IBM Informix Embedded SQLJ

User's Guide

Table of Contents

Introduction

In This Introduction

This introduction provides an overview of the information in this manual and describes the conventions it uses. Chapter 1, "Introducing Informix Embedded SQLJ," provides an introduction to the Informix Embedded SQLJ product.

About This Manual

This manual contains information about using Informix Embedded SQLJ. This section discusses the organization of the manual, the intended audience, and the associated software products that you must have to use Informix Embedded SQLJ.

Organization of This Manual

This manual includes the following chapters:

- Chapter 1, "Introducing Informix Embedded SQLJ," introduces the Informix Embedded SQLJ product.

- Chapter 2, "Preparing to Use Embedded SQLJ," 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," provides an overview of the Embedded SQLJ language and demonstrates its use with a simple program.

- Chapter 4, "The Embedded SQLJ Language," provides detailed information about the Embedded SQLJ language.

- Chapter 5, "Processing Embedded SQLJ Source Code," explains how to use the SQLJ translator and how to compile and run your Embedded SQLJ programs.
- Appendix A, "Connecting to Databases," provides background information and further details about how an Embedded SQLJ program connects to a database.
- Appendix B, "Sample Programs," provides a table describing the sample programs included with Embedded SQLJ.

An index follows at the end of the manual.

Types of Users

This guide is for programmers who want to write Java 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 *Getting Started* manual for your database server for a list of supplementary titles.

Software Dependencies

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

- Informix Dynamic Server 2000, Version 9.2
- Informix Dynamic Server with Universal Data Option, Version 9.x
- Informix Dynamic Server with Advanced Decision Support and Extended Parallel Options, Version 8.x
- Informix Dynamic Server, Version 7.x
- Informix Dynamic Server, Workgroup and Developer editions, Version 7.x
- INFORMIX-OnLine Dynamic Server, Version 5.x
- INFORMIX-SE, Versions 5.x to 7.2x

To enable your programs to connect to the server, you must use 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 *Informix JDBC Driver Programmer's Guide* for information about using Global Language Support (GLS) with Informix JDBC Driver.

Documentation Conventions

This section describes the conventions that this manual uses. The following conventions are discussed:

- Typographical conventions
- Icon conventions

Typographical Conventions

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

Convention	Meaning
KEYWORD	All primary elements in a programming language statement (keywords) appear in uppercase letters in a serif font.
italics *italics* *italics*	Within text, new terms and emphasized words appear in italics. Within syntax and code examples, variable values that you specify appear in italics.
boldface ***boldface***	Names of program entities (such as classes, events, and tables), environment variables, file and pathnames appear in boldface.
KEYSTROKE	Keys that you are to press appear in uppercase letters in a sans serif font.
♦	This symbol indicates the end of product- or platform-specific information.

Icon Conventions

Throughout the documentation, you will find text that is identified by two different types of icons:

- Comment icons
- Platform icons

Comment Icons

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

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

Platform Icons

Platform icons identify paragraphs that contain platform-specific information.

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

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

Additional Documentation

This section describes the following parts of the documentation set:

- Related manuals
- Documentation notes and release notes
- Vendor-specific documentation

Related Manuals

The following publications provide related information about the topics discussed in this manual:

- *Informix JDBC Driver Programmer's Guide.* Your programs must use Informix JDBC Driver to connect to Informix databases.
- The *Informix Guide to SQL: Tutorial, Informix Guide to SQL: Reference*, and *Informix Guide to SQL: Syntax* provide syntax and reference information about using SQL with Informix databases.

Documentation Notes and Release Notes

The following on-line files supplement the information in this manual.

On-Line File	Purpose
Documentation notes	Describe features not covered in the manual or modified since publication.
Release notes	Describe any special actions required to configure and use Informix Embedded SQLJ. This file contains information about any known problems and their workarounds.

The on-line files are located in the following directory:

IFXJLOCATION/doc/release/sqlj

IFXJLOCATION refers to the directory where you chose to install Embedded SQLJ.

Vendor-Specific Documentation

For more information about the Java language, refer to the JavaSoft Web site at http://www.javasoft.com/.

Informix Welcomes Your Comments

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

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

Write to us at the following address:

> Informix Software, Inc.
> Technical Publications Department
> 300 Lakeside Drive, Suite 2700
> Oakland, CA 94612

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

> doc@informix.com

We appreciate your suggestions.

Introducing Informix Embedded SQLJ

In This Chapter

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

What Is Embedded SQLJ?

Informix Embedded SQLJ enables you to embed SQL statements in your Java programs. 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.

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 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.

SQLJ Source	Java Source	Java .class file
Java source code with embedded SQL	Java source code with calls to JDBC	Java byte codes with calls to JDBC

SQLJ translator

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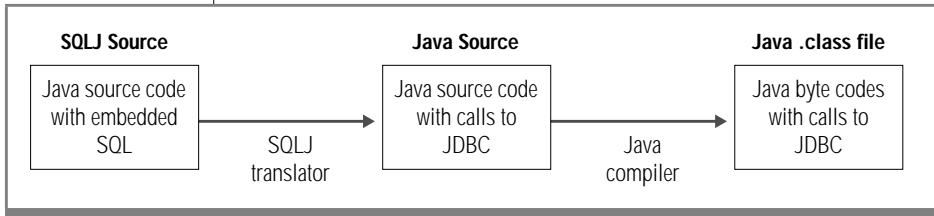
When you run your program, it uses Informix JDBC Driver to connect to an Informix database, as shown in Figure 1-2.
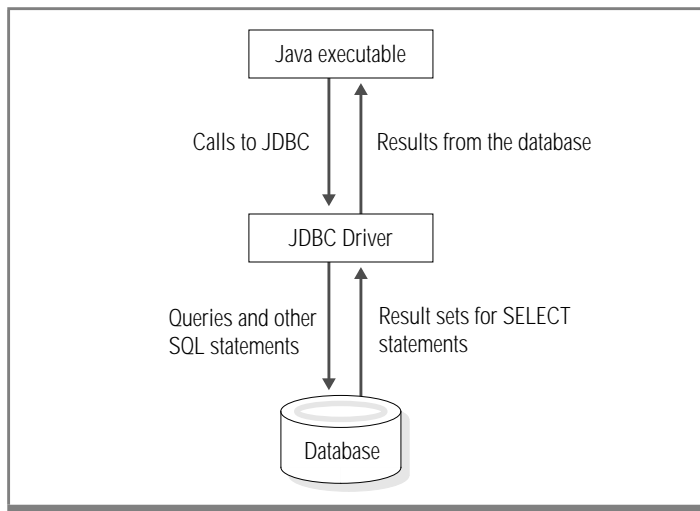
Java executable

Calls to JDBC   Results from the database

JDBC Driver

Queries and other SQL statements   Result sets for SELECT statements

Database

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

Preparing to Use Embedded SQLJ

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:

- 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*)
- 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:
    - Informix Dynamic Server 2000, Version 9.2
    - Informix Dynamic Server with Universal Data Option, Version 9.x
    - Informix Dynamic Server with Advanced Decision Support and Extended Parallel Options, Version 8.x
    - Informix Dynamic Server, Version 7.x
    - Informix Dynamic Server, Workgroup and Developer editions, Version 7.x
    - INFORMIX-OnLine Dynamic Server, Version 5.x
    - 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. (See the JavaSoft Web site, at http://www.javasoft.com/, for more information about the Java language.)

See the *Informix JDBC Driver Programmer's Guide* for further information about installing and using Informix JDBC Driver.

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

Examples

Informix Embedded SQLJ includes sample on-line 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 Informix Embedded SQLJ and Informix JDBC Driver are correctly installed. The examples in this manual are taken from these sample programs.

Building an Embedded SQLJ Program

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," 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-2.

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

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 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 3-8. 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 **Connection-Manager** class.

5. Run queries.

6. Create an iterator object and populate it by running a query.

7. Handle the results.

8. Close the iterator.

```
/****************************************************************************
 *
 *                       INFORMIX SOFTWARE, INC.
 *
 *                          PROPRIETARY DATA
 *
 *      THIS DOCUMENT CONTAINS TRADE SECRET DATA WHICH IS THE PROPERTY OF
 *      INFORMIX SOFTWARE, INC.  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 INFORMIX SOFTWARE, INC.
 *
 *      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) { }
    }
}
```

The Embedded SQLJ Language

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 *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/I. 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.

- 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 4-7), or an accessor method (described in "Named Iterators" on page 4-8) 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."

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 on-line checking, the SQLJ translator checks that the order, number, and types of the SQL expressions and host variables match. "On-Line Checking" on page 5-14 explains how to perform on-line checking.

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 on-line checking. See "On-Line Checking" on page 5-14 for information about setting up on-line checking. See "SQL and Java Type Mappings" on page 4-13 for a listing of SQL and Java type mappings.

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 3-8. 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 on-line checking. See "On-Line Checking" on page 5-14 for information about setting up on-line checking.

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. An example of how to use the **next()** method appears in "Named Iterators" on page 4-8.

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 **get*attribute*** that reads the value of the attribute, and a method called **set*attribute*** 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 num = {EXECUTE FUNCTION( func_name(func_arg) ) };
```

SQL and Java Type Mappings

When you retrieve data from a database into an iterator object (see "Handling Result Sets" on page 4-6) 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 *
LVARCHAR	String
MONEY	java.math.BigDecimal
NCHAR, NVARCHAR	String
SERIAL	int
SERIAL8	long
SMALLFLOAT	double

(1 of 2)

SQL Type	Java Type
SMALLINT	short
TEXT	String
VARCHAR	String
* IfxIntervalYM and IfxIntervalDF are Informix extensions to JDBC 2.0.	

(2 of 2)

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
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

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. Refer to your JDBC API documentation for information about using **SQLException** methods to obtain information about these errors.

Processing Embedded SQLJ Source Code

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 translation and compilation options.
- how to use property files.
- how to perform on-line checking.

Translating, Compiling, and Running Embedded SQLJ Programs

You use the command **java ifxsqlj** to create executable Java **.class** files from your Embedded SQLJ source code.

When you run the **java ifxsqlj** command with an **.sqlj** source file, the source file is processed in two stages. In the first stage, called *translation*, the SQLJ translator creates a Java source file (with the extension **.java**); for example, when you process a file called **File1.sqlj**, the SQLJ translator creates a file called **File1.java**. The SQLJ translator also creates internal resource files with the extension **.ser**.

In the second stage of processing, the SQLJ translator passes **.java** files to a Java compiler. Compilation creates files with the extension **.class**; in this example, your compiled Java program is called **File1.class**. An internal resource file named ***profilekeys*.class** is also created. If your program includes an iterator, a file called ***iterator_name*.class** is produced.

*Tip: To perform translation only, execute the **java ifxsqlj** command with the
-**compile** option set to* FALSE*. For information about the -**compile** option, see
"Advanced Options" on page 5-8.*

To create a complete application, you must include the directories that
contain the SQLJ runtime classes in **sqlj.runtime.*** in your **CLASSPATH**
environment variable definition. The SQLJ runtime files are available in
ifxsqlj.jar, the file that you installed when you first installed the Embedded
SQLJ product, as described in "Setting Up Your Software" on page 2-4.

In addition, you must include the locations of **ifxtools.jar** and the relevant
version of the JDK in your **CLASSPATH** definition. At runtime, you must also
include the location of **ifxjdbc.jar**; however, you do not need to include this
file location when translating or compiling your application.

You run your Embedded SQLJ program like any other Java program: by using
the Java interpreter, as follows:

```
java File1
```

The ifxsqlj Command

You use the **java ifxsqlj** command to transalate and compile your Embedded
SQLJ source code, as described above. You run the **java ifxsqlj** command at
the DOS or UNIX prompt.

The syntax of the **java ifxsqlj** command is as follows:

```
java ifxsqlj optionlist filelist
```

optionlist   A set of options separated by spaces. Some options have prefixes to indicate they are to be passed to other utilities than the SQLJ translator, such as the Java compiler.

filelist   A list of filenames separated by spaces: for example,
`file1.sqlj file2.sqlj`

You must include the absolute or relative path to the files in *filelist*.

The files can have the extension **.sqlj** or **.java**. You can specify **.sqlj** files together with **.java** files on the same command line.

If you have **.sqlj** and **.java** files that require access to code in each other's file, enter all of these files on the command line for the same execution of the **java ifxsqlj** command.

You can use an asterisk (*) as a wildcard to specify filenames: for example, **c*.sqlj** processes all files beginning with **c** that have the extension **.sqlj**.

When you run the **java ifxsqlj** command, your **CLASSPATH** environment variable must be set to include any directories that contain **.class** files and **.ser** files the translator needs to access for type resolution of variables in your Embedded SQLJ source code.

Command Options

Many options are available to customize how you run the **java ifxsqlj** command:

- Basic options are described in the next section.
- Advanced options are described on "Advanced Options" on page 5-**8**.

You can set options either on the command line or in property files. Options set on the command line can be passed to the SQLJ translator, the Java compiler, or the Java interpreter. Options set in property files can be passed to the SQLJ translator or the Java compiler, but not to the Java interpreter. For more information, see "Setting Options on the Command Line" on page 5-11 and "Supplying Options in Property Files" on page 5-12.

Basic Options

The following table lists the basic options available for use with the **java ifxsqlj** command.

Option	Description
-d	Specifies the root output directory for generated **.ser** and **.class** files. If you do not specify this option, files are generated under the directory of the input **.sqlj** file.
-dir	Specifies the root output directory for generated **.java** files. If you do not specify this option, files are generated under the directory of the input **.sqlj** file.
-encoding	Specifies the GLS encoding for **.sqlj** and **.java** input files and for **.java** generated files. If unspecified, the setting of the **file.encoding** property for the Java interpreter is used.
	The -**encoding** option is also passed to the Java compiler.
-help	Displays option names, descriptions, and current settings. The list displays:
	■ the name of the option.
	■ the type of the option (for example, if it is Boolean) or a selection of allowed values.
	■ the current value.
	■ a description of the option.
	■ whether the property is at its default, or was set by either a property file or the command line.
	No translation or compilation is performed when you specify the -**help** option.
-linemap	Enables the mapping of line numbers between the generated **.java** file and the original **.sqlj** file. The -**linemap** option is useful for debugging because it allows you to trace compilation and execution errors back to your Embedded SQLJ source code.
	For the -**linemap** option to be effective, the name of the **.sqlj** source code file must match the name of the class it implements.

(1 of 2)

Option	Description
-props	Specifies the name of the property file from which to read options. "The ifxprofp Tool" on page 5-15 explains how to use property files.
-status	Displays status messages while the **java ifxsqlj** command is running.
-version	Displays the version of Embedded SQLJ you are using.
	No translation or compilation is performed when you specify the **-version** option.
-warn	Specifies a list of flags in a comma-separated string for controlling the display of warning and information messages during translation. The flags are:
	▪ **all**/**none**. Turns on or off all warnings and information messages.
	▪ **null**(default)/**nonull**. Specifies whether the translator checks nullable columns and nullable Java variable types for conversion loss when data is transferred between database columns and Java host variables. The translator must connect to the database for this option to be in effect.
	▪ **precision**(default)/**noprecision**. Specifies whether the translator checks for loss of precision when data is transferred between database columns and Java variables. The translator must connect to the database for this option to be in effect.
	▪ **portable**(default)/**noportable**. Turns on or off warning messages about the portability of Embedded SQLJ statements.
	▪ **strict**(default)/**nostrict**. Specifies whether the translator checks named iterators against the columns returned by a SELECT statement and issues a warning for any mismatches. The translator must connect to the database for this option to be in effect.
	▪ **verbose**(default)/**noverbose**. Turns on or off additional information messages about the translation process. The translator must connect to the database for this option to be in effect.
	For example, the following setting of the **-warn** option turns off all warnings and then turns on the precision and nullability checks:
	``` -warn=none,null,precision ```

(2 of 2)

Advanced Options

The following table lists the advanced options available for use with the **java ifxsqlj** command. Many of these options are for on-line checking, which is discussed in "On-Line Checking" on page 5-14.

Option	Description
-cache	Turns on the caching of results from on-line checking. Caching saves you from unnecessary connections to the database in subsequent runs of the translator for the same file.
	Results are written to the file **SQLChecker.cache** in your current directory. The cache holds serialized representations of all SQL statements that translated without errors or warnings. The cache is cumulative and grows through successive invocations of the translator.
	You empty the cache by deleting the **SQLChecker.cache** file.
	Caching is off by default; you turn caching on by setting the -**cache** option to `true`, `1`, **or** `on`: for example, `-cache=true`. **You turn caching off by setting the option to** `false`, `0`, **or** `off`.
-compile	Setting this flag to `false` disables processing of **.java** files by the compiler. This applies to generated **.java** files and to **.java** files specified on the command line.
-compiler-executable	Specifies a particular Java compiler for the **java ifxsqlj** command to use. If unset, the translator uses **javac**. If you do not specify a directory path, the **java ifxsqlj** command searches for the executable according to the setting of your **PATH** environment variable.
-compiler-encoding-flag	Set this flag to `false` to prevent the value of the SQLJ -**encoding** option from being automatically passed to the compiler.
-**compiler-output-file**	If you have instructed the Java compiler to output its results to a file, use the -**compiler-output-file** option to specify the filename.
-driver	Specifies a list of JDBC drivers that can be used to interpret JDBC connection URLs for on-line checking (see "On-Line Checking" on page 5-14). You specify a class name or a comma-separated list of class names. For example, specify Informix JDBC Driver as follows:
	`-driver=com.informix.jdbc.IfxDriver`

(1 of 4)

Option	Description
-offline	Specifies a Java class to implement off-line checking. The default off-line checker class is **sqlj.semantics.OfflineChecker**.
	Off-line checking only runs when on-line checking does not (either because on-line checking was not enabled or because it stopped because of error). Off-line checking verifies SQL syntax and the usage of Java types.
	With off-line checking, there is no connection to the database.
-online	Specifies a Java class or list of classes to implement on-line checking. The default on-line checker class is **sqlj.semantics.JdbcChecker**.
	You can specify an on-line checker class for a particular connection context, as in:
	`-online@ctxclass2=sqlj.semantics.JdbcChecker`
	You must specify a user name with the -**user** option for on-line checking to occur. The -**password**, -**url**, and -**driver** options must be appropriately set as well.
-password	Specifies a password for the user name set with the -**user** option. If you specify the -**user** option, but not the -**password** option, the translator prompts you for the password.
	If you are using multiple connection contexts, the setting for -**password** for the default connection context also applies to any connection context that does not have a specific setting.
-ser2class	Setting this flag to `true` converts the generated **.ser** files to **.class** files. This is necessary if you are creating an applet to be run from a browser, such as Netscape 4.0, that does not support loading a serialized object from a resource file.
	The original **.ser** file is not saved.

(2 of 4)

Option	Description

-url	Specifies a JDBC URL for establishing a database connection for on-line checking (see "Database URLs" on page A-2 and "On-Line Checking" on page 5-14). The URL can include a host name, a port number, and an Informix database name. The format is:
	
	`jdbc:informix-sqli://{<ip-address>|`
	`    <domain-name>}:<port-number>[/<dbname>]:`
	`    INFORMIXSERVER=<server-name>[;user=<username>;`
	`    password=<password>;<name>=<value>`
	`    [;<name>=<value>]...]`
	
	If you are using multiple connection contexts, the setting for -**url** for the default context also applies to any connection context that does not have a specific setting.
	
	You can specify a URL for a particular connection context, as in `-url@ctxclass2=....`
	
	Any connection context with a URL must also have a user name set for it (using the -**user option**) for on-line checking to occur.

-user	Enables on-line checking and specifies the user name with which the translator connects to the database (see "On-Line Checking" on page 5-14).
	
	For example, to enable on-line checking on the default connection context and connect with the user name **fred**, use the following option:
	
	`-user=fred`
	
	If you are using multiple connection contexts, the setting for -**user** for the default connection context also applies to any connection context that does not have a specific setting.
	
	If you want to enable on-line checking for the default context, but turn off on-line checking for another connection—for example *ctxcon2*—you need to specify the -**user** option twice:
	
	`-user=fred -user@ctxcon2=`
	
	To enable on-line checking for a particular connection context, specify that context with the user name, as in:
	
	`-user@ctxcon3=joyce`
	
	The classes of the connection contexts you specify must all be declared in your source code or previously compiled into a **.class** file.

(3 of 4)

Option	Description
-vm	Specifies a particular Java interpreter for the **java ifxsqlj** command to use. You must also include the path to the interpreter. If you do not specify a particular Java interpreter using this option, the translator uses **java** as a default. The -**vm** option must be specified on the command line; you cannot set it in a property file.

(4 of 4)

Setting Options

You specify options for the **java ifxsqlj** command either on the command line or in a property file. Command line options are discussed in "Setting Options on the Command Line" on page 5-11. Property files are discussed in "Supplying Options in Property Files" on page 5-12.

For Boolean options (those that are either on or off), you can set the option simply by specifying the option name: for example, -linemap. You can also set the option to TRUE, as in -linemap=true. To turn off a Boolean option, you must set it to FALSE: for example, -linemap=false. You can also set Boolean options to yes or no, or to 1 or 0.

Setting Options on the Command Line

Options on the command line override any options set in default files. If the same option appears more than once on the command line, the translator uses the final (rightmost) option's value.

Command-line option names are case sensitive.

You can attach prefixes to options to pass the option to the Java compiler or to the Java interpreter. If you do not use a prefix, the option is passed to the SQLJ translator.

The prefixes are:

- **-C**

  Passes compiler options to the Java compiler, as shown in the following example:

  ```
  -C-classpath=/user/jdk/bin
  ```

- **-J**

  Passes interpreter options to the Java interpreter, as shown in the following example:

  ```
  -J-Duser.language=ja
  ```

  The options available to pass to the interpreter depend on the release and brand of Java you are using.

Do not use the **-C** prefix with the **-d** and **-encoding** options; when you specify these SQLJ translator options, they are automatically passed to the Java compiler.

Supplying Options in Property Files

You can use property files to supply options to the **java ifxsqlj** command. The default name of a property file is **sqlj.properties**; you can specify a different name by using the **-props** option on the command line (see "Basic Options" on page 5-6).

You cannot use a property file to specify:

- the **-props**, **-help**, and **-version** basic options.
- the **-vm** advanced option.
- options with the prefix **-J** (for passing options to the Java interpreter).

Precedence of Options

The **java ifxsqlj** command checks for the existence of files called **sqlj.properties** in the following directories in the following order:

1. The Java home directory
2. Your home directory
3. The current directory

The translator processes each property file it finds and overrides any previously set option if it finds a new setting for that option.

Later entries in the same property file override earlier entries.

Options on the command line override options set by property files.

If you set options on the command line or in a property file specified using the -**props** option, these options override any options set in **sqlj.properties** files.

Format of Property Files

In a property file, you

- specify one option per line.
- begin a line with the symbol # to denote a comment.

Tip: The translator ignores empty lines.

The syntax for specifying options is the same as shown in "Command Options" on page 5-5, except you replace the initial hyphen with a string followed by a period that indicates to which utility the option is passed.

You can pass options to the SQLJ translator or the Java compiler; however, you cannot pass options to the Java interpreter from a property file. The strings for specifying utilities are as follows.

Precede an option with…	To pass it to this utility…
sqlj.	SQLJ translator
compile.	Java compiler

An example property file looks like this:

```
# Turn on online checking and specify the user to connect with
sqlj.user=joyce
sqlj.password=*******
# JDBC Driver to connect with
sqlj.driver=com.informix.jdbc.IfxDriver
# Database URL
sqlj.url=jdbc:<ipaddr>:<portno>/demo_isqlj:informixserver=<$INFORMIXSERVER>
# Instruct the compiler to output status messages during compile
compile.verbose
```

On-Line Checking

On-line checking analyzes the validity of the embedded SQL statements against the database schema (user name, password, and database) you specify.

On-line checking performs the following operations:

- Passes SQL data manipulation statements (DML) to the database to verify their syntax and semantics and their validity for the database schema.
- Checks stored procedures and functions for overloading.
- Runs the checks covered by off-line checking.

Off-line checking verifies SQL syntax and usage of Java types; there is no connection to a database for off-line checking.

To set up on-line checking, you use the following options with the **java ifxsqlj** command, or set them in a property file: -**user**, -**password**, -**url**, and -**driver**. These options are described in "Advanced Options" on page 5-**8**.

Setting the -user and -password Options

You enable on-line checking by setting the -**user** option. The -**user** option also supplies the user name for the database connection to be used for checking. You do not have to specify the same database or user name for on-line checking as the application uses at runtime.

In the simplest case, you supply a user name with the -**user** option, and on-line checking is performed using the default connection context, as in:

```
-user = joyce
```

You can supply the password for the user name by using the -**password** option or by combining the password with the user name: for example, `-user = joyce/jcs123` **or** `-user = joyce -password =jcs123`.

To disable on-line checking on the command line, set the -**user** option to an empty value (as in `-user=`) or omit the option entirely. To disable on-line checking in a property file, comment out the line specifying **sqlj.user**.

To enable on-line checking against a nondefault connection context, you specify the connection context with the user name in the -**user** option. In the following example, the SQLJ translator connects to the database specified in the connection-context object, *conctx*, using the user name **fred**:

```
-user@conctx = fred
```

Setting the -url and -driver Options

The -**url** option specifies a JDBC URL for establishing a database connection (see "Database URLs" on page A-2).

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

Both of these options are shown in "Advanced Options" on page 5-8.

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
====================================================
```

Connecting to Databases

"Connecting to a Database" on page 3-3 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 3-3. 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 on-line checking are database URLs (see
"On-Line Checking" on page 5-14 for information about on-line checking).
Database URLs specify the subprotocol (the database connectivity
mechanism), the database or server identifier, and a list of properties.

Your Embedded SQLJ program uses Informix JDBC Driver to connect to an
Informix database. Informix JDBC Driver supports database URLs of the
following format:

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

In the preceding syntax:

- curly brackets ({}) together with vertical lines (|) denote more than
  one choice of variable.
- *italics* denote a variable value.
- brackets (**[]**) denote an optional value.
- words or symbols not enclosed in brackets are required
  (INFORMIXSERVER=, for example).

Important: 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.

(1 of 2)

Database URL Variable	Required?	Description
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 Informix JDBC Driver or Informix database servers. The value of *name* is case insensitive. Refer to the *Informix JDBC Driver Programmer's Guide* for information about environment variables supported by Informix JDBC Driver and how to set them.

(2 of 2)

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 **Multi-Connect.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

2. Create a connection-context object for connecting to the database.

3. 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.

```
/****************************************************************************
 *
 *                          INFORMIX SOFTWARE, INC.
 *
 *                              PROPRIETARY DATA
 *
 *      THIS DOCUMENT CONTAINS TRADE SECRET DATA WHICH IS THE PROPERTY OF
 *      INFORMIX SOFTWARE, INC.  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 INFORMIX SOFTWARE, INC.
 *
 *      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
```

MultiConnect.sqlj

```
{
    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) { }
    }
}
```

```
/***************************************************************************
 *
 *                        INFORMIX SOFTWARE, INC.
 *
 *                            PROPRIETARY DATA
 *
 *       THIS DOCUMENT CONTAINS TRADE SECRET DATA WHICH IS THE PROPERTY OF
 *       INFORMIX SOFTWARE, INC.  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 INFORMIX SOFTWARE, INC.
 *
 *       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 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;
    }
}
```

Sample Programs

The following table lists and describes the on-line sample programs that are included with 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.

Index