-62](#).

To link this Java class with the named row, create a customized type mapping using the **Connection.setTypeMap()** method or the **getObject()** method. For more information about type mapping, see [“Mapping Data Types” on page 3-65](#).

You cannot use the **SQLData** interface to access unnamed rows.

Using the Struct Interface

The JDBC 2.0 documentation does not specify that **Struct** objects can be parameters to the **PreparedStatement.setObject()** method. However, 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 **Struct** interface. However, if you do not create your own class, you must perform a fetch to retrieve the ROW data and type information before you can insert or update the ROW data.

If you create your own class to implement the **java.sql.Struct** interface, the class you create has to implement all the **java.sql.Struct** methods, including the **getSQLTypeName()** method.

If you do not create your own **Struct** class, Informix JDBC Driver automatically calls the **getSQLTypeName()** method. In this case, this method returns the type name for a named row or the row definition for an unnamed row. If you create your own class, however, you can choose what the **getSQLTypeName()** method returns.

You have to return the row definition for unnamed rows, but you can return 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 [Informix Guide to SQL: Reference](#) or the [Informix Guide to SQL: Syntax](#).

***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.*

Interval and Collection Support

Informix has extended the `java.sql.SQLOutput` and `java.sql.SQLInput` methods 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. This 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.

Caching Type Information

When a **Struct** or **SQLData** object inserts data into a ROW column and `getSQLTypeName()` returns the name of a named row, Informix JDBC Driver uses the type information to verify that the data provided matches the data the database server expects. The driver asks the database server for the type information each time.

However, you can set an environment variable in the database URL, `ENABLE_CACHE_TYPE=1`, so the driver caches the type information the first time it is retrieved. In this case, Informix JDBC Driver asks the cache for the type information before it requests the data from the database server.

SQLData Examples

The following example includes a Java class that implements the `java.sql.SQLData` interface.

Here is the 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");
```

Here 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;
    }
}
```

Here 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. The complete demonstration is in the **demo1.java** file in the **complex-types** directory. For more information, see [Appendix A, "Sample Code Files."](#)

```

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("executedQuery()...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**. 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 out 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.

Here is an example method for inserting a Java object into a named row column using the **setObject()** method. The complete demonstration is in the **demo2.java** file in the **complex-types** directory. For more information, see [Appendix A, “Sample Code Files.”](#)

```

java.util.Map map = conn.getTypeMap();
map.put("fullname_t", Class.forName("fullname"));
map.put("person_t", Class.forName("person"));
...
PreparedStatement pstmt;
System.out.println("Populate person and fullname objects");
person who = new person();
fullname name = new fullname();
name.last = "Jones";
name.first = "Sarah";
who.id = 567;
who.name = name;
who.age = 17;

String s = "insert into teachers values (?, 'physics')";
pstmt = conn.prepareStatement(s);
System.out.println("prepared...ok");

pstmt.setObject(1, who);
System.out.println("setObject()...ok");

int rowcount = pstmt.executeUpdate();
System.out.println("executeUpdate()...ok");
pstmt.close();

```

The **conn.getTypeMap()** method returns the named row mapping information from the **java.util.Map** object through the **Connection** object.

The **map.put()** method registers the mappings between the nested named row on the database server, **fullname_t**, and the Java class **fullname**, and between the named row on the database server, **person_t**, and the Java class **person**.

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. 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 out it is a named row.

The `writeSQL()` method in the `person` class calls the corresponding `SQLOutput.writeXXX()` method for each member in the class, each of which maps to one field in the named row `person_t`. The `writeSQL()` method in the class contains calls to the `SQLOutput.writeObject(name)` and `SQLOutput.writeInt(id)` methods. Each member of the class `person` is serialized and written into a stream.

Struct Examples

This example fetches an unnamed ROW column. Here is the 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");
```

Here is the rest of the example. The complete demonstration is in the **demo3.java** file in the **complex-types** directory. For more information, see [Appendix A, "Sample Code Files."](#)

```

PreparedStatement pstmt;
ResultSet rs;
pstmt = conn.prepareStatement("select person from teachers");
System.out.println("prepare ...ok");
rs = pstmt.executeQuery();
System.out.println("executedQuery()...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**, that implements the **java.sql.Struct** interface. As an alternative, you can use a **Struct** object returned from the **ResultSet.getObject()** method instead of the **GenericStruct** class.

```
import java.sql.*;
import java.util.*;
public class GenericStruct implements java.sql.Struct
{
    private Object [] attributes = null;
    private String typeName = null;

    /*
     * Constructor
     */
    GenericStruct() { }

    GenericStruct(String name, Object [] obj)
    {
        typeName = name;
        attributes = obj;
    }
    public String getSQLTypeName()
    {
        return typeName;
    }
    public Object [] getAttributes()
    {
        return attributes;
    }
    public Object [] getAttributes(Map map) throws SQLException
    {
        // this class shouldn't be used if there are elements
        // that need customized type mapping.
        return attributes;
    }
    public void setAttributes(Object [] objArray)
    {
        attributes = objArray;
    }
    public void setSQLTypeName(String name)
    {
        typeName = name;
    }
} /* GenericStruct class */
```

The following Java program inserts a ROW column. The complete demonstration is in the **demo4.java** file in the **complex-types** directory. For more information, see [Appendix A, “Sample Code Files.”](#)

```
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, struct)** method in this example, Informix JDBC Driver determines that the information is to be transported from the client to the database server as a ROW column, because the **GenericStruct** object is an instance of the **java.sql.Struct** interface.

Each element in the array is serialized, verifying that each element matches the type as defined by the **getSQLTypeName()** method.

The ClassGenerator Utility

The **ClassGenerator** utility generates a Java class for a named row type defined in the system catalog. It 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, and 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**.

Here is the **ClassGenerator** usage:

```
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.

Here is the structure of **setup.std**:

```
URL jdbc:host-name:port-number
informixserver informixservername
database database
user user
passwd password
```

Simple Named Row

First you create the named row on the database server:

```
create row type employee (name char (20), age int);
```

Next, you run the class generation:

```
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
```

Here 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

First, you create the named row on the database server:

```
create row type manager (emp employee, salary int);
```

Next, you run the class generation. 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);
    }
}
```

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)**

Mapping Data Types

This section discusses mapping issues between data types defined in a Java program and the data types supported by the Informix database server. In particular, it covers the following topics:

- Mapping between JDBC API data types and Informix data types, next
- **PreparedStatement.setXXX()** methods supported by Informix JDBC Driver on [page 3-68](#).
- **ResultSet.getXXX()** methods supported by Informix JDBC Driver on [page 3-77](#).

Mapping Between Informix and JDBC Data Types

Since 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
CHAR	CHAR(<i>n</i>)
DATE	DATE
DECIMAL	DECIMAL
DOUBLE	FLOAT
FLOAT	SMALLFLOAT

(1 of 2)

JDBC API Data Type	Informix Data Type
INTEGER	INTEGER
LONGVARBINARY	BYTE
LONGVARCHAR	TEXT
NUMERIC	DECIMAL
REAL	SMALLFLOAT
SMALLINT	SMALLINT
TIME	DATETIME
TIMESTAMP	DATETIME
TINYINT	SMALLINT
VARBINARY	BYTE
VARCHAR	VARCHAR(<i>m,r</i>)

(2 of 2)

The LONGVARBINARY and LONGVARCHAR JDBC types can also map to Informix BLOB and CLOB types, respectively.



Important: Informix JDBC Driver maps **java.sql.Timestamp** to the Informix type **DATETIME YEAR TO FRACTION(5)** and **java.sql.Time** to the Informix type **DATETIME HOUR TO SECOND**. Informix **DATETIME** types are very restrictive and are not interchangeable. If you attempt to map **java.sql.Time** to **DATETIME YEAR TO FRACTION(5)** or **java.sql.Timestamp** to **DATETIME HOUR TO SECOND**, you might get an error from the Informix database server. Any other **DATETIME** qualifiers are not supported.



Important: The **Ref** type is not supported by Informix servers.

Mapping Between Informix and JDBC Data Types

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

JDBC Type	Java Object Type	Informix Type
java.sql.Types.OTHER	boolean	BOOLEAN
java.sql.Types.SMALLINT	smallint	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.Blob	BLOB
java.sql.Types.BLOB	java.io.inputStream byte[]	IfxTypes.IFX_TYPE_BLOB
java.sql.Types.LONGVARCHAR	java.sql.Clob	CLOB
java.sql.Types.CLOB	java.io.inputStream java.lang.String	IfxTypes.IFX_TYPE_CLOB
java.sql.Types.LONGVARBINARY	java.io.inputStream	BYTE
java.sql.Types.BLOB	java.sql.Blob byte[]	IfxTypes.IFX_TYPE_BYTE
java.sql.Types.LONGVARCHAR	java.io.InputStream	TEXT
java.sql.Types.CLOB	java.sql.Clob java.sql.String	IfxTypes.IFX_TYPE_TEXT
java.sql.Types.JAVA_OBJECT	java.sql.SQLData	named row
java.sql.Types.STRUCT	java.sql.Struct	IfxTypes.IFX_TYPE_ROW

(1 of 2)

JDBC Type	Java Object Type	Informix Type
java.sql.Types.STRUCT	java.sql.Struct	unnamed row IfxTypes.IFX_TYPE_ROW
java.sql.Types.ARRAY	java.sql.Array	SET, MULTISSET
java.sql.Types.OTHER	java.util.LinkedList	IfxTypes.IFX_TYPE_SET
	java.util.HashSet	IfxTypes.IFX_TYPE_MULTISSET
	java.util.TreeSet	
java.sql.Types.ARRAY	java.sql.Array	LIST
java.sql.Types.OTHER	java.util.ArrayList	IfxTypes.IFX_TYPE_LIST
	java.util.LinkedList	

(2 of 2)

A Java **boolean** object can map to a **smallint** object or an Informix BOOLEAN data type. Informix JDBC Driver attempts to map it according to the column type. However, in cases such as **PreparedStatement** host variables, Informix JDBC Driver cannot access the column types, so the mapping is somewhat limited. Refer to “[PreparedStatement.setXXX\(\) Extensions](#),” next, for more details on data type mapping.

PreparedStatement.setXXX() Extensions

Informix Dynamic Server with Universal Data Option introduces many extended data types, such as BLOB, CLOB, BOOLEAN, LVARCHAR, and opaque 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 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, Informix JDBC Driver does not have access to the column information. In those cases, unless you use the extended methods listed next, the driver maps those data columns using the corresponding pre-Universal Data Option data types listed in the table on [page 3-65](#). 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.

To direct the driver to map to a certain data type (so there is no ambiguity in UPDATE statements and WHERE clauses), you can use extensions to the **PreparedStatement.setXXX()** methods. The Informix type must be the last parameter to the standard JDBC **PreparedStatement.setXXX()** interface.

The only data types that might have ambiguity are BOOLEAN, LVARCHAR, TEXT, BYTE, BLOB, and CLOB. Refer to the data type mapping table on [page 3-67](#) for more details on how these extended data types are mapped to Java and JDBC types.

The Mapping Extensions

The **java.sql.PreparedStatement** interface enables you to map to a specific Informix data type in cases that might result in ambiguity when you are inserting or updating data. The extensions are listed here. Each extension is followed by its full signature.

The setXXX Methods

IfmxPreparedStatement.setArray()

```
public void setArray(int parameterIndex, java.sql.Array x, int ifxType) throws  
    SQLException
```

IfmxPreparedStatement.setAsciiStream()

```
public void setAsciiStream(int i, InputStream x, int length, int ifxType) throws  
    SQLException
```

IfmxPreparedStatement.setBigDecimal()

```
public void setBigDecimal(int i, java.math.BigDecimal x, int ifxType)  
    throws SQLException
```


IfmxPreparedStatement.setBinaryStream()

```
public void setBinaryStream(int i, InputStream x, int length, int ifxType) throws  
    SQLException
```

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, java.io.Reader reader,  
    int length, int ifxType) throws SQLException
```

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
```

IfmxPreparedStatement.setDate()

```
public void setDate(int parameterIndex, java.sql.Date x, Calendar Cal,  
    int ifxType) throws SQLException
```

IfmxPreparedStatement.setDouble()

```
public void setDouble(int i, double x, int ifxType) throws SQLException
```

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
```

IfmxPreparedStatement.setTime()

```
public void setTime(int parameterIndex, java.sql.Time x, Calendar Cal,  
int ifxType) throws SQLException
```

IfmxPreparedStatement.setTimestamp()

```
public void setTimestamp(int i, Timestamp x, int ifxType) throws SQLException
```

IfmxPreparedStatement.setTimestamp()

```
public void setTimestamp(int parameterIndex, java.sql.Timestamp x, Calendar Cal)  
throws SQLException
```

The IfxSetObject Methods

IfmxPreparedStatement.IfxSetObject()

```
public void IfxSetObject(int i, Object x, int scale, int ifxType) throws  
SQLException
```

IfmxPreparedStatement.IfxSetObject()

```
public void IfxSetObject(int i, Object x, int ifxType) throws SQLException
```

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.

To use the preceding methods, you must cast your **PreparedStatement** references to **IfmxPreparedStatement**:

```
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);
```

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	SMINT
IfxTypes.IFX_TYPE_INT	INT
IfxTypes.IFX_TYPE_FLOAT	FLOAT
IfxTypes.IFX_TYPE_SMFLOAT	SMFLOAT
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

(1 of 3)

IfxTypes Constant	Informix Data Type
IfxTypes.IFX_TYPE_DATETIME	DATETIME
IfxTypes.IFX_TYPE_BYTE	BYTES
IfxTypes.IFX_TYPE_TEXT	TEXT
IfxTypes.IFX_TYPE_VARCHAR	VARCHAR
IfxTypes.IFX_TYPE_INTERVAL	SQLINTERVAL
IfxTypes.IFX_TYPE_NCHAR	NCHAR
IfxTypes.IFX_TYPE_NVCHAR	NVCHAR
IfxTypes.IFX_TYPE_INT8	INT8
IfxTypes.IFX_TYPE_SERIAL8	SERIAL8
IfxTypes.IFX_TYPE_SET	SQLSET
IfxTypes.IFX_TYPE_MULTISSET	SQLMULTISSET
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	BOOL
IfxTypes.IFX_TYPE_IMPEXP	IMPEXP

(2 of 3)

lfxTypes Constant	Informix Data Type
lfxTypes.IFX_TYPE_IMPEXPBIN	IMPEXPBIN
lfxTypes.IFX_TYPE_CLOB	CLOB
lfxTypes.IFX_TYPE_BLOB	BLOB

(3 of 3)

Extension Summary

The following table lists the **PreparedStatement.setXXX()** methods that 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 3-65](#). The table lists the **setXXX()** methods you can use to write data of a particular JDBC API data type.

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

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	
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								

(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	
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	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				
setUnicodeStream()																				
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 Informix JDBC Driver supports for the Informix extended data types, the mappings for which are shown in the table on [page 3-67](#). 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 **setXXX()** method Informix recommends you use; a lowercase **x** indicates other **setXXX()** methods supported by Informix JDBC Driver. The table does not include **setXXX()** methods that you cannot use with any of the Informix extended data types.

setXXX() 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				
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.

Supported ResultSet.getXXX() Methods

Use the **ResultSet.getXXX()** 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.getXXX()** methods that 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 3-65](#). The table lists the **getXXX()** methods you can use to retrieve data of a particular JDBC API data type.

An uppercase and bold X indicates the **getXXX()** method Informix recommends you use; a lowercase x indicates other **getXXX()** methods supported by Informix JDBC Driver.

getXXX() 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
getBytes()	X	x	x	x	x	x	x	x	x		x ¹	x ¹							
getShorts()	x	X	x	x	x	x	x	x	x		x ¹	x ¹							
getInts()	x	x	X	x	x	x	x	x	x		x ¹	x ¹							
getLongs()	x	x	x	X	x	x	x	x	x		x ¹	x ¹							
getFloats()	x	x	x	x	X	x	x	x	x		x ¹	x ¹							
getDoubles()	x	x	x	x	x	X	X	x	x		x ¹	x ¹							

(1 of 2)

getXXX() 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	
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	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				
getUnicodeStream()																				
getBinaryStream()													x	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, **getBytes(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.getXXX()** methods that Informix JDBC Driver supports for the Informix extended data types, the mappings for which are shown in the table on [page 3-67](#). The table lists the **getXXX()** methods you can use to retrieve data of a particular extended data type.

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

getXXX() 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				
getBinaryStream()	x			x		X					
getObject()	x	x	X	x	x	x	x	X	X	x	x
getArray()										x	x
getBlob()				X							
getClob()					X						

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

Troubleshooting

In This Chapter	4-3
Debugging Your JDBC API Program.	4-3
Using the Debug Version of the Driver.	4-3
Turning on Tracing	4-4
Performance Issues	4-5
Using the FET_BUF_SIZE Environment Variable	4-5
Memory Management of Large Objects	4-6
Reducing Network Traffic	4-7

In This Chapter

This chapter provides troubleshooting tips for Informix JDBC Driver. It covers the following topics:

- [“Debugging Your JDBC API Program”](#)
- [“Performance Issues”](#)

Debugging Your JDBC API Program

If your Java program contains JDBC API programming errors, you might want to use the debug version of 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 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 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 Informix JDBC Driver in a Java application or Java applet, refer to [“Using the Driver in an Application” on page 1-14](#) or [“Using the Driver in an Applet” on page 1-15](#), respectively.

Turning on Tracing

Trace output consists of the following two kinds of information:

- General information from 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 database URL or the 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 Informix JDBC Driver. Can be set to one of the following levels: <ul style="list-style-type: none">■ 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.
PROTOCOLTRACE	Traces the SQLI protocol messages sent between your Java program and the Informix database server. Can be set to the following levels: <ul style="list-style-type: none">■ 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.

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 using a database URL or a property list, refer to [“Establishing a Connection”](#) on page 2-3.

Performance Issues

This section describes issues that might affect the performance of your queries: the **FET_BUF_SIZE** environment variable, memory management of the Informix TEXT and BYTE data types, memory management of the Informix BLOB and CLOB data types, and reducing network traffic.

Using the **FET_BUF_SIZE** Environment Variable

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 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. 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.

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**.

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` environment variable to increase the size of the tuple buffer.

For more information on setting Informix environment variables, see [“Establishing a Connection” on page 2-3](#).

Memory Management of 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 as one of the name-value pairs in the database URL syntax, 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, 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 1-15](#).
- To always store the data in memory, set the `LOBCACHE` value to a negative number. If the required amount of memory is not available, 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, 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 [“Manipulating Informix BYTE and TEXT Data Types” on page 3-20](#). For programming information on how to use the **BLOB** and **CLOB** data types in a Java program, refer to [“Manipulating Informix BLOB and CLOB Data Types” on page 3-25](#).

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.

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 Informix JDBC Driver has received all rows from the database server, the **ResultSet.close()** method might still require a network round-trip when **OPTOFC** is set to 1.

Set **IFX_AUTOFREE** to 1 to specify that the **Statement.close()** method does not require a network round-trip to free the database server cursor resources if the cursor has already been closed in the database server.

You can also use the **setAutoFree(boolean flag)** and **getAutoFree()** methods to free database server cursor resources. For more information, see [“The Auto Free Feature” on page 2-47](#).

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 [“Establishing a Connection” on page 2-3](#).

Sample Code Files

UNIX

This appendix lists and describes the code examples provided with Informix JDBC Driver.

The main examples are located in the following directories:

- *\$JDBCLOCATION/demo/basic*
- *\$JDBCLOCATION/demo/clob-blob*
- *\$JDBCLOCATION/demo/udt-distinct*
- *\$JDBCLOCATION/demo/complex-types*

JDBCLOCATION refers to the directory where you installed Informix JDBC Driver.

Another set of examples is located in the directory *\$JDBCLOCATION/demo/stores7*. A README file in the **demo** directory explains the various demonstration files and how to execute them.

An RMI example is located in the directory *\$JDBCLOCATION/demo/rmi*. A README file in the **demo** directory explains how to execute the example.

Example output files of the **ClassGenerator** utility are located in the directory *\$JDBCLOCATION/demo/tools/classgenerator*. A README file in the **classgenerator** directory explains how to use the example. ♦

Windows

The main examples are located in the following directories:

- *%JDBCLOCATION%\demo\basic*
- *%JDBCLOCATION%\demo\clob-blob*
- *%JDBCLOCATION%\demo\udt-distinct*

■ **%JDBCLOCATION%\demo\complex-types**

JDBCLOCATION refers to the directory where you installed Informix JDBC Driver.

Another set of examples is located in the directory **%JDBCLOCATION%\demo\stores7**. A README file in the **demo** directory explains the various demonstration files and how to execute them.

An RMI example is located in the directory **%JDBCLOCATION%\demo\rmi**. A README file in the **demo** directory explains how to execute the example.

Example output files of the **ClassGenerator** utility are located in the directory **%JDBCLOCATION%\demo\tools\classgenerator**. A README file in the **classgenerator** directory explains how to use the example. ♦

The following table lists the files in the **basic** directory. For each code example, the table displays the name of the Java program and a brief description of what the program does.

Demo Program Name	Description
autofree.java	Shows how to use the 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 .
CreateDB.java	Creates a database called testDB .
DBCENTURYSelect.java	Shows how to retrieve a date string representation that has a four-digit year expansion based on the DBCENTURY property value from the URL string.
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 .

(1 of 2)

Demo Program Name	Description
GLDATESelect.java	Shows how to retrieve a date object and a date string representation from the database based on the GL_DATE property value from the URL string.
Intervaldemo.java	Shows how to insert and select Informix INTERVAL data.
LOCALESelect.java	Shows how to retrieve a date object and a date string representation from the database based on the CLIENT_LOCALE property value from the URL string.
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 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 ResultSet-MetaData 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.

(2 of 2)

The following table lists the files in the **lob-lob** directory. For each code example, the table displays the name of the Java program and a brief description of what the program does.

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.

The following table lists the files in the **udt-distinct** directory. For each code example, the table displays the name of the Java program and a brief description of what the program does.

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 .

(1 of 2)

Demo Program Name	Description
distinct_d2.java	Shows how to create a second distinct type without using SQLData .
dropDB.java	Drops the database that the other udt-distinct demonstration files use.
largebinUDT.java	Shows how to implement an opaque type (smart large object embedded) using SQLData .
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 (smart large object embedded).

(2 of 2)

The following table lists the files in the **complex-types** directory. For each code example, the table displays the name of the Java program and a brief description of what the program does.

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.

(1 of 2)

Demo Program Name	Description
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.
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)

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 blobpage to blobpage.
- 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 and 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 *explicit cast*, *implicit cast*.

cast function	A function that is used to implement an implicit or explicit cast. A cast function performs the necessary operations for conversion between two data types. It must be registered as a cast with the CREATE CAST statement before it can be used.
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 smart large object data type that stores blocks of text items, such as ASCII or PostScript files. See also <i>smart large object</i> .
code set	A character set of one or more natural-language alphabets with symbols for digits, punctuation, and diacritical marks. Each character set has at least one code set, which maps its characters to unique bit patterns. ASCII, ISO8559-1, Microsoft 1252, and EBCDIC are examples of code sets for the English language.
collection	An instance of a collection data type; a group of elements of the same data type stored in a SET, MULTISSET, or LIST. See also <i>collection data type</i> .
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, MULTISSET, 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 <i>explicit connection</i> , <i>implicit connection</i> .
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 <i>built-in data type</i> , <i>extended data type</i> .
DataBlade API	The C application programming interface (API) for Informix Dynamic Server with Universal Data Option. The DataBlade API is used for the development of DataBlade module applications that access data stored in this kind of database. The DataBlade API sends SQL command strings to the server for execution and processes results returned by the server to the application.
DataBlade API data types	A set of Informix-provided C data types that correspond to some of the Informix SQL data types, including extended data types. Informix recommends that you use these data types instead of the standard C data types to ensure portable applications.
dbspace	A logical collection of one or more chunks of contiguous disk space 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. Large objects are stored in sbspaces.
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 that is created with the CREATE DISTINCT TYPE statement. A distinct data type is 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.
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. An explicit cast requires a function if the internal storage representations of the two data types are not equivalent.
explicit connection	A connection made to a database environment that uses the CONNECT statement. See also <i>implicit connection</i> .
extended data type	A term used to refer to data types that are not built-in: namely, collection data types, row data types, opaque data types, and distinct data types.
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 <i>explicit cast</i> .
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 every 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.
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	<p>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. The Universal Data Option recognizes two kinds of large objects: simple large objects (TEXT, BYTE) and smart large objects (CLOB, BLOB).</p> <p>See also simple large object, smart large object.</p>
LIST data type	<p>A collection data type in which elements are ordered and duplicates are allowed.</p> <p>See also collection data type.</p>
locale	<p>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.</p> <p>See also Global Language Support (GLS).</p>
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 A fundamental data type of a predefined fixed or variable length whose internal structure is not accessible through SQL statements. Opaque data types are created with the SQL statement CREATE OPAQUE TYPE. Support functions must always be defined for opaque types.

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](#).

scroll cursor A cursor that can fetch the next row or any prior row, so it can 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	<p>A large object that is stored in a blob space, is not recoverable, and does not obey transaction isolation modes. Simple large objects include TEXT and BYTE data types.</p> <p>See also TEXT, BYTE.</p>
smart large object	<p>A large object that:</p> <ul style="list-style-type: none"> ■ 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. <p>Smart large objects include CLOB and BLOB data types.</p>
sqlhosts file	<p>An Informix file containing information that lets a client application find and connect to an Informix database server anywhere on the network. You must supply one line for each type of connection to each database server.</p>
SQLSTATE	<p>A variable that contains status values about the outcome of SQL statements.</p>
support functions	<p>The functions that the database server automatically invokes to process a data type.</p> <p>The database server uses a support function to perform operations (such as converting to and from the internal, external, and binary representations of the type) on opaque data types.</p> <p>An index access method uses a support function in an operator class to perform operations (such as building or searching) on an index.</p>
sysmaster database	<p>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. Informix recommends you do not modify this database.</p>
system catalog	<p>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.</p>

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 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.

Error Messages

- 79700 Method not supported
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
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. 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 "[Manipulating Informix INTERVAL Data Types](#)" on page 3-38 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 [“Manipulating Informix INTERVAL Data Types” on page 3-38](#) for correct values.

-79708 Can't take null input

The string you have provided is null. 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 timestamp format was passed to a JDBC escape clause. The escape clause syntax for timestamp 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 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 [“Manipulating Informix INTERVAL Data Types” on page 3-38](#) for correct values.
- 79718 Invalid qualifier start code
- The start value for an **Interval** object is incorrect. See [“Manipulating Informix INTERVAL Data Types” on page 3-38](#) for correct values.
- 79719 Invalid qualifier end code
- The end value for an **Interval** object is incorrect. See [“Manipulating Informix INTERVAL Data Types” on page 3-38](#) for correct values.
- 79720 Invalid qualifier start or end code
- The start or end value for an **Interval** object is incorrect. See [“Manipulating Informix INTERVAL Data Types” on page 3-38](#) for correct values.
- 79721 Invalid interval string
- An error occurred during conversion of a **String** value to an **IntervalDF** or **IntervalYM** object. Check [“Manipulating Informix INTERVAL Data Types” on page 3-38](#) 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 [“Manipulating Informix INTERVAL Data Types” on page 3-38](#) 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 [“Manipulating Informix INTERVAL Data Types” on page 3-38](#) 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 [“Manipulating Informix INTERVAL Data Types” on page 3-38](#) 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 [“Manipulating Informix INTERVAL Data Types” on page 3-38](#) 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 are 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()** 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 Informix 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 ["Mapping Data Types" on page 3-66](#) 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 ["Mapping Data Types" on page 3-66](#) 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 non-logging 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 non-logging 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
- 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.executeQuery()** 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.setFetchDirection(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 insensitive), as in the following example:
- ```
URL: jdbc:informix-sqli://mymachine:1234/mydatabase:user=me:password=secret
```
- 79757 Invalid sub-protocol
- The current valid subprotocol supported by Informix 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 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:
- ```
URL: 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 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.

-79763 Only CONCUR_READ_ONLY is supported

Informix JDBC Driver supports only the **ResultSet.CONCUR_READ_ONLY** method. You can only call the **Connection.createStatement(int, int)**, **Connection.prepareStatement(String, int, int)**, or **Connection.CallableStatement(String, int, int)** method with a result set concurrency value of CONCUR_READ_ONLY.

- 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. Verify your `SQLData` class or `getSQLTypeName()`
You have asked for more data than is available. Check your **SQLData** class to make sure it matches what is in the database schema. The **SQLTypeName** object 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, 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.
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 UDT's, 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 MULTISSET 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 like **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 [“Internationalization” on page 2-31](#) for valid code sets.
- 79784 Locale not supported
The locale entered in the **DB_LOCALE** or **CLIENT_LOCALE** variable is not valid. Check [“Internationalization” on page 2-31](#) 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 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 returns 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 representation 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 (1-31) 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 (1-12) 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.

Index

A

Accessing a database remotely 2-49
 Anonymous search of sqlhosts
 information 2-17
 ANSI compliance
 level Intro-10
 APPLET tag 1-16
 Applets
 and database access 2-49
 unsigned, features unavailable
 for 1-16
 using Informix JDBC Driver
 in 1-15
 ARCHIVE attribute of APPLET
 tag 1-16
 Array class 3-50
 ArrayList class 3-45
 Arrays 3-45, 3-49
 Autocommit 2-46
 AUTOFREE environment
 variable A-2
 autofree.java example
 program 4-8, A-2
 Automatically freeing the
 cursor 2-13, 2-47, 4-8

B

Batch updates to the database 2-22
 BatchUpdateException
 interface 2-23
 BatchUpdate.java example
 program 2-23, A-2
 Binary qualifiers for INTERVAL
 data types 3-39

BLOB data type
 caching 3-20, 3-33, 4-6
 examples of
 column insertion 3-34
 creation 3-33
 data retrieval 3-35
 extensions for 3-25
 Blob interface 3-32
 Boldface type Intro-6
 BOOLEAN data type 3-69
 Browsers 1-15
 BYTE data type
 caching 4-6
 examples for
 data inserts and updates 3-21
 data retrieval 3-23
 extensions for 3-20
 ByteType.java example
 program 3-22, 3-24, A-2

C

Caching large objects 4-6
 CallableStatement interface 2-22
 cancelRowUpdate() method 2-27
 cancel() method 2-27
 Catalogs
 Informix JDBC Driver
 interpretation 2-22
 systables 2-21, 2-41, 2-43
 charattrUDT.java example
 program A-4
 Classes
 Array 3-50
 ArrayList 3-45
 extensibleobject 2-16

- HashSet 3-45, 3-46
- IfmxStatement 2-48
- IfxBlob 3-32
- IfxClob 3-32
- IfxDriver 2-4
- IfxJDBCProxy 2-49, 2-50
- IfxLobDescriptor 3-26
- IfxSmartBlob 3-27
- IfxTypes 3-73
- Interval 3-39
- IntervalDF 3-43
- IntervalYM 3-40
- Java.Socket 2-19
- Locale 2-31
- Message 2-30
- Properties 2-11
- ResultSet 2-36, 2-38
- SessionMgr 2-49
- SQLException 2-30, 3-76, 3-79
- SqlhDelete 2-19
- SqlhUpload 2-19
- TreeSet 3-48
- Version 2-48
- ClassGenerator utility 1-7, 3-63, A-1, A-2
- CLASSPATH environment variable 1-14, 3-63
- Class.forName() method 2-4
- Client hosts
 - specifying the locale of 2-32
- CLIENT_LOCALE environment variable 2-12, 2-32, 2-39
- CLOB data type
 - caching 3-20, 3-33, 4-6
 - examples of
 - column insertion 3-34
 - creation 3-33
 - data retrieval 3-35
 - extensions for 3-25
- Clob interface 3-32
- close() method 2-24, 2-25
- Code sets
 - conversion of 2-40, 2-45
 - synchronizing with locales 2-31
 - table of 2-41
- Collection data type
 - examples of
 - using the array interface 3-49

- using the collection interface 3-46
 - extensions for 3-45
 - in named and unnamed rows 3-53
- Collection interface 3-45
- Comment icons Intro-7
- Compliance with industry standards Intro-10
- Concurrency
 - and multiple threads 2-25
 - of scroll cursors 2-24
- Connection interface 2-25, 2-46
- Connections
 - creating 2-4, 2-8
 - establishing 2-3
 - to a database with non-ASCII characters 2-44
- Constructors
 - IntervalDF() 3-43
 - IntervalYM() 3-41
- Contents of Informix JDBC Driver 1-7
- CORBA 2-50
- CreateDB.java example program A-2
- createDB.java example program A-4, A-5
- createTypes.java example program A-4
- Creating a connection 2-4, 2-8
- CSM option 2-20
- Cursors
 - automatically freeing 2-13, 2-47, 4-8
 - scroll 2-23

D

- Data types
 - BLOB 4-6
 - BOOLEAN 3-69
 - BYTE 3-20, 4-6
 - CLOB 4-6
 - collection 3-45
 - DataBlade API 3-8
 - DATETIME 3-67
 - distinct 3-16

- INTERVAL 3-38
- LVARCHAR 3-68, 3-78
- mapping between Informix and JDBC API 3-66
- named row 3-50
- opaque 3-5
- SERIAL 3-37
- SERIAL8 3-37
- TEXT 3-20, 4-6
- unnamed row 3-50
- Database server name, setting in database URLs 2-8
- DatabaseMetaData interface 2-21, 2-48
- Databases
 - batch updates of 2-22
 - names of, setting in database URLs 2-7
 - querying 2-22
 - remote access options 2-49
 - specifying the locale of 2-32
 - URL 2-5, 2-6
 - with non-ASCII characters 2-44
- DataBlade API data types 3-8
- Dates
 - DBDATE formats of 2-36
 - formatting directives for 2-33
 - four-digit year expansion 2-39
 - GL_DATE formats of 2-33
 - inserting values 2-35, 2-37
 - native SQL formats of 2-35, 2-37
 - nonnative SQL formats of 2-35, 2-37
 - precedence rules for end-user formats 2-39
 - represented by strings 2-36
 - retrieving values 2-36, 2-38
 - string-to-date conversion 2-39
 - support for end-user formats 2-33
- DATETIME type 3-67
- DBANSIWARN environment variable 2-12
- DBCENTURY environment variable 2-12, 2-39
- DBCENTURYSelect.java example program A-2
- DBCConnection.java example program 2-9, A-2

DBDATE environment
 variable 2-12, 2-36, 2-39
 DBDATESelect.java example
 program A-2
 DBMetaData.java example
 program A-2
 DBSPACETEMP environment
 variable 2-12
 DBUPSPACE environment
 variable 2-12
 DB_LOCALE environment
 variable 2-12, 2-32, 2-40
 Deallocating resources 2-24
 Debugging 4-3
 Default locale Intro-5
 deleteRow() method 2-27
 DELIMIDENT environment
 variable 2-12
 demo1.java example program A-4
 demo2.java example program A-4
 demo3.java example program A-4
 demo4.java example program A-4
 demo5.java example program A-4
 demo6.java example program A-4
 demo_11.java example
 program A-4
 demo_13.java example
 program A-4
 demo_14.java example
 program A-4
 Directives, formatting, for
 dates 2-33
 Distinct data type
 examples for
 inserting data 3-17
 retrieving data 3-19
 extensions for 3-16
 unsupported methods for 3-20
 distinct_d1.java example
 program A-4
 distinct_d2.java example
 program A-5
 Driver interface 2-48
 DriverManager interface 1-6, 2-4,
 2-8, 2-11
 DropDB.java example
 program A-2
 dropDB.java example
 program A-5, A-6

E

ENABLE_CACHE_TYPE
 environment variable 2-12, 3-9,
 3-17, 3-53
 Encryption of the database
 password 2-20
 End-user formats for dates
 precedence rules for 2-39
 support for 2-33
 Environment variables Intro-6
 AUTOFREE A-2
 CLASSPATH 1-14, 3-63
 CLIENT_LOCALE 2-12, 2-32,
 2-39
 DBANSIWARN 2-12
 DBCENTURY 2-12, 2-39
 DBDATE 2-12, 2-36, 2-39
 DBSPACETEMP 2-12
 DBUPSPACE 2-12
 DB_LOCALE 2-12, 2-32, 2-40
 DELIMIDENT 2-12
 ENABLE_CACHE_TYPE 2-12,
 3-9, 3-17, 3-53
 FET_BUF_SIZE 2-13, 4-5, A-3
 GL_DATE 2-13, 2-33, 2-39
 IFX_AUTOFREE 2-13, 4-7
 INFORMIXCONRETRY 2-13
 INFORMIXCONTIME 2-13
 INFORMIXOPCACHE 2-13
 INFORMIXSERVER 2-8, 2-13
 INFORMIXSTACKSIZE 2-13
 JDBCTEMP 2-14
 LOBCACHE 2-14, 3-20, 3-33, 4-6
 NODEFDAC 2-14
 OPTCOMPIND 2-14
 OPTOFC 2-15, 4-7, A-3
 PATH 2-15
 PDQPRIORITY 2-15
 PLCONFIG 2-15
 PROTOCOLTRACE 4-4
 PROTOCOLTRACEFILE 4-4
 PSORT_DBTEMP 2-15
 PSORT_NPROCS 2-15
 SECURITY 2-16, 2-20
 specifying 2-8, 2-10
 SQLH_TYPE 2-16
 supported 2-12
 TRACE 4-4

TRACEFILE 4-4
 USEV5SERVER 2-16
 en_us.8859-1 locale Intro-5
 equals() method 3-42, 3-44
 Errors 2-28, 2-29, 2-30
 Escape syntax 2-26
 Establishing a connection 2-3
 Examples
 autofree.java 4-8, A-2
 BatchUpdate.java 2-23, A-2
 BLOB and CLOB data types
 column insertion 3-34
 creation 3-33
 data retrieval 3-35
 BYTE and TEXT data types 3-21,
 3-23
 ByteType.java 3-22, 3-24, A-2
 charattrUDT.java A-4
 collection data types
 using the array interface 3-49
 using the collection
 interface 3-46
 CreateDB.java A-2
 createDB.java A-4, A-5
 createTypes.java A-4
 DBCENTURYSelect.java A-2
 DBConnection.java 2-9, A-2
 DBDATESelect.java A-2
 DBMetaData.java A-2
 demo1.java A-4
 demo2.java A-4
 demo3.java A-4
 demo4.java A-4
 demo5.java A-4
 demo6.java A-4
 demo_11.java A-4
 demo_13.java A-4
 demo_14.java A-4
 distinct data types
 inserting data 3-17
 retrieving data 3-19
 distinct_d1.java A-4
 distinct_d2.java A-5
 DropDB.java A-2
 dropDB.java A-5, A-6
 GenericStruct.java A-5
 GLDATESelect.java A-3
 Intervaldemo.java 3-45, A-3
 largebinUDT.java A-5

list1.java A-5
 list2.java A-5
 LOCALESelect.java A-3
 MultiRowCall.java A-3
 myMoney.java A-5
 named and unnamed rows
 creating a Struct class for 3-61
 using the SQLData interface for
 a named row 3-54
 using the Struct interface 3-58
 opaque data types
 inserting data 3-10
 large objects 3-13
 retrieving data 3-13
 OptimizedSelect.java A-3
 optofc.java 2-11, 4-8, A-3
 PropertyConnection.java A-3
 r1_t.java A-5
 r2_t.java A-5
 row1.java A-5
 row2.java A-5
 row3.java A-6
 RSMetaData.java A-3
 ScrollCursor.java 2-24, A-3
 Serial.java A-3
 SimpleCall.java A-3
 SimpleConnection.java A-3
 SimpleSelect.java A-3
 TextConv.java A-3
 TextType.java 3-23, 3-25, A-3
 udt_d1.java A-5
 udt_d2.java A-5
 executeBatch() method 2-23
 executeQuery() method 2-25, 2-47
 executeUpdate() method 2-9, 3-23
 execute() method 2-24, 2-28
 extensibleobject class 2-16

F

FET_BUF_SIZE environment
 variable 2-13, 4-5, A-3
 File interface 3-23
 FileInputStream interface 3-23
 Files
 IfxJDBCProxy.class 1-7, 2-49
 java.io 2-31
 java.security 2-21

java.text 2-31
 java.util 2-31
 mitypes.h 3-8
 SessionMgr.class 1-8, 2-49
 setup.std 3-63
 sqlhosts 2-16
 Firewalls and database access 2-49
 Formatting directives for
 dates 2-33
 fromString() method 3-42, 3-44

G

GenericStruct.java example
 program A-5
 getArray() method 3-45, 3-50
 getAsciiStream() method 3-36
 getAttributes() method 3-60
 getAutoFree() method 2-47, 4-8
 getBinaryStream() method 3-36
 getBlob() method 3-36
 getBytes() method 2-45, 3-36
 getCatalogName() method 2-28
 getCatalogs() method 2-22
 getClob() method 3-36
 getConnection() method 2-4, 2-8,
 2-11
 getEndCode() method 3-40
 getErrorCode() method 2-29
 getFetchSize() method 2-27
 getFieldName() method 3-40
 getIfxTypeName() method 3-40
 getJDBCVersion() method 2-48
 getLength() method 3-40
 getMajorVersion() method 2-48
 getMessage() method 2-29
 getMinorVersion() method 2-48
 getMonths() method 3-42
 getNanoSeconds() method 3-44
 getObject() method 3-45, 3-51, 3-52
 getQualifier() method 3-40
 getRef() method 2-27
 getScale() method 3-40
 getSchemaName() method 2-28
 getSchemas() method 2-21
 getSeconds() method 3-44
 getSerial80() method 3-37
 getSerial() method 3-37

getSQLTypeName() method 3-51,
 3-52, 3-57, 3-60, 3-62
 getStartCode() method 3-40
 getString() method 2-36, 2-38, 2-45,
 3-36
 getTableName() method 2-28
 getText() method 2-44
 getTypeMap() method 3-51, 3-56,
 3-57
 getUnicodeStream() method 2-27
 getUpdateCounts() method 2-23
 getXXX() method 2-26, 3-78
 GLDATESelect.java example
 program A-3
 Global Language Support
 (GLS) Intro-5, 2-31
 GL_DATE environment
 variable 2-13, 2-33, 2-39
 greaterThan() method 3-42, 3-44

H

HashSet class 3-45, 3-46
 Host names, setting in database
 URLs 2-7
 HTTP proxy 2-49

I

Icons
 Important Intro-7
 platform Intro-7
 Tip Intro-7
 Warning Intro-7
 IfmxComplexSQLInput.readObject
 () method 3-53
 IfmxComplexSQLOutput.writeObj
 ect() method 3-53
 IfmxStatement class 2-48
 IfmxUdtSQLInput interface 3-6
 IfmxUdtSQLOutput interface 3-7
 IfxBlob class 3-32
 IfxCblob class 3-32
 IfxDriver class 2-4
 ifxjdbc-g.jar 4-3
 ifxjdbc-g.jar file 1-7, 1-15
 IfxJDBCProxy class 2-49, 2-50
 IfxJDBCProxy.class file 1-7, 2-49

ifxjdbc.jar file 1-7, 1-15
 IfxLobDescriptor class 3-26
 IfxLocator object 3-26
 IfxLoClose() method 3-28
 IfxLoCreate() method 3-27
 IfxLoOpen() method 3-27
 IfxLoRead() method 3-29
 IfxLoRelease() method 3-28
 IfxLoSeek() method 3-28
 IfxLoSize() method 3-28
 IfxLoTruncate() method 3-28
 IfxLoWrite() method 3-29
 IfxSetObject() method 3-72
 IfxSmartBlob class 3-27
 ifxtools-g.jar file 1-7
 ifxtools.jar file 1-7, 3-63
 IfxTypes class 3-73
 IFX_AUTOFREE environment variable 2-13, 4-7
 Important paragraphs, icon for Intro-7
 Industry standards, compliance with Intro-10
 Informix base distinguished name 2-19
 Informix JDBC Driver contents of 1-7
 installing interactively 1-8
 installing silently 1-10
 loading 2-4
 overview of 1-6
 registering 2-4
 tracing 4-4
 uninstalling 1-13
 using debug version of 4-3
 using in an applet 1-15, 2-4
 using in an application 1-14
 INFORMIXCONRETRY environment variable 2-13
 INFORMIXCONTIME environment variable 2-13
 INFORMIXOPCACHE environment variable 2-13
 INFORMIXSERVER environment variable 2-8, 2-13
 INFORMIXSTACKSIZE environment variable 2-13
 InputStream interface 3-21

InputStreamReader() method 2-44, 2-45
 Inserting DATE values 2-35
 Inserting date values 2-37
 insertRow() method 2-27
 Installing Informix JDBC Driver 1-8, 1-10
 Interfaces
 BatchUpdateException 2-23
 Blob 3-32
 CallableStatement 2-22
 Clob 3-32
 Collection 3-45
 Connection 2-25, 2-46
 DatabaseMetaData 2-21, 2-48
 Driver 2-48
 DriverManager 1-6, 2-4, 2-8, 2-11
 File 3-23
 FileInputStream 3-23
 IfmxUdtSQLInput 3-6
 IfmxUdtSQLOutput 3-7
 InputStream 3-21
 List 3-45
 PreparedStatement 2-22, 2-23, 2-25, 3-69 to 3-78
 ResultSet 2-22, 2-24, 2-26, 3-78 to 3-80, 4-7
 ResultSetMetaData 2-22
 Set 3-45
 SQLData 3-7, 3-51, 3-57, 3-63
 SQLInput 3-56
 Statement 2-9, 2-22, 2-23, 4-7
 Struct 3-51, 3-52
 Types 3-37, 3-66
 Internationalization 2-31 to 2-46
 Interval class 3-39
 INTERVAL data type
 binary qualifiers for 3-39
 extensions for 3-38
 in named and unnamed rows 3-53
 Intervaldemo.java example program 3-45, A-3
 IntervalDF class 3-43
 IntervalDF() constructor 3-43
 IntervalYM class 3-40
 IntervalYM() constructor 3-41
 IP address, setting in database URL 2-7

isDefinitelyWriteable() method 2-28
 ISO 8859-1 code set Intro-5
 isReadOnly() method 2-27, 2-28
 isWriteable() method 2-28

J

JAR files
 for JNDI 2-16
 for LDAP SPI 2-16
 ifxjdbc-g.jar 1-7, 1-15, 4-3
 ifxjdbc.jar 1-7, 1-15
 ifxtools-g.jar 1-7
 ifxtools.jar 1-7, 3-63
 jar utility 1-15
 Java naming and directory interface (JNDI)
 and the sqlhosts file 2-16
 JAR files for 2-16
 Java virtual machine (JVM) 1-14
 javac, Java compiler 1-7
 JavaSoft 1-3, 1-15
 java.io file 2-31
 java.security file 2-21
 Java.Socket class 2-19
 java.text file 2-31
 java.util file 2-31
 JCE security package 2-20
 JDBC API 1-3, 1-4
 JDBC driver, general 1-6
 JDBCTEMP environment variable 2-14

L

largebinUDT.java example program A-5
 lessThan() method 3-42, 3-44
 Lightweight directory access protocol (LDAP) server administration requirements for 2-18
 and the sqlhosts file 2-16
 and unsigned applets 1-16
 JAR files for 2-16
 loader for 1-7
 URL syntax for 2-17

- utilities for 2-18
- version requirement 2-16
- List interface 3-45
- list1.java example program A-5
- list2.java example program A-5
- Loading Informix JDBC Driver 2-4
- LOBCACHE environment
 - variable 2-14, 3-20, 3-33, 4-6
- Locale class 2-31
- Locales
 - assumptions about Intro-5
 - client, specifying 2-32
 - database, specifying 2-32
 - synchronizing with code sets 2-31
 - table of 2-43
- LOCALESelect.java example
 - program A-3
- Localization 2-31
- Locator object 3-26
- LVARCHAR data type 3-68, 3-78

M

Mapping

- between Informix and JDBC API
 - data types 3-66
 - opaque data types 3-7
- map.put() method 3-56, 3-57
- Message class 2-30
- Metadata, accessing database 2-21

Methods

- cancelRowUpdate() 2-27
- cancel() 2-27
- Class.forName() 2-4
- close() 2-24, 2-25
- deleteRow() 2-27
- equals() 3-42, 3-44
- executeBatch() 2-23
- executeQuery() 2-25, 2-47
- executeUpdate() 2-9, 3-23
- execute() 2-24, 2-28
- fromString() 3-42, 3-44
- getArray() 3-45, 3-50
- getAsciiStream() 3-36
- getAttributes() 3-60
- getAutoFree() 2-47, 4-8
- getBinaryStream() 3-36
- getBlob() 3-36

- getBytes() 2-45, 3-36
- getCatalogName() 2-28
- getCatalogs() 2-22
- getClob() 3-36
- getConnection() 2-4, 2-8, 2-11
- getEndCode() 3-40
- getErrorCode() 2-29
- getFetchSize() 2-27
- getFieldName() 3-40
- getIxfTypeName() 3-40
- getJDBCVersion() 2-48
- getLength() 3-40
- getMajorVersion() 2-48
- getMessage() 2-29
- getMinorVersion() 2-48
- getMonths() 3-42
- getNanoSeconds() 3-44
- getObject() 3-45, 3-51, 3-52
- getQualifier() 3-40
- getRef() 2-27
- getScale() 3-40
- getSchemaName() 2-28
- getSchemas() 2-21
- getSeconds() 3-44
- getSerial8() 3-37
- getSerial() 3-37
- getSQLTypeName() 3-51, 3-52, 3-57, 3-60, 3-62
- getStartCode() 3-40
- getString() 2-36, 2-38, 2-45, 3-36
- getTableName() 2-28
- getText() 2-44
- getTypeMap() 3-51, 3-56, 3-57
- getUnicodeStream() 2-27
- getUpdateCounts() 2-23
- getXXX() 2-26, 3-78
- greaterThan() 3-42, 3-44
- IxfmxComplexSQLInput.readObject() 3-53
- IxfmxComplexSQLOutput.writeObject() 3-53
- IxfLoClose() 3-28
- IxfLoCreate() 3-27
- IxfLoOpen() 3-27
- IxfLoRead() 3-29
- IxfLoRelease() 3-28
- IxfLoSeek() 3-28
- IxfLoSize() 3-28
- IxfLoTruncate() 3-28

- IxfLoWrite() 3-29
- IxfSetObject() 3-72
- InputStreamReader() 2-44, 2-45
- insertRow() 2-27
- isDefinitelyWritable() 2-28
- isReadOnly() 2-27, 2-28
- isWritable() 2-28
- lessThan() 3-42, 3-44
- map.put() 3-56, 3-57
- moveToCurrentRow() 2-27
- moveToInsertRow() 2-27
- next() 2-26, 3-23
- OutputStreamWriter() 2-44, 2-45
- prepareStatement() 2-25
- put() 2-11
- readAsciiStream() 3-16
- readBinaryStream() 3-16
- readBytes() 3-16
- readByte() 3-65
- readCharacterStream() 3-16, 3-20, 3-65
- readObject() 3-16
- readRef() 3-16, 3-20, 3-65
- readSQL() 3-7, 3-51, 3-56, 3-63
- readString() 3-16
- refreshRow() 2-27
- registerDriver() 2-4
- registerOutParameter() 2-27
- rowDeleted() 2-27
- rowInserted() 2-27
- rowUpdated() 2-27
- setArray() 3-45, 3-70
- setAsciiStream() 3-21, 3-70
- setAutoCommit() 2-46
- setAutoFree() 2-47, 4-8
- setBigDecimal() 3-70
- setBinaryStream() 3-21, 3-71
- setBlob() 3-71
- setBoolean() 3-71
- setBytes() 3-71
- setByte() 3-71
- setCatalog() 2-27
- setCharacterStream() 3-71
- setClob() 3-71
- setDate() 3-71
- setDouble() 3-71
- setFetchSize() 2-27
- setFloat() 3-71
- setInt() 2-25, 3-71

setLong() 3-71
 setMaxFieldSize() 2-27
 setNull() 3-72
 setObject() 3-45, 3-52, 3-57, 3-62
 setQualifier() 3-42, 3-44
 setQueryTimeout() 2-27
 setReadOnly() 2-27
 setRef() 2-27
 setShort() 3-72
 setString() 3-72
 setTimestamp() 3-72
 setTime() 3-72
 setTypeMap() 3-45
 setUnicodeStream() 2-27
 setXXX() 3-70, 3-75
 set() 3-42, 3-44
 SQLInput() 3-5, 3-53
 SQLOutput() 3-5, 3-53
 toString() 3-42, 3-44
 unsupported
 for distinct data types 3-20
 for named rows 3-65
 for opaque data types 3-16
 for querying the database 2-27
 updateRow() 2-27
 updateXXX() 2-27
 writeAsciiStream() 3-16
 writeBinaryStream() 3-16
 writeBytes() 3-16
 writeByte() 3-65
 writeCharacterStream() 3-16,
 3-20, 3-65
 writeInt() 3-58
 writeObject() 3-16, 3-58
 writeRef() 3-16, 3-20, 3-65
 WriteSQL() 3-7
 writeSQL() 3-51, 3-58, 3-63
 writeString() 3-16
 writeXXX() 3-58
 mitypes.h file 3-8
 moveToCurrentRow()
 method 2-27
 moveToInsertRow() method 2-27
 MultiRowCall.java example
 program A-3
 myMoney.java example
 program A-5

N

Named row data type
 caching type information 3-9,
 3-17, 3-53
 examples of
 creating a Struct class for 3-61
 using the SQLData
 interface 3-54
 using the Struct interface 3-58
 extensions for 3-50
 generating using the
 ClassGenerator utility 3-63
 intervals and collections in 3-53
 opaque data type columns in 3-51
 unsupported methods for 3-65
 using the SQLData interface
 for 3-51
 using the Struct interface for 3-52
 Name-value pairs of database
 URL 2-8
 Native SQL date formats 2-35, 2-37
 next() method 2-26, 3-23
 NODEFDAC environment
 variable 2-14
 Nonnative SQL date formats 2-35,
 2-37

O

Objects
 IfxLocator 3-26
 Locator 3-26
 ODBC 1-6
 Opaque data type
 examples of
 inserting data 3-10
 large objects 3-13
 retrieving data 3-13
 extensions for 3-5
 mappings for 3-7
 unsupported methods 3-16
 OPTCOMPIND environment
 variable 2-14
 OptimizedSelect.java example
 program A-3
 Options
 CSM 2-20

SERVERNAME 2-20
 SPWDCSM 2-20
 OPTOFC environment
 variable 2-15, 4-7, A-3
 optofc.java example program 2-11,
 4-8, A-3
 OutputStreamWriter()
 method 2-44, 2-45

P

Passwords
 configuring the server for 2-20
 encryption of 2-20
 JCE security package for 2-20
 URL syntax of 2-8, 2-20
 versions of the server
 supporting 2-20
 PATH environment variable 2-15
 PDQPRIORITY environment
 variable 2-15
 Performance 4-5
 Platform icons Intro-7
 PLCONFIG environment
 variable 2-15
 Port numbers
 setting in database URL 2-7
 setting in sqlhosts file or LDAP
 server 2-17
 Precedence rules for date
 formats 2-39
 PREPARE statements, executing
 multiple 2-22
 PreparedStatement interface 2-22,
 2-23, 2-25, 3-69 to 3-78
 prepareStatement() method 2-25
 Properties class 2-11
 Property lists 2-11
 PropertyConnection.java example
 program A-3
 PROTOCOLTRACE environment
 variable 4-4
 PROTOCOLTRACEFILE
 environment variable 4-4
 Proxy server 2-49
 PSORT_DBTEMP environment
 variable 2-15

PSORT_NPROCS environment variable 2-15
 put() method 2-11

Q

Qualifiers, binary, for INTERVAL data types 3-39
 Querying the database 2-22

R

r1_t.java example program A-5
 r2_t.java example program A-5
 readAsciiStream() method 3-16
 readBinaryStream() method 3-16
 readBytes() method 3-16
 readByte() method 3-65
 readCharacterStream() method 3-16, 3-20, 3-65
 readObject() method 3-16
 readRef() method 3-16, 3-20, 3-65
 readSQL() method 3-7, 3-51, 3-56, 3-63
 readString() method 3-16
 Ref type 3-67
 refreshRow() method 2-27
 registerDriver() method 2-4
 Registering Informix JDBC Driver 2-4
 registerOutParameter() method 2-27
 Relative distinguished name (RDN) 2-19
 Remote database access 2-49
 Remote method invocation (RMI) 2-50, A-1, A-2
 ResultSet class 2-36, 2-38
 ResultSet interface 2-22, 2-24, 2-26, 3-78 to 3-80, 4-7
 ResultSetMetaData interface 2-22
 Retrieving
 database names 2-22
 date values 2-36, 2-38
 Informix error message text 2-30
 user names 2-21
 version information 2-48
 RMI 2-50, A-1, A-2

row1.java example program A-5
 row2.java example program A-5
 row3.java example program A-6
 rowDeleted() method 2-27
 rowInserted() method 2-27
 rowUpdated() method 2-27
 RSMetaData.java example program A-3

S

Schemas, Informix JDBC Driver interpretation 2-21
 Scroll cursors 2-23
 ScrollCursor.java example program 2-24, A-3
 Search, anonymous, of sqlhosts information 2-17
 SECURITY environment variable 2-16, 2-20
 Security, JCE package for 2-20
 SERIAL data type 3-37
 SERIAL8 data type 3-37
 Serial.java example program A-3
 SERVERNAME option 2-20
 Service provider interface (SPI) 2-16
 Servlets 2-49
 SessionMgr class 2-49
 SessionMgr.class file 1-8, 2-49
 Set interface 3-45
 setArray() method 3-45, 3-70
 setAsciiStream() method 3-21, 3-70
 setAutoCommit() method 2-46
 setAutoFree() method 2-47, 4-8
 setBigDecimal() method 3-70
 setBinaryStream() method 3-21, 3-71
 setBlob() method 3-71
 setBoolean() method 3-71
 setBytes() method 3-71
 setByte() method 3-71
 setCatalog() method 2-27
 setCharacterStream() method 3-71
 setClob() method 3-71
 setDate() method 3-71
 setDouble() method 3-71
 setFetchSize() method 2-27

setFloat() method 3-71
 setInt() method 2-25, 3-71
 setLong() method 3-71
 setMaxFieldSize() method 2-27
 setNull() method 3-72
 setObject() method 3-45, 3-52, 3-57, 3-62
 setQualifier() method 3-42, 3-44
 setQueryTimeout() method 2-27
 setReadOnly() method 2-27
 setRef() method 2-27
 setShort() method 3-72
 setString() method 3-72
 setTimestamp() method 3-72
 setTime() method 3-72
 Setting
 autocommit 2-46
 properties 2-11
 the CLASSPATH environment variable 1-14, 1-15
 setTypeMap() method 3-45
 setUnicodeStream() method 2-27
 setup.class class file 1-7, 1-8, 1-10, 1-11, 1-12
 setup.std file 3-63
 setXXX() method 3-70, 3-75
 set() method 3-42, 3-44
 SimpleCall.java example program A-3
 SimpleConnection.java example program A-3
 SimpleSelect.java example program A-3
 Specifying
 environment variables 2-8, 2-10
 the client locale 2-32
 the database locale 2-32
 SPWDCSM option 2-20
 SQL date formats
 native 2-35, 2-37
 nonnative 2-35, 2-37
 SQLData interface 3-7, 3-51, 3-57, 3-63
 SQLException class 2-30, 3-76, 3-79
 SqlhDelete utility 2-19
 sqlhosts file
 administration requirements for 2-18
 and password support 2-20

- and unsigned applets 1-16
- reading 2-16
- URL syntax for 2-17
- utilities for 2-18
- SqlhUpload utility 2-19
- SQLH_TYPE environment variable 2-16
- SQLInput interface 3-56
- SQLInput() method 3-5, 3-53
- SQLOutput() method 3-5, 3-53
- Statement interface 2-9, 2-22, 2-23, 4-7
- Strings, representing dates using 2-36
- Struct interface 3-51, 3-52
- Structured type (Struct) 3-50
- Supported environment variables 2-12
- Syntax of database URLs 2-6
- sysmaster database 2-21
- systables catalog and code set conversion 2-41, 2-43
- and metadata 2-21

T

- TEXT data type
 - caching 4-6
 - code set conversion for 2-45
 - examples for
 - data inserts and updates 3-21
 - data retrieval 3-23
 - extensions for 3-20
- TextConv.java example program A-3
- TextType.java example program 3-23, 3-25, A-3
- Threads, multiple, and concurrency 2-25
- Tip icons Intro-7
- toString() method 3-42, 3-44
- TRACE environment variable 4-4
- TRACEFILE environment variable 4-4
- Tracing 4-4
- Transactions, handling 2-46
- TreeSet class 3-48
- Tuple buffer 2-13, 4-5
- TU_DAY variable 3-39
- TU_F1 variable 3-39
- TU_F2 variable 3-39
- TU_F3 variable 3-39
- TU_F4 variable 3-39
- TU_F5 variable 3-39
- TU_FRAC variable 3-39
- TU_HOUR variable 3-39
- TU_MINUTE variable 3-39
- TU_MONTH variable 3-39
- TU_SECOND variable 3-39
- TU_YEAR variable 3-39
- Types interface 3-37, 3-66

U

- udt_d1.java example program A-5
- udt_d2.java example program A-5
- Unicode
 - and internationalization APIs 2-31
 - and the client code set 2-44
 - and the database code set 2-41
- Uninstalling Informix JDBC Driver 1-13
- Unnamed row data type
 - examples of
 - creating a Struct class for 3-61
 - using the Struct interface 3-58
 - extensions for 3-50
 - intervals and collections in 3-53
 - using the Struct interface for 3-52
- Unsupported methods
 - for distinct data types 3-20
 - for named rows 3-65
 - for opaque data types 3-16
 - for querying the database 2-27
- updateRow() method 2-27
- Updates, batch 2-22
- updateXXX() method 2-27
- URLs
 - database 2-5, 2-6
 - for a proxy server 2-49
 - syntax for LDAP server and sqlhosts file 2-17
- User names, setting in database URLs 2-8

- User-defined routine 3-53
- USEV5SERVER environment variable 2-16
- Using debug version of Informix JDBC Driver 4-3
- Using Informix JDBC Driver
 - in an applet 1-15, 2-4
 - in an application 1-14
- Using INFORMIX-OnLine 5.x database servers 2-16
- Using INFORMIX-SE 5.x database servers 2-16
- Utilities
 - ClassGenerator 1-7, 3-63
 - jar 1-15
 - SqlhDelete 2-19
 - SqlhUpload 2-19

V

- Variables for binary qualifiers 3-39
- Version class 2-48
- Version, of Informix JDBC Driver 2-48

W

- Warning icons Intro-7
- writeAsciiStream() method 3-16
- writeBinaryStream() method 3-16
- writeBytes() method 3-16
- writeByte() method 3-65
- writeCharacterStream()
 - method 3-16, 3-20, 3-65
- writeInt() method 3-58
- writeObject() method 3-16, 3-58
- writeRef() method 3-16, 3-20, 3-65
- WriteSQL() method 3-7
- writeSQL() method 3-51, 3-58, 3-63
- writeString() method 3-16
- writeXXX() method 3-58

X

- X/Open compliance level Intro-10

