DB2 IBM Informix

IBM

Version 2.90

IBM Informix Embedded SQLJ User's Guide

G251-2278-00

DB2 IBM Informix

IBM

Version 2.90

IBM Informix Embedded SQLJ User's Guide

G251-2278-00

- Chapter 1, “Introducing IBM Informix Embedded SQLJ,” on page 1-1, introduces the IBM Informix Embedded SQLJ product.
- Chapter 2, “Preparing to Use Embedded SQLJ,” on page 2-1, describes the software you need to develop and run Embedded SQLJ programs and how to set it up.
- Chapter 3, “Building an Embedded SQLJ Program,” on page 3-1, provides an overview of the Embedded SQLJ language and demonstrates its use with a simple program.
- Chapter 4, “The Embedded SQLJ Language,” on page 4-1, provides detailed information about the Embedded SQLJ language.
- Chapter 5, “Processing Embedded SQLJ Source Code,” on page 5-1, explains how to use the SQLJ translator and how to compile and run your Embedded SQLJ programs.
- Appendix A, “Connecting to Databases,” on page A-1, provides background information and further details about how an Embedded SQLJ program connects to a database.
- Appendix B, “Sample Programs,” on page B-1, provides a table describing the sample programs included with Embedded SQLJ.
- An Accessibility appendix describes how to read syntax diagrams in the HTML version of this manual using a screen reader.
- A Notices appendix provides information about IBM Informix products.
- An index follows at the end of the manual.

Types of Users

This guide is for programmers who want to write JavaTM programs that can:

- Connect to Informix databases.
- Issue SQL statements to manipulate data in the database.

This manual is written with the assumption that you have the following background:

- A working knowledge of your computer, your operating system, and the utilities that your operating system provides
- Experience with the Java programming language
- Experience working with relational databases or exposure to database concepts
- Experience with the SQL query language

If you have limited experience with relational databases, SQL, or your operating system, refer to the *IBM Informix: Getting Started Guide* for your database server for a list of supplementary titles.

Software Dependencies

To run IBM Informix Embedded SQLJ programs, you must use one of the following database servers:

- IBM Informix Dynamic Server, Version 9.x and later
- IBM Informix Extended Parallel Server, Version 8.x
- IBM Informix Dynamic Server, Version 7.x
- IBM Informix Dynamic Server, Workgroup and Developer editions, Version 7.x
- IBM Informix OnLine Dynamic Server, Version 5.x
- IBM Informix SE, Versions 5.x to 7.2x

To enable your programs to connect to the server, you must use IBM Informix JDBC Driver, Version 2.0 or later.

You must use the JavaSoft software Java Development Kit (JDK), Version 1.2 or later, or any Java software compatible with JDK 1.2, to create your programs. JDK 1.2 is also known as *Java 2*.

Global Language Support

Refer to the *IBM Informix: JDBC Driver Programmer's Guide* for information about using Global Language Support (GLS) with IBM Informix JDBC Driver.

Documentation Conventions

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

The following conventions are discussed:

- Typographical conventions
- Other conventions
- Syntax diagrams
- Command-line conventions
- Example code conventions

Typographical Conventions

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

Convention	Meaning
KEYWORD	All primary elements in a programming language statement (keywords) appear in uppercase letters in a serif font.

Convention	Meaning
italics ***italics*** `italics`	Within text, new terms and emphasized words appear in italics. Within syntax and code examples, variable values that you are to specify appear in italics.
boldface ***boldface***	Names of program entities (such as classes, events, and tables), environment variables, file and pathnames, and interface elements (such as icons, menu items, and buttons) appear in boldface.
`monospace` *`monospace`*	Information that the product displays and information that you enter appear in a monospace typeface.
`KEYSTROKE`	Keys that you are to press appear in uppercase letters in a sans serif font.
>	This symbol indicates 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.

Feature, Product, and Platform

Feature, product, and platform markup identifies paragraphs that contain feature-specific, product-specific, or platform-specific information. Some

examples of this markup follow:

Dynamic Server

Identifies information that is specific to IBM Informix Dynamic Server

End of Dynamic Server

Extended Parallel Server

Identifies information that is specific to IBM Informix Extended Parallel Server

End of Extended Parallel Server

UNIX Only

Identifies information that is specific to UNIX platforms

End of UNIX Only

Windows Only

Identifies information that is specific to the Windows environment

End of Windows Only

This markup can apply to one or more paragraphs within a section. When an entire section applies to a particular product or platform, this is noted as part of the heading text, for example:

Table Sorting (Linux Only)

Syntax Diagrams

This guide uses syntax diagrams built with the following components to describe the syntax for statements and all commands other than system-level commands.

Note: Starting in 2004, syntax diagrams have been reformatted to conform to the IBM standard.

Syntax diagrams depicting SQL and command-line statements have changed in the following ways:

- The symbols at the beginning and end of statements are now double arrows instead of a vertical line at the end.
- The symbols at the beginning and end of syntax segment diagrams are now vertical lines instead of arrows.

- How many times a loop can be repeated is now explained in a diagram footnote instead of a number in a gate symbol.
- Syntax statements that are longer than one line now continue on the next line instead of looping down with a continuous line.
- Product or condition-specific paths are now explained in diagram footnotes instead of icons.

The following table describes syntax diagram components.

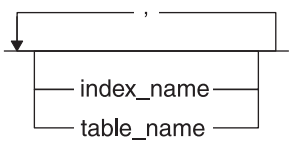
Component represented in PDF	Component represented in HTML	Meaning
▶▶————————	`>>----------`	Statement begins.
————————▶	`---------->`	Statement continues on next line.
▶————————	`>----------`	Statement continues from previous line.
————————▶◀	`----------><`	Statement ends.
———SELECT———	`--------SELECT----------`	Required item.
┬ └──LOCAL──┘	`--+-----------------+---` `  '------LOCAL------'`	Optional item.

┬
├──ALL──┤
├──DISTINCT──┤
└──UNIQUE──┘
	`---+-----ALL-------+---`
	`  +--DISTINCT-----+`
	`  '---UNIQUE------'`
		Required item with choice. One and only one item must be present.

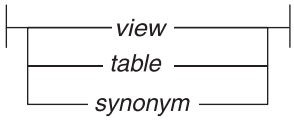
┬
├─FOR UPDATE─┤
└─FOR READ ONLY─┘
	`---+------------------+---`
	`  +--FOR UPDATE-----+`
	`  '--FOR READ ONLY--'`
		Optional items with choice are shown below the main line, one of which you might specify.

┬──NEXT──┬
├─────────┤
├──PRIOR──┤
└──PREVIOUS──┘
	`  .---NEXT---------.`
	`----+----------------+---`
	`  +---PRIOR--------+`
	`  '---PREVIOUS-----'`
		The values below the main line are optional, one of which you might specify. If you do not specify an item, the value above the line will be used as the default.

Component represented in PDF	Component represented in HTML	Meaning

	`.-------,-----------.`
	`V  |`
	`---+-----------------+---`
	`  +---index_name---+`
	`  '---table_name---'`
		Optional items. Several items are allowed; a comma must precede each repetition.

	`>>-| Table Reference |-><`	Reference to a syntax segment.

Table Reference
 
	`Table Reference`
	`|--+-----view--------+--|`
	`  +------table------+`
	`  '----synonym------'`
		Syntax segment.

How to Read a Command-Line Syntax Diagram

The following command-line syntax diagram uses some of the elements listed in the table in the previous section.

Creating a No-Conversion Job

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

►─-t─table──┬──────────────────────────────────────────────────────────────┬──►◄
            │                                                              │
            │                                                   (1)        │
            │  ┬──────────┬  ┬──────────┬  ┤ Setting the Run Mode ├        │
            │  └─-S─server─┘  └─-T─target─┘                                │
```

Notes:

1 See page 17-4

The second line in this diagram has a segment named "Setting the Run Mode," which according to the diagram footnote, is on page 17-4. This segment is shown in the following segment diagram (the diagram uses segment start and end components).

Setting the Run Mode:

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

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

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

1. Type **onpladm create job** and then the name of the job.
2. Optionally, type **-p** and then the name of the project.
3. Type the following required elements:
   - **-n**
   - **-d** and the name of the device
   - **-D** and the name of the database
   - **-t** and the name of the table
4. Optionally, you can choose one or more of the following elements and repeat them an arbitrary number of times:
   - **-S** and the server name
   - **-T** and the target server name
   - The run mode. To set the run mode, follow the Setting the Run Mode segment diagram to type **-f**, optionally type **d**, **p**, or **a**, and then optionally type **l** or **u**.
5. Follow the diagram to the terminator.

Your diagram is complete.

Keywords and Punctuation

Keywords are words reserved for statements and all commands except system-level commands. When a keyword appears in a syntax diagram, it is shown in uppercase letters. When you use a keyword in a command, you can write it in uppercase or lowercase letters, but you must spell the keyword exactly as it appears in the syntax diagram.

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

Identifiers and Names

Variables serve as placeholders for identifiers and names in the syntax diagrams and examples. You can replace a variable with an arbitrary name,

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

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

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

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

Example Code Conventions

Examples of SQL code occur throughout this manual. Except as noted, the code is not specific to any single IBM Informix application development tool.

If only SQL statements are listed in the example, they are not delimited by semicolons. For instance, you might see the code in the following example:

```
CONNECT TO stores_demo
...

DELETE FROM customer
   WHERE customer_num = 121
...

COMMIT WORK
DISCONNECT CURRENT
```

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

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

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

Additional Documentation

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

- Installation guides
- Online notes
- Informix error messages
- Manuals
- Online help

Installation Guides

Installation guides are located in the **/doc** directory of the product CD or in the **/doc** directory of the product's compressed file if you downloaded it from the IBM Web site. Alternatively, you can obtain installation guides from the IBM Informix Online Documentation site at http://www.ibm.com/software/data/informix/pubs/library/.

Online Notes

The following sections describe the online files that supplement the information in this manual. Please examine these files before you begin using your IBM Informix product. They contain vital information about application and performance issues.

Online File	Description	Format
TOC Notes	The TOC (Table of Contents) notes file provides a comprehensive directory of hyperlinks to the release notes, the fixed and known defects file, and all the documentation notes files for individual manual titles.	HTML
Documentation Notes	The documentation notes file for each manual contains important information and corrections that supplement the information in the manual or information that was modified since publication.	HTML, text
Release Notes	The release notes file describes feature differences from earlier versions of IBM Informix products and how these differences might affect current products. For some products, this file also contains information about any known problems and their workarounds.	HTML, text
Machine Notes	(Non-Windows platforms only) The machine notes file describes any platform-specific actions that you must take to configure and use IBM Informix products on your computer.	text
Fixed and Known Defects File	This text file lists issues that have been identified with the current version. It also lists customer-reported defects that have been fixed in both the current version and in previous versions.	text

Locating Online Notes

Online notes are available from the IBM Informix Online Documentation site at http://www.ibm.com/software/data/informix/pubs/library/. Additionally you can locate these files before or after installation as described below.

Before Installation

All online notes are located in the **/doc** directory of the product CD. The easiest way to access the documentation notes, the release notes, and the fixed and known defects file is through the hyperlinks from the TOC notes file.

The machine notes file and the fixed and known defects file are only provided in text format.

After Installation

On UNIX platforms in the default locale, the documentation notes, release notes, and machine notes files appear under the **$INFORMIXDIR/release/en_us/0333** directory.

Dynamic Server

On Windows the documentation and release notes files appear in the **Informix** folder. To display this folder, choose **Start > Programs > IBM Informix Dynamic Server *version* > Documentation Notes** or **Release Notes** from the taskbar.

Machine notes do not apply to Windows platforms.

End of Dynamic Server

Online Notes Filenames

Online notes have the following file formats:

Online File	File Format	Examples
TOC Notes	*prod_os*_tocnotes_*version*.html	**ids_win_tocnotes_10.0.html**
Documentation Notes	*prod_bookname*_docnotes_*version*.html/txt	**ids_hpl_docnotes_10.0.html**
Release Notes	*prod_os*_relnotes_*version*.html/txt	**ids_unix_relnotes_10.0.txt**
Machine Notes	*prod*_machine_notes_*version*.txt	**ids_machine_notes_10.0.txt**
Fixed and Known Defects File	*prod*_defects_*version*.txt	**ids_defects_10.0.txt** **client_defects_2.90.txt**
	ids_win_fixed_and_known _defects_*version*.txt	**ids_win_fixed_and_known _defects_10.0.txt**

Informix Error Messages

This file is a comprehensive index of error messages and their corrective actions for the Informix products and version numbers.

On UNIX platforms, use the **finderr** command to read the error messages and their corrective actions.

Dynamic Server

On Windows, use the Informix Error Messages utility to read error messages and their corrective actions. To display this utility, choose **Start > Programs > IBM Informix Dynamic Server *version* > Informix Error Messages** from the taskbar.

End of Dynamic Server

You can also access these files from the IBM Informix Online Documentation site at http://www.ibm.com/software/data/informix/pubs/library/.

Manuals

Online Manuals

A CD that contains your manuals in electronic format is provided with your IBM Informix products. You can install the documentation or access it directly from the CD. For information about how to install, read, and print online manuals, see the installation insert that accompanies your CD. You can also obtain the same online manuals from the IBM Informix Online Documentation site at http://www.ibm.com/software/data/informix/pubs/library/.

Printed Manuals

To order hardcopy manuals, contact your sales representative or visit the IBM Publications Center Web site at http://www.ibm.com/software/howtobuy/data.html.

Online Help

IBM Informix online help, provided with each graphical user interface (GUI), displays information about those interfaces and the functions that they perform. Use the help facilities that each GUI provides to display the online help.

Accessibility

IBM is committed to making our documentation accessible to persons with disabilities. Our books are available in HTML format so that they can be accessed with assistive technology such as screen reader software. The syntax diagrams in our manuals are available in dotted decimal format, which is an accessible format that is available only if you are using a screen reader. For more information about the dotted decimal format, see the Accessibility appendix.

IBM Informix Dynamic Server Version 10.0 and CSDK Version 2.90 Documentation Set

The following tables list the manuals that are part of the IBM Informix Dynamic Server, Version 10.0 and the CSDK Version 2.90, documentation set. PDF and HTML versions of these manuals are available at http://www.ibm.com/software/data/informix/pubs/library/. You can order hardcopy versions of these manuals from the IBM Publications Center at http://www.ibm.com/software/howtobuy/data.html.

Table 1. Database Server Manuals

Manual	Subject
Administrator's Guide	Understanding, configuring, and administering your database server.
Administrator's Reference	Reference material for Informix Dynamic Server, such as the syntax of database server utilities **onmode** and **onstat**, and descriptions of configuration parameters, the **sysmasters** tables, and logical-log records.
Backup and Restore Guide	The concepts and methods you need to understand when you use the **ON-Bar** and **ontape** utilities to back up and restore data.
DB-Access User's Guide	Using the **DB-Access** utility to access, modify, and retrieve data from Informix databases.
DataBlade API Function Reference	The DataBlade API functions and the subset of ESQL/C functions that the DataBlade API supports. You can use the DataBlade API to develop client LIBMI applications and C user-defined routines that access data in Informix databases.
DataBlade API Programmer's Guide	The DataBlade API, which is the C-language application-programming interface provided with Dynamic Server. You use the DataBlade API to develop client and server applications that access data stored in Informix databases.
Database Design and Implementation Guide	Designing, implementing, and managing your Informix databases.
Enterprise Replication Guide	How to design, implement, and manage an Enterprise Replication system to replicate data between multiple database servers.
Error Messages file	Causes and solutions for numbered error messages you might receive when you work with IBM Informix products.
Getting Started Guide	Describes the products bundled with IBM Informix Dynamic Server and interoperability with other IBM products. Summarizes important features of Dynamic Server and the new features for each version.
Guide to SQL: Reference	Information about Informix databases, data types, system catalog tables, environment variables, and the stores_demo demonstration database.
Guide to SQL: Syntax	Detailed descriptions of the syntax for all Informix SQL and SPL statements.
Guide to SQL: Tutorial	A tutorial on SQL, as implemented by Informix products, that describes the basic ideas and terms that are used when you work with a relational database.
High-Performance Loader User's Guide	Accessing and using the High-Performance Loader (HPL), to load and unload large quantities of data to and from Informix databases.
Installation Guide for Microsoft Windows	Instructions for installing IBM Informix Dynamic Server on Windows.
Installation Guide for UNIX and Linux	Instructions for installing IBM Informix Dynamic Server on UNIX and Linux.

Table 1. Database Server Manuals (continued)

Manual	Subject
J/Foundation Developer's Guide	Writing user-defined routines (UDRs) in the Java programming language for Informix Dynamic Server with J/Foundation.
Large Object Locator DataBlade Module User's Guide	Using the Large Object Locator, a foundation DataBlade module that can be used by other modules that create or store large-object data. The Large Object Locator enables you to create a single consistent interface to large objects and extends the concept of large objects to include data stored outside the database.
Migration Guide	Conversion to and reversion from the latest versions of Informix database servers. Migration between different Informix database servers.
Optical Subsystem Guide	The Optical Subsystem, a utility that supports the storage of BYTE and TEXT data on optical disk.
Performance Guide	Configuring and operating IBM Informix Dynamic Server to achieve optimum performance.
R-Tree Index User's Guide	Creating R-tree indexes on appropriate data types, creating new operator classes that use the R-tree access method, and managing databases that use the R-tree secondary access method.
SNMP Subagent Guide	The IBM Informix subagent that allows a Simple Network Management Protocol (SNMP) network manager to monitor the status of Informix servers.
Storage Manager Administrator's Guide	Informix Storage Manager (ISM), which manages storage devices and media for your Informix database server.
Trusted Facility Guide	The secure-auditing capabilities of Dynamic Server, including the creation and maintenance of audit logs.
User-Defined Routines and Data Types Developer's Guide	How to define new data types and enable user-defined routines (UDRs) to extend IBM Informix Dynamic Server.
Virtual-Index Interface Programmer's Guide	Creating a secondary access method (index) with the Virtual-Index Interface (VII) to extend the built-in indexing schemes of IBM Informix Dynamic Server. Typically used with a DataBlade module.
Virtual-Table Interface Programmer's Guide	Creating a primary access method with the Virtual-Table Interface (VTI) so that users have a single SQL interface to Informix tables and to data that does not conform to the storage scheme of Informix Dynamic Server.

Table 2. Client/Connectivity Manuals

Manual	Subject
Client Products Installation Guide	Installing IBM Informix Client Software Developer's Kit (Client SDK) and IBM Informix Connect on computers that use UNIX, Linux, and Windows.
Embedded SQLJ User's Guide	Using IBM Informix Embedded SQLJ to embed SQL statements in Java programs.

Table 2. Client/Connectivity Manuals (continued)

Manual	Subject
ESQL/C Programmer's Manual	The IBM Informix implementation of embedded SQL for C.
GLS User's Guide	The Global Language Support (GLS) feature, which allows IBM Informix APIs and database servers to handle different languages, cultural conventions, and code sets.
JDBC Driver Programmer's Guide	Installing and using Informix JDBC Driver to connect to an Informix database from within a Java application or applet.
.NET Provider Reference Guide	Using Informix .NET Provider to enable .NET client applications to access and manipulate data in Informix databases.
ODBC Driver Programmer's Manual	Using the Informix ODBC Driver API to access an Informix database and interact with the Informix database server.
OLE DB Provider Programmer's Guide	Installing and configuring Informix OLE DB Provider to enable client applications, such as ActiveX Data Object (ADO) applications and Web pages, to access data on an Informix server.
Object Interface for C++ Programmer's Guide	The architecture of the C++ object interface and a complete class reference.

Table 3. DataBlade Developer's Kit Manuals

Manual	Subject
DataBlade Developer's Kit User's Guide	Developing and packaging DataBlade modules using BladeSmith and BladePack.
DataBlade Module Development Overview	Basic orientation for developing DataBlade modules. Includes an example illustrating the development of a DataBlade module.
DataBlade Module Installation and Registration Guide	Installing DataBlade modules and using BladeManager to manage DataBlade modules in Informix databases.

Compliance with Industry Standards

The American National Standards Institute (ANSI) and the International Organization of Standardization (ISO) have jointly established a set of industry standards for the Structured Query Language (SQL). IBM Informix SQL-based products are fully compliant with SQL-92 Entry Level (published as ANSI X3.135-1992), which is identical to ISO 9075:1992. In addition, many features of IBM Informix database servers comply with the SQL-92 Intermediate and Full Level and X/Open SQL Common Applications Environment (CAE) standards.

IBM Welcomes Your Comments

We want to know about any corrections or clarifications that you would find useful in our manuals, which will help us improve future versions. Include the following information:

- The name and version of the manual that you are using
- Section and page number
- Your suggestions about the manual

Send your comments to us at the following email address:

docinf@us.ibm.com

This email address is reserved for reporting errors and omissions in our documentation. For immediate help with a technical problem, contact IBM Technical Support.

We appreciate your suggestions.

Chapter 1. Introducing IBM Informix Embedded SQLJ

What Is Embedded SQLJ? 1-1
How Does Embedded SQLJ Work? 1-1
Embedded SQLJ Versus JDBC. 1-2

In This Chapter

This chapter explains what IBM Informix Embedded SQLJ allows you to do and provides an overview of how it works.

What Is Embedded SQLJ?

IBM Informix Embedded SQLJ enables you to embed SQL statements in your Java programs. IBM Informix Embedded SQLJ consists of:

- The SQLJ translator, which translates SQLJ code into Java code
- A set of Java classes that provide runtime support for SQLJ programs

IBM Informix Embedded SQLJ includes the standard SQLJ implementation, as defined by the SQLJ consortium, plus specific Informix extensions. The rest of this manual refers to IBM Informix Embedded SQLJ as *Embedded SQLJ*. The standard SQLJ implementation is referred to as *traditional Embedded SQLJ*.

How Does Embedded SQLJ Work?

When you use Embedded SQLJ, you embed SQL statements in your Java source code. You use the SQLJ translator to convert the embedded SQL statements to Java source code with calls to JDBC. JDBC is the JavaSoft specification of a standard application programming interface (API) that allows Java programs to access database management systems.

Finally, you use the Java compiler to compile your translated Java program into an executable Java **.class** file, as shown in Figure 1-1.

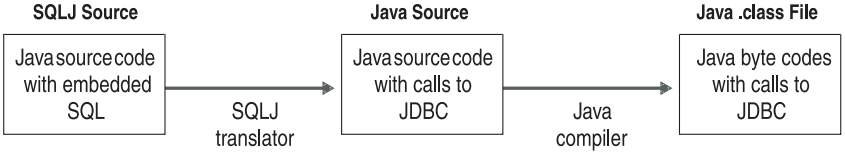


Figure 1-1. Translation and Compilation of an Embedded SQLJ Program

© Copyright IBM Corp. 1996, 2004

When you run your program, it uses the IBM Informix JDBC Driver to connect to an Informix database, as shown in Figure 1-2.

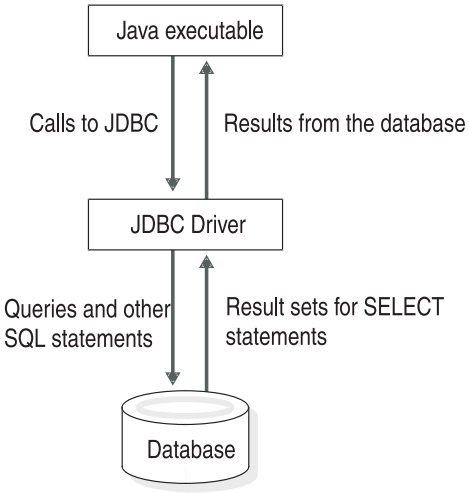


Figure 1-2. Runtime Architecture for Embedded SQLJ Programs

See the *IBM Informix: JDBC Driver Programmer's Guide* for information about using the IBM Informix JDBC Driver.

Embedded SQLJ Versus JDBC

Embedded SQLJ does not support dynamic SQL; you must use the JDBC API if you want to use dynamic SQL. Your Embedded SQLJ program can call the JDBC API to perform a dynamic operation (the SQLJ connection-context object that you use to connect an Embedded SQLJ program to the database contains a JDBC **Connection** object that you can use to create JDBC statement objects).

If you are using static SQL, Embedded SQLJ provides the following advantages:

- **Default connection context.** You only need to set the default connection context once within a program; then every subsequent Embedded SQLJ statement uses this connection context unless you specify otherwise.
- **Reduced statement complexity.** For example, you do not need to explicitly bind each variable; Embedded SQLJ performs binding for you. Generally, this feature allows you to create smaller programs than with the JDBC API.
- **Compile-time syntax and semantics checking.** The Embedded SQLJ translator checks the syntax of SQL statements.
- **Compile-time type checking.** The Embedded SQLJ translator and the Java compiler check that the Java data types of arguments are compatible with the SQL data types of the SQL operation.

- **Compile-time schema checking.** You can connect to a sample database schema during translation to check that your program uses valid SQL statements for the tables, views, columns, stored procedures, and so on in your sample.

Chapter 2. Preparing to Use Embedded SQLJ

What Components Do You Need? 2-1
Setting Up Your Software 2-1
Examples 2-2

In This Chapter

This chapter describes the software you must have to develop Embedded SQLJ programs and how to set up this software.

What Components Do You Need?

You need the following software to create and run SQLJ programs:

- IBM Informix Embedded SQLJ
- The JavaSoft software Java Development Kit (JDK), Version 1.2 or later, or any Java software compatible with JDK 1.2 (also known as *Java 2*)
- IBM Informix JDBC Driver, Version 2.0 or later, to enable your programs to connect to the database server
- One of the following Informix database servers:
  - IBM Informix Dynamic Server, Version 9.x and later
  - IBM Informix Extended Parallel Server, Version 8.x
  - IBM Informix Dynamic Server, Version 7.x
  - IBM Informix Dynamic Server, Workgroup and Developer editions, Version 7.x
  - IBM Informix OnLine Dynamic Server, Version 5.x
  - IBM Informix SE, Versions 5.x to 7.2x

Setting Up Your Software

Before you install Embedded SQLJ, you must already have installed the JavaSoft software Java Development Kit (JDK), Version 1.2 or later. (For more information about the Java language, see the JavaSoft Web site at http://java.sun.com/.)

For further information about installing and using IBM Informix JDBC Driver, see the *IBM Informix: JDBC Driver Programmer's Guide*.

If you do not already have your Informix server installed, refer to the *IBM Informix: Installation Guide* that accompanies that software.

© Copyright IBM Corp. 1996, 2004

Examples

IBM Informix Embedded SQLJ includes sample online programs in the **/demo/sqlj** directory. The README file in this directory briefly explains what each of the programs demonstrates and how to set up, compile, and run the programs. The programs also enable you to verify that IBM Informix Embedded SQLJ and IBM Informix JDBC Driver are correctly installed. The examples in this manual are taken from these sample programs.

Chapter 3. Building an Embedded SQLJ Program

Fundamentals of Embedded SQLJ 3-1
SQLJ Statement Identifier 3-1
Connecting to a Database 3-1
Embedding SQL Statements 3-2
Handling Result Sets. 3-3
Positional Iterators 3-3
Named Iterators 3-3
A Simple Embedded SQLJ Program. 3-4

In This Chapter

This chapter explains the fundamentals of building an Embedded SQLJ program and includes a demonstration program.

Fundamentals of Embedded SQLJ

This chapter introduces simple Embedded SQLJ statements; see Chapter 4, "The Embedded SQLJ Language," on page 4-1, for detailed information about the language.

SQLJ Statement Identifier

Each SQLJ statement in an Embedded SQLJ program is identified by **#sql** at the beginning of the statement. The SQLJ translator recognizes **#sql** and translates the rest of the statement into Java code using JDBC calls.

Connecting to a Database

You can use a class called **ConnectionManager** (located in a file in the **/demo/sqlj** directory) to initiate a JDBC connection. The **ConnectionManager** class uses a JDBC driver and a database URL to connect to a database. Database URLs are described in "Database URLs" on page A-1.

To enable your Embedded SQLJ program to connect to a database, you assign values to the following data members of the **ConnectionManager** class in the file **/demo/sqlj/ConnectionManager.java**:

UID	The user name
PWD	The password for the user name
DRIVER	The JDBC driver
DBURL	The URL for the database

© Copyright IBM Corp. 1996, 2004

You must include the directory that contains your **ConnectionManager.class** file (produced when you compile **ConnectionManager.java**) in your **CLASSPATH** environment variable definition.

Your Embedded SQLJ program connects to the database by calling the **initContext()** method of the **ConnectionManager** class, as follows:

```
ConnectionManager.initContext();
```

"The ConnectionManager Class" on page A-1 provides details about the functionality of the **initContext()** method.

As an alternative to using the **ConnectionManager** class, you can write your own input methods to read the values of user name, password, driver, and database URL from a file or from the command line.

The connection context that you set up is the *default* connection context; all **#sql** statements execute within this context, unless you specify a different context. For information about using nondefault connection contexts, see "Using Nondefault Connection Contexts" on page A-3.

Embedding SQL Statements

Embedded SQL statements can appear anywhere that Java statements can legally appear. SQL statements must appear within curly braces, as follows:

```
#sql
{
INSERT INTO customer VALUES
( 101, "Ludwig", "Pauli", "All Sports Supplies",
"213 Erstwild Court", "", "Sunnyvale", "CA",
"94086", "408-789-8075"
)
};
```

You can use the SELECT...INTO statement to retrieve data into Java variables (*host variables*). Host variables within SQL statements are designated by a preceding colon (:). For example, the following query places values in the variables *customer_num*, *fname*, *lname*, *company*, *address1*, *address2*, *city*, *state*, *zipcode*, and *phone*:

```
#sql
{
SELECT *  INTO :customer_num, :fname, :lname, :company,
:address1, :address2, :city, :state, :zipcode,
:phone
FROM customer
WHERE customer_num = 101
};
```

SQL statements are case insensitive and can be written in uppercase, lowercase, or mixed-case letters. Java statements are case sensitive (and so are host variables).

You use SELECT...INTO statements for queries that return a single record; for queries that return multiple rows (a *result set*), you use an iterator object, described in the next section.

Handling Result Sets

Embedded SQLJ uses result-set iterator objects rather than cursors to manage result sets (cursors are used by languages such as IBM Informix ESQL/C). A result-set iterator is a Java object from which you can retrieve the data returned by a SELECT statement. Unlike cursors, iterator objects can be passed as parameters to a method.

Important: Names of iterator classes must be unique within an application.

When you declare an iterator class, you specify a set of Java variables to match the SQL columns that your SELECT statement returns. There are two types of iterators: positional and named.

Positional Iterators

The order of declaration of the Java variables of a positional iterator must match the order in which the SQL columns are returned. You use a FETCH...INTO statement to retrieve data from a positional iterator.

For example, the following statement generates a positional iterator class with five columns, called **CustIter**:

```
#sql iterator CustIter( int , String, String, String, String, String );
```

This iterator can hold the result set from the following SELECT statement:

```
SELECT customer_num, fname, lname,  address1,
address2, phone
FROM   customer
```

Named Iterators

The name of each Java variable of a named iterator must match the name of a column returned by your SELECT statement; order is irrelevant. The matching of SQL column name and iterator column name is case insensitive.

You use accessor methods of the same name as each iterator column to obtain the returned data, as shown in the example in "A Simple Embedded SQLJ Program" on page 3-4. The SQLJ translator uses the iterator column names to create accessor methods. Iterator column names are case sensitive; therefore, you must use the correct case when you specify an accessor method.

You cannot use the FETCH...INTO statement with named iterators.

For example, the following statement generates a named iterator class called **CustRec**:

```
#sql iterator CustRec(
int    customer_num,
String fname,
String lname ,
String company ,
String address1 ,
String address2 ,
String city ,
String state ,
String zipcode ,
String phone
);
```

This iterator class can hold the result set of any query that returns the columns defined in the iterator class. The result set from the query can have more columns than the iterator class, but the iterator class cannot have more columns than the result set. For example, this iterator class can hold the result set of the following query because the iterator columns include all of the columns in the **customer** table:

```
SELECT *  FROM customer
```

A Simple Embedded SQLJ Program

This sample program, **Demo03.sqlj**, demonstrates the use of a named iterator to retrieve data from a database. This simple program outlines a standard sequence for many Embedded SQLJ programs:

1. Import necessary Java classes.
2. Declare an iterator class.
3. Define the **main()** method.

   All Java applications have a method called **main**, which is the entry point for the application (where the interpreter starts executing the program).
4. Connect to the database.

   The constructor of the application makes the connection to the database by calling the **initContext()** method of the **ConnectionManager** class.
5. Run queries.
6. Create an iterator object and populate it by running a query.
7. Handle the results.
8. Close the iterator.

```
/*************************************************************************
 *
 *                        IBM CORPORATION
 *
```

```
 *                        PROPRIETARY DATA
 *
 *      THIS DOCUMENT CONTAINS TRADE SECRET DATA WHICH IS THE PROPERTY OF
 *     IBM CORPORATION.  THIS DOCUMENT IS SUBMITTED TO RECIPIENT IN
 *      CONFIDENCE.  INFORMATION CONTAINED HEREIN MAY NOT BE USED, COPIED OR
 *      DISCLOSED IN WHOLE OR IN PART EXCEPT AS PERMITTED BY WRITTEN AGREEMENT
 *      SIGNED BY AN OFFICER OF IBM CORPORATION.
 *
 *      THIS MATERIAL IS ALSO COPYRIGHTED AS AN UNPUBLISHED WORK UNDER
 *      SECTIONS 104 AND 408 OF TITLE 17 OF THE UNITED STATES CODE.
 *      UNAUTHORIZED USE, COPYING OR OTHER REPRODUCTION IS PROHIBITED BY LAW.
 *
 *
 *  Title:        Demo03.sqlj
 *
 *  Description:  This demonstrates simple iterator use
 *
 *
 ***************************************************************************
*/
import java.sql.*;
import sqlj.runtime.*; //SQLJ runtime classes

#sql iterator CustRec(
    int    customer_num,
    String fname,
    String lname ,
    String company ,
    String address1 ,
    String address2 ,
    String city ,
    String state ,
    String zipcode ,
    String phone
    );

public class Demo03
{
    public static void main (String args[]) throws SQLException
    {
        Demo03 demo03 = new Demo03();
        try
        {
            demo03.runDemo();
        }
        catch (SQLException s)
        {
            System.err.println( "Error running demo program: " + s );
            System.err.println( "Error Code                : " +
                                s.getErrorCode());
            System.err.println( "Error Message             : " +
                                s.getMessage());
        }
    }


    // Initialize database connection thru Connection Manager
    Demo03()
    {
        ConnectionManager.initContext();
    }
    void runDemo() throws SQLException
    {
```

```
drop_db();

#sql { CREATE DATABASE demo_sqlj WITH LOG MODE ANSI };

#sql
{
    create table customer
    (
    customer_num            serial(101),
    fname                   char(15),
    lname                   char(15),
    company                 char(20),
    address1                char(20),
    address2                char(20),
    city                    char(15),
    state                   char(2),
    zipcode                 char(5),
    phone                   char(18),
    primary key (customer_num)
    )
};

// Insert 4 Records in a try block
try
{
    #sql
    {
    INSERT INTO customer VALUES
        ( 101, "Ludwig", "Pauli", "All Sports Supplies",
          "213 Erstwild Court", "", "Sunnyvale", "CA",
          "94086", "408-789-8075"
        )
    };

    #sql
    {
    INSERT INTO customer VALUES
        ( 102, "Carole", "Sadler", "Sports Spot",
          "785 Geary St", "", "San Francisco", "CA",
          "94117", "415-822-1289"
        )
    };

    #sql
    {
    INSERT INTO customer VALUES
        ( 103, "Philip", "Currie", "Phil's Sports",
          "654 Poplar", "P. O. Box 3498", "Palo Alto",
          "CA", "94303", "415-328-4543"
        )
    };

    #sql
    {
    INSERT INTO customer VALUES
        ( 104, "Anthony", "Higgins", "Play Ball!",
          "East Shopping Cntr.", "422 Bay Road", "Redwood City",
          "CA", "94026", "415-368-1100"
        )
    };

}
catch (SQLException e)
```

```
        {
            System.out.println("INSERT Exception: " + e + "\n");
            System.out.println("Error Code               : " +
                               e.getErrorCode());
            System.err.println("Error Message            : " +
                               e.getMessage());

        }

        System.out.println();
        System.out.println( "Running demo program Demo03...." );
        System.out.println();

        // Declare Iterator of type CustRec
        CustRec cust_rec;

        #sql cust_rec = { SELECT *  FROM customer };

        int row_cnt = 0;
        while ( cust_rec.next() )
            {
            System.out.println("==================================");
            System.out.println("CUSTOMER NUMBER :" + cust_rec.customer_num());
            System.out.println("FIRST NAME      :" + cust_rec.fname());
            System.out.println("LAST NAME       :" + cust_rec.lname());
            System.out.println("COMPANY         :" + cust_rec.company());
            System.out.println("ADDRESS         :" + cust_rec.address1() +"\n" +
                               "                 " + cust_rec.address2());
            System.out.println("CITY            :" + cust_rec.city());
            System.out.println("STATE           :" + cust_rec.state());
            System.out.println("ZIPCODE         :" + cust_rec.zipcode());
            System.out.println("PHONE           :" + cust_rec.phone());
            System.out.println("==================================");
            System.out.println("\n\n");
            row_cnt++;
            }
        System.out.println("Total No Of rows Selected :" + row_cnt);
        cust_rec.close() ;
        System.out.println("\n\n\n\n\n");

        drop_db();
    }
    void drop_db() throws SQLException
    {
        try
        {
            #sql { drop database demo_sqlj };
        }
        catch (SQLException s) { }
    }
}
```

Chapter 4. The Embedded SQLJ Language

Embedded SQLJ Versus Traditional Embedded SQL 4-1
Embedded SQLJ Source Files 4-2
Identifying Embedded SQLJ Statements 4-2
SQL Statements 4-2
Host Variables 4-3
SELECT Statements That Return a Single Row 4-3
Handling Result Sets. 4-4
Positional Iterators 4-4
Named Iterators 4-5
Using Column Aliases 4-6
Iterator Methods 4-6
Positioned Updates and Deletes 4-7
Monitoring the Execution of an SQL Query 4-7
Calling SPL Routines and Functions 4-8
SQL and Java Type Mappings. 4-8
Language Character Sets 4-10
Importing Java Packages 4-11
SQLJ Reserved Names 4-11
Parameter, Field, and Variable Names. 4-11
Class Names and Filenames 4-11
Handling Errors 4-12

In This Chapter

This chapter provides detailed information about using the Embedded SQLJ language. For syntax and reference information about specific statements, refer to the *IBM Informix: Guide to SQL Syntax*.

Embedded SQLJ Versus Traditional Embedded SQL

Embedded SQLJ has some differences from the earlier embedded SQL languages defined by ANSI/ISO: ESQL/C, ESQL/ADA, ESQL/FORTRAN, ESQL/COBOL, and ESQL/PL/1. The major differences are as follows:

- The SQL connection statement of traditional embedded SQL is replaced by a Java connection-context object. This approach enables Embedded SQLJ programs to open multiple database connections simultaneously.
- In Embedded SQLJ there is no host variable definition section (preceded by a BEGIN DECLARE SECTION statement and terminated by an END DECLARE SECTION statement). All legal Java variables can be used as host variables.

© Copyright IBM Corp. 1996, 2004

- Embedded SQLJ does not include the WHENEVER...GOTO/CONTINUE statement, because Java has well-developed rules for declaring and handling exceptions.
- Embedded SQLJ uses iterator objects rather than cursors to manage result sets. A result-set iterator is a Java object from which you can retrieve the data returned by a SELECT statement. Unlike cursors, iterator objects can be passed as parameters to methods.
- Embedded SQLJ supports access to data in columns of iterator objects by name, through generated accessor methods. You can also access this data by position using the FETCH...INTO statement, as used by traditional embedded SQL.
- Unlike other host languages, Java allows null data. Therefore, you do not need to use null indicator variables with Embedded SQLJ.
- Embedded SQLJ does not include dynamic SQL; you must use JDBC instead.

The rest of this chapter describes how to use the Embedded SQLJ language.

Embedded SQLJ Source Files

The files containing your Embedded SQLJ source code must have the extension .**sqlj**; for example, **custapp.sqlj**.

Identifying Embedded SQLJ Statements

To identify Embedded SQLJ statements to the SQLJ translator, each SQLJ statement must begin with **#sql**. The SQLJ translator recognizes **#sql** and translates the statement into Java code.

SQL Statements

Embedded SQLJ supports SQL statements at the SQL92 Entry level, with the following additions:

- The EXECUTE PROCEDURE statement, for calling SPL routines and user-defined routines
- The EXECUTE FUNCTION statement, for calling stored functions
- The BEGIN...END block

SQL statements must appear within curly braces, as follows:

```
#sql
{
create table customer
(
customer_num            serial(101),
fname                   char(15),
```

```
lname                   char(15),
company                 char(20),
address1                char(20),
address2                char(20),
city                    char(15),
state                   char(2),
zipcode                 char(5),
phone                   char(18),
primary key (customer_num)
)
};
```

An SQL statement that is not enclosed within curly braces will generate a syntax error.

SQL statements are case insensitive (unless delimited by double quotes) and can be written in uppercase, lowercase, or mixed-case letters. Java statements are case sensitive.

Host Variables

Host variables are variables of the host language (in this case Java) that appear within SQL statements. A host variable represents a parameter, variable, or field and is prefixed by a colon (:), as in the following example:

```
#sql [ctx] { DELETE FROM customer WHERE customer_num = :cust_no };
```

You use the SELECT...INTO statement (as shown in this example), the FETCH...INTO statement (described in “Positional Iterators” on page 4-4), or an accessor method (described in “Named Iterators” on page 4-5) to retrieve data into host variables.

SELECT Statements That Return a Single Row

You use the SELECT...INTO statement for queries that return a single record of data. For queries that return multiple rows (called a *result set*) you use an iterator object, as described in the next section, “Handling Result Sets” on page 4-4.

The SELECT...INTO statement includes a list of host variables in the INTO clause to which the selected data is assigned. For example:

```
#sql
{
SELECT *  INTO :customer_num, :fname, :lname, :company,
:address1, :address2, :city, :state, :zipcode,
:phone
FROM customer
WHERE customer_num = 101
};
```

The number of selected expressions must match the number of host variables. The SQL types must be compatible with the host variable types. If you use online checking, the SQLJ translator checks that the order, number, and types of the SQL expressions and host variables match. For information on how to perform online checking, see "Online Checking" on page 5-11.

Handling Result Sets

Embedded SQLJ uses iterator objects to manage result sets returned by SELECT statements. A result-set iterator is a Java object from which you can retrieve the data returned from the database. Iterator objects can be passed as parameters to methods and manipulated like other Java objects.

Important: Names of iterator classes must be unique within an application.

When you declare an iterator object, you specify a set of Java variables to match the SQL columns that your SELECT statement returns. There are two types of iterators: positional and named.

Positional Iterators

The order of declaration of the Java variables in a positional iterator must match the order in which the SQL columns are returned.

For example, the following statement generates a positional iterator class called **CustIter** with six columns:

```
#sql iterator CustIter( int , String, String, String, String, String );
```

This iterator can hold the result set from the following SELECT statement:

```
SELECT customer_num, fname, lname,  address1,
address2, phone
FROM   customer
```

You run the SELECT statement and populate the iterator object with the result set by using an Embedded SQLJ statement of the form:

```
#sql iterator-object = { SELECT ...};
```

For example:

```
CustIter cust_rec;
#sql  [ctx] cust_rec = { SELECT customer_num, fname, lname,  address1,
address2, phone
FROM   customer
};
```

You retrieve data from a positional iterator into host variables using the FETCH...INTO statement:

```
#sql { FETCH :cust_rec
INTO :customer_num, :fname, :lname,
:address1, :address2, :phone
};
```

The SQLJ translator checks that the types of the host variables in the INTO clause of the FETCH statement match the types of the iterator columns in corresponding positions.

The types of the SQL columns in the SELECT statement must be compatible with the types of the iterator. These type conversions are checked at translation time if you perform online checking. For information about setting up online checking, see "Online Checking" on page 5-11. For a listing of SQL and Java type mappings, see "SQL and Java Type Mappings" on page 4-8.

Named Iterators

The name of each Java variable of a named iterator must match the name of a column returned by your SELECT statement; order is irrelevant. The matching of SQL column names and iterator column names is case insensitive.

For example, the following statement generates a named iterator class called **CustRec**:

```
#sql iterator CustRec(
int    customer_num,
String fname,
String lname ,
String company ,
String address1 ,
String address2 ,
String city ,
String state ,
String zipcode ,
String phone
);
```

This iterator can hold the result set of any query that returns the columns defined in the iterator class. You use accessor methods of the same name as each iterator column to obtain the returned data, as shown in the example in "A Simple Embedded SQLJ Program" on page 3-4. The SQLJ translator uses the iterator column names to create accessor methods. Iterator column names are case sensitive; therefore, you must use the correct case when you specify an accessor method.

You cannot use the FETCH...INTO statement with named iterators.

The following example illustrates the use of named iterators:

```
// Declare Iterator of type CustRec
CustRec cust_rec;
```

```
#sql cust_rec = { SELECT *  FROM customer };

int row_cnt = 0;
while ( cust_rec.next() )
{
System.out.println("==================================");
System.out.println("CUSTOMER NUMBER :" + cust_rec.customer_num());
System.out.println("FIRST NAME      :" + cust_rec.fname());
System.out.println("LAST NAME       :" + cust_rec.lname());
System.out.println("COMPANY         :" + cust_rec.company());
System.out.println("ADDRESS         :" + cust_rec.address1() +"\n" +
"                 " + cust_rec.address2());
System.out.println("CITY            :" + cust_rec.city());
System.out.println("STATE           :" + cust_rec.state());
System.out.println("ZIPCODE         :" + cust_rec.zipcode());
System.out.println("PHONE           :" + cust_rec.phone());
System.out.println("==================================");
System.out.println("\n\n");
row_cnt++;
}
System.out.println("Total No Of rows Selected :" + row_cnt);
cust_rec.close() ;
```

The **next()** method of the iterator object advances processing to successive rows of the result set. It returns `FALSE` after it fails to find a row to retrieve.

The Java compiler detects type mismatches for the accessor methods.

The validity of the types and names of the iterator columns and their related columns in the SELECT statement are checked at translation time if you perform online checking. For information about setting up online checking, see "Online Checking" on page 5-11.

Using Column Aliases

When an expression returned by a SELECT statement has an SQL name that is not a valid Java identifier, use SQL column aliases to rename them. For example, the name **Not valid for Java** is acceptable as a column name in SQL, but not as a Java identifier. You can use a column alias that has a name acceptable as a Java identifier by using the AS clause:

```
SELECT "Not valid for Java" AS "Col1" FROM tablename
```

When you create a named iterator class for this query, you specify the column alias name for the Java variable, as in:

```
#sql iterator Iterator_name (String Col1);
```

Iterator Methods

Both named and positional iterator objects have the following methods:

- **rowCount()**

  Returns the number of rows retrieved by the iterator object

- **close()**

  Closes the iterator; raises **SQLException** if the iterator is already closed
- **isClosed()**

  Returns `TRUE` after the iterator's **close()** method has been called; otherwise, it returns `FALSE`

Positional iterators also have the **endFetch()** method. The **endFetch()** method returns `TRUE` when no more rows are available.

Named iterators also have the **next()** method. The **next()** method advances processing to successive rows of the result set. It returns `FALSE` after it fails to find a row to retrieve. For an example of how to use the **next()** method, see "Named Iterators" on page 4-5.

Positioned Updates and Deletes

To perform positioned updates and deletes in a result set, you use the WHERE CURRENT OF clause with a host variable that contains an iterator object. For example:

```
#sql { delete_statement/update_statement
       WHERE CURRENT OF :iter };
```

At runtime, the variable *:iter* must contain an open iterator object that contains a result set selected from the same table accessed by the query in either *delete_statement* or *update_statement*. The current row of that iterator object is deleted or updated.

Monitoring the Execution of an SQL Query

You can monitor and modify the execution of an SQL query by using the *execution context* associated with it. An execution context is an instance of the class **sqlj.runtime.ExecutionContext**; an execution context is associated with each executable SQL operation in an Embedded SQLJ program.

You can supply an execution context explicitly for an SQL statement:

```
#sql [execCtx] {SQL_statement};
```

If you do not explicitly supply an execution context, the SQL statement uses the default execution context for the connection context you are using.

If you want to supply an explicit connection context and an explicit execution context, the SQL statement looks like this:

```
#sql [connCtx, execCtx] {SQL_statement };
```

You use the **getExecutionContext()** method of the connection context to obtain that connection's default execution context.

The execution-context object has attributes and methods that provide information about an SQL operation and the ability to modify its execution.

For each of the following attributes, there is a method called **getattribute** that reads the value of the attribute, and a method called **setattribute** that sets its value. The attributes are:

MaxRows	The maximum number of rows a query can return
MaxFieldSize	The maximum number of bytes that can be returned as data for any column or output variable
QueryTimeout	The number of seconds to wait for an SQL operation to complete
SQLWarnings	Any warnings that occurred during the last SQL operation
UpdateCount	The number of rows updated, inserted, or deleted during the last SQL operation

Calling SPL Routines and Functions

You can call a Stored Procedure Language (SPL) procedure by using the EXECUTE PROCEDURE statement. For example:

```
#sql { EXECUTE PROCEDURE proc_name(:arg_name) };
```

You can call a stored function by using the EXECUTE FUNCTION statement. For example:

```
#sql {EXECUTE FUNCTION func_name (func_arg ) into :num };
```

SQL and Java Type Mappings

When you retrieve data from a database into an iterator object (see "Handling Result Sets" on page 4-4) or into a host variable, you must use Java types that are compatible with the SQL types. The following table shows valid conversions from SQL types to Java types.

SQL Type	Java Type
BLOB	byte[]
BOOLEAN	boolean
BYTE	byte[]
CHAR, CHARACTER	String
CHARACTER VARYING	String
CLOB	byte[]
DATE	java.sql.Date
DATETIME	java.sql.Timestamp
DECIMAL, NUMERIC, DEC	java.math.BigDecimal
FLOAT, DOUBLE PRECISION	double
INT8	long
INTEGER, INT	int
INTERVAL	IfxIntervalDF, IfxIntervalYM[1]
LVARCHAR	String
MONEY	java.math.BigDecimal
NCHAR, NVARCHAR	String
SERIAL	int
SERIAL8	long
SMALLFLOAT	float[2]
SMALLINT	short
TEXT	String
VARCHAR	String

[1] IfxIntervalYM and IfxIntervalDF are Informix extensions to JDBC 2.0.[2] This mapping is JDBC compliant. You can use IBM Informix JDBC Driver to map SMALLFLOAT data type (via the JDBC FLOAT data type) to the Java double data type for backward compatibility by setting the IFX_GET_SMFLOAT_AS_FLOAT environment variable to 1.

You must also use compatible Java types for host variables that are arguments to SQL operations. This table shows valid conversions from Java types to SQL types.

Java Type	SQL Type
java.math.BigDecimal	DECIMAL
boolean	BOOLEAN
byte[]	BYTE
java.sql.Date	DATE
double	FLOAT[1]
float	SMALLFLOAT
int	INT
long	INT8
short	SMALLINT
String	CHAR
java.sql.Time	DATETIME
java.sql.Timestamp	DATETIME
com.informix.jdbc.IfxIntervalDF	INTERVAL
com.informix.jdbc.IfxIntervalYM	INTERVAL

[1] This mapping is JDBC compliant. You can use IBM Informix JDBC Driver to map the Java double data type (via the JDBC FLOAT data type) to the Informix SMALLFLOAT data type for backward compatibility by setting the IFX_GET_SMFLOAT_AS_FLOAT environment variable to 1.

Important: Unlike other host languages (for example, C), Java allows null data. Therefore, you do not need to use null indicator variables with Embedded *SQLJ*. The Java `null` value is equivalent to the *SQL* `NULL` value.

Language Character Sets

Embedded SQLJ supports Java's Unicode escape sequences. Also, if you set your Java property **file.encoding** to `8859_1` (or do not set it at all), you can use the Latin-1 character set.

To process files with a different encoding—for example, SJIS—you have the following choices:

- Use the Sun JDK tool **native2ascii** to convert the native encoded source to a source with ASCII encoding.
- Set `file.encoding=SJIS` in **java.properties** in the Java home directory.
- Invoke the SQLJ translator using the following command:

  ```
  java ifxsqlj -Dfile.encoding=SJIS file.sqlj
  ```

Importing Java Packages

Your Embedded SQLJ programs need to import the JDBC API (**java.sql.***) and SQLJ runtime (**sqlj.runtime.***) packages to which they refer. The classes you are likely to commonly use are:

- In package **java.sql** for the JDBC API:

  The **SQLException** class—includes all runtime exceptions raised by Embedded SQLJ—and classes you explicitly use, such as **java.sql.Date**, **java.sql.ResultSet**.
- In package **sqlj.runtime** for SQLJ runtime:

  SQLJ stream types (explicitly referenced): for example, **BinaryStream**, the **ConnectionContext** class, and the reference implementation of Embedded SQLJ classes (in **sqlj.runtime.ref**).

SQLJ Reserved Names

This section lists names reserved by the SQLJ translator. Do not use these names in your Embedded SQLJ programming.

Parameter, Field, and Variable Names

The string **_sJT** is a reserved prefix for generated variable names. Do not use this prefix for the names of:

- Variables declared within blocks that include SQL statements
- Parameters to methods that contain SQL statements
- Fields in classes that contain SQL statements or whose subclasses contain SQL statements

Class Names and Filenames

Do not declare classes that conflict with the names of internal classes. Do not create files that conflict with generated internal resource files.

The SQLJ translator creates internal classes and resource files for use by generated code. The names of these files and classes have a prefix composed of the name of the original input file followed by the string **_SJ**. For example, if you translate a file called **File1.sqlj** that uses the package **COM.foo**, the names of some of the internal classes produced are:

- **COM.foo.File1_SJInternalClass**
- **COM.foo.File1_SJProfileKeys**
- **COM.foo.File1_SJInternalClass$Inner**
- **COM.foo.File1_SJProfile0**
- **COM.foo.File1_SJProfile1**

Generated files for these internal classes, which are created in the same directory as the input file, **File1.sqlj**, are called:

- **File1_SJInternalClass.java** (includes the class **COM.foo.File1_SJInternalClass$Inner**)
- **File1_SJProfileKeys.java**
- **File1_SJProfile0.ser**
- **File1_SJProfile1.ser**

Files with the **.ser** extension are internal resource files that contain information about SQL operations in an **.sqlj** file.

Handling Errors

Some iterator and connection-context methods might raise exceptions specified by the JDBC API **SQLException** class. For information about using **SQLException** methods to obtain information about these errors, refer to your JDBC API documentation.

Chapter 5. Processing Embedded SQLJ Source Code

Translating, Compiling, and Running Embedded SQLJ Programs 5-1
The ifxsqlj Command 5-2
Command Options 5-3
Basic Options 5-3
Advanced Options 5-5
Setting Options 5-8
Setting Options on the Command Line 5-9
Supplying Options in Property Files 5-9
Precedence of Options 5-9
Format of Property Files 5-10
Online Checking 5-11
Setting the -user and -password Options 5-11
Setting the -url and -driver Options 5-11
The ifxprofp Tool 5-12

In This Chapter

This chapter describes how to create executable Java programs from your Embedded SQLJ source code. It explains:

- How to use the SQLJ translator
- Basic translation and compilation options
- Advanced

The **-driver** option specifies a list of JDBC drivers that can be used to interpret JDBC connection URLs for online checking.

Both of these options are shown in "Advanced Options" on page 5-5.

The ifxprofp Tool

Embedded SQLJ includes the **ifxprofp** tool. The tool **ifxprofp** enables you to print out the information stored in internal resource **.ser** files, for debugging purposes. You invoke the tool as follows:

```
java ifxprofp filename.ser
```

Here is an example of the output of the **ifxprofp** tool:

```
==================================================
printing contents of profile Demo02_SJProfile0
created 918584057644 (2/9/99 10:14 AM)
associated context is sqlj.runtime.ref.DefaultContext
profile loader is sqlj.runtime.profile.DefaultLoader@1f7f1941
contains no customizations
original source file:Demo02.sqlj
contains 8 entries
==================================================
profile Demo02_SJProfile0 entry 0
#sql { CREATE DATABASE demo_sqlj WITH LOG MODE ANSI
         };
line number:59
PREPARED_STATEMENT executed via EXECUTE_UPDATE
role is STATEMENT
descriptor is null
contains no parameters
result set type is NO_RESULT
result set name is null
contains no result columns
==================================================
```

Appendix A. Connecting to Databases

"Connecting to a Database" on page 3-1 describes how Embedded SQLJ programs connect to databases. This appendix provides background information and information about using nondefault connection contexts.

The ConnectionManager Class

You use the **ConnectionManager** class to make a connection to a database, as described in "Connecting to a Database" on page 3-1. The **ConnectionManager** class has two methods:

- **newConnection()**
- **initContext()**

The **newConnection()** method creates and returns a new JDBC **Connection** object using the current values of the DRIVER, DBURL, UID, and PWD attributes. If any of the needed attributes is null or a connection cannot be established, an error message is printed to **System.out**, and the program exits.

The **initContext()** method returns the currently installed default context. If the current default context is null, a new default context instance is created and installed using a connection obtained from a call to **getConnection**.

Database URLs

The DBURL data member of the **ConnectionManager** class and the value for the **-url** option that you specify for online checking are database URLs. (For information about online checking, see "Online Checking" on page 5-11.) Database URLs specify the subprotocol (the database connectivity mechanism), the database or server identifier, and a list of properties.

Your Embedded SQLJ program uses IBM Informix JDBC Driver to connect to an Informix database. IBM 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.

© Copyright IBM Corp. 1996, 2004

- Brackets (**[]**) denote an optional value.
- Words or symbols not enclosed in brackets are required (`INFORMIXSERVER=`, for example).

Important: Spaces are not allowed in the database URL.

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

Database URL Variable	Required?	Description
ip-address or *domain-name*	Yes	The IP address or the domain name of the computer running the Informix database server An example of an IP address is 123.45.67.89. An example of a domain name is myhost.com.
port-number	Yes	The port number of the Informix database server
dbname	No	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.
server-name	Yes	The name of the Informix 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.
username	Yes	The name of the user you want to connect to the Informix database or database server as
password	Yes	The password of the user specified by *username*
name=value	No	A name-value pair that specifies a *value* for the Informix environment variable contained in the *name* variable, recognized by either IBM Informix JDBC Driver or Informix database servers The value of *name* is case insensitive. For information about environment variables supported by IBM Informix JDBC Driver and how to set them, refer to the *IBM Informix: JDBC Driver Programmer's Guide*.

Using Nondefault Connection Contexts

This section explains how to use nondefault connection contexts. Embedded SQLJ uses a connection-context object to manage the connection to the database in which you want an SQL statement to execute. You can specify different connection-context objects for different SQL statements in the same Embedded SQLJ program, as shown in the sample program **MultiConnect.sqlj** included in this section.

To use a nondefault connection context:

1. Define the connection-context class by using an Embedded SQLJ connection statement. The syntax of the connection statement is as follows:

   `#sql [modifiers] context java_class_name;`

modifiers A list of Java class modifiers: for example, **public**

java_class_name
The name of the Java class of the new connection context

1. Create a connection-context object for connecting to the database.
2. Specify the connection-context object in your Embedded SQLJ statement in parentheses following the **#sql** string.

MultiConnect.sqlj

The sample program **MultiConnect.sqlj** creates two databases with two tables, **Orders** and **Items**, and inserts two records in the **Orders** table with corresponding records in the **Items** table. The program prints the order line items for all the orders from both tables, which reside in different databases, by creating separate connection contexts for each database.

MultiConnect.sqlj calls the methods **executeSQLScript()** and **getConnect()**. These methods are contained in **demoUtil.java**, which follows this program.

```
/*****************************************************************************
 *
 *                           IBM CORPORATION
 *
 *                           PROPRIETARY DATA
 *
 *      THIS DOCUMENT CONTAINS TRADE SECRET DATA WHICH IS THE PROPERTY OF
 *      IBM CORPORATION.  THIS DOCUMENT IS SUBMITTED TO RECIPIENT IN
 *      CONFIDENCE.  INFORMATION CONTAINED HEREIN MAY NOT BE USED, COPIED OR
 *      DISCLOSED IN WHOLE OR IN PART EXCEPT AS PERMITTED BY WRITTEN AGREEMENT
 *      SIGNED BY AN OFFICER OF IBM CORPORATION.
 *
 *      THIS MATERIAL IS ALSO COPYRIGHTED AS AN UNPUBLISHED WORK UNDER
 *      SECTIONS 104 AND 408 OF TITLE 17 OF THE UNITED STATES CODE.
 *      UNAUTHORIZED USE, COPYING OR OTHER REPRODUCTION IS PROHIBITED BY LAW.
 *
 *
```

```
 *  Title:        MultiConnect.sqlj
 *
 *  Description:  This demonstrates usage of 2 connection contexts using
 *                different URLs.
 *
 *
 ***************************************************************************
*/

import java.sql.*;
import java.math.*;
import java.lang.*;
import sqlj.runtime.*;  //SQLJ runtime classes
import sqlj.runtime.ref.*;

/* Declare  ConnectionContext classes OrdersCtx and ItemsCtx.
 * OrdersCtx is related to the orders table which is in orders_db database
 * ItemsCtx  is related to the items table which is in items_db database
 * Instances of these  classes are used to specify where SQL operations
 * on orders table or items table shld should execute.
 * We create the 2 databases using a default context using ConnectionManager
 *
 * For an order (from the orders table in the orders_db database), we try
 * to query the items table(in the items_db database) for the line items which
 * make up that order
 *
 */

#sql context OrdersCtx;
#sql context ItemsCtx;

// Declare 2 named iterators for Items and Orders

#sql iterator OrdersRec (
    Integer     order_num,
    Date        order_date,
    String      po_num,
    Date        paid_date
    );

#sql iterator ItemsRec (
    Short       item_num,
    int         order_num,
    Short       stock_num,
    String      manu_code,
    Integer     quantity,
    BigDecimal  total_price
    );



public class MultiConnect extends demoUtil
{
    private OrdersCtx o_ctx = null;
    private ItemsCtx  i_ctx = null;
    private DefaultContext ctx = null;

    // The constructor sets up a default database context

    MultiConnect()
    {
        /* Initialize database connection thru Connection Manager
         * and create a default context
```

```
     */
    ctx = ConnectionManager.initContext();
}

public static void main (String args[]) throws SQLException
{
    MultiConnect mc_ob = new MultiConnect();
    try
    {
        System.out.println( "Running demo program MultiConnect...." );
        mc_ob.runDemo();

        //Close the connection
        mc_ob.o_ctx.close() ;
        mc_ob.i_ctx.close() ;
    }
    catch (SQLException s)
    {
        System.err.println( "Error running demo program: " + s );
        System.err.println( "Error Code                : " +
                        s.getErrorCode());
        System.err.println( "Error Message             : " +
                        s.getMessage());
    }
}
void runDemo() throws SQLException
{
    // We drop the 2 databases using the default context

    drop_db();

    /*
     * We create the 2 databases needed for the program using the
     * default Connection Context
     */

    #sql [ctx] { CREATE DATABASE orders_db WITH LOG MODE ANSI };
    #sql [ctx] { CREATE DATABASE items_db WITH LOG MODE ANSI };
    ctx.close();

    String driver = "com.informix.jdbc.IfxDriver";
    String url = "jdbc:158.58.9.121:1527:informixserver=tulua2";
    String user = "rdtest";
    String password = "1RDSRDS";
    set_driver(driver);
    set_url(url);
    set_user(user);
    set_passwd(password);
    getConnect();

    // Create the schema and the tables by running the SQL scripts
    executeSQLScript("./schema.sql");
    conn.close();

    // We now set up the Connection context OrdersCtx
    url = "jdbc:158.58.9.121:1527/orders_db:informixserver=tulua2";
    set_url(url);
    o_ctx = new OrdersCtx(getConnect());

    /* Change the url to reflect items database
     * Here we are changing the database name
     * the machine name and the port no could also be different
     */
```

```
    url = "jdbc:158.58.9.121:1527/items_db:informixserver=tulua2";
    set_url(url);
    i_ctx = new ItemsCtx(getConnect());

    // Declare orders_rec of type OrdersRec
    OrdersRec orders_rec;

    // Using context o_ctx query orders

    #sql [o_ctx] orders_rec =
    { SELECT order_num, order_date, po_num, paid_date
      FROM orders
    };
    while ( orders_rec.next() )
    {
    System.out.println("================================"+
                       "====================");
    System.out.print("ORDER NUMBER:" + orders_rec.order_num() + "\t\t");
    System.out.println("ORDER DATE:" + orders_rec.order_date() );
    System.out.print("PURCHASE ORDER NUMBER:" +
                     orders_rec.po_num() + "\t");
    System.out.println("PAID DATE:" + orders_rec.paid_date() );
    System.out.println("================================"+
                       "====================");
    System.out.print("\n");
    int ord_no = orders_rec.order_num().intValue();
    printItemRec( fetchItemRec(ord_no) ) ;
    }
    System.out.println("\n");
}
ItemsRec fetchItemRec(int ord_no) throws SQLException
{

    ItemsRec  items_rec;
    #sql  [i_ctx] items_rec =
    { SELECT item_num, order_num, stock_num, manu_code, quantity,
             total_price
      FROM   items
      WHERE  order_num = :ord_no
    };
    return items_rec;
}
void printItemRec(ItemsRec items_rec) throws SQLException
{
    System.out.print("ITEM NUMBER   ");
    System.out.print("STOCK NUMBER   ");
    System.out.print("MANUFACTURER CODE   ");
    System.out.print("QUANTITY   ");
    System.out.print("TOTAL PRICE   ");
    System.out.println("\n--------------------------------"+
                       "---------------------------------------");
    while ( items_rec.next() )
    {
        System.out.print(items_rec.item_num() + "\t\t");
        System.out.print(items_rec.stock_num() + "\t\t" );
        System.out.print(items_rec.manu_code()+ "\t\t");
        System.out.print(items_rec.quantity() + "   " + "\t\t");
        System.out.print(items_rec.total_price() + "\t\t");
        System.out.print("\n");
    }
    System.out.println("\n");
```

```
    }
    void drop_db() throws SQLException
    {
        try
        {
            #sql [ctx]  { drop database orders_db };
            #sql [ctx] { drop database items_db };
        }
        catch (SQLException s) { }
    }
}
/*****************************************************************************
 *
 *                            IBM CORPORATION
 *
 *                            PROPRIETARY DATA
 *
 *      THIS DOCUMENT CONTAINS TRADE SECRET DATA WHICH IS THE PROPERTY OF
 *      IBM CORPORATION  THIS DOCUMENT IS SUBMITTED TO RECIPIENT IN
 *      CONFIDENCE.  INFORMATION CONTAINED HEREIN MAY NOT BE USED, COPIED OR
 *      DISCLOSED IN WHOLE OR IN PART EXCEPT AS PERMITTED BY WRITTEN AGREEMENT
 *      SIGNED BY AN OFFICER OF IBM CORPORATION.
 *
 *      THIS MATERIAL IS ALSO COPYRIGHTED AS AN UNPUBLISHED WORK UNDER
 *      SECTIONS 104 AND 408 OF TITLE 17 OF THE UNITED STATES CODE.
 *      UNAUTHORIZED USE, COPYING OR OTHER REPRODUCTION IS PROHIBITED BY LAW.
 *
 *
 *  Title:        demoUtil.java
 *
 *  Description:  Utilities used in the demo programs
 *
 *
 *
 *****************************************************************************
*/

import java.io.*;
import java.util.*;
import java.lang.*;
import java.sql.*;

public class demoUtil
{
    private String driver;
    private String URL;
    private String myURL;
    private String user;
    private String passwd;
    private int count = 0;
    private int lineno = 0;
    private int errors = 0;
    private boolean end_of_file = false;
    private FileInputStream fs = null;
    private DataInputStream in = null;
    private BufferedReader br = null;
    private String line = null;
    private StringBuffer read_line = null;
    public Connection conn;


    public void executeSQLScript(String SQLscript)
    {
```

```
        try
        {
            fs = new FileInputStream(SQLscript);
        }
        catch (Exception e)
        {
            System.out.println("Script File Not Found");
        e.printStackTrace();
        }

        in = new DataInputStream(fs);
        br = new BufferedReader(new InputStreamReader(in));
        line = getNextLine();
        read_line = (line==null) ? new StringBuffer() : new StringBuffer(line);
        while (!end_of_file)
        {
            if (line!=null && line.indexOf(';')==line.length()-1)
            {
                tryExecute(read_line);
                read_line = new StringBuffer();
            }
            line = getNextLine();
            if (line!=null)
                read_line.append(line).append(" ");

        }
        if (read_line!=null && read_line.length()>0)
        {
            tryExecute(read_line);
        }
        System.out.println("\n");
    }
    private boolean isComment(String s)
    {
        if (s!=null)
            s.trim();
        return (
               s==null || s.equals("")
               || (s.length()>=2 && s.substring(0,2).equals("--"))
               || (s.length()>=4 && s.substring(0,4).toUpperCase().equals(
                   "REM "))
               );
    }

    private String getNextLine()
    {
        String line = null;
        lineno++;

        try
        {
            line = br.readLine();
            if (line==null)
                end_of_file=true;
        }
        catch (IOException e)
        {
            line = null;
            end_of_file=true;
        }

        return ( (isComment(line)) ? null : line);
    }
```

```
private String bufferToCommand(StringBuffer sb)
{
    String s = sb.toString().trim();

    // chop off trailing semicolon
    if (s.substring(s.length()-1,s.length()).equals(";"))
        s = s.substring(0,s.length()-1);

    return s;
}
private void tryExecute(StringBuffer sb)
{
    String cmd = bufferToCommand(sb);
    System.out.print(".");
    System.out.flush();

    try
    {
        count++;
        Statement stmt = conn.createStatement();
        stmt.executeUpdate(cmd);
        stmt.close();
    }
    catch (SQLException e)
    {
        errors++;
        System.out.println("SQL Error line "+lineno+": "+e.getMessage());
        System.out.println("SQLState: " + e.getSQLState());
        System.out.println("ErrorCode: " + e.getErrorCode());
        System.out.println("Offending statement: '"+cmd+"'");
        e.printStackTrace();
    }
}
public void set_driver(String driver)
{
    this.driver = driver;
}
public void set_url(String url)
{
    this.URL = url;
}
public void set_user(String userName)
{
    this.user = userName;
}
public void set_passwd(String passwd)
{
    this.passwd = passwd;
}
public void connSetup()
{
    try
    {
        Class.forName(driver);
    }
    catch (Exception e)
    {
        System.out.println("Failed to load IBM Informix JDBC driver.");
    e.printStackTrace();
    }
myURL = URL ;
myURL = myURL + ";user=" + user + ";password=" + passwd;
```

```
    }
    public Connection getConnect()
    {

        connSetup();
        try
    {
            conn = DriverManager.getConnection(myURL);
        }
        catch (SQLException e)
        {
            System.out.println("Connect Error : " + e.getErrorCode());
            System.out.println("Failed to connect: " + e.toString());
            e.printStackTrace();
        }
        return conn;
    }
    public Connection getConnect(Connection i_conn)
    {
        connSetup();
        try
    {
            i_conn = DriverManager.getConnection(myURL);
        }
        catch (SQLException e)
        {
            System.out.println("Connect Error : " + e.getErrorCode());
            System.out.println("Failed to connect: " + e.toString());
            e.printStackTrace();
        }
        return i_conn;
    }
}
```

Appendix B. Sample Programs

The following table lists and describes the online sample programs that are included with IBM Informix Embedded SQLJ.

Demo Program Name	Description
Demo01.sqlj	Demonstrates a simple connection to the database
Demo02.sqlj	Demonstrates a simple SELECT statement and the use of host variables
Demo03.sqlj	Demonstrates the use of a named iterator
Demo04.sqlj	Demonstrates the use of a positional iterator
Demo05.sqlj	Demonstrates interoperability between a JDBC ResultSet object and an SQLJ iterator
Demo06.sqlj	Demonstrates interoperability between a JDBC **Connection** object and an SQLJ connection-context object

The sample programs are located in the **IFXJLOCATION/demo/sqlj** directory (**IFXJLOCATION** refers to the directory where you chose to install Embedded SQLJ). The README file in the directory explains how to compile and run the programs.

© Copyright IBM Corp. 1996, 2004

Appendix C. Accessibility

The syntax diagrams in the HTML version of this manual are available in dotted decimal syntax format, which is an accessible format that is available only if you are using a screen reader.

Dotted Decimal Syntax Diagrams

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

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

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

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

Characters such as commas, which are used to separate a string of syntax elements, are shown in the syntax just before the items they separate. These characters can appear on the same line as each item, or on a separate line with the same dotted decimal number as the relevant items. The line can also show another symbol that provides information about the syntax elements. For example, the lines `5.1*`, `5.1 LASTRUN,` and `5.1 DELETE` mean that if you use more than one of the `LASTRUN` and `DELETE` syntax elements, the elements

© Copyright IBM Corp. 1996, 2004

must be separated by a comma. If no separator is given, assume that you use a blank to separate each syntax element.

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

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

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

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

***** Specifies a syntax element that can be repeated zero or more times. A dotted decimal number followed by the * symbol indicates that this syntax element can be used zero or more times; that is, it is optional and can be repeated. For example, if you hear the line `5.1* data-area`, you know that you can include more than one data area or

you can include none. If you hear the lines `3*`, `3 HOST`, and `3 STATE`, you know that you can include `HOST`, `STATE`, both together, or nothing.

Notes:

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

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

Notices

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

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

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

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

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

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

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make

© Copyright IBM Corp. 1996, 2004

improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

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

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

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

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

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

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

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

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

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

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

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

COPYRIGHT LICENSE:

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

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

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

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

Trademarks

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

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

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

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

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

Index

Special characters

__sJT prefix 4-11
-C prefix 5-9
-cache option 5-5
-compile option 5-6
-compiler-encoding-flag option 5-6
-compiler-executable option 5-6
-compiler-output-file option 5-6
-d option 5-3
-dir option 5-3
-driver option 5-6
-encoding option 5-3
-help option 5-3
-J prefix 5-9
-linemap option 5-4
-offline option 5-6
-online option 5-6
-password option 5-7
-props option 5-4, 5-9
-ser2class option 5-7
-status option 5-4
-url option 5-7
-user option 5-7
-version option 5-4
-vm option 5-8
-warn option 5-4
.class files 1-1
.ser files 5-1, 5-7, 5-12
.sqlj file extension 4-2

A

Accessibility xvii
  dotted decimal format of syntax diagrams C-1
  syntax diagrams, reading in a screen reader C-1
Accessor methods 3-3, 4-2, 4-5

B

BEGIN DECLARE SECTION statement 4-1
BEGIN...END block 4-2
Binding of variables 1-2
Boldface type viii
Boolean options 5-8

C

CLASSPATH environment variable 3-2, 5-2
close() method 4-7
Code, sample, conventions for xiii
Column aliases 4-6
Command options, ifxsqlj 5-3
Command-line conventions
  how to read xi
  sample diagram xi
Compiling code 5-1
Compliance
  with industry standards xx
Connecting to a database 3-1
Connection-context class A-3
Connection-context object A-3
ConnectionManager class 3-1, 3-4, A-1
ConnectionManager.java file 3-1
Contact information xxi
Conventions
  command-line xi
  documentation vii
  sample-code xiii
  syntax diagrams ix
  syntax notation ix
  typographical vii
Curly braces, {} 4-2
Cursors 3-3, 4-2

D

Database server names, setting in database URLs A-2
Database servers 2-1
Database URLs 3-1, A-1
Databases, connecting to 3-1
Default connection context 1-2, 3-2
Deletes, positioned 4-7
Demo01.sqlj program B-1
Demo02.sqlj program B-1
Demo03.sqlj program 3-4, B-1
Demo04.sqlj program B-1
Demo05.sqlj program B-1
Demo06.sqlj program B-1
demoUtil.java program A-3
Dependencies, software vii
Disabilities, visual
  reading syntax diagrams C-1
Documentation conventions vii
Documentation Notes xv
Documentation set of all manuals xvii
Documentation, types of xiv
  machine notes xv
  online manuals xvii
  printed manuals xvii
Domain names, setting in database URLs A-2
Dotted decimal format of syntax diagrams C-1
Dynamic SQL 4-2

© Copyright IBM Corp. 1996, 2004

E

Embedded SQL, traditional 4-1
END DECLARE SECTION statement 4-1
endFetch() method 4-7
Environment variables viii
Error messages xvi
Errors 4-12
ESQL/C 4-1
EXECUTE FUNCTION statement 4-2, 4-8
EXECUTE PROCEDURE statement 4-2, 4-8
Execution context 4-7

F

FETCH statement 3-3, 4-2, 4-3, 4-4
file.encoding property 4-10, 5-3
Files
  .ser 5-1, 5-7, 5-12
  ConnectionManager.java 3-1
  ifxjdbc.jar 5-2
  ifxsqlj.jar 5-2
  ifxtools.jar 5-2
  iterator_name.class 5-1
  java.properties 4-10
  profilekeys.class 5-1
  Property files 5-9
  SQLChecker.cache 5-5
  sqlj.properties 5-9
Fixed and Known Defects File xv
Functions 4-8

G

getExecutionContext() method 4-7
getMaxFieldSize() method 4-8
getMaxRows() method 4-8
getQueryTimeout() method 4-8
getSQLWarnings() method 4-8
getUpdateCount() method 4-8
GLS 5-3

H

Help xvii
Host variables 3-2, 4-1, 4-3

I

IBM Informix JDBC Driver 1-2, 2-1, A-1
ifxjdbc.jar file 5-2
ifxprofp tool 5-12
ifxsqlj command 5-1
ifxsqlj.jar file 5-2
ifxtools.jar file 5-2
Industry standards, compliance with xx
Informix database servers 2-1
Informix Dynamic Server documentation set xvii
INFORMIXSERVER environment variable A-2
initContext() method 3-2, 3-4, A-1
Installation Guides xiv
Internal resource files 5-1
IP addresses, setting in database URLs A-2
isClosed() method 4-7
Iterator objects 3-3, 3-4, 4-2, 4-4
iterator_name.class file 5-1

J

Java compiler 1-1
Java Development Kit (JDK) 2-1
Java interpreter 5-2
Java types 4-8
java.properties file 4-10
JDBC 1-1, 1-2, 4-11, 5-6, 5-7

K

Keywords
  in syntax diagrams xii

L

Language character sets 4-10
Latin-1 character set 4-10
Line numbers 5-4

M

Machine notes xv
main() method 3-4
MultiConnect.sqlj program A-3
Multiple database connections 4-1

N

Name-value pairs of database URLs A-2
Named iterators 3-3, 4-5
native2ascii tool 4-10
newConnection() method A-1
next() method 4-6, 4-7
Nondefault connections A-3
Null data 4-2
Null indicator variables 4-10

O

Off-line checking 5-11
On-line checking 5-7, 5-8, 5-11
Online checking 5-6
Online help xvii
Online manuals xvii
Online notes xiv, xv
Output directory 5-3

P

Passwords, setting in database URLs A-2
PATH environment variable 5-6
Port numbers, setting in database URLs A-2
Positional iterators 3-3, 4-4
Positioned updates, deletes 4-7
Preprocessing source code 5-1

Printed manuals xvii
profilekeys.class file 5-1
Property files 5-9

R

README file 2-2, B-1
Release Notes xv
Reserved names 4-11
Result sets 3-3, 4-4
Root output directory 5-3
rowCount() method 4-6
Running Embedded SQLJ programs 5-1

S

Sample programs 2-2, 3-4, B-1
Sample-code conventions xiii
Schema checking 1-3
Screen reader
  reading syntax diagrams C-1
SELECT statement 3-3
SELECT...AS statement 4-6
SELECT...INTO statement 3-2, 4-3
Semantics checking 1-2, 5-11
Servers 2-1
setMaxFieldSize() method 4-8
setMaxRows() method 4-8
setQueryTimeou() method 4-8
setUpdateCount() method 4-8
Software dependencies vii
Specifying environment variables A-2
SPL routines 4-8
SQL code xiii
SQL statements 3-2
SQL types 4-8
SQL92 Entry level 4-2
SQLChecker.cache file 5-5
SQLException class 4-11
SQLException methods 4-12
SQLJ consortium 1-1
SQLJ runtime package 4-11
SQLJ translator 1-1, 4-11, 5-1
sqlj.properties file 5-9
sqlj.semantics.JdbcChecker class 5-6
sqlj.semantics.OfflineChecker class 5-6
Stored functions 4-8
Syntax checking 1-2, 5-11
Syntax diagrams
  conventions for ix
  keywords in xii
  reading in a screen reader C-1
  variables in xii
Syntax segment xi
System requirements vii

T

TOC Notes xv
Translating source code 5-1
Type checking 1-2, 4-5, 4-6
Type mappings 4-8
Typographical conventions vii

U

Unicode escape sequences 4-10
Updates, positioned 4-7
User names, setting in database URLs A-2

V

Variables, in syntax diagrams xii
Visual disabilities
  reading syntax diagrams C-1

W

WHENEVER...GOTO/CONTINUE statement 4-2
WHERE CURRENT OF clause 4-7

Printed in USA

G251-2278-00