Oracle® Database

SQLJ Developer's Guide and Reference

11*g* Release 1 (11.1)

B31227-01

July 2007

ORACLE®

Oracle Database SQLJ Developer's Guide and Reference, 11*g* Release 1 (11.1)

B31227-01

Primary Author: Tulika Das

Contributing Author: Venkatasubramaniam Iyer, Brian Wright, Janice Nygard

Contributor: Quan Wang, Ekkehard Rohwedder, Amit Bande, Krishna Mohan, Amoghavarsha Ramappa, Brian Becker, Alan Thiesen, Lei Tang, Julie Basu, Pierre Dufour, Jerry Schwarz, Risto Lakinen, Cheuk Chau, Vishu Krishnamurthy, Rafiul Ahad, Jack Melnick, Tim Smith, Thomas Pfaeffle, Tom Portfolio, Ellen Barnes, Susan Kraft, Sheryl Maring, Angela Barone

Contents

3 Key Programming Considerations

4 Basic Language Features

5 Type Support

6 Objects, Collections, and OPAQUE Types

7 Advanced Language Features

8 Translator Command Line and Options

9 Translator and Run Time Functionality

10 Performance and Debugging

11 SQLJ in the Server

A Customization and Specialized Customizers

List of Tables

Preface

This preface introduces you to the *Oracle Database SQLJ Developer's Guide and Reference*, discussing the intended audience and conventions of this document. A list of related Oracle documents is also provided.

Audience

This manual is intended for anyone with an interest in SQLJ programming but assumes at least some prior knowledge of the following:

- Java
- SQL
- PL/SQL
- Oracle Database

Although general knowledge of SQL is sufficient, any knowledge of JDBC and Oracle-specific SQL features would be helpful as well.

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible, with good usability, to the disabled community. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Accessibility standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For more information, visit the Oracle Accessibility Program Web site at

`http://www.oracle.com/accessibility/`

Accessibility of Code Examples in Documentation

Screen readers may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, some screen readers may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation

This documentation may contain links to Web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

TTY Access to Oracle Support Services

Oracle provides dedicated Text Telephone (TTY) access to Oracle Support Services within the United States of America 24 hours a day, seven days a week. For TTY support, call 800.446.2398.

Related Documents

Also available from the Oracle Java Platform group are the following Oracle resources:

- *Oracle Database Java Developer's Guide*

  This book introduces the basic concepts of Java in Oracle Database 11*g* and provides general information about server-side configuration and functionality. Information that pertains to Oracle Database Java environment in general, rather than to a particular product such as JDBC or SQLJ, is in this book.

  It also discusses Java stored procedures, which are programs that run directly in Oracle Database. With stored procedures, Java developers can implement business logic at the server level, thereby improving application performance, scalability, and security.

- *Oracle Database JDBC Developer's Guide and Reference*

  This book covers programming syntax and features of the Oracle implementation of the JDBC standard. This includes an overview of the Oracle JDBC drivers, details of the Oracle implementation of JDBC 1.22, 2.0, 3.0, and 4.0 features, and discussion of Oracle JDBC type extensions and performance extensions.

- *Oracle Database JPublisher User's Guide*

  This book describes how to use the Oracle JPublisher utility to translate user-defined SQL types or PL/SQL packages into Java classes. If you are developing SQLJ or JDBC applications that use object types, VARRAY types, nested table types, object reference types, or PL/SQL packages, then JPublisher can generate custom Java classes to map to them.

The following documents are from the Oracle Server Technologies group:

- *Oracle XML DB Developer's Guide*
- *Oracle XML Developer's Kit Programmer's Guide*
- *Oracle Database XML Java API Reference*
- *Oracle Database Advanced Application Developer's Guide*
- *Oracle Database SecureFiles and Large Objects Developer's Guide*
- *Oracle Database Object-Relational Developer's Guide*
- *Oracle Database PL/SQL Packages and Types Reference*
- *Oracle Database PL/SQL Language Reference*
- *Oracle Database SQL Language Reference*
- *Oracle Database Net Services Administrator's Guide*
- *Oracle Database Advanced Security Administrator's Guide*

- *Oracle Database Globalization Support Guide*
- *Oracle Database Reference*
- *Oracle Database Sample Schemas*

> **Note:** Oracle error message documentation is available in HTML only. If you have access to the Oracle Documentation CD only, you can browse the error messages by range. Once you find the specific range, use the "find in page" feature of your browser to locate the specific message. When connected to the Internet, you can search for a specific error message using the error message search feature of the Oracle online documentation.

For documentation of SQLJ standard features and syntax, refer to the following ANSI specification:

Information Technology - Database Languages - SQL - Part 10: Object Language Bindings (SQL/OLB)

You can obtain this from ANSI through the following Web site:

`http://www.ansi.org/`

(Click "eStandards Store" and search for the specification number.)

The following location has SQLJ sample applications:

`http://www.oracle.com/technology/sample_code/tech/java/sqlj_jdbc/sqlj.html`

Conventions

This section describes the conventions used in the text and code examples of this documentation set. It describes:

- Conventions in Text
- Conventions in Code Examples

> **Note:** Also note that command-line examples are for a UNIX environment with a system prompt of "`%`". This is only by convention and can be adjusted as appropriate for your operating system.

Conventions in Text

There are various conventions in text to help you more quickly identify special terms. The following table describes those conventions and provides examples of their use.

Convention	Meaning	Example
Italics	Italic typeface indicates book titles or emphasis, or terms that are defined in the text.	*Oracle Database Concepts* Ensure that the recovery catalog and target database do *not* reside on the same disk.

Convention	Meaning	Example
UPPERCASE monospace (fixed-width) font	Uppercase monospace typeface indicates elements supplied by the system. Such elements include parameters, privileges, data types, RMAN keywords, SQL keywords, SQL*Plus or utility commands, packages and methods, as well as system-supplied column names, database objects and structures, usernames, and roles.	You can specify this clause only for a NUMBER column. You can back up the database by using the BACKUP command. Query the TABLE_NAME column in the USER_TABLES data dictionary view. Use the DBMS_STATS.GENERATE_STATS procedure.
lowercase monospace (fixed-width) font	Lowercase monospace typeface indicates executables, filenames, directory names, and sample user-supplied elements. Such elements include computer and database names, net service names, and connect identifiers, as well as user-supplied database objects and structures, column names, packages and classes, usernames and roles, program units, and parameter values. **Note:** Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.	Enter sqlplus to open SQL*Plus. The password is specified in the orapwd file. Back up the data files and control files in the /disk1/oracle/dbs directory. The department_id, department_name, and location_id columns are in the hr.departments table. Set the QUERY_REWRITE_ENABLED initialization parameter to true. Connect as oe user. The JRepUtil class implements these methods.
lowercase italic monospace (fixed-width) font	Lowercase italic monospace font represents place holders or variables.	You can specify the *parallel_clause*. Run *old_release*.SQL where *old_release* refers to the release you installed prior to upgrading.

Conventions in Code Examples

Code examples illustrate SQL, PL/SQL, SQL*Plus, or other command-line statements. They are displayed in a monospace (fixed-width) font and separated from standard text as shown in this example:

```
SELECT username FROM dba_users WHERE username = 'MIGRATE';
```

The following table describes typographic conventions used in code examples and provides examples of their use.

Convention	Meaning	Example
< >	In this document, angle brackets are used instead of regular brackets to enclose one or more optional items. Do not enter the angle brackets. (Regular brackets are not used due to SQLJ syntax considerations.)	DECIMAL (*digits* < , *precision* >)
|	A vertical bar represents a choice of two or more options within brackets or braces. Enter one of the options. Do not enter the vertical bar.	{ENABLE | DISABLE} [COMPRESS | NOCOMPRESS]

Convention	Meaning	Example
`...`	Horizontal ellipsis points indicate either: ▪ Omission of parts of the code that are not directly related to the example ▪ That you can repeat a portion of the code	`CREATE TABLE ... AS subquery;` `SELECT col1, col2, ... , coln FROM employees;`
Other notation	You must enter symbols other than brackets, braces, vertical bars, and ellipsis points as shown.	`acctbal NUMBER(11,2);` `acct CONSTANT NUMBER(4) := 3;`
Italics	Italicized text indicates place holders or variables for which you must supply particular values.	`CONNECT SYSTEM/system_password` `DB_NAME = database_name`

`UPPERCASE`	Uppercase typeface indicates elements supplied by the system. These terms are in uppercase to distinguish them from terms you define. Unless terms appear in brackets, enter them in the order and with the spelling shown. However, because these terms are not case-sensitive, you can enter them in lowercase.	`SELECT last_name, employee_id FROM employees;`
		`SELECT * FROM USER_TABLES;`
		`DROP TABLE hr.employees;`

`lowercase`	Lowercase typeface indicates programmatic elements that you supply. For example, lowercase indicates names of tables, columns, or files.
	Note: Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.
		`SELECT last_name, employee_id FROM employees;`
		`sqlplus hr/hr`
		`CREATE USER mjones IDENTIFIED BY ty3MU9;`

What's New in 11g Release 1 (11.1)

The following new SQLJ-related features have been introduced in Oracle Database 11*g* Release 1 (11.1):

- Support for JDK 1.5 and JDK 1.6
- Enhanced Profile Print Option
- Outline Generation
- Support for Global Transactions

Support for JDK 1.5 and JDK 1.6

In Oracle Database 11*g* Release 1 (11.1) SQLJ programs, the following JDK1.5 specific language structures are allowed outside the #SQL region:

JDK 1.5

- strictfp
- Assert
- Generics
- Enhanced for Loop
- Autoboxing/Unboxing
- Typesafe Enums
- Varargs
- Static Import
- Annotation

Note: Visit the following Sun site more information about all the JDK 1.5 features:

`http://java.sun.com/j2se/1.5.0/docs/relnotes/features.html`

JDK 1.6

Starting from Oracle Database 11*g* Release 1 (11.1), SQLJ programs are supported in JDK 1.6.x environment. Refer to Chapter 1, "Getting Started" for more information.

Enhanced Profile Print Option

The SQLJ profile print option has been enhanced to output all the options set on the customizer. This feature is discussed in detail in "Profile Customization (ISO Code Generation)" on page 9-7.

Outline Generation

You can use the outline feature of Oracle to support plan stability. This feature is discussed in detail in "Execution Plan Fixing" on page 7-54.

Support for Global Transactions

The X/Open Distributed Transaction Processing coordinates the work between application programs and resource managers into global transactions. This feature is discussed in detail in"SQLJ Support for Global Transactions" on page 7-17.

1

Getting Started

This chapter guides you through the basics of testing your Oracle SQLJ installation and configuration and running a simple application.

This chapter discusses the following topics:

- Assumptions and Requirements
- Checking the Installation and Configuration
- Testing the Setup

Assumptions and Requirements

This section discusses basic assumptions about your environment and requirements of your system so that you can run SQLJ, covering the following topics:

- Assumptions About Your Environment
- Requirements for Using the Oracle SQLJ Implementation
- SQLJ Environment
- Environment Considerations
- SQLJ Backward Compatibility

Assumptions About Your Environment

The following assumptions are made about the system on which you will be running the Oracle SQLJ implementation:

- You have a standard Java environment that is operational on your system. This would typically be using a Sun Microsystems Java Development Kit (JDK), but other implementations of Java will work. Ensure that you can run Java (typically `java`) and the Java compiler (typically `javac`).

  To translate and run SQLJ applications on a Sun JDK, you must use JDK 1.5.x or JDK 1.6.x. You must use the JDBC driver of the same version as that of SQLJ, can be thin or OCI8 driver

  See Also: "SQLJ Environment" on page 1-2

> **Note:** A Java run-time environment (JRE), such as the one installed with Oracle Database 11*g*, is not by itself sufficient for translating SQLJ programs. However, a JRE *is* sufficient for running SQLJ programs that have already been translated and compiled.

- You can already run JDBC applications in your environment.

  > **See also:** *Oracle Database JDBC Developer's Guide and Reference* for more information about JDBC drivers

Requirements for Using the Oracle SQLJ Implementation

The following are required to use the Oracle SQLJ implementation:

- A database system that is accessible using your JDBC driver
- Class files for the SQLJ translator

  Translator-related classes are available in the following file:

  ORACLE_HOME`/sqlj/lib/translator.jar`

  > **Note:** For more information about `translator.jar`, refer to "Set the Path and Classpath" on page 1-4.

- Class files for the SQLJ run time.

  ORACLE_HOME`/sqlj/lib/runtime12.jar`

  > **Note:** `runtime12ee.jar` has been deprecated in Oracle Database 11*g* Release 1 (11.1). Use `runtime12.jar` instead.

SQLJ Environment

To ensure that you have a fully working environment, you must consider several aspects of your environment: SQLJ and its code generation mode, JDBC, and the JDK.

> **Note:** Code generation is determined by the SQLJ `-codegen` option. Refer to "Code Generation (-codegen)" on page 8-41 for more information.

The following is a typical environment setup for Oracle-specific code generation:

- SQLJ code generation: `-codegen=oracle` (default)
- SQLJ translation library: `translator.jar`
- SQLJ run time library: `runtime12.jar`
- JDBC drivers: Oracle 11*g* release 1 (11.1)
- JDK version: 1.5.x or 1.6.x

Note: If you are running against different JDBC versions, then translate against the earlier version.

Environment Considerations

You can run the application against a JDK version that is at least as high as the version you translated the code under.

Note: For more information about `translator.jar`, refer to "Set the Path and Classpath" on page 1-4.

SQLJ Backward Compatibility

You must keep in mind the following points regarding backward compatibility of the Oracle SQLJ implementation:

- Code generated with an earlier release of the SQLJ translator can continue to run and compile against current run time libraries. However, this is subject to the cross-compatibility limitations discussed in "Environment Considerations" on page 1-3.
- Oracle-specific translator output, that is, code generated with the default `-codegen=oracle` setting, must be created and executed using the `runtime12.jar` library. In addition:
  - Such code will be executable under future Oracle JDBC and SQLJ implementations.
  - Such code, however, will *not* be executable under earlier releases of Oracle JDBC drivers and the Oracle SQLJ run time. In these circumstances, you will have to retranslate the code.

Checking the Installation and Configuration

After you have verified that the preceding assumptions and requirements are satisfied, you must check your SQLJ installation. You must:

- Check for Availability of SQLJ and Demo Applications
- Check for Installed Directories and Files
- Set the Path and Classpath
- Verify Installation of the sqljutl Package

Check for Availability of SQLJ and Demo Applications

Following are the release-specific notes regarding availability of SQLJ and its demo applications:

- SQLJ and its demo applications are available from the Oracle Technology Network (OTN) at the following location:

  `http://www.oracle.com/technology/sample_code/tech/java/sqlj_jdbc/sqlj.html`

- For Oracle Database 11*g*, SQLJ and its demo applications are included with the installation.

Check for Installed Directories and Files

Verify that the following directories have been installed and are populated:

Directories for JDBC

Refer to the *Oracle Database JDBC Developer's Guide and Reference* for information about JDBC files that should be installed on your system.

Directories for SQLJ

Installing the Oracle Database 11*g* Java environment includes, among other things, installing a `sqlj` directory under your *`ORACLE_HOME`* directory. The `sqlj` directory contains the following subdirectories:

- `demo` (demo applications, including some referenced in this chapter)
- `lib` (`.jar` files containing class files for SQLJ)

Check whether all these directories have been created and populated, especially `lib`.

The *`ORACLE_HOME`*`/bin` directory contains utilities for all Java product areas, including the SQLJ and JPublisher executable files.

Set the Path and Classpath

Ensure that the `PATH` and `CLASSPATH` environment variables have the necessary settings for the Oracle SQLJ implementation. Set the `PATH` and `CLASSPATH` environment variables as follows for the Oracle SQLJ implementation:

- Setting `PATH`

  To run the `sqlj` script, which invokes the SQLJ translator, without having to fully specify its path, verify that the `PATH` environment variable has been updated to include the following:

  `ORACLE_HOME``/bin`

  Use backslash (`\`) for Microsoft Windows. Replace *`ORACLE_HOME`* with your actual Oracle home directory.

- Setting `CLASSPATH`

  Update the `CLASSPATH` environment variable to include the current directory as well as the following:

  `ORACLE_HOME``/sqlj/lib/translator.jar`

  Use backslash (`\`) for Microsoft Windows. Replace *`ORACLE_HOME`* with your actual Oracle home directory.

  Include the following run time library in the `CLASSPATH`:

  `ORACLE_HOME``/sqlj/lib/runtime12.jar`

  In addition, you must include one of the following JDBC JARs in the `CLASSPATH`:

  ```
  ORACLE_HOME/jdbc/lib/ojdbc5*.jar
  ORACLE_HOME/jdbc/lib/ojdbc6*.jar
  ```

Note: To translate or run SQLJ programs in JDK 1.5.x environment, you should have `ojdbc5.jar` in the classpath and to translate or run SQLJ programs in JDK 1.6.x environment, you should have `ojdbc6.jar` in the classpath. Ensure that the correct JDBC JAR is picked up at runtime for connecting to Oracle Database.

See Also: "Requirements for Using the Oracle SQLJ Implementation" on page 1-2

Note: You will not be able to run the SQLJ translator if you do not add a run time library. You must specify a run time library as well as the translator library in the `CLASSPATH`.

To see if SQLJ is installed correctly, and to see the version information for SQLJ, JDBC, and Java, run the following command:

```
% sqlj -version-long
```

Verify Installation of the sqljutl Package

The `sqljutl` package is required for online checking of stored procedures and functions in Oracle Database instance. Beginning with Oracle8*i* Database release 8.1.5, it is installed automatically under the `SYS` schema during installation of the server-side Java virtual machine (JVM) for a Java-enabled database. If your database is not Java-enabled, then you will have to manually install this package.

If you want to verify the installation of `sqljutl`, then issue the following SQL command from SQL*Plus:

```
describe sys.sqljutl
```

This should result in a brief description of the package.

If you get a message indicating that the package cannot be found, or if you want to install an updated version of the package, then you can install it by using SQL*Plus to run the `sqljutl.sql` script (or `sqljutl8.sql` for Oracle8*i* Database), which is located at:

ORACLE_HOME`/sqlj/lib/sqljutl.sql`

Testing the Setup

You can test your database, JDBC, and SQLJ setup using demo applications defined in the following source files:

- `TestInstallCreateTable.java`
- `TestInstallJDBC.java`
- `TestInstallSQLJ.sqlj`
- `TestInstallSQLJChecker.sqlj`

There is also a Java properties file, `connect.properties`, that helps you set up your database connection. You must edit this file to set appropriate user, password, and URL values.

The demo applications discussed here are provided with your SQLJ installation in the `demo` directory:

ORACLE_HOME`/sqlj/demo`

You may have to edit some of the source files and translate and compile them, as appropriate. The demo applications provided with the Oracle SQLJ implementation refer to tables on Oracle Database account with user name `scott` and password `tiger`. Most Oracle Database installations have this account. You can substitute other values for `scott` and `tiger` if desired.

> **Note:** Running the demo applications requires that the `demo` directory be the current directory, and that the current directory (`"."`) should be specified in the `CLASSPATH`.

This section covers the following topics:

- Set Up the Run Time Connection
- Create a Table to Verify the Database
- Verify the JDBC Driver
- Verify the SQLJ Translator and Run Time
- Verify the SQLJ Translator Connection to the Database

See Also: "Check for Availability of SQLJ and Demo Applications" on page 1-3

Set Up the Run Time Connection

This section describes how to update the `connect.properties` file to configure your Oracle connection for run time. The file is in the `demo` directory and looks something like the following:

> **Note:** In the Oracle Database 11*g* JDBC implementation, database URL connect strings using SIDs are deprecated. Following is an example, where `orcl` is the SID:
>
> `jdbc:oracle:thin:@localhost:1521:orcl`
>
> This would now generate a warning, but not a fatal error. Instead, you are encouraged to use database service names, such as `myservice` in the following example:
>
> `jdbc:oracle:thin:@localhost:1521/myservice`
>
> Refer to the *Oracle Database JDBC Developer's Guide and Reference* for information about database service names.

```
# Users should uncomment one of the following URLs or add their own.
# (If using Thin, edit as appropriate.)
#sqlj.url=jdbc:oracle:thin:@localhost:1521/myservice
#sqlj.url=jdbc:oracle:oci:@
#
# User name and password here
sqlj.user=scott
sqlj.password=tiger
```

Connecting with an Oracle JDBC Driver

Use `oci` in the connect string for the Oracle JDBC OCI driver in any new code. For backward compatibility, however, `oci8` is still accepted. Therefore, you do not have to change existing code.

If you are using the JDBC Thin driver, then uncomment the `thin` URL line in `connect.properties` and edit it as appropriate for your Oracle connection. Use the same URL that was specified when your JDBC driver was set up.

Create a Table to Verify the Database

The following tests assume a table called `SALES`. Compile and run `TestInstallCreateTable` as follows:

```
% javac TestInstallCreateTable.java
% java TestInstallCreateTable
```

This will create the table for you if the database and the JDBC driver are working and the connection is set up properly in the `connect.properties` file.

> **Note:** If you already have a table called `SALES` in your schema and do not want it altered, edit `TestInstallCreateTable.java` to change the table name. Otherwise, your original table will be dropped and replaced.

If you do not want to use `TestInstallCreateTable`, then you can create the `SALES` table using the following SQL statement:

```
CREATE TABLE SALES (
      ITEM_NUMBER NUMBER,
      ITEM_NAME CHAR(30),
      SALES_DATE DATE,
      COST NUMBER,
      SALES_REP_NUMBER NUMBER,
      SALES_REP_NAME CHAR(20));
```

Verify the JDBC Driver

If you want to further test the Oracle JDBC driver, then use the `TestInstallJDBC` demo. Verify that your connection is set up properly in `connect.properties`. Then, compile and run `TestInstallJDBC`, as follows:

```
% javac TestInstallJDBC.java
% java TestInstallJDBC
```

The program should print:

```
Hello, JDBC!
```

Verify the SQLJ Translator and Run Time

Now translate and run the `TestInstallSQLJ` demo, a SQLJ application that has functionality similar to that of `TestInstallJDBC`. Use the following command to translate the source:

```
% sqlj TestInstallSQLJ.sqlj
```

Note that this command also compiles the application.

On a UNIX environment, the `sqlj` script is in *ORACLE_HOME*`/bin`, which should already be in the `PATH`. On Windows, use the `sqlj.exe` executable in the `bin` directory. The SQLJ `translator.jar` file has the class files for the SQLJ translator and run time. It is located in *ORACLE_HOME*`/sqlj/lib` and should already be in the `CLASSPATH`.

See Also: "Set the Path and Classpath" on page 1-4

Now run the application as follows:

```
% java TestInstallSQLJ
```

The program should print:

```
Hello, SQLJ!
```

Verify the SQLJ Translator Connection to the Database

If the SQLJ translator is able to connect to a database, then it can provide online semantics-checking of your SQL operations during translation. The SQLJ translator is written in Java and uses JDBC to get information it needs from a database connection that you specify. You provide the connection parameters for online semantics-checking using the `sqlj` script command line or using a SQLJ properties file, which is `sqlj.properties` by default.

While still in the `demo` directory, edit the `sqlj.properties` file and update, comment, or uncomment the `sqlj.password`, `sqlj.url`, and `sqlj.driver` lines, as appropriate, to reflect your database connection information. For assistance, refer to the comments in the `sqlj.properties` file.

Following is an example of what the appropriate driver, URL, and password settings might be if you are using the Oracle JDBC OCI driver.

```
sqlj.url=jdbc:oracle:oci:@
sqlj.driver=oracle.jdbc.OracleDriver
sqlj.password=tiger
```

Online semantics-checking is enabled as soon as you specify a user name for the translation-time connection. You can specify the user name either by uncommenting the `sqlj.user` line in the `sqlj.properties` file or by using the `-user` command-line option. The `user`, `password`, `url`, and `driver` options all can be set either on the command line or in the properties file.

See Also: "Connection Options" on page 8-25

You can test online semantics-checking by translating the `TestInstallSQLJChecker.sqlj` file located in the `demo` directory, as follows (or using another user name, if appropriate):

```
% sqlj -user=scott TestInstallSQLJChecker.sqlj
```

This should produce the following error message if you are using one of the Oracle JDBC drivers:

```
TestInstallSQLJChecker.sqlj:41: Warning: Unable to check SQL query. Error returned
by database is: ORA-00904:
invalid column name
```

Edit `TestInstallSQLJChecker.sqlj` to fix the error on line 41. The column name should be `ITEM_NAME` instead of `ITEM_NAMAE`. Once you make this change, you can translate and run the application without error using the following commands:

```
% sqlj -user=scott TestInstallSQLJChecker.sqlj
% java  TestInstallSQLJChecker
```

If everything works, then the following line is displayed:

```
Hello, SQLJ Checker!
```

2

Overview

This chapter provides a general overview of SQLJ features and scenarios. The following topics are discussed:

Introduction to SQLJ

This section introduces the basic concepts of SQLJ and discusses the complementary relationship between Java and PL/SQL in Oracle Database applications.

SQLJ enables applications programmers to embed SQL statements in Java code in a way that is compatible with the Java design philosophy. A SQLJ program is a Java program containing embedded SQL statements that comply with the International Standardization Organization (ISO) standard SQLJ Language Reference syntax. The Oracle SQLJ implementation supports the ISO SQLJ standard specification. The standard covers only **static SQL** operations, which are predefined SQL operations that do not change in real time while a user runs the application. The Oracle SQLJ implementation also offers extensions to support **dynamic SQL** operations, which are *not* predefined and the operations can change in real time. It is also possible to use dynamic SQL operations through Java Database Connectivity (JDBC) code or PL/SQL code within a SQLJ application. Typical applications contain more static SQL operations than dynamic SQL operations.

SQLJ consists of a translator and a run-time component and is smoothly integrated into your development environment. You can run the translator to translate, compile, and customize the code in a single step using the `sqlj` front-end utility. The translation process replaces embedded SQL statements with calls to the SQLJ run time, which processes the SQL statements. In ISO standard SQLJ this is typically, but not necessarily, performed through calls to a JDBC driver. To access Oracle Database, you would typically use an Oracle JDBC driver. When you run the SQLJ application, the run time is started to handle the SQL operations.

The SQLJ translator is conceptually similar to other Oracle precompilers and enables you to check SQL syntax, verify SQL operations against what is available in the

schema, and check the compatibility of Java types with corresponding database types. In this way, you can catch errors during development rather than a user catching the errors at run time. The translator checks the following:

- Syntax of the embedded SQL statements
- SQL constructs, against a specified database schema to ensure consistency within a particular set of SQL entities (optional)

  For example, it verifies table names and column names.
- Data types, to ensure that the data exchanged between Java and SQL have compatible types and proper type conversions

The SQLJ methodology of embedding SQL statements directly in Java code is very convenient and concise in a way that it reduces development and maintenance costs in Java programs that require database connectivity.

Java programs can call PL/SQL stored procedures and anonymous blocks through JDBC or SQLJ. In particular, SQLJ provides syntax for calling stored procedures and functions from within a SQLJ statement and also supports embedded PL/SQL anonymous blocks within a SQLJ statement.

Note: Using PL/SQL anonymous blocks within SQLJ statements is one way to support dynamic SQL operations in a SQLJ application. However, the Oracle SQLJ implementation includes extensions to support dynamic SQL directly.

Overview of SQLJ Components

This section introduces the main two major SQLJ components in Oracle SQLJ implementation. It covers the following topics:

- SQLJ Translator
- SQLJ Run Time

SQLJ Translator

This component is a precompiler that you run after creating SQLJ source code.

The translator, which is written in pure Java, supports a programming syntax that enables you to embed SQL statements in SQLJ executable statements. SQLJ executable statements and SQLJ declarations are preceded by the `#sql` token and can be interspersed with Java statements in a SQLJ source code file. SQLJ source code file names must have the `.sqlj` extension. The following is a sample SQLJ statement:

```
#sql { INSERT INTO emp (ename, sal) VALUES ('Joe', 43000) };
```

The translator produces a `.java` file.

You can invoke the translator using the `sqlj` command-line utility. On the command line, specify the files that need to be translated and any desired SQLJ option settings.

See Also: Chapter 8, "Translator Command Line and Options"

SQLJ Run Time

This component is also written in pure Java and is invoked automatically each time you run a SQLJ application.

Oracle JDBC calls are generated directly into the translated code and the SQLJ run time plays a much smaller role.

See Also: "SQLJ Run Time" on page 9-11

Note: From Oracle Database 10*g* release 1 (10.1), only Oracle JDBC drivers are supported with SQLJ.

Overview of Oracle Extensions to the SQLJ Standard

The Oracle SQLJ implementation supports the ISO SQLJ specification. Because the ISO SQLJ standard is a superset of the American National Standards Institute (ANSI) SQLJ standard, using its features requires a Java Development Kit (JDK) 1.5.x or later environment that complies with Java2 Platform, Enterprise Edition (J2EE). The SQLJ translator accepts a broader range of SQL syntax than the ANSI SQLJ standard specifies.

Note: Oracle SQLJ implementation is supported only with JDK 1.5.x.

The ANSI standard addresses not only the SQL92 dialect of SQL, but also enables extension beyond that. The Oracle SQLJ implementation supports the Oracle SQL dialect, which is a superset of SQL92. If you need to create SQLJ programs that work with other databases, then avoid using SQL syntax and SQL types that are not in the standard and, therefore, may not be supported in other environments.

This section covers the following topics:

- SQLJ Type Extensions
- SQLJ Functionality Extensions

See Also: Chapter 5, "Type Support", and Chapter 6, "Objects, Collections, and OPAQUE Types" for information about SQLJ extensions provided by Oracle Database

SQLJ Type Extensions

The Oracle SQLJ implementation supports the following Java types as extensions to the SQLJ standard:

- Instances of `oracle.sql.*` classes as wrappers for SQL data.

  See Also: "Support for JDBC 2.0 LOB Types and Oracle Type Extensions" on page 5-20

- Custom Java classes, typically produced by the JPublisher utility to correspond to SQL objects, object references, and collections. For example, classes that implement the `oracle.sql.ORAData` interface or the JDBC standard `java.sql.SQLdata` interface.

Note: The `SQLData` interface is standard. Classes that implement it are supported by JDBC drivers and databases of other vendors.

See Also: "Custom Java Classes" on page 6-4

- Stream instances: `BinaryStream` and `CharacterStream`, the latter of which replaces the deprecated `AsciiStream` and `UnicodeStream`, used as output parameters.

  See Also: "Support for Streams" on page 5-8

- Iterator and result set instances as input or output parameters. The SQLJ standard specifies them only in result expressions or cast statements.

  See Also: "Using Iterators and Result Sets as Host Variables" on page 4-37 and "Using Iterators and Result Sets as Stored Function Returns" on page 4-45

- Unicode character types: `NString`, `NCHAR`, `NCLOB`, and `NcharCharacterStream`, the latter of which replaces the deprecated `NcharAsciiStream` and `NcharUnicodeStream`.

  See Also: "SQLJ Extended Globalization Support" on page 9-18

Using any of these extensions requires Oracle-specific code generation or Oracle customization during translation, as well as the Oracle SQLJ run time and an Oracle JDBC driver when your application runs. Do not use these or other types if you want to use your code in other environments. To ensure that your application is portable, use the SQLJ `-warn=portable` flag.

See Also: See "Translator Command Line and Options" on page 8-1

SQLJ Functionality Extensions

The Oracle SQLJ implementation also supports the following extended functionality:

- Oracle-specific code generation

  This generates JDBC code directly. Much of the SQLJ run time functionality is bypassed during program execution.

  See Also: "Oracle-Specific Code Generation (No Profiles)" on page 3-28

- Dynamic SQL in SQLJ statements

  See Also: "Support for Dynamic SQL" on page 7-50

- Scrollable result set iterators with additional navigation methods, and `FETCH` syntax from result set iterators and scrollable result set iterators

  See Also: "Scrollable Iterators" on page 7-36

- Optimization flags for column and parameter size definitions

> **See Also:** "Column Definitions" on page 10-16, "Parameter Size Definitions" on page 10-17, and "Options for Code Generation, Optimizations, and CHAR Comparisons" on page 8-40

- Flags for modified translator behavior, such as for binding host expressions by identifier or accounting for blank padding in `CHAR` comparisons for `WHERE` clauses

  > **See Also:** "Binding Host Expressions by Identifier (-bind-by-identifier)" on page 8-55 and "CHAR Comparisons with Blank Padding (-fixedchar)" on page 8-46

- SQLJ statement caching on connection contexts

  > **See Also:** "Statement Caching" on page 10-3

Basic Translation Steps and Run-Time Processing

SQLJ source code contains a mixture of standard Java source together with SQLJ class declarations and SQLJ executable statements containing embedded SQL statements. SQLJ source files have the `.sqlj` file name extension. The file name must be a legal Java identifier. If the source file declares a public class, then the file name must match the name of this class. If the source file does not declare a public class, then the file name should match the name of the first defined class.

This section covers the following topics:

- SQLJ Translation Steps
- Summary of Translator Input and Output
- SQLJ Run-Time Processing

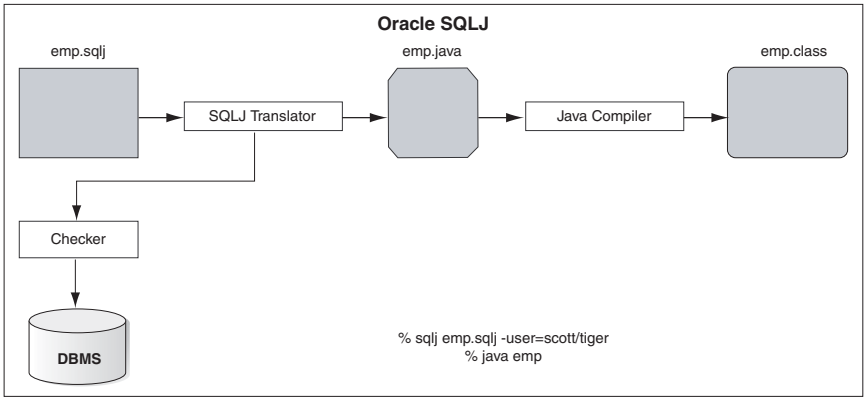
SQLJ Translation Steps

After you have written your `.sqlj` file, you must run SQLJ to process the files. The following example shows SQLJ being run in its simplest form with no command-line options for the `Foo.sqlj` source file with the public class `Foo`:

```
% sqlj Foo.sqlj
```

This command runs a front-end script or utility depending on the platform. The script or utility reads the command line, invokes a Java virtual machine (JVM), and passes arguments to it. The JVM invokes the SQLJ translator and acts as a front end.

Figure 2–1



The following sequence of events occurs, presuming each step completes without error:

1. The JVM invokes the SQLJ translator.
2. The translator parses the SQLJ and Java code in the `.sqlj` file, checking for proper SQLJ syntax and looking for type mismatches between the declared SQL data types and corresponding Java host variables. **Host variables** are Java local variables that are used as input or output parameters in SQL operations.

   See Also: "Java Host, Context, and Result Expressions" on page 4-12

3. Depending on the SQLJ option settings, the translator invokes the online semantics-checker, the offline parser, neither, or both. This is to verify syntax of embedded SQL and PL/SQL statements and to check the use of database elements in the code against an appropriate database schema, for online checking. Even when neither is specified, some basic level of checking is performed.

   When online checking is specified, SQLJ will connect to a specified database schema to verify that the database supports all the database tables, stored procedures, and SQL syntax that the application uses. It also verifies that the host variable types in the SQLJ application are compatible with data types of corresponding database columns.
4. For Oracle-specific SQLJ code generation (`-codegen=oracle`, which is default), SQL operations are converted directly into Oracle JDBC calls.

   See Also: "Oracle-Specific Code Generation (No Profiles)" on page 3-28

   Generated Java code is put into a `.java` output file containing the following:

   - Any class definitions and Java code from the `.sqlj` source file
   - Class definitions created as a result of the SQLJ iterator and connection context declarations

     See Also: "Overview of SQLJ Declarations" on page 4-1

   - Calls to Oracle JDBC drivers to implement the actions of the embedded SQL operations
5. The JVM invokes the Java compiler, which is usually, but not necessarily, the standard `javac` provided with the Sun Microsystems JDK.

6. The compiler compiles the Java source file generated in Step 4 and produces Java `.class` files as appropriate. This will include a `.class` file for each class that is defined, each of the SQLJ declarations.

> **See Also:** "Internal Translator Operations" on page 9-1

General SQLJ Notes

Consider the following when translating and running SQLJ applications:

- It is also possible to specify existing `.java` files on the command line to be compiled and to be available for type resolution as well.

> **See Also:** "Translator Command Line and Properties Files" on page 8-1

- Your application requires an Oracle JDBC driver when it runs, even if your code does not use Oracle-specific features.

Summary of Translator Input and Output

This section summarizes what the SQLJ translator takes as input, what it produces as output, and where it places its output. This section covers the following topics:

- Translator Input
- Translator Output
- Output File Locations

> **Note:** This discussion mentions iterator class and connection context class declarations. Iterators are similar to JDBC result sets and connection contexts are used for database connections.

Translator Input

The SQLJ translator takes one or more `.sqlj` source files as input, which can be specified on the command line. The name of the main `.sqlj` file is based on the public class it defines, if any, else on the first class it defines.

If the main `.sqlj` file defines the `MyClass` class, then the source file name must be:

```
MyClass.sqlj
```

This must also be the file name if there are no public class definitions, but `MyClass` is the first class defined. You must define each public class in separate `.sqlj` files. When you run SQLJ, you can also specify numerous SQLJ options on the command line or in the properties files.

> **See Also:** "Translator Command Line and Properties Files" on page 8-1

Translator Output

The translation step produces a Java source file for each `.sqlj` file in the application, presuming the source code uses SQLJ executable statements.

SQLJ generates Java source files as follows:

- Java source files are `.java` files with the same base names as the `.sqlj` files.

For example, the translator produces `MyClass.java` corresponding to `MyClass.sqlj`, which defines the `MyClass` class. The output `.java` file also contains class definitions for any iterators or connection context classes declared in the `.sqlj` file.

The compilation step compiles the Java source file into multiple class files. One `.class` file is generated for each class defined in the `.sqlj` source file. Additional `.class` files are produced if you declared any SQLJ iterators or connection contexts. Also, separate `.class` files will be produced for any inner classes or anonymous classes in the code.

See Also: "Overview of SQLJ Declarations" on page 4-1

The `.class` files are named as follows:

- The class file for each class defined consists of the name of the class with the `.class` extension. For example, the translator output file `MyClass.java` is compiled into the `MyClass.class` class file.
- The translator names iterator classes and connection context classes according to how you declare them. For example, if you declare an iterator `MyIter`, then the compiler will generate a corresponding `MyIter.class` class file.

Output File Locations

By default, SQLJ places the generated `.java` files in the same directory as the `.sqlj` file. You can specify a different `.java` file location using the SQLJ `-dir` option.

By default, SQLJ places the generated `.class` and `.ser` files in the same directory as the generated `.java` files. You can specify a different location for `.class` and `.ser` files using the SQLJ `-d` option. This option setting is passed to the Java compiler so that `.class` files and `.ser` files will be in the same location.

For both the `-d` and `-dir` option, you must specify a directory that already exists.

See Also: "Options for Output Files and Directories" on page 8-21

SQLJ Run-Time Processing

This section discusses run-time processing during program execution.

When you translate with the default `-codegen=oracle` setting, your program performs the following at run time:

- Executes Oracle-specific application programming interfaces (APIs) that ensure batching support and proper creation and closing of Oracle JDBC statements
- Directly calls Oracle JDBC APIs for registering, passing, and retrieving parameters and result sets

See Also: "Oracle-Specific Code Generation (No Profiles)" on page 3-28

SQLJ Sample Code

This section presents a side-by-side comparison of two versions of the same sample code, where one version is written in SQLJ and the other in JDBC. The objective of this section is to point out the differences in coding requirements between SQLJ and JDBC. This section covers:

- SQLJ Version of the Sample Code
- JDBC Version of the Sample Code

> **Note:** The particulars of SQLJ statements and features used here are described later in this manual, but this example is still useful here to give you a general idea in comparing and contrasting SQLJ and JDBC. You can look at it again when you are more familiar with SQLJ concepts and features.

In the sample, two methods are defined: `getEmployeeAddress()`, which selects and returns an employee's address from a table based on the employee's number, and `updateAddress()`, which takes the retrieved address, calls a stored procedure, and returns the updated address to the database.

In both versions of the sample code, the following assumptions are made:

- A SQL script has been run to create the schema in the database and populate the tables. Both versions of the sample code refer to objects and tables created by this script.
- The `UPDATE_ADDRESS()` PL/SQL stored function exists, and it updates a given address.
- The `Connection` object (for JDBC) and default connection context (for SQLJ) have been created previously by the caller.
- Exceptions are handled by the caller.
- The value of the address argument, `addr`, passed to the `updateAddress()` method can be `null`.

> **Note:** The JDBC and SQLJ versions of the sample code are only partial samples and cannot run independently. There is no `main()` method in either.

SQLJ Version of the Sample Code

The SQLJ version of the sample code that defines methods to retrieve an employee's address from the database, update the address, and return it to the database is as follows:

```
import java.sql.*;

/**
  This is what you have to do in SQLJ
  **/
public class SimpleDemoSQLJ                                    // line 6
{
  //TO DO: make a main that calls this

  public Address getEmployeeAddress(int empno)                 // line 10
    throws SQLException
  {
    Address addr;                                              // line 13
    #sql { SELECT office_addr INTO :addr FROM employees
           WHERE empnumber = :empno };
    return addr;
  }
```

```
                                                                  // line 18
  public Address updateAddress(Address addr)
    throws SQLException
  {
    #sql addr = { VALUES(UPDATE_ADDRESS(:addr)) };          // line 22
    return addr;
  }
}
```

Line 10

The `getEmployeeAddress()` method does not require an explicit `Connection` object. SQLJ can use a default connection context instance, which should be initialized somewhere earlier in the application.

Lines 13-15

The `getEmployeeAddress()` method retrieves an employee address according to the employee number. Use standard SQLJ `SELECT INTO` syntax to select an employee's address from the `employee` table if the employee number matches the one (`empno`) passed in to `getEmployeeAddress()`. This requires a declaration of the `Address` object (`addr`) that will receive the data. The `empno` and `addr` variables are used as input host variables.

Line 16

The `getEmployeeAddress()` method returns the `addr` object.

Line 19

The `updateAddress()` method also uses the default connection context instance.

Lines 19-22

The address is passed to the `updateAddress()` method, which passes it to the database. The database updates the address and passes it back. The actual updating of the address is performed by the `UPDATE_ADDRESS()` stored function. Use standard SQLJ function-call syntax to receive the `addr` address object returned by `UPDATE_ADDRESS()`.

Line 23

The `updateAddress()` method returns the `addr` object.

Specific Features of the SQLJ Version of the Code

Note the following features of the SQLJ version of the sample code:

- An explicit connection is not required. SQLJ can use a default connection context that has been initialized previously in the application.
- No data type casting is required.
- SQLJ does not require knowledge of `_SQL_TYPECODE`, `_SQL_NAME`, or factories.
- `NULL` value data is processed implicitly.
- No explicit code for resource management (for example, closing statements or results sets) is required.
- SQLJ embeds host variables, in contrast to JDBC, which uses parameter markers.
- String concatenation for long SQL statements is not required.
- You do not have to register output parameters.

- SQLJ syntax is simpler. For example, `SELECT INTO` statements are supported and ODBC-style escapes are not used.
- You do not have to implement your own statement cache. By default, SQLJ will automatically cache `#sql` statements. This results in improved performance, for example, if you repeatedly call `getEmployeeAddress()` and `updateAddress()`.

JDBC Version of the Sample Code

If you are familiar with JDBC, then you can check the following the JDBC version of the sample code, which defines methods to retrieve an employee's address from the database, update the address, and return it to the database.

> **Note:** The `TO DO` items in the comment lines indicate where you might want to add additional code to increase the usefulness of the code sample.

```
import java.sql.*;
import oracle.jdbc.*;

/**
  This is what you have to do in JDBC
  **/
public class SimpleDemoJDBC                                    // line 7
{

//TO DO: make a main that calls this

  public Address getEmployeeAddress(int empno, Connection conn)
    throws SQLException                                        // line 13
  {
    Address addr;
    PreparedStatement pstmt =                                  // line 16
      conn.prepareStatement("SELECT office_addr FROM employees" +
       " WHERE empnumber = ?");
    pstmt.setInt(1, empno);
    OracleResultSet rs = (OracleResultSet)pstmt.executeQuery();
    rs.next();                                                 // line 21
     //TO DO: what if false (result set contains no data)?
    addr = (Address)rs.getORAData(1, Address.getORADataFactory());
    //TO DO: what if additional rows?
    rs.close();                                                // line 25
    pstmt.close();
    return addr;                                               // line 27
  }
  public Address updateAddress(Address addr, Connection conn)
    throws SQLException                                        // line 30

  {
    OracleCallableStatement cstmt = (OracleCallableStatement)
      conn.prepareCall("{ ? = call UPDATE_ADDRESS(?) }");   //line 34
    cstmt.registerOutParameter(1, Address._SQL_TYPECODE, Address._SQL_NAME);
                                                               // line 36
    if (addr == null) {
      cstmt.setNull(2, Address._SQL_TYPECODE, Address._SQL_NAME);
    } else {
      cstmt.setORAData(2, addr);
```

```
    }

    cstmt.executeUpdate();                                    // line 43
    addr = (Address)cstmt.getORAData(1, Address.getORADataFactory());
    cstmt.close();                                            // line 45
    return addr;
  }
}
```

Alternative Deployment Scenarios

Although this manual mainly discusses writing for client-side SQLJ applications, you may find it useful to run SQLJ code in the following scenarios:

- From an applet
- In the server (optionally running the SQLJ translator in the server as well)

This section covers the following topics:

- Running SQLJ in Applets
- Introduction to SQLJ in the Server

Running SQLJ in Applets

Because the SQLJ run time is pure Java, you can use SQLJ source code in applets as well as applications. However, there are a few considerations.

> **See Also:** *Oracle Database JDBC Developer's Guide and Reference* for applet issues that apply to Oracle JDBC drivers.

This section covers the following topics:

- General Development and Deployment Considerations
- General End User Considerations
- Java Environment and the Java Plug-In

General Development and Deployment Considerations

The following general considerations apply to the use of SQLJ in applets:

- You must package all the SQLJ run time packages with your applet. The packages are:

  ```
  sqlj.runtime
  sqlj.runtime.ref
  sqlj.runtime.error
  ```

  Also package the following if you used Oracle customization:

  ```
  oracle.sqlj.runtime
  oracle.sqlj.runtime.error
  ```

  These packages are included with your Oracle installation in one of several run time libraries in the *ORACLE_HOME*`/lib` directory.

  > **See Also:** "Requirements for Using the Oracle SQLJ Implementation" on page 1-2

- You must specify a pure Java JDBC driver, such as the Oracle JDBC Thin driver, for your database connection.
- You must explicitly specify a connection context instance for each SQLJ executable statement in an applet. This is a requirement because you could conceivably run two SQLJ applets in a single browser and, thus, in the same JVM.

  See Also: "Connection Considerations" on page 3-4

- The default translator setting `-codegen=oracle` generates Oracle-specific code. This will eliminate the use of Java reflection at run time and, thus, increase portability across different browser environments.

  See Also: "Code Generation (-codegen)" on page 8-41 and "Oracle-Specific Code Generation (No Profiles)" on page 3-28

General End User Considerations

When end users run your SQLJ applet, classes in their `CLASSPATH` may conflict with classes that are downloaded with the applet. Therefore, Oracle recommends that end users clear their `CLASSPATH` before running the applet.

Java Environment and the Java Plug-In

The following are some additional considerations regarding the Java environment and use of Oracle-specific features:

- SQLJ requires the run-time environment of JDK 1.5. Users cannot run SQLJ applets in browsers using earlier JDK versions, without a plug-in. One option is to use a Java plug-in offered by Sun Microsystems. For information, refer to the following:

  `http://java.sun.com/products/plugin/`

- Applets using Oracle-specific features require the Oracle SQLJ run time to work. The Oracle SQLJ run time consists of the classes in the SQLJ run time library file under `oracle.sqlj.*`. The Oracle SQLJ `runtime.jar` library requires the Java Reflection API, `java.lang.reflect.*`. Most browsers do not support the Reflection API or impose security restrictions, but the Sun Microsystems Java plug-in provides support for the Reflection API.

  Note: The term "Oracle-specific features" refers to the use of Oracle type extensions (discussed in Chapter 5, "Type Support") and the use of SQLJ features that require Oracle-specific code generation or, for ISO code generation, require your application to be customized to work against Oracle Database instance. (For example, this is true of the `SET` statement, discussed in Chapter 4, "Basic Language Features".)

The preceding issues can be summarized as follows, focusing on users with Internet Explorer and Netscape browsers:

- The SQLJ and JDBC versions should match. For example, to use the SQLJ 9.0.0 run time, you must have an Oracle 9.0.0 or earlier JDBC driver.

  See Also: "Requirements for Using the Oracle SQLJ Implementation" on page 1-2

- If you use object types, JDBC 2.0 types, REF CURSORs, or the `CAST` statement in your SQLJ statements, then you must adhere to your choice of the following:
  - Use the default `-codegen=oracle` setting when you translate your applet.
  - Ensure that the browser in which you run supports JDK 1.5.x and permits reflection.
  - Run your applet through a browser Java plug-in.

Introduction to SQLJ in the Server

In addition to its use in client applications, SQLJ code can run within a target Oracle Database in stored procedures, stored functions, or triggers. Server-side access occurs through an Oracle JDBC driver that runs inside the server itself. Additionally, Oracle Database 11*g* (and preceding versions) has an embedded SQLJ translator so that SQLJ source files for server-side use can optionally be translated directly in the server.

The two main areas to consider are the following:

- Creating SQLJ code for use within the server

  Coding a SQLJ application for use within the target Oracle Database is similar to coding for client-side use. The issues that exist are due to general JDBC characteristics, as opposed to SQLJ-specific characteristics. The main differences involve connections:

  - You have only one connection.
  - The connection is to the database in which the code is running.
  - The connection is implicit (does not have to be explicitly initialized, unlike on a client).
  - The connection cannot be closed. Any attempt to close it will be ignored.

  Additionally, the JDBC server-side driver used for connections within the server does not support auto-commit mode.

  > **Note:** There is also a server-side Thin driver for connecting to one server from code that runs in another. This case is effectively the same as using a Thin driver from a client and is coded in the same way. Refer "Overview of the Oracle JDBC Drivers" on page 3-1 for further information.

- Translating and loading SQLJ code for server-side use

  You can translate and compile your code either on a client or in the server. If you do this on a client, then you can load the class and resource files into the server from your client, either by pushing them from the client using the Oracle `loadjava` utility or pulling them in from the server using SQL commands.

  Alternatively, you can translate and load in one step using the embedded server-side SQLJ translator. If you load a SQLJ source file instead of class or resource files, then translation and compilation are done automatically. In general, `loadjava` or SQL commands can be used for class and resource files or for source files. From a user perspective, `.sqlj` files are treated the same as `.java` files, with translation taking place implicitly.

  > **See Also:** "Loading SQLJ Source Code into the Server" on page 11-11 for information about using the embedded server-side translator

Note: The server-side translator does not support the SQLJ `-codegen` option and generates Oracle-specific code. To use ISO standard code in the server, you must translate on a client and load the individual components into the server. Also note restrictions on interoperability when running code generated with different settings. For more information, refer to "Translating SQLJ Source on a Client and Loading Components" on page 11-5 and "Oracle-Specific Code Generation (No Profiles)" on page 3-28.

Alternative Development Scenarios

The discussion in this book assumes that you are coding manually on a UNIX environment for English-language deployment. However, you can use SQLJ on other platforms and with integrated development environments (IDEs). There is also globalization support for deployment to other languages. This section covers the following topics:

- SQLJ Globalization Support
- SQLJ in Oracle JDeveloper 10g and Other IDEs
- Windows Considerations

SQLJ Globalization Support

Support for native languages and character encodings by the Oracle SQLJ implementation is based on Java built-in globalization support capabilities.

The standard `user.language` and `file.encoding` properties of the JVM determine appropriate language and encoding for translator and run-time messages. The SQLJ `-encoding` option determines encoding for interpreting and generating source files during translation.

> **See Also:** "Globalization Support in the Translator and Run Time" on page 9-13

SQLJ in Oracle JDeveloper 10*g* and Other IDEs

The Oracle SQLJ implementation includes a programmatic API so that it can be embedded in IDEs, such as Oracle JDeveloper 10*g*. The IDE takes on a role similar to that of the front-end `sqlj` script, invoking the translator, semantics-checker, compiler, and customizer (as applicable).

JDeveloper is a Windows-based visual development environment for Java programming. The JDeveloper Suite enables developers to build multitier, scalable Internet applications using Java across the Oracle Internet Platform. The core product of the suite, the JDeveloper IDE, excels in creating, debugging, and deploying component-based applications.

The Oracle JDBC OCI and Thin drivers are included with JDeveloper. The compilation functionality of JDeveloper includes an integrated SQLJ translator so that your SQLJ application is translated automatically as it is compiled.

Information about JDeveloper is available at the following URL:

`http://www.oracle.com/technology/products/jdev/index.html`

Windows Considerations

Note the following if you are using a Microsoft Windows environment instead of a UNIX environment:

- This manual uses UNIX syntax. Use platform-specific file names and directory separators, such as "\" on Microsoft Windows, that are appropriate for your platform, because your JVM expects file names and paths in the platform-specific format. This is true even if you are using a shell, such as `ksh`, that permits a different file name syntax.
- For UNIX, the Oracle SQLJ implementation provides a front-end script, `sqlj`, that you use to invoke the SQLJ translator. On Microsoft Windows, Oracle instead provides an executable file, `sqlj.exe`. Using a script is not feasible on Microsoft Windows because `.bat` files on these platforms do not support embedded equals signs (=) in arguments, string operations on arguments, or wildcard characters in file name arguments.
- How to set environment variables is specific to the operating system. There may also be OS-specific restrictions. In Windows 95, use the `Environment` tab in the `System` control panel. Additionally, because Windows 95 does not support the "=" character in variable settings, SQLJ supports the use of "#" instead of "=" in setting `SQLJ_OPTIONS`, an environment variable that SQLJ can use for option settings. Consult your operating system documentation regarding settings and syntax for environment variables, and be aware of any size limitations.
- As with any operating system and environment you use, be aware of specific limitations. In particular, the complete, expanded SQLJ command line must not exceed the maximum command-line size, which is 250 characters for Windows 95 and 4000 characters for Windows NT. Consult your operating system documentation.

Refer to the release notes for Windows for additional information.

3

Key Programming Considerations

This chapter discusses key issues to consider before developing and running your SQLJ application, and also provides a summary and sample applications. The following topics are discussed:

Selection of the JDBC Driver

You must consider which Java Database Connectivity (JDBC) driver will be appropriate for your situation and whether it may be advantageous to use different drivers for translation and run time. You must choose or register the appropriate driver class for each and then specify the driver in your connection URL.

> **Note:** Your application will require an Oracle JDBC driver if you use Oracle-specific code generation or if you use ISO code generation with the Oracle customizer, even if your code does not actually use Oracle-specific features.

This section covers the following topics:

Overview of the Oracle JDBC Drivers

Oracle provides the following JDBC drivers:

- Oracle Call Interface (OCI) driver: For client-side use with an Oracle client installation.
- Thin driver: A pure Java driver for client-side use, particularly with applets. It does not require an Oracle client installation.
- Server-side Thin driver: Is functionally the same as the client-side Thin driver, but is for code that runs inside Oracle Database instance and needs to access a remote server.
- Server-side internal driver: For code that runs inside the target server, that is, inside Oracle Database instance that it must access.

Oracle Database 11*g* provides client-side drivers compatible with Java Development Kit (JDK) 1.5.

> **See Also:** *Oracle Database JDBC Developer's Guide and Reference*

> **Note:** Remember that your choices may differ between translation time and run time. For example, you may want to use the Oracle JDBC OCI driver at translation time for semantics-checking, but the Oracle JDBC Thin driver at run time.

Core JDBC Functionality

The core functionality of all Oracle JDBC drivers is the same. They support the same feature set, syntax, programming interfaces, and Oracle extensions.

All Oracle JDBC drivers are supported by the `oracle.jdbc.OracleDriver` class.

JDBC OCI Driver

The Oracle JDBC OCI driver accesses the database by calling the OCI directly from Java, providing the highest compatibility with the different Oracle Database versions. These drivers support installed Oracle Net adapters, including interprocess communication (IPC), named pipes, TCP/IP, and IPX/SPX.

The use of native methods to call C entry points makes the OCI driver dependent on the Oracle platform, requiring an Oracle client installation that includes Oracle Net. Therefore it is not suitable for applets.

Connect strings for the OCI driver are of the following form, where *`tns`* is an optional TNS alias or full TNS specification:

```
jdbc:oracle:oci:@<tns>
```

> **Note:** For backward compatibility, `oci8` is still acceptable instead of `oci`.

JDBC Thin Driver

The Oracle JDBC Thin driver is a platform-independent, pure Java implementation that uses Java sockets to connect directly to Oracle Database from any Oracle or non-Oracle client. It can be downloaded into a browser simultaneously with the Java applet being run.

The JDBC Thin driver supports only TCP/IP protocol and requires a TNS listener to be listening on TCP/IP sockets from the database server. When the JDBC Thin driver is

used with an applet, the client browser must have the capability to support Java sockets.

Connect strings for the JDBC Thin driver are typically of the following form:

```
jdbc:oracle:thin:@host:port/servicename
```

> **See Also:** *Oracle Database JDBC Developer's Guide and Reference* for information about database service names

In Oracle Database 11*g*, connect strings using SIDs are deprecated, but are still supported for backward compatibility:

```
jdbc:oracle:thin:@host:port:sid
```

JDBC Server-Side Thin Driver

The Oracle JDBC server-side Thin driver offers the same functionality as the client-side JDBC Thin driver, but runs inside the database and accesses a remote server. This is useful in accessing one Oracle Database instance from inside another, such as from a Java stored procedure.

Connect strings for the server-side Thin driver are the same as for the client-side Thin driver.

> **Note:** In order to leave the originating database when using the server-side Thin driver, the user account must have `SocketPermission` assigned. Refer to the *Oracle Database JDBC Developer's Guide and Reference* for more information. Also, refer to the *Oracle Database Java Developer's Guide* for general information about `SocketPermission` and other permissions.

JDBC Server-Side Internal Driver

The Oracle JDBC server-side internal driver provides support for any Java code that runs inside the target Oracle Database instance where the SQL operations are to be performed. The server-side internal driver enables the Oracle Java virtual machine (JVM) to communicate directly with the SQL engine. This driver is the default JDBC driver for SQLJ code running as a stored procedure, stored function, or trigger in Oracle Database 11*g*.

Connect strings for the server-side internal driver are of the following form:

```
jdbc:oracle:kprb:
```

If your SQLJ code uses the default connection context, then SQLJ automatically uses this driver for code running in the Oracle JVM.

Driver Selection for Translation

Use SQLJ option settings, either on the command line or in a properties file, to choose the driver manager class and specify a driver for translation.

Use the SQLJ `-driver` option to choose any driver manager class other than `OracleDriver`, which is the default.

Specify the particular JDBC driver to choose, such as JDBC Thin or JDBC OCI for Oracle Database, as part of the connection URL you specify in the SQLJ `-url` option.

See Also: "Connection Options" on page 8-25

You will typically, but not necessarily, use the same driver that you use in your source code for the run time connection.

Note: Remember that the `-driver` option does not choose a particular driver. It registers a driver class with the driver manager. One driver class might be used for multiple driver protocols, such as `OracleDriver`, which is used for all of Oracle JDBC protocols.

Driver Selection and Registration for Run Time

To connect to the database at run time, you must *register* one or more drivers that will understand the URLs you specify for any of your connection instances, whether they are instances of the `sqlj.runtime.ref.DefaultContext` class or of any connection context classes that you declare.

If you are using an Oracle JDBC driver and create a default connection using the `Oracle.connect()` method, then SQLJ handles this automatically. The `Oracle.connect()` method registers the `oracle.jdbc.OracleDriver` class.

If you are using an Oracle JDBC driver, but do not use `Oracle.connect()`, then you must manually register the `OracleDriver` class, as follows:

```
DriverManager.registerDriver(new oracle.jdbc.OracleDriver());
```

If you are not using an Oracle JDBC driver, then you must register some appropriate driver class, as follows:

```
DriverManager.registerDriver(new mydriver.jdbc.driver.MyDriver());
```

In any case, you must also set your connection URL, user name, and password.

See Also: "Single Connection or Multiple Connections Using DefaultContext" on page 3-5

Note: As an alternative to using the JDBC driver manager in establishing JDBC connections, you can use data sources. You can specify a data source in a `with` clause, as described in "Declaration WITH Clause" on page 4-4. For general information about data sources, refer to the *Oracle Database JDBC Developer's Guide and Reference*.

Connection Considerations

When deciding what database connection or connections you will need for your SQLJ application, consider the following:

- Will you need just one database connection or multiple connections?
- If using multiple connections (possibly to multiple schemas), then will each connection use SQL entities of the same name: tables of the same name, columns of the same name and data types, stored procedures of the same name and signature, and so on?
- Will you need different connections for translation and run time or will the same suffice for both?

A SQLJ executable statement can specify a particular connection context instance, either of `DefaultContext` or of a declared connection context class, for its database connection. Alternatively, it can omit the connection context specification and use the default connection, which is an instance of `DefaultContext` that was previously set as the default.

> **Note:** If your operations will use different sets of SQL entities, then you will typically want to declare and use additional connection context classes.

This section covers the following topics:

Single Connection or Multiple Connections Using DefaultContext

This section discusses scenarios where you will use connection instances of only the `DefaultContext` class.

This is typical if you are using a single connection, or multiple connections that use SQL entities with the same names and data types.

Single Connection

For a single connection, use one instance of the `DefaultContext` class specifying the database URL, user name, and password, when you construct your `DefaultContext` object.

You can use the `connect()` method of the `oracle.sqlj.runtime.Oracle` class to accomplish this. Calling this method automatically initializes the default connection context instance. This method has several signatures, including ones that allow you to specify user name, password, and URL, either directly or using a properties file. In the following example, the properties file `connect.properties` is used:

```
Oracle.connect(MyClass.class, "connect.properties");
```

> **Note:** The `connect.properties` file is searched for relative to the specified class. In the example, if `MyClass` is located in `my-package`, then `connect.properties` must be found in the same package location, `my-package`.

If you use `connect.properties`, then you must edit it appropriately and package it with your application. In this example, you must also import the `oracle.sqlj.runtime.Oracle` class.

Alternatively, you can specify user name, password, and URL directly:

```
Oracle.connect("jdbc:oracle:thin:@localhost:1521/myservice", "scott", "tiger");
```

In this example, the connection will use the JDBC Thin driver to connect the `scott` user with the password, `tiger`, to a database on the computer, `localhost`, through port `1521`, where `myservice` is the name of the database service for the connection.

Either of these examples creates a special static instance of the `DefaultContext` class and installs it as your default connection. It is not necessary to do anything with this `DefaultContext` instance directly.

Once you have completed these steps, you do not need to specify the connection for any of the SQLJ executable statements in your application, if you want them all to use the default connection.

Note that in using a JDBC Thin driver, the URL must include the host name, port number, and service name (or SID, which is deprecated in Oracle Database 11*g*), as in the preceding example. Also, the database must have a listener running at the specified port. In using the JDBC OCI driver, no service name (or SID) is required if you intend to use the default account of the client, as will be the case in examples in this document. Alternatively, you can use name-value pairs.

> **See Also:** *Oracle Database JDBC Developer's Guide and Reference* for more information

The following URL will connect to the default account of the client:

```
jdbc:oracle:oci:@
```

Note:

- `Oracle.connect()` will not set your default connection if one had already been set. In that case, it returns `null`. This enables you to use the same code on a client or in the server. If you do want to override your default connection, then use the static `setDefaultContext()` method of `DefaultContext`.
- The `Oracle.connect()` method defaults to a `false` setting of the auto-commit flag. However, it also has signatures to set it explicitly. In the Oracle JDBC implementation, the auto-commit flag defaults to `true`.
- You can optionally specify `getClass()` instead of `MyClass.class` in the `Oracle.connect()` call, as long as you are not calling `getClass()` from a static method. The `getClass()` method is used in some of the SQLJ demo applications.
- You can access the static `DefaultContext` instance, which corresponds to your default connection, as follows:

  ```
  DefaultContext.getDefaultContext();
  ```

Multiple Connections

For multiple connections, you can create and use additional instances of the `DefaultContext` class, while optionally still using the default connection.

You can use the `Oracle.getConnection()` method to instantiate `DefaultContext`, as in the following examples.

First, consider a case where you want most statements to use the default connection, but other statements to use a different connection. You must create one additional instance of `DefaultContext`:

```
DefaultContext ctx = Oracle.getConnection (
   "jdbc:oracle:thin:@localhost2:1521/myservice2", "bill", "lion");
```

> **Note:** `ctx` could also use the `scott/tiger` schema, if you want to perform multiple sets of operations on the same schema.

When you want to use the default connection, it is not necessary to specify a connection context:

```
#sql { SQL operation };
```

This is actually a shortcut for the following:

```
#sql [DefaultContext.getDefaultContext()] { SQL operation };
```

When you want to use the additional connection, specify `ctx` as the connection:

```
#sql [ctx] { SQL operation };
```

Next, consider situations where you want to use multiple connections, where each of them is a named `DefaultContext` instance. This enables you to switch your connection back and forth.

The following statements establish multiple connections to the same schema (in case you want to use multiple Oracle Database sessions or transactions, for example). Instantiate the `DefaultContext` class for each connection you will need:

```
DefaultContext ctx1 = Oracle.getConnection
   ("jdbc:oracle:thin:@localhost1:1521/myservice1", "scott", "tiger");
DefaultContext ctx2 = Oracle.getConnection
   ("jdbc:oracle:thin:@localhost1:1521/myservice1", "scott", "tiger");
```

This creates two connection context instances that would use the same schema, connecting to `scott/tiger` using service `myservice1` on the computer `localhost1`, using the Oracle JDBC Thin driver.

Now, consider a case where you would want multiple connections to different schemas. Again, instantiate the `DefaultContext` class for each connection you will need:

```
DefaultContext ctx1 = Oracle.getConnection
   ("jdbc:oracle:thin:@localhost1:1521/myservice1", "scott", "tiger");
DefaultContext ctx2 = Oracle.getConnection
   ("jdbc:oracle:thin:@localhost2:1521/myservice2", "bill", "lion");
```

This creates two connection context instances that use the Oracle JDBC Thin driver but use different schemas. The `ctx1` object connects to `scott/tiger` using service `myservice1` on the computer `localhost1`, while the `ctx2` object connects to `bill/lion` using service `myservice2` on the computer `localhost2`.

There are two ways to switch back and forth between these connections for the SQLJ executable statements in your application:

- If you switch back and forth frequently, then you can specify the connection for each statement in your application:

```
#sql [ctx1] { SQL operation };
...
#sql [ctx2] { SQL operation };
```

> **Note:** Include the square brackets around the connection context instance name; they are part of the syntax.

- If you use either of the connections several times in a row within your code flow, then you can periodically use the static `setDefaultContext()` method of the `DefaultContext` class to reset the default connection. This method initializes the default connection context instance. This way, you can avoid specifying connections in your SQLJ statements.

```
DefaultContext.setDefaultContext(ctx1);
#sql { SQL operation };   // These three statements all use ctx1
#sql { SQL operation };
#sql { SQL operation };
...
DefaultContext.setDefaultContext(ctx2);
#sql { SQL operation };   // These three statements all use ctx2
#sql { SQL operation };
#sql { SQL operation };
```

> **Note:** Because the preceding statements do not specify connection contexts, at translation time they will all be checked against the default connection context.

Closing Connections

It is advisable to close your connection context instances when you are done, preferably in a `finally` clause of a `try` block (in case your application terminates with an exception).

The `DefaultContext` class, as well as any connection context classes that you declare, includes a `close()` method. Calling this method closes the SQLJ connection context instance and, by default, also closes the underlying JDBC connection instance and the physical connection.

In addition, the `oracle.sqlj.runtime.Oracle` class has a static `close()` method to close the default connection only. In the following example, presume `ctx` is an instance of any connection context class:

```
...
finally
{
   ctx.close();
}
...
```

Alternatively, if the `finally` clause is not within a `try` block in case a SQL exception is encountered:

```
...
```

```
finally
{
   try { ctx.close(); } catch(SQLException ex) {...}
}
...
```

Or, to close the default connection, the `Oracle` class also provides a `close()` method:

```
...
finally
{
   Oracle.close();
}
...
```

Always commit or roll back any pending changes before closing the connection. Whether there would be an implicit `COMMIT` operation as the connection is closed is not specified in the JDBC standard and may vary from vendor to vendor. For Oracle, there is an implicit `COMMIT` when a connection is closed, and an implicit `ROLLBACK` when a connection is garbage-collected without being closed, but it is not advisable to rely on these mechanisms.

Note: It is also possible to close a connection context instance without closing the underlying connection (in case the underlying connection is shared). Refer to "Closing Shared Connections" on page 7-47 for more information.

Multiple Connections Using Declared Connection Context Classes

For multiple connections that use different sets of SQL entities, it is advantageous to use connection context declarations to define additional connection context classes. Having a separate connection context class for each set of SQL entities that you use enables SQLJ to do more rigorous semantics-checking of your code.

See Also: "Connection Contexts" on page 7-1

More About the Oracle Class

The Oracle SQLJ implementation provides the `oracle.sqlj.runtime.Oracle` class to simplify the process of creating and using instances of the `DefaultContext` class.

The static `connect()` method initializes the default connection context instance, instantiating a `DefaultContext` object and installing it as your default connection. You do not need to assign or use the `DefaultContext` instance returned by `connect()`. If you had already established a default connection, then `connect()` returns `null`.

The static `getConnection()` method simply instantiates a `DefaultContext` object and returns it. You can use the returned instance as desired.

Both methods register the Oracle JDBC driver manager automatically if the `oracle.jdbc.OracleDriver` class is found in the `CLASSPATH`. The static `close()` method closes the default connection.

Signatures of the Oracle.connect() and Oracle.getConnection() Methods

Both the method have signatures that take the following parameter sets as input:

- URL (`String`), user name (`String`), password (`String`)
- URL (`String`), user name (`String`), password (`String`), auto-commit flag (`boolean`)
- URL (`String`), `java.util.Properties` object containing properties for the connection
- URL (`String`), `java.util.Properties` object, auto-commit flag (`boolean`)
- URL (`String`) fully specifying the connection, including user name and password

  The following is an example of the format of a URL string specifying user name (`scott`) and password (`tiger`) when using the Oracle JDBC drivers, in this case the JDBC Thin driver:

  ```
  "jdbc:oracle:thin:scott/tiger@localhost:1521/myservice"
  ```

- URL (`String`), auto-commit flag (`boolean`)
- A `java.lang.Class` object for the class relative to which the properties file is loaded, name of properties file (`String`)
- A `java.lang.Class` object, name of properties file (`String`), auto-commit flag (`boolean`)
- A `java.lang.Class` object, name of properties file (`String`), user name (`String`), password (`String`)
- A `java.lang.Class` object, name of properties file (`String`), user name (`String`), password (`String`), auto-commit flag (`boolean`)
- JDBC connection object (`Connection`)
- SQLJ connection context object

These last two signatures inherit an existing database connection. When you inherit a connection, you will also inherit the auto-commit setting of that connection.

The auto-commit flag specifies whether SQL operations are automatically committed. For the `Oracle.connect()` and `Oracle.getConnection()` methods only, the default is `false`. If that is the setting you want, then you can use one of the signatures that does not take auto-commit as input. However, anytime you use a constructor to create an instance of a connection context class, including `DefaultContext`, you must specify the auto-commit setting. In the Oracle JDBC implementation, the default for the auto-commit flag is `true`.

> **See Also:** "Basic Transaction Control" on page 3-18 and "Single Connection or Multiple Connections Using DefaultContext" on page 3-5

Optional Oracle.close() Method Parameters

In using the `Oracle.close()` method to close the default connection, you have the option of specifying whether or not to close the underlying physical database connection. By default it is closed. This is relevant if you are sharing this physical connection between multiple connection objects, either SQLJ connection context instances or JDBC connection instances.

You can keep the underlying physical connection open as follows:

```
Oracle.close(ConnectionContext.KEEP_CONNECTION);
```

You can close the underlying physical connection (default behavior) as follows:

```
Oracle.close(ConnectionContext.CLOSE_CONNECTION);
```

`KEEP_CONNECTION` and `CLOSE_CONNECTION` are static constants of the `ConnectionContext` interface.

See Also: "Closing Shared Connections" on page 7-47

More About the DefaultContext Class

The `sqlj.runtime.ref.DefaultContext` class provides a complete default implementation of a connection context class. As with classes created using a connection context declaration, the `DefaultContext` class implements the `sqlj.runtime.ConnectionContext` interface. The `DefaultContext` class has the same class definition that would have been generated by the SQLJ translator from the declaration:

```
#sql public context DefaultContext;
```

DefaultContext Methods

The following are the key methods of the `DefaultContext` class:

- `getConnection()`

  Gets the underlying JDBC connection object. This is useful if you want to have JDBC code in your application, which is one way to use dynamic SQL operations. You can also use the `setAutoCommit()` method of the underlying JDBC connection object to set the auto-commit flag for the connection.

- `setDefaultContext()`

  Sets the default connection your application uses. This is a `static` method and takes a `DefaultContext` instance as input. SQLJ executable statements that do not specify a connection context instance will use the default connection that you define using this method or the `Oracle.connect()` method.

- `getDefaultContext()`

  Returns the `DefaultContext` instance currently defined as the default connection for your application. This is a `static` method.

- `close()`

  Closes the connection context instance.

The `getConnection()` and `close()` methods are specified in the `sqlj.runtime.ConnectionContext` interface.

Note: On a client, `getDefaultContext()` returns `null` if `setDefaultContext()` was not previously called. However, if a data source object has been bound under "`jdbc/defaultDataSource`" in JNDI, then the client will use this data source object as its default connection.

In the server, `getDefaultContext()` returns the default connection, which is the connection to the server itself.

DefaultContext Constructors

It is typical to instantiate `DefaultContext` using the `Oracle.connect()` or `Oracle.getConnection()` method. However, if you want to create an instance

directly, then there are five constructors for `DefaultContext`. The different input parameter sets for these constructors are:

- URL (`String`), user name (`String`), password (`String`), auto-commit (`boolean`)
- URL (`String`), `java.util.Properties` object, auto-commit (`boolean`)
- URL (`String` fully specifying connection and including user name and password), auto-commit setting (`boolean`)

  The following is an example of the format of a URL specifying user name and password when using the Oracle JDBC drivers, in this case the JDBC Thin driver:

  ```
  "jdbc:oracle:thin:scott/tiger@localhost:1521/myservice"
  ```

- JDBC connection object (`Connection`)
- SQLJ connection context object

The last two signatures inherit an existing database connection. When you inherit a connection, you will also inherit the auto-commit setting of that connection.

Following is an example of constructing a `DefaultContext` instance:

```
DefaultContext defctx = new DefaultContext
   ("jdbc:oracle:thin:@localhost:1521/myservice", "scott", "tiger", false);
```

Notes About Connection Context Constructors:

Note: You must keep the following in mind when using connection context constructors:

- It is important to note that connection context class constructors, unlike the `Oracle.connect()` method, require an auto-commit setting.
- To use any of the first three constructors listed, you must first register your JDBC driver. This happens automatically if you are using an Oracle JDBC driver and call `Oracle.connect()`. Refer to "Driver Selection and Registration for Run Time" on page 3-4.
- Connection context classes that you declare generally have the same constructor signatures as the `DefaultContext` class. However, if you declare a connection context class to be associated with a data source, a different set of constructors is provided. Refer to "Standard Data Source Support" on page 7-9 for more information.
- When using the constructor that takes a JDBC connection object, do not initialize the connection context instance with a null JDBC connection.
- The auto-commit setting determines whether SQL operations are automatically committed. Refer to "Basic Transaction Control" on page 3-18 for more information.

Optional DefaultContext close() Method Parameters

When you close a connection context instance, you have the option of specifying whether or not to close the underlying physical connection. By default it is closed. This is relevant if you are sharing the physical connection between multiple connection

objects, either SQLJ connection context instances or JDBC connection instances. The following examples presume a `DefaultContext` instance `defctx`.

To keep the underlying physical connection open, use the following:

```
defctx.close(ConnectionContext.KEEP_CONNECTION);
```

To close the underlying physical connection, which is the default behavior, use the following:

```
defctx.close(ConnectionContext.CLOSE_CONNECTION);
```

`KEEP_CONNECTION` and `CLOSE_CONNECTION` are static constants of the `ConnectionContext` interface.

> **See Also:** "Closing Shared Connections" on page 7-47 for more information about using these parameters and about shared connections

Connection for Translation

If you want to use online semantics-checking during translation, then you must specify a database connection for SQLJ to use. These are referred to as **exemplar schemas**.

> **See Also:** "Connection Context Concepts" on page 7-2

You can use different connections for translation and run time. In fact, it is often necessary or preferable to do so. It might be necessary if you are not developing the application in the same kind of environment that it will run in. But even if the run time connection is available during translation, it might be preferable to create an account with a narrower set of resources so that your online checking will be tighter. This would be true if your application uses only a small subset of the SQL entities available in the run time connection. Your online checking would be tighter and more meaningful if you create an exemplar schema consisting only of SQL entities that your application actually uses.

Use the SQLJ translator connection options, either on the command line or in a properties file, to specify a connection for translation.

> **See Also:** "Connection Options" on page 8-25

Connection for Customization

Generally, Oracle customization does not require a database connection. However, the Oracle SQLJ implementation does support customizer connections. This is useful in two circumstances:

- If you are using the Oracle customizer with the `optcols` option enabled, then a connection is required. This option allows iterator column type and size definitions for performance optimization.
- If you are using `SQLCheckerCustomizer`, a specialized customizer that performs semantics-checking on profiles, then a connection is required if you are using an online checker, which is true by default.

For Oracle-specific code generation, the SQLJ translator has an `-optcols` option with the same functionality. The `SQLCheckerCustomizer` is invoked through the Oracle customizer harness `verify` option. Use the customizer harness `user`, `password`,

`url`, and `driver` options to specify connection parameters for whatever customizer you are using, as appropriate.

See Also:

- "Oracle Customizer Column Definition Option (optcols)" on page A-20.
- "SQLCheckerCustomizer for Profile Semantics-Checking" on page A-30
- "Customizer Harness Options for Connections" on page A-12

NULL-Handling

Java primitive types, such as `int`, `double`, or `float`, cannot have null values. You must consider this in choosing your result expression and host expression types.

This section covers the following topics:

- Wrapper Classes for NULL-Handling
- Examples of NULL-Handling

Wrapper Classes for NULL-Handling

SQLJ consistently enforces retrieving SQL `NULL` as Java `null`, in contrast to JDBC, which retrieves `NULL` as `0` or `false` for certain data types. Therefore, do not use Java primitive types in SQLJ for output variables in situations where a SQL `NULL` may be received, because Java primitive types cannot take `null` values.

This pertains to result expressions, output or input-output host expressions, and iterator column types. If the receiving Java type is primitive and an attempt is made to retrieve a SQL `NULL`, then a `sqlj.runtime.SQLNullException` is thrown and no assignment is made.

To avoid the possibility of `NULL` being assigned to Java primitives, use the following wrapper classes instead of primitive types:

- `java.lang.Boolean`
- `java.lang.Byte`
- `java.lang.Short`
- `java.lang.Integer`
- `java.lang.Long`
- `java.lang.Double`
- `java.lang.Float`

In case you must convert back to a primitive value, each of these wrapper classes has an *xxx*`Value()` method. For example, `intValue()` returns an `int` value from an `Integer` object and `floatValue()` returns a `float` value from a `Float` object. For example, presuming `intobj` is an `Integer` object:

```
int j = intobj.intValue();
```

Note:

- `SQLNullException` is a subclass of the standard `java.sql.SQLException` class.
- Because Java objects can have `null` values, there is no need for indicator variables in SQLJ, such as those used in other host languages like C, C++, and COBOL.

Examples of NULL-Handling

The following examples show the use of the `java.lang` wrapper classes to handle `NULL`.

Example: Null Input Host Variable

In the following example, a `Float` object is used to pass a `null` value to the database:

```
int empno = 7499;
Float commission = null;

#sql { UPDATE emp SET comm = :commission WHERE empno = :empno };
```

You cannot use the Java primitive type `float` to accomplish this.

Example: Null Iterator Rows

In the following example, a `Double` column type is used in an iterator to allow for the possibility of `null` data.

For each employee in the `emp` table whose salary is at least $50,000, the employee name (`ENAME`) and commission (`COMM`) are selected into the iterator. Then each row is tested to determine if the `COMM` field is, in fact, null. If so, then it is processed accordingly.

```
#sql iterator EmployeeIter (String ename, Double comm);

EmployeeIter ei;
#sql ei = { SELECT ename, comm FROM emp WHERE sal >= 50000 };

while (ei.next())
{
   if (ei.comm() == null)
      System.out.println(ei.ename() + " is not on commission.");
}
ei.close();
...
```

Note: To execute a `WHERE` clause comparison against `NULL`, use the following SQL syntax:

```
...WHERE :x IS NULL
```

Exception-Handling Basics

This section covers the basics of handling exceptions in SQLJ application, including requirements for error-checking. This section covers the following topics:

SQLJ and JDBC Exception-Handling Requirements

Because SQLJ executable statements result in JDBC calls through `sqlj.runtime`, and JDBC requires SQL exceptions to be caught or thrown, SQLJ also requires SQL exceptions to be caught or thrown in any block containing SQLJ executable statements. Your source code will generate errors during compilation if you do not include appropriate exception-handling.

Handling SQL exceptions requires the `SQLException` class, which is included in the standard JDBC `java.sql.*` package.

Example: Exception Handling

This example demonstrates the basic exception-handling required in SQLJ applications. The code declares a `main` method with a `try/catch` block and another method, which throws `SQLException` when an exception is encountered. The code is as follows:

```
    /* Import SQLExceptions class.  The SQLException comes from
       JDBC. Executable #sql clauses result in calls to JDBC, so methods
       containing executable #sql clauses must either catch or throw
       SQLException.
    */
import java.sql.* ;
import oracle.sqlj.runtime.Oracle;

    // iterator for the select

#sql iterator MyIter (String ITEM_NAME);

public class TestInstallSQLJ
{
    //Main method
  public static void main (String args[])
  {
    try {

    // Set the default connection to the URL, user, and password
    // specified in your connect.properties file
      Oracle.connect(TestInstallSQLJ.class, "connect.properties");

      TestInstallSQLJ ti = new TestInstallSQLJ();

    // This method throws SQLException. Therefore, it ic called within a try block
      ti.runExample();

    } catch (SQLException e) {
      System.err.println("Error running the example: " + e);
    }

  } //End of method main

  //Method that runs the example
  void runExample() throws SQLException
  {
```

```
      //Issue SQL command to clear the SALES table
    #sql { DELETE FROM SALES };
    #sql { INSERT INTO SALES(ITEM_NAME) VALUES ('Hello, SQLJ!')};

    MyIter iter;
    #sql iter = { SELECT ITEM_NAME FROM SALES };

    while (iter.next()) {
      System.out.println(iter.ITEM_NAME());
    }
  }
}
```

Processing Exceptions

This section discusses ways to process and interpret exceptions in your SQLJ application. During run time, exceptions may be raised from any of the following:

- SQLJ run time
- JDBC driver
- RDBMS

Printing Error Text

The example in the previous section showed how to catch SQL exceptions and output the error messages. Part of that code is as follows:

```
...
try {
...
} catch (SQLException e) {
      System.err.println("Error running the example: " + e);
}
...
```

This will print the error text from the `SQLException` object.

You can also retrieve error information using the `getMessage()`, `getErrorCode()`, and `getSQLState()` methods the `SQLException` class.

Printing the error text, as in this example, prints the error message with some additional text, such as `SQLException`.

Retrieving SQL States and Error Codes

The `java.sql.SQLException` class and subclasses include the `getMessage()`, `getErrorCode()`, and `getSQLState()` methods. Depending on where the exception or error originated and how they are implemented there, the following methods provide additional information:

- `String getMessage()`

  If the error originates in the SQLJ run time or JDBC driver, then this method returns the error message with no prefix. If the error originates in the RDBMS, then it returns the error message prefixed by the `ORA` number.

- `int getErrorCode()`

If the error originates in the SQLJ run time, then this method returns no meaningful information. If the error originates in the JDBC driver or RDBMS, then it returns the five-digit `ORA` number as an integer.

- `String getSQLState()`

  If the error originates in the SQLJ run time, then this method returns a string with a five-digit code indicating the SQL state. If the error originates in the JDBC driver, then it returns no meaningful information. If the error originates in the RDBMS, then it returns the five-digit SQL state. Your application should have appropriate code to handle `null` values returned.

The following example prints the error message and also checks the SQL state:

```
...
try {
...
} catch (SQLException e) {
      System.err.println("Error running the example: " + e);
      String sqlState = e.getSQLState();
      System.err.println("SQL state = " + sqlState);
}
...
```

Using SQLException Subclasses

For more specific error-checking, use any available and appropriate subclasses of the `java.sql.SQLException` class.

SQLJ provides the `sqlj.runtime.NullException` class, which is a subclass of `java.sql.SQLException`. You can use this exception in situations where a `NULL` might be returned into a Java primitive variable.

For batch-enabled environments, there is also the standard `java.sql.BatchUpdateException` subclass. Refer to "Error Conditions During Batch Execution" on page 10-16 for further information.

When you use a subclass of `SQLException`, catch the subclass exception before catching `SQLException`, as in the following example:

```
...
try {
...
} catch (SQLNullException ne) {
     System.err.println("Null value encountered: " + ne); }
  catch (SQLException e) {
     System.err.println("Error running the example: " + e); }
...
```

This is because a subclass exception can also be caught as a `SQLException`. If you catch `SQLException` first, then execution will not proceed to the part where you have coded special processing for the subclass exception.

Basic Transaction Control

This section discusses how to manage data updates. It covers the following topics:

- Overview of Transactions
- Automatic Commits Versus Manual Commits

See Also: "Advanced Transaction Control" on page 7-41

Overview of Transactions

A **transaction** is a sequence of SQL operations that Oracle treats as a single unit. A transaction begins with the first executable SQL statement after any of the following:

- Connection to the database
- `COMMIT` (committing data updates, either automatically or manually)
- `ROLLBACK` (canceling data updates)

A transaction ends with a `COMMIT` or `ROLLBACK` operation.

Note: In Oracle Database 11*g*, all data definition language (DDL) statements, such as `CREATE` and `ALTER`, include an implicit `COMMIT`. This will commit not only the DDL statement, but all the preceding data manipulation language (DML) statements, such as `INSERT`, `DELETE`, and `UPDATE`, that have not yet been committed or rolled back.

Automatic Commits Versus Manual Commits

In using SQLJ or JDBC, you can either have your data updates automatically committed or commit them manually. In either case, each `COMMIT` operation starts a new transaction. You can specify that changes be committed automatically by enabling the auto-commit flag. This can be done either when you define a SQLJ connection or by using the `setAutoCommit()` method of the underlying JDBC connection object of an existing connection. You can use manual control by disabling the auto-commit flag and using SQLJ `COMMIT` and `ROLLBACK` statements.

Enabling auto-commit may be more convenient, but gives you less control. For example, you have no option to roll back changes. In addition, some SQLJ or JDBC features are incompatible with auto-commit mode. For example, you must disable the auto-commit flag for update batching or `SELECT FOR UPDATE` syntax to work properly.

Specifying Auto-Commit as You Define a Connection

When you use the `Oracle.connect()` or `Oracle.getConnection()` method to create a `DefaultContext` instance and define a connection, the auto-commit flag is set to `false` by default. However, there are signatures of these methods that enable you to set this flag explicitly. The auto-commit flag is always the last parameter.

The following is an example of instantiating `DefaultContext` and using the default `false` setting for auto-commit mode:

```
Oracle.getConnection
   ("jdbc:oracle:thin:@localhost:1521/myservice", "scott", "tiger");
```

Alternatively, you can specify a `true` setting as follows:

```
Oracle.getConnection
   ("jdbc:oracle:thin:@localhost:1521/myservice", "scott", "tiger", true);
```

See Also: "More About the Oracle Class" on page 3-9

If you use a constructor to create a connection context instance, either of `DefaultContext` or of a declared connection context class, then you must specify the auto-commit setting. Again, it is the last parameter, as in the following example:

```
DefaultContext ctx = new DefaultContext
   ("jdbc:oracle:thin:@localhost:1521/myservice", "scott", "tiger", false);
```

See Also: "More About the DefaultContext Class" on page 3-11

If you have reason to create a JDBC `Connection` instance directly, then the auto-commit flag is set to `true` by default if your program runs on a client, or `false` by default if it runs in the server. You cannot specify an auto-commit setting when you create a JDBC `Connection` instance directly, but you can use the `setAutoCommit()` method to alter the setting.

Note: Auto-commit functionality is *not* supported by the JDBC server-side internal driver.

Modifying Auto-Commit in an Existing Connection

There is typically no reason to change the auto-commit flag setting for an existing connection, but you can if you desire. You can do this by using the `setAutoCommit()` method of the underlying JDBC connection object.

You can retrieve the underlying JDBC connection object by using the `getConnection()` method of any SQLJ connection context instance, whether it is an instance of the `DefaultContext` class or of a connection context class that you declared.

You can accomplish these two steps at once, as follows:

```
ctx.getConnection().setAutoCommit(false);
```

or:

```
ctx.getConnection().setAutoCommit(true);
```

In these examples, `ctx` is a SQLJ connection context instance.

Note: Do *not* alter the auto-commit setting in the middle of a transaction.

Using Manual COMMIT and ROLLBACK

If you disable the auto-commit flag, then you must manually commit any data updates. To commit any changes that have been executed since the last `COMMIT` operation, use the SQLJ `COMMIT` statement, as follows:

```
#sql { COMMIT };
```

To roll back any changes that have been executed since the last COMMIT operation, use the SQLJ ROLLBACK statement, as follows:

```
#sql { ROLLBACK };
```

Note:

- Do not use the COMMIT and ROLLBACK commands when auto-commit is enabled. This will result in unspecified behavior, or even SQL exceptions could be raised.
- You can also roll back to a specified savepoint. Refer to "Using Savepoints" on page 3-21.
- All DDL statements in Oracle SQL syntax include an implicit COMMIT operation. There is no special SQLJ functionality in this regard. Such statements follow standard Oracle SQL rules.
- If auto-commit mode is off and you close a connection context instance from a client application, then any changes since your last COMMIT will be committed, unless you close the connection context instance with KEEP_CONNECTION. Refer to "Closing Shared Connections" on page 7-47 for more information.

Effect of Commits and Rollbacks on Iterators and Result Sets

COMMIT and ROLLBACK operations do *not* affect open result sets and iterators. The result sets and iterators will still be open. Usually, all that is relevant to their content is the state of the database at the time of execution of the SELECT statements that populated them.

Note: An exception to this is if you declared an iterator class with sensitivity=SENSITIVE. In this case, changes to the underlying result set may be seen whenever the iterator is scrolled outside of its window size. For more information about scrollable iterators, refer to "Scrollable Iterators" on page 7-36. For more information about the underlying scrollable result sets, refer to the *Oracle Database JDBC Developer's Guide and Reference*.

This also applies to UPDATE, INSERT, and DELETE statements that are executed after the SELECT statements. Execution of these statements does not affect the contents of open result sets and iterators.

Consider a situation where you SELECT, then UPDATE, and then COMMIT. A nonsensitive result set or iterator populated by the SELECT statement will be unaffected by the UPDATE and COMMIT.

As a further example, consider a situation where you UPDATE, then SELECT, and then ROLLBACK. A nonsensitive result set or iterator populated by the SELECT will still contain the updated data, regardless of the subsequent ROLLBACK.

Using Savepoints

The JDBC 3.0 specification added support for savepoints. A **savepoint** is a defined point in a transaction that you can roll back to, if desired, instead of rolling back the

entire transaction. The savepoint is the point in the transaction where the SAVEPOINT statement appears.

In Oracle9*i* Database Release 2 (9.2), SQLJ first included Oracle-specific syntax to support savepoints. In Oracle Database 11*g*, SQLJ adds support for ISO savepoint syntax.

Support for ISO SQLJ Savepoint Syntax

In ISO syntax, use a string literal in a SAVEPOINT statement to designate a name for a savepoint. This can be done as follows:

```
#sql { SAVEPOINT savepoint1 };
```

If you want to roll back changes to that savepoint, then you can refer to the specified name later in a ROLLBACK TO statement, as follows:

```
#sql { ROLLBACK TO savepoint1 };
```

Use a RELEASE SAVEPOINT statement if you no longer need the savepoint:

```
#sql { RELEASE SAVEPOINT savepoint1 };
```

Savepoints are saved in the SQLJ execution context, which has methods that parallel the functionality of these three statements.

> **See Also:** "Savepoint Methods" on page 7-29

Because any COMMIT operation ends the transaction, this also releases all savepoints of the transaction.

Oracle SQLJ Savepoint Syntax

In addition to the ISO syntax, the following Oracle-specific syntax for savepoints is supported. Note that the Oracle syntax uses string host expressions, rather than string literals.

You can set a savepoint as follows:

```
#sql { SET SAVEPOINT :savepoint };
```

The host expression, *savepoint* in this example, is a variable that specifies the name of the savepoint as a Java String.

You can roll back to a savepoint as follows:

```
#sql { ROLLBACK TO :savepoint };
```

To release a savepoint, use the following SQLJ statement:

```
#sql { RELEASE :savepoint };
```

Note: Oracle-specific syntax will continue to be supported for backward compatibility. Note the following differences between Oracle syntax and ISO syntax:

- Oracle syntax takes string variables rather than string literals.
- Oracle syntax uses SET SAVEPOINT instead of SAVEPOINT.
- Oracle syntax uses RELEASE instead of RELEASE SAVEPOINT.

Summary: First Steps in SQLJ Code

The best way to summarize the SQLJ executable statement features and functionality discussed to this point is by examining short but complete programs. This section presents two such examples.

The first example, presented one step at a time and then again in its entirety, uses a `SELECT INTO` statement to perform a single-row query of two columns from a table of employees. If you want to run the example, ensure that you change the parameters in the `connect.properties` file to settings that will let you connect to an appropriate database.

The second example, slightly more complicated, will make use of a SQLJ iterator for a multi-row query.

Import Required Classes

Import any JDBC or SQLJ packages you will need. You will need at least some of the classes in the `java.sql` package:

```
import java.sql.*;
```

You may not need all the `java.sql` package. Key classes are `java.sql.SQLException` and any classes that you refer to explicitly. For example, `java.sql.Date` and `java.sql.ResultSet`.

You will need the following package for the `Oracle` class, which you typically use to instantiate `DefaultContext` objects and establish your default connection:

```
import oracle.sqlj.runtime.*;
```

If you will be using any SQLJ run time classes directly in your code, then import the following packages:

```
import sqlj.runtime.*;
import sqlj.runtime.ref.*;
```

However, even if your code does not use any SQLJ run time classes directly, it will be sufficient to have them in the `CLASSPATH`.

Key run time classes include `ResultSetIterator` and `ExecutionContext` in the `sqlj.runtime` package and `DefaultContext` in the `sqlj.runtime.ref` package.

Register JDBC Drivers and Set Default Connection

Declare the `SimpleExample` class with a constructor that uses the static `Oracle.connect()` method to set the default connection. This also registers the Oracle JDBC drivers.

This uses a signature of `connect()` that takes the URL, user name, and password from the `connect.properties` file. An example of this file is in the directory *`ORACLE_HOME`*`/sqlj/demo` and also in "Set Up the Run Time Connection" on page 1-6.

```
public class SimpleExample {

  public SimpleExample() throws SQLException {
    // Set default connection (as defined in connect.properties).
    Oracle.connect(getClass(), "connect.properties");
  }
```

Set Up Exception Handling

Create a `main()` that calls the `SimpleExample` constructor and then sets up a `try/catch` block to handle any SQL exceptions thrown by the `runExample()` method, which performs the real work of this application:

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

   try {
      SimpleExample o1 = new SimpleExample();
      o1.runExample();
   }
   catch (SQLException ex) {
      System.err.println("Error running the example: " + ex);
   }
}
...
```

You can also use a `try/catch` block inside a `finally` clause when you close the connection, presuming the `finally` clause is not already inside a `try/catch` block in case of SQL exceptions:

```
finally
{
   try { Oracle.close(); } catch(SQLException ex) {...}
}
```

Set Up Host Variables, Execute SQLJ Clause, Process Results

Create a `runExample()` method that performs the following:

1. Throws any SQL exceptions to the `main()` method for processing.
2. Declares Java host variables.
3. Executes a SQLJ clause that binds the Java host variables into an embedded `SELECT` statement and selects the data into the host variables.
4. Prints the results.

The code for this method is as follows:

```
void runExample() throws SQLException {

     System.out.println( "Running the example--" );

     // Declare two Java host variables--
     Float salary;
     String empname;

     // Use SELECT INTO statement to execute query and retrieve values.
      #sql { SELECT ename, sal INTO :empname, :salary FROM emp
             WHERE empno = 7499 };

     // Print the results--
     System.out.println("Name is " + empname + ", and Salary is " + salary);
  }
}    // Closing brace of SimpleExample class
```

This example declares `salary` and `empname` as Java host variables. The SQLJ clause then selects data from the `ename` and `sal` columns of the `emp` table and places the data into the host variables. Finally, the values of `salary` and `empname` are printed.

Note that this SELECT statement could select only one row of the emp table, because the empno column in the WHERE clause is the primary key of the table.

Example of Single-Row Query using SELECT INTO

This section presents the entire SimpleExample class from the previous step-by-step sections. Because this is a single-row query, no iterator is required.

```
// Import SQLJ classes:
import sqlj.runtime.*;
import sqlj.runtime.ref.*;
import oracle.sqlj.runtime.*;

// Import standard java.sql package:
import java.sql.*;

public class SimpleExample {

  public SimpleExample() throws SQLException {
    // Set default connection (as defined in connect.properties).
    Oracle.connect(getClass(), "connect.properties");
  }

  public static void main (String [] args) throws SQLException {

    try {
      SimpleExample o1 = new SimpleExample();
      o1.runExample();
    }
    catch (SQLException ex) {
      System.err.println("Error running the example: " + ex);
    }
  }

  finally
  {
     try { Oracle.close(); } catch(SQLException ex) {...}
  }

  void runExample() throws SQLException {

     System.out.println( "Running the example--" );

     // Declare two Java host variables--
     Float salary;
     String empname;

     // Use SELECT INTO statement to execute query and retrieve values.
        #sql { SELECT ename, sal INTO :empname, :salary FROM emp
              WHERE empno = 7499 };

     // Print the results--
     System.out.println("Name is " + empname + ", and Salary is " + salary);
  }
}
```

Set Up a Named Iterator

This example builds on the previous example by adding a named iterator and using it for a multiple-row query.

First, declare the iterator class. Use object types `Integer` and `Float`, instead of primitive types `int` and `float`, wherever there is the possibility of `NULL` values.

```
#sql iterator EmpRecs(
      int empno,       // This column cannot be null, so int is OK.
                       // (If null is possible, use Integer.)
      String ename,
      String job,
      Integer mgr,
      Date hiredate,
      Float sal,
      Float comm,
      int deptno);
```

Next, instantiate the `EmpRecs` class and populate it with query results.

```
EmpRecs employees;

#sql employees = { SELECT empno, ename, job, mgr, hiredate,
                   sal, comm, deptno FROM emp };
```

Then, use the `next()` method of the iterator to print the results.

```
    while (employees.next())  {
      System.out.println( "Name:       " + employees.ename() );
      System.out.println( "EMPNO:      " + employees.empno() );
      System.out.println( "Job:        " + employees.job() );
      System.out.println( "Manager:    " + employees.mgr() );
      System.out.println( "Date hired: " + employees.hiredate() );
      System.out.println( "Salary:     " + employees.sal() );
      System.out.println( "Commission: " + employees.comm() );
      System.out.println( "Department: " + employees.deptno() );
      System.out.println();
    }
```

Finally, close the iterator.

```
employees.close();
```

Example of Multiple-Row Query Using Named Iterator

This example uses a named iterator for a multiple-row query that selects several columns of data from a table of employees.

Apart from use of the named iterator, this example is conceptually similar to the previous single-row query example.

```
// Import SQLJ classes:
import sqlj.runtime.*;
import sqlj.runtime.ref.*;
import oracle.sqlj.runtime.*;

// Import standard java.sql package:
import java.sql.*;

// Declare a SQLJ iterator.
// Use object types (Integer, Float) for mgr, sal, And comm rather
// than primitive types to allow for possible null selection.

#sql iterator EmpRecs(
      int empno,       // This column cannot be null, so int is OK.
                       // (If null is possible, Integer is required.)
      String ename,
```

```
      String job,
      Integer mgr,
      Date hiredate,
      Float sal,
      Float comm,
      int deptno);

// This is the application class.
public class EmpDemo1App {

   public EmpDemo1App() throws SQLException {
      // Set default connection (as defined in connect.properties).
      Oracle.connect(getClass(), "connect.properties");
   }

  public static void main(String[] args) {

    try {
      EmpDemo1App app = new EmpDemo1App();
      app.runExample();
    }
    catch( SQLException exception ) {
      System.err.println( "Error running the example: " + exception );
    }
  }

  finally
  {
     try { Oracle.close(); } catch(SQLException ex) {...}
  }

  void runExample() throws SQLException  {
    System.out.println("\nRunning the example.\n" );

    // The query creates a new instance of the iterator and stores it in
    // the variable 'employees' of type 'EmpRecs'.  SQLJ translator has
    // automatically declared the iterator so that it has methods for
    // accessing the rows and columns of the result set.

    EmpRecs employees;

    #sql employees = { SELECT empno, ename, job, mgr, hiredate,
                       sal, comm, deptno FROM emp };

    // Print the result using the iterator.

    // Note how the next row is accessed using method 'next()', and how
    // the columns can be accessed with methods that are named after the
    // actual database column names.

    while (employees.next())  {
      System.out.println( "Name:       " + employees.ename() );
      System.out.println( "EMPNO:      " + employees.empno() );
      System.out.println( "Job:        " + employees.job() );
      System.out.println( "Manager:    " + employees.mgr() );
      System.out.println( "Date hired: " + employees.hiredate() );
      System.out.println( "Salary:     " + employees.sal() );
      System.out.println( "Commission: " + employees.comm() );
      System.out.println( "Department: " + employees.deptno() );
      System.out.println();
```

```
    }

    // You must close the iterator when it's no longer needed.
    employees.close() ;
  }
}
```

Oracle-Specific Code Generation (No Profiles)

Throughout this manual there is general and standard discussion of the SQLJ run time layer and SQLJ profiles. However, the Oracle SQLJ implementation, by default, generates Oracle-specific code with direct calls to the Oracle JDBC driver instead of generating ISO standard code that calls the SQLJ run time. With Oracle-specific code generation, there are no profile files, and the role of the SQLJ run time layer is greatly reduced during program execution. Oracle-specific code supports all Oracle-specific extended features.

Code generation is determined through the SQLJ translator `-codegen` option. The default setting for Oracle-specific code generation is `-codegen=oracle`. Alternatively, you can set `-codegen=iso` for code generation according to the ISO standard.

> **See Also:** "Code Generation (-codegen)" on page 8-41.

This section covers the following topics:

- Code Considerations and Limitations with Oracle-Specific Code Generation
- SQLJ Usage Changes with Oracle-Specific Code Generation
- Server-Side Considerations with Oracle-Specific Code Generation
- Advantages and Disadvantages of Oracle-Specific Code Generation

Code Considerations and Limitations with Oracle-Specific Code Generation

When coding a SQLJ application where Oracle-specific code generation will be used, be aware of the following programming considerations and restrictions:

- To use a nondefault statement cache size, you must include appropriate method calls in your code, because the Oracle customizer `stmtcache` option is unavailable. Refer to "SQLJ Usage Changes with Oracle-Specific Code Generation" on page 3-30.
- Do not mix Oracle-specific generated code with ISO standard generated code in the same application. However, if Oracle-specific code and ISO standard code *must* share the same connection, do one of the following:
  - Ensure that the Oracle-specific code and ISO standard code use different SQLJ execution context instances. Refer to "Execution Contexts" on page 7-24 for information about SQLJ execution contexts.
  - Place a transaction boundary, that is, as a manual `COMMIT` or `ROLLBACK` statement, between the two kinds of code.

  This limitation regarding mixing code is especially significant for server-side code, because all Java code running in a given session uses the same JDBC connection and SQLJ connection context.

See Also: "Server-Side Considerations with Oracle-Specific Code Generation" on page 3-31

- Do not rely on side effects in parameter expressions when values are returned from the database. Oracle-specific code generation does not create temporary variables for evaluation of `OUT` parameters, `IN OUT` parameters, `SELECT INTO` variables, or return arguments on SQL statements.

  For example, avoid statements such as the following:

  ```
  #sql { SELECT * FROM EMP INTO :(x[i++]), :(f_with_sideffect()[i++]),
                                :(a.b[i]) };
  ```

  or:

  ```
  #sql x[i++] = { VALUES f(:INOUT (x[i++]), :OUT (f_with_sideffect())) };
  ```

  Evaluation of arguments is performed *in place* in the generated code. This may result in different behavior than when evaluation is according to ISO SQLJ standards.

  See Also: "Evaluation of Java Expressions at Run Time" on page 4-17 and "Examples of Evaluation of Java Expressions at Run Time (ISO Code Generation)" on page 4-18

- Type maps for Oracle object functionality assumes that the corresponding Java classes implement the `java.sql.SQLData` interface, given that JPublisher-generated Java classes do not otherwise require a type map. If you use type maps for Oracle object functionality, then your iterator declarations and connection context declarations must specify the same type maps. Specify this through the `with` clause.

  For example, if you declare a connection context class as follows:

  ```
  #sql context TypeMapContext with (typeMap="MyTypeMap");
  ```

  and you populate an iterator instance from a SQLJ statement that uses an instance of this connection context class, as follows:

  ```
  TypeMapContext tmc = new TypeMapContext(...);
  ...
  MyIterator it;
  #sql [tmc] it = ( SELECT pers, addr FROM tab WHERE ...);
  ```

  then the iterator declaration is required to have specified the same type map, as follows:

  ```
  #sql iterator MyIterator with (typeMap="MyTypeMap")
               (Person pers, Address addr);
  ```

  See Also: "Custom Java Class Requirements" on page 6-8 and "Declaration WITH Clause" on page 4-4

Note: The reason for this restriction is that with Oracle-specific code generation, all iterator getter methods are fully generated as Oracle JDBC calls during translation. To generate the proper calls, the SQLJ translator must know whether an iterator will be used with a particular type map.

SQLJ Usage Changes with Oracle-Specific Code Generation

Some options that were previously available only as Oracle customizer options are useful with Oracle-specific code generation as well. Because profile customization is not applicable with Oracle-specific code generation, these options have been made available through other means.

To alter the statement cache size or disable statement caching when generating Oracle-specific code, use method calls in your code instead of using the customizer `stmtcache` option. The `sqlj.runtime.ref.DefaultContext` class, as well as any connection context class you declare, now has the following static methods:

- `setDefaultStmtCacheSize(int)`
- `int getDefaultStmtCacheSize()`

It also has the following instance methods:

- `setStmtCacheSize(int)`
- `int getStmtCacheSize()`

By default, statement caching is enabled.

See Also: "Statement Caching" on page 10-3

In addition, the following options are available as front-end Oracle SQLJ translator options as well as Oracle customizer options:

- `-optcols`: Enable iterator column type and size definitions to optimize performance.
- `-optparams`: Enable parameter size definitions to optimize JDBC resource allocation. This option is used in conjunction with `optparamdefaults`.
- `-optparamdefaults`: Set parameter size defaults for particular data types. This option is used in conjunction with `optparams`.
- `-fixedchar`: Enable `CHAR` comparisons with blank padding for `WHERE` clauses.

See Also: "Options for Code Generation, Optimizations, and CHAR Comparisons" on page 8-40

Be aware of the following:

- Use the `-optcols` option only if you are using online semantics-checking, where you have used the SQLJ translator `-user`, `-password`, and `-url` options appropriately to request a database connection during translation.
- The functionality of the `-optcols`, `-optparams`, and `-optparamdefaults` options, including default values, is the same as for the corresponding customizer options.

Server-Side Considerations with Oracle-Specific Code Generation

Consider the following if your SQLJ code will run in the server:

- The server-side SQLJ translator no longer supports ISO standard generated code. SQLJ source code that is loaded into the server and compiled there will always be translated with the default `-codegen=oracle` setting.

  Therefore, to use ISO standard generated code in the server, you must translate and compile the SQLJ code on a client and then load the individual components into the server.

  See Also: "Translating SQLJ Source on a Client and Loading Components" on page 11-5

- The caution against mixing Oracle-specific generated code with ISO standard generated code applies to server-side Java code that calls a Java stored procedure or stored function, even if the stored procedure is invoked through a PL/SQL wrapper. This constitutes a recursive call-in. By default, the `ExecutionContext` object is shared by both the calling module and the called module. Therefore, both modules should be translated with the same `-codegen` setting.

  If you want to ensure interoperability with code that has been translated with ISO standard code generation, then it is advisable to explicitly instantiate execution context instances, as in the following example:

```
public static method() throws SQLException
{
   Execution Context ec = new ExecutionContext();
   ...
   try {
      ...
      #sql [ec] { SQL operation };
      ...
   } finally { ec.close(); }
}
```

  Note: To avoid resource leakage when using an explicit `ExecutionContext` instance, ensure that you use the `close()` method, as shown in this example.

  See Also: "Code Considerations and Limitations with Oracle-Specific Code Generation" on page 3-28

Advantages and Disadvantages of Oracle-Specific Code Generation

Oracle-specific code generation offers following advantages over ISO standard code generation:

- Applications run more efficiently. The code calls JDBC application programming interfaces (APIs) directly, placing run-time performance directly at the JDBC level. The role of the intermediate SQLJ run time layer is greatly reduced during program execution.
- Applications are smaller in size.

- No profile files (`.ser`) are produced. This is especially convenient if you are loading a translated application into the database or porting it to another system, because there are fewer components.
- Translation is faster, because there is no profile customization step.
- During execution, the Oracle SQLJ run time and Oracle JDBC driver use the same statement cache resources, so partitioning resources between the two is unnecessary.
- Having the SQL-specific information appear in the Java class files instead of in separate profile files avoids potential security issues.
- You need not have to rewrite your code to take advantage of possible future Oracle JDBC performance enhancements, such as enhancements being considered for execution of static SQL code. Future releases of the Oracle SQLJ translator will handle this automatically.
- The use of Java reflection at run time is eliminated, and thus, provides full portability to browser environments.

However. there are a few disadvantages:

- Oracle-specific generated code may not be portable to generic JDBC platforms.
- Profile-specific functionality is not available. For example, you cannot perform customizations at a later date to use the Oracle customizer harness `-debug`, `-verify`, and `-print` options.

> **See Also:** "Customizer Harness Options for Connections" on page A-12 and "AuditorInstaller Customizer for Debugging" on page A-33

ISO Standard Code Generation

This section covers the following topics:

Environment Requirements for ISO Standard Code Generation

The Oracle SQLJ implementation, by default, generates Oracle-specific code with direct calls to the Oracle JDBC driver instead of generating ISO standard code that calls the SQLJ run time. The following is a typical environment setup for ISO standard code generation:

- SQLJ code generation: `-codegen=iso`
- SQLJ translation library: `translator.jar`
- SQLJ run time library: `runtime12.jar` with JDK 1.5.x and Oracle JDBC driver 11g release 1 (11.1).

- JDBC drivers: Oracle 11g release 1 (11.1)
- JDK version: 1.5.x

SQLJ Translator and SQLJ Run Time

The following section describes the differences in Oracle SQLJ implementation in case of ISO standard code generation:

- SQLJ translator: Along with the `.java` file, the translator also produces one or more SQLJ profiles for ISO standard code generation. These profiles contain information about the embedded SQL operations. SQLJ then automatically invokes a Java compiler to produce `.class` files from the `.java` file.

  See Also: "SQLJ Translator" on page 2-2

- SQLJ run time: For ISO standard code generation, the SQLJ run time implements the desired actions of the SQL operations by accessing the database using a JDBC driver. The generic ISO SQLJ standard does not require that a SQLJ run time use a JDBC driver to access the database.

  See Also: "SQLJ Run Time" on page 9-11

In addition to the translator and run time, there is a component known as the **customizer** that plays a role. A customizer tailors SQLJ profiles for a particular database implementation and vendor-specific features and data types. By default, for ISO standard code, the SQLJ front end invokes an Oracle customizer to tailor your profiles for Oracle Database instance and Oracle-specific features and data types.

When you use the Oracle customizer during translation, your application will require the SQLJ run time and an Oracle JDBC driver when it runs.

Note: From Oracle Database 10*g* release 1 (10.1), only Oracle JDBC drivers are supported with SQLJ.

SQLJ Profiles

With ISO standard code generation, SQLJ profiles are serialized Java resources or classes generated by the SQLJ translator, which contain details about the embedded SQL statements. The translator creates these profiles. Then, depending on the translator option settings, it either serializes the profiles and puts them into binary resource files or puts them into `.class` files.

This section covers the following topics:

- Overview of Profiles
- Binary Portability

Overview of Profiles

SQLJ profiles are used in ISO standard code for implementing the embedded SQL operations in SQLJ executable statements. Profiles contain information about the SQL operations and the types and modes of data being accessed. A profile consists of a collection of entries, where each entry maps to one SQL operation. Each entry fully specifies the corresponding SQL operation, describing each of the parameters used in processing this instruction.

SQLJ generates a profile for each connection context class in your application, where each connection context class corresponds to a particular set of SQL entities you use in your database operations. There is one default connection context class, and you can declare additional classes. The ISO SQLJ standard requires that the profiles be of standard format and content. Therefore, for your application to use vendor-specific extended features, your profiles must be customized. By default, this occurs automatically, with your profiles being customized to use Oracle-specific extended features.

Profile customization enables vendors to add value in the following ways:

- Vendors can support their own specific data types and SQL syntax. For example, the Oracle customizer maps standard JDBC `PreparedStatement` method calls in translated SQLJ code to `OraclePreparedStatement` method calls, which provide support for Oracle type extensions.
- Vendors can improve performance through specific optimizations.

Note:

- By default, SQLJ profile file names have the `.ser` extension, but this does not mean that all `.ser` files are profiles. Other serialized objects can use this extension, and a SQLJ program unit can use serialized objects other than its profiles. Optionally, profiles can be converted to `.class` files instead of `.ser` files.
- A SQLJ profile is not produced if there are no SQLJ executable statements in the source code.

Binary Portability

SQLJ-generated profile files support binary portability. That is, you can port them as is and use them with other kinds of databases or in other environments, if you have not used vendor-specific data types or features. This is true for generated `.class` files as well.

SQLJ Translation Steps

For ISO standard code generation (`-codegen=iso`), the translator processes the SQLJ source code, converts SQL operations to SQLJ run-time calls, and generates Java output code and one or more SQLJ profiles. A separate profile is generated for each connection context class in the source code, where a different connection context class is typically used for each interrelated set of SQL entities that is used in the operations.

Generated Java code is put into a `.java` output file containing the following:

- Any class definitions and Java code from the `.sqlj` source file
- Class definitions created as a result of the SQLJ iterator and connection context declarations

  See Also: "Overview of SQLJ Declarations" on page 4-1

- A class definition for a specialized class known as the **profile-keys** class that SQLJ generates and uses in conjunction with the profiles (for ISO standard SQLJ code generation only)
- Calls to the SQLJ run time to implement the actions of the embedded SQL operations

Generated profiles contain information about all the embedded SQL statements in the SQLJ source code, such as actions to take, data types being manipulated, and tables being accessed. When the application is run, the SQLJ run time accesses the profiles to retrieve the SQL operations and passes them to the JDBC driver.

By default, profiles are put into `.ser` serialized resource files, but SQLJ can optionally convert the `.ser` files to `.class` files as part of the translation.

The compiler compiles the generated Java source file and produces Java `.class` files as appropriate. This includes a `.class` file for each class that is defined, each of the SQLJ declarations, and the profile-keys class. The JVM then invokes the Oracle customizer or other specified customizer to customize the profiles generated.

See Also: "Internal Translator Operations" on page 9-1

General SQLJ Notes

Consider the following when translating and running SQLJ applications for ISO specific code generation:

- Along with compiling existing `.java` files on the command line and making them available for type resolution, as for Oracle-specific code generation, you need to:
  - Customize the existing profiles
  - Customize the Java Archive (JAR) files containing profiles

  See Also: "Translator Command Line and Properties Files" on page 8-1

- SQLJ generates profiles and the profile-keys class only if your source code includes SQLJ executable statements.
- If you use the Oracle customizer during translation, then your application requires the Oracle SQLJ run time and an Oracle JDBC driver when it runs, even if your code does not use Oracle-specific features. You can avoid this by specifying `-profile=false` when you translate, to bypass Oracle-specific customization.

Summary of Translator Input and Output

We have seen what the SQLJ translator takes as input, what it produces as output, and where it places its output in case of Oracle-specific code generation. This section covers the same topics for ISO standard code generation:

- Translator Input
- Translator Output
- Output File Locations

See Also: "Summary of Translator Input and Output" on page 2-7

Translator Input

Similar to Oracle -specific code generation, the SQLJ translator takes one or more `.sqlj` source files as input, which can be specified on the command line. The name of the main `.sqlj` file is based on the public class it defines, if any, else on the first class it defines.

See Also: "Translator Input" on page 2-7

Translator Output

The translation step produces a Java source file for each `.sqlj` file in the application and at least one application profile for ISO standard code generation, presuming the source code uses SQLJ executable statements.

SQLJ generates Java source files and application profiles as follows:

See Also: "Translator Output" on page 2-7

- Similar to Oracle-specific code generation, Java source files are `.java` files with the same base names as the `.sqlj` files.
- The application profile files, if applicable, contain information about the SQL operations of the SQLJ application. There is one profile for each connection class that is used in the application. The profiles have names with the same base name as the main `.sqlj` file and the following extensions:

  ```
  _SJProfile0.ser
  _SJProfile1.ser
  _SJProfile2.ser
  ...
  ```

  For example, for `MyClass.sqlj` the translator produces:

  ```
  MyClass_SJProfile0.ser
  ```

  The `.ser` file extension indicates that the profiles are serialized. The `.ser` files are binary files.

  Note: The `-ser2class` translator option instructs the translator to generate profiles as `.class` files instead of `.ser` files. Other than the file name extension, the naming is the same.

Similar to the compilation step of Oracle-specific code generation, compiling the Java source file into multiple class files generates one `.class` file for each class defined in the `.sqlj` source file. But in case of ISO code generation, a `.class` file is also generated for a class known as the **profile-keys** class that the translator generates and uses with the profiles to implement the SQL operations. Additional `.class` files are produced if you declare any SQLJ iterators or connection contexts. Also, like Oracle-specific code generation, separate `.class` files are produced for any inner classes or anonymous classes in the code.

See Also: "Overview of SQLJ Declarations" on page 4-1

The `.class` files are named as follows:

- Like Oracle-specific code generation, the class file for each class defined consists of the name of the class with the `.class` extension.
- The profile-keys class that the translator generates is named according to the base name of the main `.sqlj` file, plus the following:

  ```
  _SJProfileKeys
  ```

  So, the class file has the following extension:

  ```
  _SJProfileKeys.class
  ```

For example, for `MyClass.sqlj`, the translator together with the compiler produces:

```
MyClass_SJProfileKeys.class
```

- Like Oracle-specific code generation, the translator names iterator classes and connection context classes according to how you declare them.

The customization step alters the profiles but produces no additional output.

> **See Also:** "Profile Customization (ISO Code Generation)" on page 9-7

Note: It is not necessary to reference SQLJ profiles or the profile-keys class directly. This is all handled automatically.

Output File Locations

The output file locations are the same for both Oracle-specific code generation and ISO standard code generation.

> **See Also:** "Output File Locations" on page 2-8

SQLJ Run-Time Processing

This section discusses run-time processing for ISO standard code during program execution.

For ISO standard SQLJ applications, the SQLJ run time reads the profiles and creates connected profiles, which incorporate database connections. Then the following occurs each time the application must access the database:

1. SQLJ-generated application code uses methods in a SQLJ-generated profile-keys class to access the connected profile and read the relevant SQL operations. There is a mapping between SQLJ executable statements in the application and SQL operations in the profile.
2. The SQLJ-generated application code calls the SQLJ run time, which reads the SQL operations from the profile.
3. The SQLJ run time calls the JDBC driver and passes the SQL operations to the driver.
4. The SQLJ run time passes any input parameters to the JDBC driver.
5. The JDBC driver executes the SQL operations.
6. If any data is to be returned, then the database sends it to the JDBC driver, which sends it to the SQLJ run time for use by your application.

Note: Passing input parameters can also be referred to as binding input parameters or binding host expressions. The terms host variables, host expressions, bind variables, and bind expressions are all used to describe Java variables or expressions that are used as input or output for SQL operations.

Deployment Scenarios

We have discussed how to run Oracle-specific SQLJ code in the following scenarios:

- From an applet
- In the server (optionally running the SQLJ translator in the server as well)

There are a few considerations that you need to make while running your ISO standard code from an applet:

> **See Also:** "Alternative Deployment Scenarios" on page 2-12

- You must package all the SQLJ run time packages with your applet. The packages are:

  ```
  sqlj.runtime
  sqlj.runtime.ref
  sqlj.runtime.profile
  sqlj.runtime.profile.ref
  sqlj.runtime.error
  ```

  Also package the following if you used Oracle customization:

  ```
  oracle.sqlj.runtime
  oracle.sqlj.runtime.error
  ```

  These packages are included with your Oracle installation in one of several run time libraries in the *ORACLE_HOME*`/lib` directory.

  > **See Also:** "Requirements for Using the Oracle SQLJ Implementation" on page 1-2

- Some browsers, such as Netscape Navigator 4.x, do not support resource files with a `.ser` extension, which is the extension used by the SQLJ serialized object files that are used for profiles. However, the Sun Microsystems Java plug-in supports `.ser` files.

  Alternatively, if you do not want to use the plug-in, then the Oracle SQLJ implementation offers the `-ser2class` option to convert `.ser` files to `.class` files during translation.

  > **See Also:** "Conversion of .ser File to .class File (-ser2class)" on page 8-53

  Note: This consideration *does not* apply to the default Oracle-specific code generation, where no profiles are produced.

- Applets using Oracle-specific features require the Oracle SQLJ run time to work. The Oracle SQLJ run time consists of the classes in the SQLJ run time library file under `oracle.sqlj.*`. The Oracle SQLJ `runtime.jar` library requires the Java Reflection API, `java.lang.reflect.*`. Most browsers do not support the Reflection API or impose security restrictions, but the Sun Microsystems Java plug-in provides support for the Reflection API.

  With ISO standard code generation, the following SQLJ language features always require the Java Reflection API, regardless of the version of the SQLJ run time you are using:

  – The `CAST` statement

- `REF CURSOR` parameters or `REF CURSOR` columns being retrieved from the database as instances of a SQLJ iterator
- Retrieval of `java.sql.Ref`, `Struct`, `Array`, `Blob`, or `Clob` objects
- Retrieval of SQL objects as instances of Java classes implementing the `oracle.sql.ORAData` or `java.sql.SQLData` interfaces

---

Note: ■An exception to the preceding is if you use SQLJ in a mode that is fully compatible with ISO. That is, if you use SQLJ in an environment that complies with J2EE and you translate and run your program with the SQLJ `runtime12ee.jar` library, and you employ connection context type maps as specified by ISO. In this case, instances of `java.sql.Ref`, `Struct`, `Array`, `Blob`, `Clob`, and `SQLData` are being retrieved without the use of reflection.

- If you use Oracle-specific code generation, then you will eliminate the use of reflection in all of the instances listed.

Oracle-Specific Code Generation Versus ISO Standard Code Generation

The Oracle SQLJ implementation provides the option of Oracle-specific code generation, where Oracle JDBC calls are generated directly in the code. This is the default behavior. In the case of Oracle-specific code generation, you must be aware of the following:

- There are no profile files, and therefore, there is no customization step during translation.
- At run time, SQL operations do not have to go through the SQLJ run time layer, because JDBC calls, instead of the SQLJ run time calls, are directly generated in the translated code.

Requirements and Restrictions for Naming

There are four areas to consider in discussing naming requirements, naming restrictions, and reserved words:

- The Java namespace, including additional restrictions imposed by SQLJ on the naming of local variables and classes
- The SQLJ namespace
- The SQL namespace
- Source file names

This section covers the following topics:

Java Namespace: Local Variable and Class Naming Restrictions

The Java namespace applies to all standard Java statements and declarations, including the naming of Java classes and local variables. All standard Java naming restrictions apply, and you should avoid the use of Java reserved words.

In addition, SQLJ places minor restrictions on the naming of local variables and classes.

> **Note:** Naming restrictions particular to host variables are discussed in "Restrictions on Host Expressions" on page 4-24.

Local Variable Naming Restrictions

Some of the functionality of the SQLJ translator results in minor restrictions in naming local variables. The SQLJ translator replaces each SQLJ executable statement with a statement block, where the SQLJ executable statement is of the standard syntax:

```
#sql { SQL operation };
```

SQLJ may use temporary variable declarations within a generated statement block. The name of any such temporary variables will include the following prefix:

```
__sJT_
```

> **Note:** There are two underscores at the beginning and one at the end.

The following declarations are examples of those that might occur in a SQLJ-generated statement block:

```
int __sJT_index;
Object __sJT_key;
java.sql.PreparedStatement __sJT_stmt;
```

The string `__sJT_` is a reserved prefix for SQLJ-generated variable names. SQLJ programmers must not use this string as a prefix for the following:

- Names of variables declared in blocks that include executable SQL statements
- Names of parameters to methods that contain executable SQL statements
- Names of fields in classes that contain executable SQL statements, or whose subclasses or enclosed classes contain executable SQL statements

Class Naming Restrictions

Be aware of the following minor restrictions in naming classes in SQLJ applications:

- You must not declare class names that may conflict with SQLJ internal classes. In particular, a top-level class cannot have a name of the following form, where `a` is the name of an existing class in the SQLJ application:

  ```
  a_SJb
  ```

  where, `a` and `b` are legal Java identifiers.

  For example, if your application class is `Foo` in file `Foo.sqlj`, then SQLJ generates a profile-keys class called `Foo_SJProfileKeys`. Do not declare a class name that conflicts with this.

- A class containing SQLJ executable statements must not have a name that is the same as the first component of the name of any package that includes a Java type used in the application. Examples of class names to avoid are `java`, `sqlj`, and `oracle` (case-sensitive). As another example, if your SQLJ statements use host variables whose type is `abc.def.MyClass`, then you cannot use `abc` as the name of the class that uses these host variables.

  To avoid this restriction, follow Java naming conventions recommending that package names start in lowercase and class names start in uppercase.

SQLJ Namespace

The **SQLJ namespace** refers to `#sql` class declarations and the portion of `#sql` executable statements outside the curly braces.

> **Note:** Restrictions particular to the naming of iterator columns are discussed in "Using Named Iterators" on page 4-31.

Avoid using the following SQLJ reserved words as class names for declared connection context classes or iterator classes, in `with` or `implements` clauses, or in iterator column type declaration lists:

- `iterator`
- `context`
- `with`

For example, do not have an iterator class or instance called `iterator` or a connection context class or instance called `context`.

However, note that it is permissible to have a stored function return variable whose name is any of these words.

SQL Namespace

The **SQL namespace** refers to the portion of a SQLJ executable statement inside the curly braces. Standard SQL naming restrictions apply here.

> **See Also:** *Oracle Database SQL Language Reference*

However, note that host expressions follow rules of the Java namespace, not the SQL namespace. This applies to the name of a host variable and to everything between the outer parentheses of a host expression.

File Name Requirements and Restrictions

SQLJ source files have the `.sqlj` file name extension. If the source file declares a public class (maximum of one), then the base name of the file must match the name of this class (case-sensitive). If the source file does not declare a public class, then the file name must still be a legal Java identifier, and it is recommended that the file name match the name of the first defined class.

For example, if you define the public class `MySource` in your source file, then your file name must be:

```
MySource.sqlj
```

Note: These file naming requirements follow the Java Language Specification (JLS) and are not SQLJ-specific. These requirements do not directly apply in Oracle Database 11*g*, but it is still advisable to adhere to them.

Considerations for SQLJ in the Middle Tier

There are special considerations if you run SQLJ in the middle tier, such as in an Oracle9*i* Application Server Containers for J2EE (OC4J) environment.

The Oracle JDBC drivers provide Oracle-specific interfaces in the `oracle.jdbc` package. The Oracle SQLJ libraries `runtime12.jar` and `runtime12ee.jar` make full use of these interfaces, but these libraries are not compatible with Oracle JDBC implementations prior to Oracle9*i* Application Server.

In Oracle9*i* Application Server, connections are established through data sources, which typically return instances of the `oracle.jdbc.OracleConnection` interface instead of the older `oracle.jdbc.driver.OracleConnection` class. This is necessary for certain connection functionality, such as distributed transactions (XA). To support such features, connection objects must implement the new interface.

This has the following consequences, relevant in an Oracle9*i* Application Server middle-tier environment, or any situation where data sources are used:

- For maximum portability and flexibility of your code, use `oracle.jdbc.OracleXXX` types instead of `oracle.jdbc.driver.OracleXXX` types.
- For custom Java types (typically for SQL objects and collections), implement `oracle.sql.ORAData` instead of the deprecated `oracle.sql.CustomDatum` interface.

For general information about SQLJ support for data sources and connection JavaBeans, refer to the following sections:

- "Standard Data Source Support" on page 7-9
- "SQLJ-Specific Data Sources" on page 7-11
- "SQLJ-Specific Connection JavaBeans for JavaServer Pages" on page 7-14

4

Basic Language Features

SQLJ statements always begin with a `#sql` token and can be broken into two main categories:

- Declarations: Used for creating Java classes for iterators, which is similar to Java Database Connectivity (JDBC) result sets, or connection contexts, which is designed to help you create strongly typed connections according to the sets of SQL entities being used.
- Executable statements: Used to execute embedded SQL operations.

This chapter discusses the following topics:

Overview of SQLJ Declarations

A SQLJ declaration consists of the `#sql` token followed by the declaration of a class. SQLJ declarations introduce specialized Java types into your application. There are currently two kinds of SQLJ declarations, iterator declarations and connection context declarations, defining Java classes as follows:

- Iterator declarations define iterator classes. Iterators are conceptually similar to JDBC result sets and are used to receive multi-row query data. An iterator is implemented as an instance of an iterator class.
- Connection context declarations define connection context classes. Each connection context class is typically used for connections whose operations use a particular set of SQL entities, such as tables, views, and stored procedures. That is to say, instances of a particular connection context class are used to connect to schemas that include SQL entities with the same names and characteristics. SQLJ implements each database connection as an instance of a connection context class.

  SQLJ includes the predefined `sqlj.runtime.DefaultContext` connection context class. If you only require one connection context class, then you can use `DefaultContext`, which does not require a connection context declaration.

In any iterator or connection context declaration, you may optionally include the following clauses:

- The `implements` clause: Specifies one or more interfaces that the generated class will implement.
- The `with` clause: Specifies one or more initialized constants to be included in the generated class.

This section covers the following topics:

Rules for SQLJ Declarations

SQLJ declarations are allowed in your SQLJ source code anywhere that a class definition would be allowed in standard Java. For example:

```
SQLJ declaration;   // OK (top level scope)

class Outer
{
   SQLJ declaration; // OK (class level scope)

   class Inner
   {
      SQLJ declaration; // OK (nested class scope)
   }

   void func()
   {
      SQLJ declaration; // OK (method block)
   }
}
```

Note: As with standard Java, any public class should be declared in one of the following ways:

- Declare it in a separate source file. The base name of the file should be the same as the class name.
- Declare it at class-level scope or nested-class-level scope. In this case, it may be advisable to use `public static` modifiers.

This is a requirement if you are using the standard `javac` compiler provided with the Sun Microsystems JDK.

Iterator Declarations

An iterator declaration creates a class that defines a kind of iterator for receiving query data. The declaration will specify the column types of the iterator instances, which must match the column types being selected from the database table.

Basic iterator declarations use the following syntax:

```
#sql <modifiers> iterator iterator_classname (type declarations);
```

Modifiers are optional and can be any standard Java class modifiers, such as `public`, `static`, and so on. Type declarations are separated by commas.

There are two categories of iterators, named iterators and positional iterators. For named iterators, you must specify column names and types. For positional iterators, you need to specify only types.

The following is an example of a named iterator declaration:

```
#sql public iterator EmpIter (String ename, double sal);
```

This statement results in the SQLJ translator creating a public `EmpIter` class with a `String` attribute `ename` and a `double` attribute `sal`. You can use this iterator to select data from a database table with corresponding employee name and salary columns of matching names (`ENAME` and `SAL`) and data types (`CHAR` and `NUMBER`).

Declaring `EmpIter` as a positional iterator, instead of a named iterator, can be done as follows:

```
#sql public iterator EmpIter (String, double);
```

See Also: "Multirow Query Results: SQLJ Iterators" on page 4-26

Connection Context Declarations

A connection context declaration creates a connection context class, whose instances are typically used for database connections that use a particular set of SQL entities. Basic connection context declarations use the following syntax:

```
#sql <modifiers> context context_classname;
```

As for iterator declarations, modifiers are optional and can be any standard Java class modifiers. For example:

```
#sql public context MyContext;
```

As a result of this statement, the SQLJ translator creates a public `MyContext` class. In your SQLJ code you can use instances of this class to create database connections to schemas that include a desired set of entities, such as tables, views, and stored procedures. Different instances of `MyContext` might be used to connect to different schemas, but each schema might be expected, for example, to include an `EMP` table, a `DEPT` table, and a `TRANSFER_EMPLOYEE` stored procedure.

Declared connection context classes are an advanced topic and are not necessary for basic SQLJ applications that use only one interrelated set of SQL entities. In basic scenarios, you can use multiple connections by creating multiple instances of the `sqlj.runtime.ref.DefaultContext` class, which does not require any connection context declarations.

See Also: "Connection Considerations" on page 3-4 and "Connection Contexts" on page 7-1

Declaration IMPLEMENTS Clause

When you declare any iterator class or connection context class, you can specify one or more interfaces to be implemented by the generated class.

Use the following syntax for an iterator class:

```
#sql <modifiers> iterator iterator_classname implements intfc1,..., intfcN
     (type declarations);
```

The portion `implements intfc1,..., intfcN` is known as the `implements` clause. Note that in an iterator declaration, the `implements` clause precedes the iterator type declarations.

Here is the syntax for a connection context declaration:

```
#sql <modifiers> context context_classname implements intfc1,..., intfcN;
```

The `implements` clause is potentially useful in either an iterator declaration or a connection context declaration, but is more likely to be useful in iterator declarations, particularly in implementing the `sqlj.runtime.Scrollable` or `sqlj.runtime.ForUpdate` interface. Scrollable iterators are supported in the Oracle SQLJ implementation.

> **Note:** The SQLJ `implements` clause corresponds to the Java `implements` clause.

The following example uses an `implements` clause in declaring a named iterator class. Presume you have created a package, `mypackage`, that includes an iterator interface, `MyIterIntfc`.

```
#sql public iterator MyIter implements mypackage.MyIterIntfc
     (String ename, int empno);
```

The declared class `MyIter` will implement the `mypackage.MyIterIntfc` interface.

The following example declares a connection context class that implements an interface named `MyConnCtxtIntfc`. Presume that it is in the package `mypackage`.

```
#sql public context MyContext implements mypackage.MyConnCtxtIntfc;
```

> **See Also:** "Using the IMPLEMENTS Clause in Iterator Declarations" on page 7-35 and "Using the IMPLEMENTS Clause in Connection Context Declarations" on page 7-8

Declaration WITH Clause

In declaring a connection context class or iterator class, you can use a `with` clause to specify and initialize one or more constants to be included in the definition of the generated class. Most of this usage is standard, although Oracle implementation adds some extended functionality for iterator declarations.

This section covers the following topics:

- Standard WITH Clause Usage
- Oracle-Specific WITH Clause Usage
- Example: Returnability

Standard WITH Clause Usage

In using a `with` clause, the constants that are produced are always `public static final`. Use the following syntax for an iterator class:

```
#sql <modifiers> iterator iterator_classname with (var1=value1,..., varN=valueN)
     (type declarations);
```

The portion `with (var1=value1,..., varN=valueN)` is the `with` clause. Note that in an iterator declaration, the `with` clause precedes the iterator type declarations.

Where there is both a `with` clause and an `implements` clause, the `implements` clause must come first. Note that parentheses are used to enclose `with` lists, but not `implements` lists.

Here is the syntax for a connection context declaration that uses a `with` clause:

```
#sql <modifiers> context context_classname with (var1=value1,..., varN=valueN);
```

Note: A predefined set of standard SQLJ constants can be defined in a `with` clause. However, not all of these constants are meaningful to Oracle Database 11*g* or to the Oracle SQLJ run time.

Attempts to define constants other than the standard constants is legal with Oracle Database 11*g*, but might not be portable to other SQLJ implementations and will generate a warning if you have the `-warn=portable` flag enabled. For information about this flag, refer to "Translator Warnings (-warn)" on page 8-33.

Supported WITH Clause Constants

The Oracle SQLJ implementation supports the following standard constants in connection context declarations:

- `typeMap`: a `String` literal defining the name of a type map properties resource

  Oracle also supports the use of `typeMap` in iterator declarations.

  See Also: "Oracle-Specific WITH Clause Usage" on page 4-6

- `dataSource`: a `String` literal defining the name under which a data source is looked up in the `InitialContext`

  See Also: "Standard Data Source Support" on page 7-9

The Oracle SQLJ implementation supports the following standard constants in iterator declarations:

- `sensitivity`: `SENSITIVE/ASENSITIVE/INSENSITIVE`, to define the sensitivity of a scrollable iterator

  See Also: "Scrollable Iterator Sensitivity" on page 7-37

- `returnability`: `true/false`, to define whether an iterator can be returned from a Java stored procedure or function

  See Also: "Example: Returnability" on page 4-7

Unsupported WITH Clause Constants

If you have SQLJ code that uses these constants, then they will not cause an error but will result in no operation. The Oracle SQLJ implementation does *not* support the following standard constants in connection context declarations:

- `path`: a `String` literal defining the name of a path to be prepended for resolution of Java stored procedures and functions
- `transformGroup`: a `String` literal defining the name of a SQL transformation group that can be applied to SQL types

The Oracle SQLJ implementation does *not* support the following standard constants, involving cursor states, in iterator declarations:

- `holdability`: `true/false`, determining cursor holdability

  The concept of holdability is defined in the SQL specification. A cursor that is holdable can, subject to application request, be kept open and positioned on the current row even when a transaction is completed. Use of the cursor can then be continued in the next transaction of the same SQL session, however, subject to some limitations.

- `updateColumns`: a `String` literal containing a comma-delimited list of column names

  An iterator declaration having a `with` clause that specifies `updateColumns` must also have an `implements` clause that specifies the `sqlj.runtime.ForUpdate` interface. The Oracle SQLJ implementation enforces this, even though `updateColumns` is currently unsupported.

The following is a sample connection context declaration using `typeMap`:

```
#sql public context MyContext with (typeMap="MyPack.MyClass");
```

The declared class `MyContext` will define a `String` attribute `typeMap` that will be `public static final` and initialized to the value `MyPack.MyClass`. This value is the fully qualified class name of a `ListResourceBundle` implementation that provides the mapping between SQL and Java types for statements executed on instances of the `MyContext` class.

The following is a sample iterator declaration using `sensitivity`:

```
#sql public iterator MyAsensitiveIter with (sensitivity=ASENSITIVE)
     (String ename, int empno);
```

This declaration sets the cursor sensitivity to `ASENSITIVE` for the `MyAsensitiveIter` named iterator class.

The following example uses both an `implements` clause and a `with` clause:

```
#sql public iterator MyScrollableIterator implements sqlj.runtime.Scrollable
     with (holdability=true) (String ename, int empno);
```

This declaration implements the interface `sqlj.runtime.Scrollable` and enables the cursor `holdability` for a named iterator class.

> **Note:** `holdability` is currently *not* supported.

Oracle-Specific WITH Clause Usage

In addition to the standard `with` clause usage in a connection context declaration to associate a type map with the connection context class, the Oracle SQLJ implementation enables you to use a `with` clause to associate a type map with the iterator class in an iterator declaration. For example:

```
#sql iterator MyIterator with (typeMap="MyTypeMap") (Person pers, Address addr);
```

If you use Oracle-specific code generation and use type maps in your application, then your iterator and connection context declarations must use the same type maps.

> **See Also:** "Code Considerations and Limitations with Oracle-Specific Code Generation" on page 3-28

Example: Returnability

Use `returnability=true` in the `with` clause of a SQLJ iterator declaration to specify that the iterator can be returned from a Java stored procedure to a SQL or PL/SQL statement as a REF CURSOR. With the default `returnability=false` setting, the iterator cannot be returned in this manner, and an attempt to do so will result in a SQL exception at run time.

Create the following database table:

```
create table sqljRetTab(str varchar2(30));
insert into sqljRetTab values ('sqljRetTabCol');
```

Define the `RefCursorSQLJ` class in the `RefCursorSQLJ.sqlj` source file as follows. Note that the iterator type `MyIter` uses `returnability=true`.

```
public class RefCursorSQLJ
{
   #sql static public iterator MyIter with (returnability=true) (String str);

   static public MyIter sqljUserRet() throws java.sql.SQLException
   {
        MyIter iter=null;
        try {
        #sql iter = {select str from sqljRetTab};
        } catch (java.sql.SQLException e)
        {
           e.printStackTrace();
           throw e;
        }
        System.err.println("iter is " + iter);
        return iter;
   }
}
```

Load `RefCursorSQLJ.sqlj` into the Oracle Java virtual machine (JVM) inside the database as follows:

```
% loadjava -u scott/tiger -r -f -v RefCursorSQLJ.sqlj
```

Invoke the Java stored procedure defined for the `sqljUserRet()` method:

```
create or replace package refcur_pkg as
     type refcur_t is ref cursor;
end;
/
create or replace function sqljUserRet
return refcur_pkg.refcur_t as
language java
name 'RefCursorSQLJ.sqljUserRet() return
RefCursorSQLJ.MyIter';
/
select scott.sqljUserRet from dual;
```

Here is the result of the `SELECT` statement:

```
SQLJRET1
--------------------
CURSOR STATEMENT : 1

STR
------------------------------
sqljRetTabCol
```

Overview of SQLJ Executable Statements

A SQLJ executable statement consists of the `#sql` token followed by a SQLJ clause, which uses syntax that follows a specified standard for embedding executable SQL statements in Java code. The embedded SQL operation of a SQLJ executable statement can be any SQL operation supported by the JDBC driver.

This section covers the following topics:

- Rules for SQLJ Executable Statements
- SQLJ Clauses
- Specifying Connection Context Instances and Execution Context Instances
- Executable Statement Examples
- PL/SQL Blocks in Executable Statements

Rules for SQLJ Executable Statements

A SQLJ executable statement must adhere to the following rules:

- It is permitted in Java code wherever Java block statements are permitted. That is, it is permitted inside method definitions and static initialization blocks.
- Its embedded SQL operation must be enclosed in curly braces: `{ . . . }`.
- It must be terminated with a semi-colon (;).

Note:

- It is recommended that you do *not* close the SQL operation with a semi-colon. The parser will detect the end of the operation when it encounters the closing curly brace of the SQLJ clause.
- Everything inside the curly braces of a SQLJ executable statement is treated as SQL syntax and must follow SQL rules, with the exception of Java host expressions.
- During offline parsing of SQL operations, all SQL syntax is checked. However, during online semantics-checking only data manipulation language (DML) operations can be parsed and checked. Data definition language (DDL) operations, transaction-control operations, or any other kinds of SQL operations cannot be parsed and checked.

SQLJ Clauses

A SQLJ clause is the executable part of a statement, consisting of everything to the right of the `#sql` token. This consists of embedded SQL inside curly braces, preceded by a Java result expression if appropriate, such as `result` in the following example:

```
#sql { SQL operation };   // For a statement with no output, like INSERT
...
#sql result = { SQL operation };   // For a statement with output, like SELECT
```

A clause without a result expression, such as in the first SQLJ statement in the example, is known as a statement clause. A clause that does have a result expression, such as in the second SQLJ statement in the example, is known as an assignment clause.

A result expression can be anything from a simple variable that takes a stored-function return value to an iterator that takes several columns of data from a multi-row `SELECT`, where the iterator can be an instance of an iterator class or subclass.

A SQL operation in a SQLJ statement can use standard SQL syntax only or can use a clause with syntax specific to SQLJ.

Table 4–1 lists supported SQLJ statement clauses and Table 4–2 lists supported SQLJ assignment clauses. The last two entries in Table 4–1 are general categories for statement clauses that use standard SQL syntax or Oracle PL/SQL syntax, as opposed to SQLJ-specific syntax.

Table 4–1 SQLJ Statement Clauses

Category	Functionality	More Information
`SELECT INTO` clause	Select data into Java host expressions.	"Single-Row Query Results: SELECT INTO Statements" on page 4-24
`FETCH` clause	Fetch data from a positional iterator.	"Using Positional Iterators" on page 4-34
`COMMIT` clause	Commit changes to the data.	"Using Manual COMMIT and ROLLBACK" on page 3-20
`ROLLBACK` clause	Cancel changes to the data.	"Using Manual COMMIT and ROLLBACK" on page 3-20
`SAVEPOINT` `RELEASE SAVEPOINT` `ROLLBACK TO` clauses	Set a savepoint for future rollbacks, release a specified savepoint, roll back to a savepoint.	"Using Savepoints" on page 3-21
`SET TRANSACTION` clause	Use advanced transaction control for access mode and isolation level.	"Advanced Transaction Control" on page 7-41
Procedure clause	Call a stored procedure.	"Calling Stored Procedures" on page 4-43
Assignment clause	Assign values to Java host expressions.	"Assignment Statements (SET)" on page 4-41
SQL clause	Use standard SQL syntax and functionality: `UPDATE`, `INSERT`, `DELETE`.	*Oracle Database SQL Language Reference*
PL/SQL block	Use `BEGIN..END` or `DECLARE..BEGIN..END` anonymous block inside SQLJ statement.	"PL/SQL Blocks in Executable Statements" on page 4-11 *Oracle Database PL/SQL Language Reference*

Table 4–2 SQLJ Assignment Clauses

Category	Functionality	More Information
Query clause	Select data into a SQLJ iterator.	"Multirow Query Results: SQLJ Iterators" on page 4-26
Function clause	Call a stored function.	"Calling Stored Functions" on page 4-44
Iterator conversion clause	Convert a JDBC result set to a SQLJ iterator.	"Converting from Result Sets to Named or Positional Iterators" on page 7-48

Note: A SQLJ statement is referred to by the same name as the clause that makes up the body of that statement. For example, an executable statement consisting of `#sql` followed by a `SELECT INTO` clause is referred to as a `SELECT INTO` statement.

Specifying Connection Context Instances and Execution Context Instances

If you have defined multiple database connections and want to specify a particular connection context instance for an executable statement, then use the following syntax:

```
#sql [conn_context_instance] { SQL operation };
```

See Also: "Connection Considerations" on page 3-4

If you have defined one or more execution context instances of the `sqlj.runtime.ExecutionContext` class and want to specify one of them for use with an executable statement, then use the following syntax:

```
#sql [exec_context_instance] { SQL operation };
```

You can use an execution context instance to provide status or control of the SQL operation of a SQLJ executable statement. For example, you can use execution context instances in multithreading situations where multiple operations are occurring on the same connection.

See Also: "Execution Contexts" on page 7-24

You can also specify both a connection context instance and an execution context instance:

```
#sql [conn_context_instance, exec_context_instance] { SQL operation };
```

Note:

- Include the square brackets around connection context instances and execution context instances. They are part of the syntax.
- If you specify both a connection context instance and an execution context instance, then the connection context instance must come first.

Executable Statement Examples

This section provides examples of elementary SQLJ executable statements.

Elementary INSERT

The following example demonstrates a basic `INSERT`. The statement clause does not require any syntax specific to SQLJ.

Consider an employee table `EMP` with the following rows:

```
CREATE TABLE EMP (
   ENAME VARCHAR2(10),
   SAL NUMBER(7,2) );
```

Use the following SQLJ executable statement, which uses only standard SQL syntax, to insert Joe as a new employee into the `EMP` table, specifying his name and salary:

```
#sql { INSERT INTO emp (ename, sal) VALUES ('Joe', 43000) };
```

Elementary INSERT with Connection Context or Execution Context Instances

The following examples use `ctx` as a connection context instance, which is an instance of either the default `sqlj.runtime.ref.DefaultContext` or a class that you have previously declared in a connection context declaration, and `execctx` as an execution context instance:

```
#sql [ctx] { INSERT INTO emp (ename, sal) VALUES ('Joe', 43000) };

#sql [execctx] { INSERT INTO emp (ename, sal) VALUES ('Joe', 43000) };

#sql [ctx, execctx] { INSERT INTO emp (ename, sal) VALUES ('Joe', 43000) };
```

A Simple SQLJ Method

This example demonstrates a simple method using SQLJ code, demonstrating how SQLJ statements interrelate with and are interspersed with Java statements. The SQLJ statement uses standard `INSERT INTO` *`table`* `VALUES` syntax supported by the Oracle SQL implementation. The statement also uses Java host expressions, marked by colons (:), to define the values. Host expressions are used to pass data between the Java code and SQL instructions.

```
public static void writeSalesData (int[] itemNums, String[] itemNames)
       throws SQLException
{
  for (int i =0; i < itemNums.length; i++)
    #sql { INSERT INTO sales VALUES(:(itemNums[i]), :(itemNames[i]), SYSDATE) };
}
```

Note:

- The `throws SQLException` *is* required.
- SQLJ function calls also use a `VALUES` token, but these situations are not related semantically.

PL/SQL Blocks in Executable Statements

PL/SQL blocks can be used within the curly braces of a SQLJ executable statement just as SQL operations can, as in the following example:

```
#sql {
   DECLARE
      n NUMBER;
   BEGIN
      n := 1;
      WHILE n <= 100 LOOP
         INSERT INTO emp (empno) VALUES(2000 + n);
         n := n + 1;
      END LOOP;
   END
};
```

This example goes through a loop that inserts new employees in the `emp` table, creating employee numbers `2001` through `2100`. It presumes data other than the employee number will be filled in later.

Simple PL/SQL blocks can also be coded in a single line as follows:

```
#sql { <DECLARE ...> BEGIN ... END; };
```

Using PL/SQL anonymous blocks within SQLJ statements is one way to use dynamic SQL in your application. You can also use dynamic SQL directly through SQLJ extensions provided by Oracle or through JDBC code within a SQLJ application.

> **See Also:** "Support for Dynamic SQL" on page 7-50 and "SQLJ and JDBC Interoperability" on page 7-44

Note: Remember that using PL/SQL in your SQLJ code would prevent portability to other platforms, because PL/SQL is Oracle-specific.

Java Host, Context, and Result Expressions

This section discusses three categories of Java expressions used in SQLJ code: host expressions, context expressions, and result expressions. Host expressions are the most frequently used Java expressions. Another category of expressions, called meta bind expressions, are used specifically for dynamic SQL operations and use syntax similar to that of host expressions.

> **See Also:** "Support for Dynamic SQL" on page 7-50

SQLJ uses Java host expressions to pass arguments between Java code and SQL operations. This is how you pass information between Java and SQL. Host expressions are interspersed within the embedded SQL operations in the SQLJ source code.

The most basic kind of host expression, consisting of only a Java identifier, is referred to as a host variable. A context expression specifies a connection context instance or execution context instance to be used for a SQLJ statement. A result expression specifies an output variable for query results or a function return.

This section covers the following topics:

- Overview of Host Expressions
- Basic Host Expression Syntax
- Examples of Host Expressions
- Overview of Result Expressions and Context Expressions

Overview of Host Expressions

Any valid Java expression can be used as a host expression. In the simplest case, the expression consists of just a single Java variable. Other kinds of host expressions include the following:

- Arithmetic expressions
- Java method calls with return values
- Java class field values
- Array elements
- Conditional expressions (`a ? b : c`)
- Logical expressions
- Bitwise expressions

Java identifiers used as host variables or in host expressions can represent any of the following:

- Local variables
- Declared parameters
- Class fields
- Static or instance method calls

Local variables used in host expressions can be declared anywhere that other Java variables can be declared. Fields can be inherited from a superclass.

Java variables that are legal in the Java scope where the SQLJ executable statement appears can be used in a host expression in a SQL statement, presuming its type is convertible to or from a SQL data type. Host expressions can be input, output, or input-output.

See Also: "Supported Types for Host Expressions" on page 5-1

Basic Host Expression Syntax

A host expression is preceded by a colon (:). If the desired mode of the host expression is not the default, then the colon must be followed by `IN`, `OUT`, or `INOUT`, as appropriate, before the host expression itself. These are referred to as mode specifiers. The default is `OUT` if the host expression is part of an `INTO`-list or is the assignment expression in a `SET` statement. Otherwise, the default is `IN`. Any `OUT` or `INOUT` host expression must be assignable.

Note: When using the default, you can still include the mode specifier if desired.

The SQL code that surrounds a host expression can use any vendor-specific SQL syntax. Therefore, no assumptions can be made about the syntax when parsing the SQL operations and determining the host expressions. To avoid any possible

ambiguity, any host expression that is not a simple host variable (in other words, that is more complex than a nondotted Java identifier) must be enclosed in parentheses.

To summarize the basic syntax:

- For a simple host variable without a mode specifier, put the host variable after the colon, as in the following example:

  ```
  :hostvar
  ```

- For a simple host variable with a mode specifier, put the mode specifier after the colon and put white space (space, tab, newline, or comment) between the mode specifier and the host variable, as in the following example:

  ```
  :INOUT hostvar
  ```

  The white space is required to distinguish between the mode specifier and the variable name.

- For any other host expression, enclose the expression in parentheses and place it after the mode specifier or after the colon, if there is no mode specifier, as in the following examples:

  ```
  :IN(hostvar1+hostvar2)
  :(hostvar3*hostvar4)
  :(index--)
  ```

  White space is not required after the mode specifier in this example, because the parenthesis is a suitable separator. However, a white space after the mode specifier is allowed.

  An outer set of parentheses is needed even if the expression already starts with a begin-parenthesis, as in the following examples:

  ```
  :((x+y).z)
  :(((y)x).myOutput())
  ```

Syntax Notes

Keep the following in mind regarding the host expression syntax:

- White space is always allowed after the colon as well as after the mode specifier. Wherever white space is allowed, you can also have a comment.

  You can have any of the following in the SQL namespace:

  – SQL comments after the colon and before the mode specifier

  – SQL comments after the colon and before the host expression if there is no mode specifier

  – SQL comments after the mode specifier and before the host expression

  You can have the following in the Java namespace:

  – Java comments within the host expression (inside the parentheses)

- The `IN`, `OUT`, and `INOUT` syntax used for host variables and expressions are not case-sensitive. These tokens can be in uppercase, lowercase, or mixed.

- Do not confuse the `IN`, `OUT`, and `INOUT` syntax of SQLJ host expressions with similar `IN`, `OUT`, and `IN OUT` syntax used in PL/SQL declarations to specify the mode of parameters passed to PL/SQL stored functions and procedures.

Usage Notes

Keep the following in mind regarding the usage of host expressions:

- A simple host variable can appear multiple times in the same SQLJ statement, as follows:
  - If the host variable appears only as an input variable, then there are no restrictions or complications.
  - If at least one appearance of the host variable is as an output variable in a PL/SQL block, then you will receive a portability warning if the translator `-warn=portability` flag is set. SQLJ run-time behavior in this situation is vendor-specific. The Oracle SQLJ run time uses value semantics, as opposed to reference semantics, for all occurrences of the host variable.
  - If at least one appearance of the host variable is as an output variable in a stored procedure call, stored function call, `SET` statement, or `INTO`-list, then you will *not* receive any warning. SQLJ run-time behavior in this situation is standardized, using value semantics.

  Note: The term output refers to `OUT` or `INOUT` variables, as applicable.

- If a host expression that is a simple host variable appears multiple times in a SQLJ statement, then by default each appearance is treated completely independently of the other appearances, using value semantics. However, if you use the SQLJ translator `-bind-by-identifier=true` setting, then this is not the case. With a `true` setting, multiple appearances of the same host variable in a given SQLJ statement or PL/SQL block are treated as a single bind occurrence.

  See Also: "Binding Host Expressions by Identifier (-bind-by-identifier)" on page 8-55

- When binding a string host expression into a `WHERE` clause for comparison against `CHAR` data, be aware that there is a SQLJ option, `-fixedchar`, that accounts for blank padding in the `CHAR` column when the comparison is made.

  See Also: "CHAR Comparisons with Blank Padding (-fixedchar)" on page 8-46

Examples of Host Expressions

The following examples will help clarify the preceding syntax discussion.

Note: Some of these examples use `SELECT INTO` statements, which are described in "Single-Row Query Results: SELECT INTO Statements" on page 4-24.

Example 1

In this example, two input host variables are used, one as a test value for a `WHERE` clause and one to contain new data to be sent to the database.

Presume you have a database employee table `emp` with an `ename` column for employee names and a `sal` column for employee salaries. The relevant Java code that defines the host variables is as follows:

```
String empname = "SMITH";
double salary = 25000.0;
...
#sql { UPDATE emp SET sal = :salary WHERE ename = :empname };
```

IN is the default, but you can state it explicitly as well:

```
#sql { UPDATE emp SET sal = :IN salary WHERE ename = :IN empname };
```

As you can see, the colon (:) can immediately precede the variable when not using the IN token, but :IN must be followed by white space before the host variable.

Example 2

This example uses an output host variable in a SELECT INTO statement, where you want to find out the name of the employee whose employee number 28959.

```
String empname;
...
#sql { SELECT ename INTO :empname FROM emp WHERE empno = 28959 };
```

OUT is the default for an INTO-list, but you can state it explicitly as well:

```
#sql { SELECT ename INTO :OUT empname FROM emp WHERE empno = 28959 };
```

This statement looks in the empno column of the emp table for employee number 28959, selects the name in the ename column of that row, and outputs it to the empname output host variable, which is a Java String.

Example 3

This example uses an arithmetic expression as an input host expression. The Java variables balance and minPmtRatio are multiplied, and the result is used to update the minPayment column of the creditacct table for account number 537845.

```
float balance = 12500.0;
float minPmtRatio = 0.05;
...
#sql { UPDATE creditacct SET minPayment = :(balance * minPmtRatio)
       WHERE acctnum = 537845 };
```

Alternatively, to use the IN token:

```
#sql { UPDATE creditacct SET minPayment = :IN (balance * minPmtRatio)
       WHERE acctnum = 537845 };
```

Example 4

This example shows the use of the output of a method call as an input host expression and also uses an input host variable. This statement uses the value returned by getNewSal() to update the sal column in the emp table for the employee who is specified by the Java empname variable. Java code initializing the host variables is also shown.

```
String empname = "SMITH";
double raise = 0.1;
...
#sql {UPDATE emp SET sal = :(getNewSal(raise, empname))
      WHERE ename = :empname};
```

Overview of Result Expressions and Context Expressions

A context expression is an input expression that specifies the name of a connection context instance or an execution context instance to be used in a SQLJ executable statement. Any legal Java expression that yields such a name can be used.

A result expression is an output expression used for query results or a function return. It can be any legal Java expression that is assignable, meaning that it can logically appear on the left side of an equals sign. This is sometimes referred to as an l-value.

The following examples can be used for either result expressions or context expressions:

- Local variables
- Declared parameters
- Class fields
- Array elements

Result expressions and context expressions appear lexically in the SQLJ space, unlike host expressions, which appear lexically in the SQL space, that is, inside the curly brackets of a SQLJ executable statement. Therefore, a result expression or context expression must *not* be preceded by a colon.

Evaluation of Java Expressions at Run Time

This section discusses the evaluation of Java host expressions, connection context expressions, execution context expressions, and result expressions when your application executes.

Following is a simplified representation of a SQLJ executable statement that uses all these kinds of expressions:

```
#sql [connctxt_exp, execctxt_exp] result_exp = { SQL with host expression };
```

Java expressions can be used as any of the following, as appropriate:

- Connection context expression: Evaluated to specify the connection context instance to be used
- Execution context expression: Evaluated to specify the execution context instance to be used
- Result expression: To receive results, for example, from a stored function
- Host expression

For ISO standard code generation, the evaluation of Java expressions is well-defined, even for the use of any side effects that depend on the order in which expressions are evaluated.

For Oracle-specific code generation, evaluation of Java expressions follows the ISO standard when there are no side effects, except when the `-bind-by-identifier` option is enabled, but is implementation-specific and subject to change when there are side effects.

Note: The following discussion and the related examples later do *not* apply to Oracle-specific code generation. If you use side effects as described here, then request ISO code generation during translation.

The following is a summary, for ISO code, of the overall order of evaluation, execution, and assignment of Java expressions for each statement that executes during run time.

1. If there is a connection context expression, then it is evaluated immediately, before any other Java expressions are evaluated.
2. If there is an execution context expression, then it is evaluated after any connection context expression, but before any result expression.
3. If there is a result expression, then it is evaluated after any context expressions, but before any host expressions.
4. After evaluation of any context or result expressions, host expressions are evaluated from left to right as they appear in the SQL operation. As each host expression is encountered and evaluated, its value is saved to be passed to SQL.

   Each host expression is evaluated once and only once.
5. `IN` and `INOUT` parameters are passed to SQL, and the SQL operation is executed.
6. After execution of the SQL operation, the output parameters, Java `OUT` and `INOUT` host expressions, are assigned output in order from left to right as they appear in the SQL operation.

   Each output host expression is assigned once and only once.
7. The result expression, if there is one, is assigned output last.

> **Note:** Host expressions inside a PL/SQL block are all evaluated together before any statements within the block are executed. They are evaluated in the order in which they appear, regardless of the control flow within the block.

Once the expressions in a statement have been evaluated, input and input-output host expressions are passed to SQL, and then the SQL operation is executed. After execution of the SQL operation, assignments are made to Java output host expressions, input-output host expressions, and result expressions as follows:

1. `OUT` and `INOUT` host expressions are assigned output in order from left to right.
2. The result expression, if there is one, is assigned output last.

Note that during run time, all host expressions are treated as distinct values, even if they share the same name or reference the same object. The execution of each SQL operation is treated as if invoking a remote method, and each host expression is taken as a distinct parameter. Each input or input-output parameter is evaluated and passed as it is first encountered, before any output assignments are made for that statement, and each output parameter is also taken as distinct and is assigned exactly once.

It is also important to remember that each host expression is evaluated only once. An `INOUT` expression is evaluated when it is first encountered. When the output assignment is made, the expression itself is not reevaluated nor are any side-effects repeated.

Examples of Evaluation of Java Expressions at Run Time (ISO Code Generation)

This section discusses, for ISO code generation, how Java expressions are evaluated when your application executes.

Note: Do *not* count on these effects if you use Oracle-specific code generation. Request ISO code generation during translation if you depend on such effects.

Evaluation of Prefix and Postfix Operators

When a Java expression contains a Java postfix increment or decrement operator, the incrementing or decrementing occurs *after* the expression has been evaluated. Similarly, when a Java expression contains a Java prefix increment or decrement operator, the incrementing or decrementing occurs *before* the expression is evaluated. This is equivalent to how these operators are handled in standard Java code.

The following is an example of postfix operator:

```
int indx = 1;
...
#sql { ... :OUT (array[indx]) ... :IN (indx++) ... };
```

This example is evaluated as follows:

```
#sql { ... :OUT (array[1]) ... :IN (1) ... };
```

The `indx` variable is incremented to 2 and will have that value the next time it is encountered, but not until after `:IN (indx++)` has been evaluated.

The following is the example of postfix operator:

```
int indx = 1;
...
#sql { ... :OUT (array[indx++]) ... :IN (indx++) ... };
```

This example is evaluated as follows:

```
#sql { ... :OUT (array[1]) ... :IN (2) ... };
```

The variable `indx` is incremented to 2 after the first expression is evaluated, but before the second expression is evaluated. It is incremented to 3 after the second expression is evaluated and will have that value the next time it is encountered.

The following example consists of both prefix and postfix operators:

```
int indx = 1;
...
#sql { ... :OUT (array[++indx]) ... :IN (indx++) ... };
```

This example is evaluated as follows:

```
#sql { ... :OUT (array[2]) ... :IN (2) ... };
```

The variable `indx` is incremented to 2 before the first expression is evaluated. It is incremented to 3 after the second expression is evaluated and will have that value the next time it is encountered.

Evaluation Order of IN, INOUT, and OUT Host Expressions

Host expressions are evaluated from left to right. Whether an expression is `IN`, `INOUT`, or `OUT` makes no difference when it is evaluated. All that matters is its position in the left-to-right order.

Consider the following example:

```
int[5] arry;
```

```
int n = 0;
...
#sql { SET :OUT (arry[n]) = :(++n) };
```

This example is evaluated as follows:

```
#sql { SET :OUT (arry[0]) = 1 };
```

One might expect input expressions to be evaluated before output expressions, but that is not the case. The expression `:OUT (arry[n])` is evaluated first because it is the left-most expression. Then `n` is incremented prior to evaluation of `++n`, because it is being operated on by a prefix operator. Then `++n` is evaluated as 1. The result will be assigned to `arry[0]`, not `arry[1]`, because `0` was the value of `n` when it was originally encountered.

Expressions in PL/SQL Blocks Are Evaluated Before Statements Are Executed

Host expressions in a PL/SQL block are all evaluated in one sequence, before any have been executed. Consider the following example:

```
int x=3;
int z=5;
...
#sql { BEGIN :OUT x := 10; :OUT z := :x; END };
System.out.println("x=" + x + ", z=" + z);
```

This example is evaluated as follows:

```
#sql { BEGIN :OUT x := 10; :OUT z := 3; END };
```

Therefore, it would print `x=10, z=3`.

All expressions in a PL/SQL block are evaluated before any are executed. In this example, the host expressions in the second statement, `:OUT z` and `:x`, are evaluated before the first statement is executed. In particular, the second statement is evaluated while `x` still has its original value of `3`, before it has been assigned the value `10`.

Consider another example of how expressions are evaluated within a PL/SQL block:

```
int x=1, y=4, z=3;
...
#sql { BEGIN
         :OUT x := :(y++) + 1;
         :OUT z := :x;
       END };
```

This example is evaluated as follows:

```
#sql { BEGIN
          :OUT x := 4 + 1;
          :OUT z := 1;
       END };
```

The postfix increment operator is executed after `:(y++)` is evaluated, so the expression is evaluated as `4`, which is the initial value of `y`. The second statement, `:OUT z := :x`, is evaluated before the first statement is executed. Therefore, `x` still has its initialized value of `1`. After execution of this block, `x` will have the value `5` and `z` will have the value `1`.

The following example demonstrates the difference between two statements appearing in a PL/SQL block in one SQLJ executable statement, and the same statements appearing in separate (consecutive) SQLJ executable statements.

First, consider the following, where two statements are in a PL/SQL block.

```
int y=1;
...
#sql { BEGIN :OUT y := :y + 1; :OUT x := :y + 1; END };
```

This example is evaluated as follows:

```
#sql { BEGIN :OUT y := 1 + 1; :OUT x := 1 + 1; END };
```

The `:y` in the second statement is evaluated before either statement is executed. Therefore, `y` has not yet received its output from the first statement. After execution of this block, both `x` and `y` have the value `2`.

Now, consider the situation where the same two statements are in PL/SQL blocks in separate SQLJ executable statements.

```
int y=1;
#sql { BEGIN :OUT y := :y + 1; END };
#sql { BEGIN :OUT x := :y + 1; END };
```

The first statement is evaluated as follows:

```
#sql { BEGIN :OUT y := 1 + 1; END };
```

Then, it is executed and `y` is assigned the value `2`.

After execution of the first statement, the second statement is evaluated as follows:

```
#sql { BEGIN :OUT x := 2 + 1; END };
```

This time, as opposed to the previous PL/SQL block example, `y` has already received the value `2` from execution of the previous statement. Therefore, `x` is assigned the value `3` after execution of the second statement.

Expressions in PL/SQL Blocks Are Always Evaluated Once Only

Each host expression is evaluated once, and only once, regardless of program flow and logic.

Consider the following example of evaluation of host expression in a loop:

```
int count = 0;
...
#sql {
   DECLARE
      n NUMBER
   BEGIN
      n := 1;
      WHILE n <= 100 LOOP
         :IN (count++);
         n := n + 1;
      END LOOP;
   END
};
```

The Java `count` variable will have the value `0` when it is passed to SQL, because it is operated on by a postfix operator, as opposed to a prefix operator. It will then be incremented to `1` and will hold that value throughout execution of this PL/SQL block.

It is evaluated only once as the SQLJ executable statement is parsed and then is replaced by the value 1 prior to SQL execution.

Consider the following example that illustrates the evaluation of host expressions in conditional blocks. This example demonstrates how each expression is always evaluated, regardless of the program flow. As the block is executed, only one branch of the IF...THEN...ELSE construct can be executed. However, before the block is executed, all expressions in the block are evaluated in the order that the statements appear.

```
int x;
...
(operations on x)
...
#sql {
   DECLARE
      n NUMBER
   BEGIN
      n := :x;
      IF n < 10 THEN
         n := :(x++);
      ELSE
         n := :x * :x;
      END LOOP;
   END
};
```

Say the operations performed on x resulted in x having a value of 15. When the PL/SQL block is executed, the ELSE branch will be executed and the IF branch will not. However, all expressions in the PL/SQL block are evaluated before execution, regardless of program logic or flow. Therefore, x++ is evaluated, then x is incremented, and then each x is evaluated in the (x * x) expression. The IF...THEN...ELSE block is evaluated as follows:

```
IF n < 10 THEN
   n := 15;
ELSE
   n := :16 * :16;
END LOOP;
```

After execution of this block, given an initial value of 15 for x, n will have the value 256.

Output Host Expressions Are Assigned Left to Right, Before Result Expression

Remember that OUT and INOUT host expressions are assigned in order from left to right, and then the result expression, if any, is assigned last. If the same variable is assigned more than once, then it will be overwritten according to this order, with the last assignment taking precedence.

The following example contains multiple output host expressions referencing the same variable:

```
#sql { CALL foo(:OUT x, :OUT x) };
```

If foo() outputs the values 2 and 3, respectively, then x will have the value 3 after the SQLJ executable statement has finished executing. The right-hand assignment will be performed last, thereby taking precedence.

The following example contains multiple output host expressions referencing the same object:

```
MyClass x = new MyClass();
MyClass y = x;
...
#sql { ... :OUT (x.field):=1 ... :OUT (y.field):=2 ... };
```

After execution of the SQLJ executable statement, `x.field` will have a value of `2`, and not `1`, because `x` is the same object as `y`, and `field` was assigned the value of `2` after it was assigned the value of `1`.

The following example demonstrates the difference between having the output results of a function assigned to a result expression and having the results assigned to an `OUT` host expression. Consider the following function, with the `invar` input parameter, the `outvar` output parameter, and a return value:

```
CREATE FUNCTION fn(invar NUMBER, outvar OUT NUMBER)
   RETURN NUMBER AS BEGIN
      outvar := invar + invar;
      return (invar * invar);
   END fn;
```

Now consider an example where the output of the function is assigned to a result expression:

```
int x = 3;
#sql x = { VALUES(fn(:x, :OUT x)) };
```

The function will take `3` as the input, will calculate `6` as the output, and will return `9`. After execution, the `:OUT x` will be assigned first, giving `x` a value of `6`. But finally the result expression is assigned, giving `x` the return value of `9` and overwriting the value of `6` previously assigned to `x`. So `x` will have the value `9` the next time it is encountered.

Now consider an example where the output of the function is assigned to an `OUT` host variable instead of a result expression:

```
int x = 3;
#sql { BEGIN :OUT x := fn(:x, :OUT x); END };
```

In this case, there is no result expression and the `OUT` variables are simply assigned left to right. After execution, the first `:OUT x`, on the left side of the equation, is assigned first, giving `x` the function return value of `9`. However, proceeding left to right, the second `:OUT x`, on the right side of the equation, is assigned last, giving `x` the output value of `6` and overwriting the value of `9` previously assigned to `x`. Therefore, `x` will have the value `6` the next time it is encountered.

Note: Some unlikely cases have been used in these examples to explain the concepts of how host expressions are evaluated. In practice, it is not advisable to use the same variable in both an `OUT` or `INOUT` host expression or in an `IN` host expression inside a single statement or PL/SQL block. The behavior in such cases is well defined in the Oracle SQLJ implementation, but this practice is not covered in the SQLJ specification. Therefore, code written in this manner will not be portable. Such code will generate a warning from the SQLJ translator if the `portable` flag is set during semantics-checking.

Restrictions on Host Expressions

Do not use `in`, `out`, and `inout` as identifiers in host expressions unless they are enclosed in parentheses. Otherwise, they might be mistaken for mode specifiers. This is not case-sensitive.

For example, you could use an input host variable called `in`, as follows:

```
:(in)
```

or:

```
:IN(in)
```

Single-Row Query Results: SELECT INTO Statements

When only a single row of data is being returned, SQLJ enables you to assign selected items directly to Java host expressions inside SQL syntax. This is done using the `SELECT INTO` statement. This section covers the following topics:

- SELECT INTO Syntax
- Examples of SELECT INTO Statements
- Examples with Host Expressions in SELECT-List
- SELECT INTO Error Conditions

SELECT INTO Syntax

The syntax for a `SELECT INTO` statement is as follows:

```
#sql { SELECT expression1,..., expressionN  INTO :host_exp1,..., :host_expN
       FROM table <optional_clauses> };
```

Keep in mind the following:

- The items *`expression1`* through *`expressionN`* are expressions specifying what is to be selected from the database. These can be any expressions valid for any `SELECT` statement. This list of expressions is referred to as the `SELECT`-list. In a simple case, these would be names of columns from a database table. It is also legal to include a host expression in the `SELECT`-list.
- The items *`host_exp1`* through *`host_expN`* are target host expressions, such as variables or array elements. This list of host expressions is referred to as the `INTO`-list.
- The item *`table`* is the name of the database table, view, or snapshot from which you are selecting the data.
- The item *`optional_clauses`* is for any additional clauses you want to include that are valid in a `SELECT` statement, such as a `WHERE` clause.

A `SELECT INTO` statement must return one, and only one, row of data, otherwise an error will be generated at run time.

The default is `OUT` for a host expression in an `INTO`-list, but you can optionally state this explicitly:

```
#sql { SELECT column_name1, column_name2  INTO :OUT host_exp1, :OUT host_exp2
       FROM table WHERE condition };
```

Trying to use an IN or INOUT token in the INTO-list will result in an error at translation time.

Note:

- Permissible syntax for *expression1* through *expressionN*, the *table*, and the optional clauses is the same as for any SQL SELECT statement.
- There can be any number of SELECT-list and INTO-list items, as long as they match. That is, one INTO-list item per SELECT-list item, with compatible types.

Examples of SELECT INTO Statements

The examples in this section use an employee table EMP with the following rows:

```
CREATE TABLE EMP (
   EMPNO NUMBER(4),
   ENAME VARCHAR2(10),
   HIREDATE DATE );
```

The following is an example of a SELECT INTO statement with a single host expression in the INTO-list:

```
String empname;
#sql { SELECT ename INTO :enpname FROM emp WHERE empno=28959 };
```

The following is an example of a SELECT INTO statement with multiple host expressions in the INTO-list:

```
String empname;
Date hdate;
#sql { SELECT ename, hiredate INTO :empname, :hdate FROM emp
       WHERE empno=28959 };
```

Examples with Host Expressions in SELECT-List

It is legal to use Java host expressions in the SELECT-list as well as in the INTO-list. For example, you can select directly from one host expression into another, though this is of limited usefulness, as follows:

```
...
#sql { SELECT :name1 INTO :name2 FROM emp WHERE empno=28959 };
...
```

More realistically, you may want to perform an operation or concatenation on the data selected, as in the following examples. Assume Java variables were previously declared and assigned, as necessary.

```
...
#sql { SELECT sal + :raise INTO :newsal FROM emp WHERE empno=28959 };
...

...
#sql { SELECT :(firstname + " ") || emp_last_name INTO :name FROM myemp
       WHERE empno=28959 };
...
```

In the second example, presume `myemp` is a table much like the `emp` table but with an `emp_last_name` column instead of an `ename` column. In the `SELECT` statement, `firstname` is prepended to a single space (" "), using a Java host expression and the Java string concatenation operator (+). This result is then passed to the SQL engine, which uses SQL string concatenation operator (||) to append the last name.

SELECT INTO Error Conditions

Remember that `SELECT INTO` statements are intended for queries that return exactly one row of data only. A `SELECT INTO` query that finds zero rows or multiple rows will result in an exception, as follows:

- A `SELECT INTO` finding no rows will return an exception with a SQL state of `2000`, representing a "no data" condition.
- A `SELECT INTO` finding multiple rows will return an exception with a SQL state of `21000`, representing a cardinality violation.

You can retrieve the SQL state through the `getSQLState()` method of the `java.sql.SQLException` class.

See Also: "Retrieving SQL States and Error Codes" on page 3-17

This is vendor-independent behavior that is specified in the ISO SQLJ standard. There is no vendor-specific error code in these cases. The error code is always `0`.

Multirow Query Results: SQLJ Iterators

A large number of SQL operations are multirow queries. Processing multirow query results in SQLJ requires a SQLJ iterator. A SQLJ iterator is a strongly typed version of a JDBC result set and is associated with the underlying database cursor. SQLJ iterators are primarily used to take query results from a `SELECT` statement.

Additionally, Oracle offers SQLJ extensions that enable you to use SQLJ iterators and result sets in the following ways:

- As `OUT` host variables in executable SQL statements
- As `INTO`-list targets, such as in a `SELECT INTO` statement
- As a return type from a stored function call
- As column types in iterator declarations (essentially, nested iterators)

Note: To use a SQLJ iterator in any of these ways, its class must be declared as `public`. If you declared it at the class level or nested-class level, then it might be advisable to declare it as `public static`.

This section covers the following topics:

- Iterator Concepts
- General Steps in Using an Iterator
- Named, Positional, and Result Set Iterators
- Using Named Iterators
- Using Positional Iterators

- Using Iterators and Result Sets as Host Variables
- Using Iterators and Result Sets as Iterator Columns

See Also: "Iterator Class Implementation and Advanced Functionality" on page 7-33

Iterator Concepts

Using a SQLJ iterator declaration results in a strongly typed iterator. This is the typical usage for iterators and takes particular advantage of SQLJ semantics-checking features during translation. It is also possible, and at times advantageous, to use weakly typed iterators. There are generic classes you can instantiate in order to use a weakly typed iterator.

This section covers the following topics:

- Introduction to Strongly Typed Iterators
- Introduction to Weakly Typed Iterators

Introduction to Strongly Typed Iterators

Before using a strongly typed iterator object, you must declare an iterator class. An iterator declaration specifies a Java class that SQLJ constructs for you, where the class attributes define the type and, optionally, the name of the columns of data in the iterator.

A SQLJ iterator object is an instance of such a specifically declared iterator class, with a fixed number of columns of predefined type. This is as opposed to a JDBC result set object, which is a standard `java.sql.ResultSet` instance and can, in principle, contain any number of columns of any type.

When you declare an iterator, you specify either just the data type of the selected columns, or both the data type and the name of the selected columns:

- Specifying the names and data types defines a named iterator class.
- Specifying just the data types defines a positional iterator class.

The data types and names, if applicable, that you declare determine how query results will be stored in iterator objects you instantiate from that class. SQL data retrieved into an iterator object are converted to the Java types specified in the iterator declaration.

When you query to populate a named iterator object, the name and data type of the columns in the `SELECT` statement must match the name and data type of the iterator columns. However, this is not case-sensitive. The order of the columns in the `SELECT` statement is irrelevant. All that matters is that each column name in the `SELECT` statement matches an iterator column name. In the simplest case, the database column names directly match the iterator column names.

For example, data from an `ENAME` column in a database table can be selected and put into an iterator `ename` column. Alternatively, you can use an alias to map a database column name to an iterator column name if the names differ. Also, in a more complicated query, you can perform an operation between two columns and alias the result to match the corresponding iterator column name.

Because SQLJ iterators are strongly typed, they offer the benefit of Java type-checking during the SQLJ semantics-checking phase.

As an example, consider the following table:

```
CREATE TABLE EMPSAL (
```

```
   EMPNO NUMBER(4),
   ENAME VARCHAR2(10),
   OLDSAL NUMBER(10),
   RAISE NUMBER(10) );
```

Given this table, you can declare a named iterator as follows.

```
#sql iterator SalNamedIter (int empno, String ename, float raise);
```

Once declared, you can use this named iterator as follows:

```
class MyClass {
   void func() throws SQLException {
      ...
      SalNamedIter niter;
      #sql niter = { SELECT ename, empno, raise FROM empsal };

      ... process niter ...
   }
}
```

This is a simple case where the iterator column names match the table column names. Note that the order of items in the `SELECT` statement does not matter when you use a named iterator. Data is matched by name, not position.

When you query to populate a positional iterator object, the data is retrieved according to the order in which you select the columns. Data from the first column selected from the database table is placed into the first column of the iterator, and so on. The data types of the table columns must be convertible to the types of the iterator columns, but the names of the database columns are irrelevant, as the iterator columns have no names.

Given the `EMPSAL` table, you can declare a positional iterator as follows:

```
#sql iterator SalPosIter (int, String, float);
```

You can use this positional iterator as follows:

```
class MyClass {
   void func() throws SQLException {
      ...
      SalPosIter piter;
      #sql piter = { SELECT empno, ename, raise FROM empsal };

      ... process piter ...
   }
}
```

Note that the order of the data items in the `SELECT` statement must be the same as in the iterator. The processing differs between named iterators and positional iterators.

General Iterator Notes

In addition to the preceding concepts, be aware of the following general notes about iterators:

- The `SELECT *` syntax is allowed in populating an iterator, but is not recommended. In the case of a positional iterator, this requires that the number of columns in the table be equal to the number of columns in the iterator, and that the data types match in order. In the case of a named iterator, this requires that the number of columns in the table be greater than or equal to the number of columns in the iterator and that the name and data type of each iterator column match a

database table column. However, if the number of columns in the table is greater, then a warning will be generated unless the translator `-warn=nostrict` flag is set.

> **See Also:** "Translator Warnings (-warn)" on page 8-33

- Positional and named iterators are distinct and incompatible kinds of Java classes. An iterator object of one kind cannot be cast to an iterator object of the other kind.
- Unlike a SQL cursor, an iterator instance is a first-class Java object. That is, it can be passed and returned as a method parameter, for example. Also, an iterator instance can be declared using Java class modifiers, such as `public` or `private`.
- SQLJ supports interoperability and conversion between SQLJ iterators and JDBC result sets.

  > **See Also:** "SQLJ Iterator and JDBC Result Set Interoperability" on page 7-48

- Generally speaking, the contents of an iterator is determined only by the state of the database at the time of execution of the `SELECT` statement that populated it. Subsequent `UPDATE`, `INSERT`, `DELETE`, `COMMIT`, or `ROLLBACK` operations have no effect on the iterator or its contents. The exception to this is if you declare an iterator to be scrollable and sensitive to changes in the data.

  > **See Also:** "Effect of Commits and Rollbacks on Iterators and Result Sets" on page 3-21, "Declaring Scrollable Iterators" on page 7-36, and "Scrollable Iterator Sensitivity" on page 7-37.

Introduction to Weakly Typed Iterators

In case you do not want to declare an iterator class, the Oracle SQLJ implementation enables you to use a weakly typed iterator. Such iterators are known as result set iterators. To use a plain, that is, nonscrollable result set iterator, instantiate the `sqlj.runtime.ResultSetIterator` class. To use a scrollable result set iterator, instantiate the `sqlj.runtime.ScrollableResultSetIterator` class.

The drawback to using result set iterators, compared to strongly typed iterators, is that SQLJ cannot perform as much semantics-checking for your queries.

> **See Also:** "Scrollable Iterators" on page 7-36 and "Result Set Iterators" on page 7-36

General Steps in Using an Iterator

You must follow the following general steps to use SQLJ named or positional iterator:

1. Use a SQLJ declaration to define the iterator class (in other words, to define the iterator type).
2. Declare a variable of the iterator class.
3. Populate the iterator variable with the results from a SQL query, using a `SELECT` statement.
4. Access the query columns in the iterator. How to accomplish this differs between named iterators and positional iterators.
5. When you finish processing the results of the query, close the iterator to release its resources.

Named, Positional, and Result Set Iterators

There are advantages and appropriate situations for each kind of SQLJ iterator.

Named iterators enable greater flexibility. Because data selection into a named iterator matches the columns in the `SELECT` statement to iterator columns by name, you need not be concerned about the order in your query. This is less prone to error, as it is not possible for data to be placed into the wrong column. If the names do not match, then the SQLJ translator will generate an error when it checks the SQL statements against the database.

Positional iterators offer a familiar paradigm and syntax to developers who have experience with other embedded-SQL languages. With named iterators you use a `next()` method to retrieve data, while with positional iterators you use `FETCH INTO` syntax similar to that of Pro*C, for example. Each fetch implicitly advances to the next available row of the iterator before retrieving the next set of values.

However, positional iterators do offer less flexibility than named iterators, because you are selecting data into iterator columns by position, instead of by name. You must be certain of the order of items in your `SELECT` statement. Also, you must select data into all columns of the iterator. It is possible to have data written into the wrong iterator column, if the data type of that column happens to match the data type of the table column being selected.

Access to individual data elements is also less convenient with positional iterators. Named iterators, because they store data by name, are able to have convenient accessor methods for each column. For example, there would be an `ename()` method to retrieve data from an `ename` iterator column. With positional iterators, you must fetch data directly into Java host expressions with the `FETCH INTO` statement, and the host expressions must be in the correct order.

If you do not want to declare strongly typed iterator classes for your queries, then you can choose the alternative of using weakly typed result set iterators. Result set iterators are most convenient when converting JDBC code to SQLJ code. You must balance this consideration against the fact that result set iterators, either `ResultSetIterator` instances or `ScrollableResultSetIterator` instances, do not allow complete SQLJ semantics-checking during translation. With named or positional iterators, SQLJ verifies that the data types of columns in the `SELECT` statement match the Java types into which the data will be materialized. With result set iterators, this is not possible.

See Also: "Result Set Iterators" on page 7-36

Comparative Iterator Notes

Be aware of the following notes regarding SQLJ iterators:

- In populating a positional iterator, the number of columns you select from the database must equal the number of columns in the iterator. In populating a named iterator, the number of columns you select from the database can never be less than the number of columns in the iterator, but can be greater than the number of columns in the iterator if you have the translator `-warn=nostrict` flag set. Unmatched columns are ignored in this case.

  See Also: "Translator Warnings (-warn)" on page 8-33

- Although the term "fetching" often refers to fetching data from a database, remember that a `FETCH INTO` statement for a positional iterator does not necessarily involve a round trip to the server. This depends on the row-prefetch value. This is because you are fetching data from the iterator, and not the database.

However, if the row-prefetch value is `1`, then each fetch does involve a separate trip to the database. The row-prefetch value determines how many rows are retrieved with each trip to the database.

See Also: "Row Prefetching" on page 10-2

- Result set iterators use the same `FETCH INTO` syntax that is used with positional iterators and are subject to the same restriction at run time. That is, the number of data items in the `SELECT`-list must match the number of variables that are assigned data in the `FETCH` statement.

Using Named Iterators

When you declare a named iterator class, you declare the name as well as the data type of each column of the iterator. When you select data into a named iterator, the columns in the `SELECT` statement must match the iterator columns in two ways:

- The name of each data item in the `SELECT` statement, either a table column name or an alias, must match an iterator column name. However, this is not case-sensitive. That is, `ename` or `Ename` would match `ENAME`).
- The data type of each iterator column must be compatible with the data type of the corresponding data item in the `SELECT` statement according to standard JDBC type mappings.

The order in which attributes are declared in the named iterator class declaration is irrelevant. Data is selected into the iterator based on name alone.

A named iterator has a `next()` method to retrieve data row by row and an accessor method for each column to retrieve the individual data items. The accessor method names are identical to the column names. Unlike most accessor method names in Java, accessor method names in named iterator classes do not start with `get`. For example, a named iterator object with a column `sal` would have a `sal()` accessor method.

Note: The following restrictions apply in naming the columns of a named iterator:

- Column names cannot use Java reserved words.
- Column names cannot have the same name as utility methods provided in named iterator classes, such as the `next()`, `close()`, `getResultSet()`, and `isClosed()` methods. For scrollable named iterators, this includes additional methods such as `previous()`, `first()`, and `last()`.

Declaring Named Iterator Classes

Use the following syntax to declare a named iterator class:

```
#sql <modifiers> iterator classname <implements clause> <with clause>
     ( type-name-list );
```

In this syntax, *`modifiers`* is an optional sequence of legal Java class modifiers, *`classname`* is the desired class name for the iterator, and *`type-name-list`* is a list of the Java types and names equivalent to or compatible with the column types and column names in a database table.

The `implements` clause and `with` clause are optional, specifying interfaces to implement and variables to define and initialize, respectively.

> **See Also:** "Declaration IMPLEMENTS Clause" on page 4-3 and "Declaration WITH Clause" on page 4-4

Consider the following table:

```
CREATE TABLE PROJECTS (
   ID NUMBER(4),
   PROJNAME VARCHAR(30),
   START_DATE DATE,
   DURATION NUMBER(3) );
```

You can declare the following named iterator to use with this table:

```
#sql public iterator ProjIter (String projname, int id, Date deadline);
```

This will result in an iterator class with columns of data accessible, using the following provided accessor methods: `projname()`, `id()`, and `deadline()`.

> **Note:** As with standard Java, any public class should be declared in one of the following ways:
>
> - Declare it in a separate source file. The base name of the file should be the same as the class name.
> - Declare it at class-level scope or nested-class-level scope, with `public static` modifiers.
>
> This is a requirement if you are using the standard `javac` compiler provided with the Sun Microsystems JDK.

Instantiating and Populating Named Iterators

Continuing to use the `PROJECTS` table and `ProjIter` iterator defined in the preceding section, note that there are columns in the table whose names and data types match the `id` and `projname` columns of the iterator. However, you must use an alias and perform an operation to populate the `deadline` column of the iterator. Following is an example:

```
ProjIter projsIter;

#sql projsIter = { SELECT start_date + duration AS deadline, projname, id
                   FROM projects WHERE start_date + duration >= sysdate };
```

This calculates a deadline for each project by adding its duration to its start date, then aliases the results as `deadline` to match the `deadline` iterator column. It also uses a `WHERE` clause so that only future deadlines are processed, that is, deadlines beyond the current system date in the database.

Similarly, you must create an alias if you want to use a function call. Suppose you have a `MAXIMUM()` function that takes a `DURATION` entry and an integer as input and returns the maximum of the two. For example, you could input the value `3` to ensure that each project has at least a three-month duration in your application.

Now, presume you are declaring your iterator as follows:

```
#sql public iterator ProjIter2 (String projname, int id, float duration);
```

You could use the `MAXIMUM()` function in your query, with an alias for the result, as follows:

```
ProjIter2 projsIter2;
```

```
#sql projsIter2 = { SELECT id, projname, maximum(duration, 3) AS duration
                    FROM projects };
```

Generally, you must use an alias in your query for any data item in the `SELECT` statement whose name is not a legal Java identifier or does not match a column name in the iterator.

Remember that in populating a named iterator, the number of columns you select from the database can never be less than the number of columns in the iterator. The number of columns you select can be greater than the number of columns in the iterator, because unmatched columns are ignored. However, this will generate a warning, unless you have the SQLJ `-warn=nostrict` option set.

Accessing Named Iterators

Use the `next()` method of the named iterator object to step through the data that was selected into it. To access each column of each row, use the accessor methods generated by SQLJ, typically inside a `while` loop.

Whenever `next()` is called:

- If there is another row to retrieve from the iterator, then `next()` retrieves the row and returns `true`.
- If there are no more rows to retrieve, `next()` returns `false`.

The following is an example of how to access the data of a named iterator, repeating the declaration, instantiation, and population code illustrated in the preceding section.

> **Note:** Each iterator has a `close()` method that you must always call when you finish retrieving data from the iterator. This is necessary to close the iterator and free its resources.

Presume the following iterator class declaration:

```
#sql public iterator ProjIter (String projname, int id, Date deadline);
```

Populate and then access an instance of this iterator class as follows:

```
// Declare the iterator variable
ProjIter projsIter;

// Instantiate and populate iterator; order of SELECT doesn't matter
#sql projsIter = { SELECT start_date + duration AS deadline, projname, id
                   FROM projects WHERE start_date + duration >= sysdate };

// Process the results
while (projsIter.next()) {
   System.out.println("Project name is " + projsIter.projname());
   System.out.println("Project ID is " + projsIter.id());
   System.out.println("Project deadline is " + projsIter.deadline());
}

// Close the iterator
projsIter.close();
...
```

Note the convenient use of the `projname()`, `id()`, and `deadline()` accessor methods to retrieve the data. Note also that the order of the `SELECT` items does not matter, nor does the order in which the accessor methods are used.

However, remember that accessor method names are created with the case exactly as in your declaration of the iterator class. The following will generate compilation errors.

Consider the following declaration of the iterator:

```
#sql iterator Cursor1 (String NAME);
```

The code for using the iterator is as follows:

```
...
Cursor1 c1;
#sql c1 = { SELECT NAME FROM TABLE };
while (c1.next()) {
   System.out.println("The name is " + c1.name());
}
...
```

The `Cursor1` class has a method called `NAME()`, and not `name()`. You will have to use `c1.NAME()` in the `System.out.println` statement.

Using Positional Iterators

When you declare a positional iterator class, you declare the data type of each column but not the column name. The Java types into which the columns of the SQL query results are selected must be compatible with the data types of the SQL data. The names of the database columns or data items in the `SELECT` statement are irrelevant. Because names are not used, the order in which you declare your positional iterator Java types must exactly match the order in which the data is selected.

To retrieve data from a positional iterator once data has been selected into it, use a `FETCH INTO` statement followed by an `endFetch()` method call to determine if you have reached the end of the data.

Declaring Positional Iterator Classes

Use the following syntax to declare a positional iterator class:

```
#sql <modifiers> iterator classname <implements clause> <with clause>
     ( position-list );
```

In this syntax, *`modifiers`* is an optional sequence of legal Java class modifiers and the *`position-list`* is a list of Java types compatible with the column types in a database table.

The `implements` clause and `with` clause are optional, specifying interfaces to implement and variables to define and initialize, respectively.

> **See Also:** "Declaration IMPLEMENTS Clause" on page 4-3 and "Declaration WITH Clause" on page 4-4

Now consider an employee table `EMP` with the following rows:

```
CREATE TABLE EMP (
   EMPNO NUMBER(4),
   ENAME VARCHAR2(10),
   SAL NUMBER(7,2) );
```

And consider the following positional iterator declaration:

```
#sql public iterator EmpIter (String, int, float);
```

This example defines the `EmpIter` Java class with unnamed `String`, `int`, and `float` columns. Note that the table columns and iterator columns are in a different order, with the `String` corresponding to `ENAME` and the `int` corresponding to `EMPNO`. The order of the iterator columns determines the order in which you must select the data.

Note: As with standard Java, any public class should be declared in one of the following ways:

- Declare it in a separate source file. The base name of the file should be the same as the class name.
- Declare it at class-level scope or nested-class-level scope, with `public static` modifiers.

This is a requirement if you are using the standard `javac` compiler provided with the Sun Microsystems JDK.

Instantiating and Populating Positional Iterators

Instantiating and populating a positional iterator is no different than doing so for a named iterator, except that you must be certain that the data items in the `SELECT` statement are in the proper order.

The three data types in the `EmpIter` iterator class are compatible with the types of the `EMP` table, but be careful how you select the data, because the order is different. The following will work, because the data items in the `SELECT` statement are in the same order as the iterator columns:

```
EmpIter empsIter;

#sql empsIter = { SELECT ename, empno, sal FROM emp };
```

Remember that in populating a positional iterator, the number of columns you select from the database must equal the number of columns in the iterator.

Accessing Positional Iterators

Access the columns defined by a positional iterator using SQL `FETCH INTO` syntax. The `INTO` part of the command specifies Java host variables that receive the results columns. The host variables must be in the same order as the corresponding iterator columns. Use the `endFetch()` method provided with all positional iterator classes to determine whether the last fetch reached the end of the data.

Note:

- The `endFetch()` method initially returns `true` before any rows have been fetched, then returns `false` once a row has been successfully retrieved, and then returns `true` again when a `FETCH` finds no more rows to retrieve. Therefore, you must perform the `endFetch()` test *after* the `FETCH INTO` statement. If your `endFetch()` test precedes the `FETCH INTO` statement, then you will never retrieve any rows, because `endFetch()` would be true before your first `FETCH` and you would immediately break out of the `while` loop.
- The `endFetch()` test must be *before* the results are processed, however, because the `FETCH` does not throw a SQL exception when it reaches the end of the data, it just triggers the next `endFetch()` call to return `true`. If there is no `endFetch()` test before results are processed, then your code will try to process `NULL` or invalid data from the first `FETCH` attempt after the end of the data had been reached.
- Each iterator has a `close()` method that you must always call once you finish retrieving data from it. This is necessary to close the iterator and free its resources.

The following is an example, repeating the declaration, instantiation, and population code illustrated in the preceding section. Note that the Java host variables in the `SELECT` statement are in the same order as the columns of the positional iterator, which is mandatory.

First, presume the following iterator class declaration:

```
#sql public iterator EmpIter (String, int, float);
```

Populate and then access an instance of this iterator class as follows:

```
// Declare and initialize host variables
int empnum=0;
String empname=null;
float salary=0.0f;

// Declare an iterator instance
EmpIter empsIter;

#sql empsIter = { SELECT ename, empno, sal FROM emp };

while (true) {
   #sql { FETCH :empsIter INTO :empnum, :empname, :salary };
   if (empsIter.endFetch()) break;  // This test must be AFTER fetch,
                                    // but before results are processed.
   System.out.println("Name is " + empname);
   System.out.println("Employee number is " + empnum);
   System.out.println("Salary is " + salary);
}

// Close the iterator
empsIter.close();
...
```

The `empname`, `empnum`, and `salary` variables are Java host variables whose types must match the types of the iterator columns.

Do not use the `next()` method for a positional iterator. A `FETCH` operation calls it implicitly to move to the next row.

Note: Host variables in a `FETCH INTO` statement must always be initialized because they are assigned in one branch of a conditional statement. Otherwise, you will get a compiler error indicating they may never be assigned. `FETCH` can assign the variables only if there was a row to be fetched.

Positional Iterator Navigation with the next() Method

The positional iterator `FETCH` clause discussed in the preceding section performs a movement, an implicit `next()` call, before it populates the host variables, if any. As an alternative, the Oracle SQLJ implementation supports a special `FETCH` syntax in conjunction with explicit `next()` calls in order to use the same movement logic as with JDBC result sets and SQLJ named iterators. Using this special `FETCH` syntax, the semantics differ. There is no implicit `next()` call before the `INTO`-list is populated.

See Also: "FETCH CURRENT Syntax: from JDBC Result Sets to SQLJ Iterators" on page 7-40

Using Iterators and Result Sets as Host Variables

SQLJ supports SQLJ iterators and JDBC result sets as host variables. Using iterators and result sets is fundamentally the same, with differences in declarations and in accessor methods to retrieve the data.

Note:

- Additionally, SQLJ supports iterators and result sets as return variables for stored functions.
- The Oracle JDBC drivers currently do *not* support result sets as input host variables. There is a `setCursor()` method in the `OraclePreparedStatement` class, but it raises an exception at run time if called.

For the examples in this section, consider the following department and employee tables:

```
CREATE TABLE DEPT (
   DEPTNO NUMBER(2),
   DNAME VARCHAR2(14) );

CREATE TABLE EMP (
   EMPNO NUMBER(4),
   ENAME VARCHAR2(10),
   SAL NUMBER(7,2),
   DEPTNO NUMBER(2) );
```

Example: Use of Result Set as OUT Host Variable

This example uses a JDBC result set as an output host variable.

```
...
ResultSet rs;
...
#sql { BEGIN
          OPEN :OUT rs FOR SELECT ename, empno FROM emp;
       END };

while (rs.next())
{
   String empname = rs.getString(1);
   int empnum = rs.getInt(2);
}
rs.close();
...
```

This example opens the result set `rs` in a PL/SQL block to receive data from a `SELECT` statement, selects data from the `ENAME` and `EMPNO` columns of the `EMP` table, and then loops through the result set to retrieve data into local variables.

Example: Use of Iterator as OUT Host Variable

This example uses a named iterator as an output host variable.

The iterator can be declared as follows:

```
#sql public <static> iterator EmpIter (String ename, int empno);
```

The `public` modifier is required, and the `static` modifier may be advisable if your declaration is at class level or nested-class level.

This iterator can be used as follows:

```
...
EmpIter iter;
...
#sql { BEGIN
          OPEN :OUT iter FOR SELECT ename, empno FROM emp;
       END };

while (iter.next())
{
   String empname = iter.ename();
   int empnum = iter.empno();

   ...process/output empname and empnum...
}
iter.close();
...
```

This example opens the iterator `iter` in a PL/SQL block to receive data from a `SELECT` statement, selects data from the `ENAME` and `EMPNO` columns of the `EMP` table, and then loops through the iterator to retrieve data into local variables.

Example: Use of Iterator as OUT Host Variable for SELECT INTO

This example uses a named iterator as an output host variable, taking data through a `SELECT INTO` statement. `OUT` is the default for host variables in an `INTO`-list.

The iterator can be declared as follows:

```
#sql public <static> iterator ENameIter (String ename);
```

The public modifier is required, and the static modifier may be advisable if your declaration is at class level or nested-class level.

This iterator can be used as follows:

```
...
ENameIter enamesIter;
String deptname;
...

#sql { SELECT dname, cursor
      (SELECT ename FROM emp WHERE deptno = dept.deptno)
       INTO :deptname, :enamesIter FROM dept WHERE deptno = 20 };

System.out.println(deptname);
while (enamesIter.next())
{
   System.out.println(enamesIter.ename());
}
enamesIter.close();
...
```

This example uses nested SELECT statements to accomplish the following:

- Select the name of department number 20 from the DEPT table, selecting it into the deptname output host variable.
- Query the EMP table to select all employees whose department number is 20, selecting the resulting cursor into the enamesIter output host variable, which is a named iterator.
- Print the department name.
- Loop through the named iterator printing employee names. This prints the names of all employees in the department.

In most cases, using SELECT INTO is more convenient than using nested iterators if you are retrieving a single row in the outer SELECT, although that option is also available. Also, with nested iterators, you would have to process the data to determine how many rows there are in the outer SELECT. With SELECT INTO you are assured of just one row.

Using Iterators and Result Sets as Iterator Columns

The Oracle SQLJ implementation includes extensions that allow iterator declarations to specify columns of ResultSet type or columns of other iterator types declared within the current scope. In other words, iterators and result sets can exist within iterators. These column types are used to retrieve a column in the form of a cursor. This is useful for nested SELECT statements that return nested table information.

The following examples are functionally identical. Each uses a nested result set or iterator, that is, result sets or iterators in a column within an iterator, to print all the employees in each department in the DEPT table. The first example uses result sets within a named iterator, the second example uses named iterators within a named iterator, and the third example uses named iterators within a positional iterator.

Following are the steps:

1. Select each department name (DNAME) from the DEPT table.
2. Do a nested SELECT into a cursor to get all employees from the EMP table for each department.

3. Put the department names and sets of employees into the outer iterator (`iter`), which has a name column and an iterator column. The cursor with the employee information for any given department goes into the iterator column of the row of the outer iterator corresponding to the department.
4. Go through a nested loop that, for each department, prints the department name and then loops through the inner iterator to print all employee names for that department.

Example: Result Set Column in a Named Iterator

This example uses a column of type `ResultSet` in a named iterator.

The iterator can be declared as follows:

```
#sql iterator DeptIter (String dname, ResultSet emps);
```

The code that uses the iterator is as follows:

```
...
DeptIter iter;
...
#sql iter = { SELECT dname, cursor
              (SELECT ename FROM emp WHERE deptno = dept.deptno)
               AS emps FROM dept };

while (iter.next())
{
   System.out.println(iter.dname());
   ResultSet enamesRs = iter.emps();
   while (enamesRs.next())
   {
      String empname = enamesRs.getString(1);
      System.out.println(empname);
   }
   enamesRs.close();
}
iter.close();
...
```

Example: Named Iterator Column in a Named Iterator

This example uses a named iterator that has a column whose type is that of a previously defined named iterator (nested iterators).

The iterator declaration is as follows:

```
#sql iterator ENameIter (String ename);
#sql iterator DeptIter (String dname, ENameIter emps);
```

The code that uses this iterator is as follows:

```
...
DeptIter iter;
...
#sql iter = { SELECT dname, cursor
              (SELECT ename FROM emp WHERE deptno = dept.deptno)
               AS emps FROM dept };

while (iter.next())
{
   System.out.println(iter.dname());
   ENameIter enamesIter = iter.emps();
```

```
   while (enamesIter.next())
   {
      System.out.println(enamesIter.ename());
   }
   enamesIter.close();
}
iter.close();
...
```

Example: Named Iterator Column in a Positional Iterator

This example uses a positional iterator that has a column whose type is that of a previously defined named iterator (nested iterators). This uses the `FETCH INTO` syntax of positional iterators. This example is functionally equivalent to the previous two.

Note that because the outer iterator is a positional iterator, there does not have to be an alias to match a column name, as was required when the outer iterator was a named iterator in the previous example.

The iterator declaration is as follows:

```
#sql iterator ENameIter (String ename);
#sql iterator DeptIter (String, ENameIter);
```

The code that uses this iterator is as follows:

```
...
DeptIter iter;
...
#sql iter = { SELECT dname, cursor
              (SELECT ename FROM emp WHERE deptno = dept.deptno)
               FROM dept };

while (true)
{
   String dname = null;
   ENameIter enamesIter = null;
   #sql { FETCH :iter INTO :dname, :enamesIter };
   if (iter.endFetch()) break;
   System.out.println(dname);
   while (enamesIter.next())
   {
      System.out.println(enamesIter.ename());
   }
   enamesIter.close();
}
iter.close();
...
```

Assignment Statements (SET)

SQLJ enables you to assign a value to a Java host expression inside a SQL operation. This is known as an assignment statement and is accomplished using the following syntax:

```
#sql { SET :host_exp = expression };
```

The *host_exp* is the target host expression, such as a variable or array index. The *expression* could be a number, host expression, arithmetic expression, function call, or other construct that yields a valid result into the target host expression.

The default is OUT for a target host expression in an assignment statement, but you can optionally state this explicitly:

```
#sql { SET :OUT host_exp = expression };
```

Trying to use an IN or INOUT token in an assignment statement will result in an error at translation time.

The preceding statements are functionally equivalent to the following PL/SQL code:

```
#sql { BEGIN :OUT host_exp := expression; END };
```

Here is a simple example of an assignment statement:

```
#sql { SET :x = foo1() + foo2() };
```

This statement assigns to x the sum of the return values of foo1() and foo2() and assumes that the type of x is compatible with the type of the sum of the outputs of these functions.

Consider the following additional examples:

```
int i2;
java.sql.Date dat;
...
#sql { SET :i2 = TO_NUMBER(substr('750 etc.', 1, 3)) +
       TO_NUMBER(substr('250 etc.', 1, 3)) };
...
#sql { SET :dat = sysdate };
...
```

The first statement will assign to i2 the value 1000. The substr() calls takes the first three characters of the strings, that is, "750" and "250". The TO_NUMBER() calls convert the strings to the numbers 750 and 250.

The second statement will read the database system date and assign it to dat.

An assignment statement is especially useful when you are performing operations on return variables from functions stored in the database. You do not need an assignment statement to simply assign a function result to a variable, because you can accomplish this using standard function call syntax. You also do not need an assignment statement to manipulate output from Java functions, because you can accomplish that in a typical Java statement. So you can presume that foo1() and foo2() are stored functions in the database, not Java functions.

Stored Procedure and Function Calls

SQLJ provides convenient syntax for calling stored procedures and stored functions in the database. These procedures and functions could be written in Java, PL/SQL, or any other language supported by the database.

A stored function requires a result expression in your SQLJ executable statement to accept the return value and, optionally, can take input, output, or input-output parameters as well.

A stored procedure does not have a return value. Optionally, it can take input, output, or input-output parameters. A stored procedure can return output through any output or input-output parameter.

> **Note:** Remember that instead of using the following procedure-call and function-call syntax, you can optionally use JPublisher to create Java wrappers for PL/SQL stored procedures and functions, and then call the Java wrappers as you would call any other Java methods. JPublisher is discussed in "JPublisher and the Creation of Custom Java Classes" on page 6-20. For additional information, refer to the *Oracle Database JPublisher User's Guide*.

This section covers the following topics:

- Calling Stored Procedures
- Calling Stored Functions
- Using Iterators and Result Sets as Stored Function Returns

Calling Stored Procedures

Stored procedures do not have a return value but can take a list with input, output, and input-output parameters. Stored procedure calls use the `CALL` token. The `CALL` token is followed by white space and then the procedure name. There must be a space after the `CALL` token to differentiate it from the procedure name. There *cannot* be a set of outer parentheses around the procedure call. This differs from the syntax for function calls. The syntax for the `CALL` token is as follows:

```
#sql { CALL PROC(<PARAM_LIST>) };
```

PROC is the name of the stored procedure, which can optionally take a list of input, output, and input-output parameters. *PROC* can include a schema or package name as well, such as `SCOTT.MYPROC()`.

Presume that you have defined the following PL/SQL stored procedure:

```
CREATE OR REPLACE PROCEDURE MAX_DEADLINE (deadline OUT DATE) IS
   BEGIN
      SELECT MAX(start_date + duration) INTO deadline FROM projects;
   END;
```

This reads the `PROJECTS` table, looks at the `START_DATE` and `DURATION` columns, calculates `start_date + duration` in each row, then takes the maximum `START_DATE + DURATION` total, and assigns it to `DEADLINE`, which is an output parameter of type `DATE`.

In SQLJ, you can call this `MAX_DEADLINE` procedure as follows:

```
java.sql.Date maxDeadline;
...
#sql { CALL MAX_DEADLINE(:out maxDeadline) };
```

For any parameters, you must use the host expression tokens `IN`, `OUT`, and `INOUT` appropriately, to match the input, output, and input-output designations of the stored procedure. Additionally, the types of the host variables you use in the parameter list must be compatible with the parameter types of the stored procedure.

Note: If you want your application to be compatible with Oracle7 Database, then do *not* include empty parentheses for the parameter list if the procedure takes no parameters. For example:

```
#sql { CALL MAX_DEADLINE };
```

not:

```
#sql { CALL MAX_DEADLINE() };
```

Calling Stored Functions

Stored functions have a return value and can also take a list of input, output, and input-output parameters. Stored function calls use the `VALUES` token. The `VALUES` token is followed by the function call. In standard SQLJ, the function call must be enclosed in a set of outer parentheses. In the Oracle SQLJ implementation, the outer parentheses are optional. When using the outer parentheses, it does not matter if there is white space between the `VALUES` token and the begin-parenthesis. The syntax for the `VALUES` token is as follows:

```
#sql result = { VALUES(FUNC(PARAM_LIST)) };
```

In this syntax, *`result`* is the result expression, which takes the function return value. *`FUNC`* is the name of the stored function, which can optionally take a list of input, output, and input-output parameters. *`FUNC`* can include a schema or package name, such as `SCOTT.MYFUNC()`.

Note: A `VALUES` token can also be used in `INSERT INTO` *`table`* `VALUES` syntax supported by the Oracle SQL implementation, but these situations are unrelated semantically and syntactically.

Referring back to the example in "Calling Stored Procedures" on page 4-43, consider defining the stored procedure as a stored function instead, as follows:

```
CREATE OR REPLACE FUNCTION GET_MAX_DEADLINE RETURN DATE IS
   deadline DATE;
   BEGIN
      SELECT MAX(start_date + duration) INTO deadline FROM projects;
      RETURN deadline;
   END;
```

In SQLJ, you can call this `GET_MAX_DEADLINE` function as follows:

```
java.sql.Date maxDeadline;
...
#sql maxDeadline = { VALUES(GET_MAX_DEADLINE) };
```

The result expression must have a type compatible with the return type of the function.

In the Oracle SQLJ implementation, the following syntax is also allowed:

```
#sql maxDeadline = { VALUES GET_MAX_DEADLINE };
```

Note that the outer parentheses is omitted.

For stored function calls, as with stored procedures, you must use the host expression tokens `IN`, `OUT`, and `INOUT` appropriately, to match the input, output, and input-output parameters of the stored function. Additionally, the types of the host

variables you use in the parameter list must be compatible with the parameter types of the stored function.

> **Note:** If you want your stored function to be portable to non-Oracle environments, then you should use only input parameters in the calling sequence, not output or input-output parameters.

Using Iterators and Result Sets as Stored Function Returns

SQLJ supports assigning the return value of a stored function to an iterator or result set variable, if the function returns a REF CURSOR type.

The following example uses an iterator to take a stored function return. Using a result set is similar.

Example: Iterator as Stored Function Return

This example uses an iterator as a return type for a stored function, using a `REF CURSOR` type in the process.

Presume the following function definition:

```
CREATE OR REPLACE PACKAGE sqlj_refcursor AS
   TYPE EMP_CURTYPE IS REF CURSOR;
   FUNCTION job_listing (j varchar2) RETURN EMP_CURTYPE;
END sqlj_refcursor;

CREATE OR REPLACE PACKAGE BODY sqlj_refcursor AS
   FUNCTION job_listing (j varchar) RETURN EMP_CURTYPE IS
   DECLARE
      rc EMP_CURTYPE;
   BEGIN
      OPEN rc FOR SELECT ename, empno FROM emp WHERE job = j;
      RETURN rc;
   END;
END sqlj_refcursor;
```

Declare the iterator as follows:

```
#sql public <static> iterator EmpIter (String ename, int empno);
```

The `public` modifier is required, and the `static` modifier may be advisable if your declaration is at class level or nested-class level.

The code that uses the iterator and the function is as follows:

```
EmpIter iter;
...
#sql iter = { VALUES(sqlj_refcursor.job_listing('SALES')) };

while (iter.next())
{
   String empname = iter.ename();
   int empnum = iter.empno();

   ... process empname and empnum ...
}
iter.close();
...
```

This example calls the `job_listing()` function to return an iterator that contains the name and employee number of each employee whose job title is SALES. It then retrieves this data from the iterator.

5

Type Support

This chapter documents data types supported by the Oracle SQLJ implementation, listing supported SQL types and the Java types that correspond to them. This is followed by details about support for streams and Oracle type extensions. SQLJ support of Java types refers to types that can be used in host expressions.

This chapter covers the following topics:

- Supported Types for Host Expressions
- Support for Streams
- Support for JDBC 2.0 LOB Types and Oracle Type Extensions

> **See Also:** Chapter 6, "Objects, Collections, and OPAQUE Types"

Supported Types for Host Expressions

This section summarizes the types supported by the Oracle SQLJ implementation, including information about new support for Java Database Connectivity (JDBC) 2.0 types.

> **See Also:** *Oracle Database JDBC Developer's Guide and Reference* for a complete list of legal Java mappings for each Oracle SQL type

---

Note: SQLJ performs implicit conversions between SQL and Java types. Although this is generally useful and helpful, it can produce unexpected results. Do not rely on translation-time type-checking alone to ensure the correctness of your code.

This section covers the following topics:

- Summary of Supported Types
- Supported Types and Requirements for JDBC 2.0
- Using PL/SQL BOOLEAN, RECORD Types, and TABLE Types

Summary of Supported Types

Table 5–1 lists the Java types that you can use in host expressions when employing the Oracle JDBC drivers. This table also documents the correlation between Java types, SQL types whose type codes are defined in the `oracle.jdbc.OracleTypes` class, and data types in Oracle Database 11*g*.

Note: The `OracleTypes` class simply defines a type code, which is an integer constant, for each Oracle data type. For standard JDBC types, the `OracleTypes` value is identical to the standard `java.sql.Types` value.

SQL data output to a Java variable is converted to the corresponding Java type. A Java variable input to SQL is converted to the corresponding Oracle data type.

Table 5–1 Type Mappings for Supported Host Expression Types

Java Type	OracleTypes Definition	Oracle SQL Data Type
STANDARD JDBC 1.*x* TYPES		
`boolean`	`BIT`	`NUMBER`
`byte`	`TINYINT`	`NUMBER`
`short`	`SMALLINT`	`NUMBER`
`int`	`INTEGER`	`NUMBER`
`long`	`BIGINT`	`NUMBER`
`float`	`REAL`	`NUMBER`
`double`	`FLOAT`, `DOUBLE`	`NUMBER`

`java.lang.String`	`CHAR`
	`VARCHAR`
	`LONGVARCHAR`
		`CHAR`
		`VARCHAR2`
		`LONG`

`byte[]`	`BINARY`
	`VARBINARY`
	`LONGVARBINARY`
		`RAW`
		`RAW`
		`LONGRAW`

`java.sql.Date`	`DATE`	`DATE`
`java.sql.Time`	`TIME`	`DATE`
`java.sql.Timestamp`	`TIMESTAMP` `TIMESTAMP`	`DATE` `TIMESTAMP`
`java.math.BigDecimal`	`NUMERIC` `DECIMAL`	`NUMBER` `NUMBER`
STANDARD JDBC 2.0 TYPES		
`java.sql.Blob`	`BLOB`	`BLOB`
`java.sql.Clob`	`CLOB`	`CLOB`
`java.sql.Struct`	`STRUCT`	Object types
`java.sql.Ref`	`REF`	Reference types
`java.sql.Array`	`ARRAY`	Collection types
Custom object classes implementing `java.sql.SQLData`	`STRUCT`	Object types
JAVA WRAPPER CLASSES		
`java.lang.Boolean`	`BIT`	`NUMBER`
`java.lang.Byte`	`TINYINT`	`NUMBER`
`java.lang.Short`	`SMALLINT`	`NUMBER`
`java.lang.Integer`	`INTEGER`	`NUMBER`

Table 5–1 (Cont.) Type Mappings for Supported Host Expression Types

Java Type	OracleTypes Definition	Oracle SQL Data Type
`java.lang.Long`	`BIGINT`	`NUMBER`
`java.lang.Float`	`REAL`	`NUMBER`
`java.lang.Double`	`FLOAT`, `DOUBLE`	`NUMBER`
SQLJ STREAM CLASSES		
`sqlj.runtime.BinaryStream`	`LONGVARBINARY`	`LONG RAW`
`sqlj.runtime.CharacterStream`	`LONGVARCHAR`	`LONG`
`sqlj.runtime.AsciiStream` (Deprecated; use `CharacterStream`.)	`LONGVARCHAR`	`LONG`
`sqlj.runtime.UnicodeStream` (Deprecated; use `CharacterStream`.)	`LONGVARCHAR`	`LONG`
ORACLE EXTENSIONS		
`oracle.sql.NUMBER`	`NUMBER`	`NUMBER`
`oracle.sql.CHAR`	`CHAR`	`CHAR`
`oracle.sql.RAW`	`RAW`	`RAW`
`oracle.sql.DATE`	`DATE`	`DATE`
`oracle.sql.TIMESTAMP`	`TIMESTAMP`	`TIMESTAMP`
`oracle.sql.TIMESTAMPTZ`	`TIMESTAMPTZ`	`TIMESTAMP-WITH-TIMEZONE`
`oracle.sql.TIMESTAMPLTZ`	`TIMESTAMPLTZ`	`TIMESTAMP-WITH-LOCAL-TIMEZONE`
`oracle.sql.ROWID`	`ROWID`	`ROWID`
`oracle.sql.BLOB`	`BLOB`	`BLOB`
`oracle.sql.CLOB`	`CLOB`	`CLOB`
`oracle.sql.BFILE`	`BFILE`	`BFILE`
`oracle.sql.STRUCT`	`STRUCT`	Object types
`oracle.sql.REF`	`REF`	Reference types
`oracle.sql.ARRAY`	`ARRAY`	Collection types
`oracle.sql.OPAQUE`	`OPAQUE`	`OPAQUE` types
Custom object classes implementing `oracle.sql.ORAData`	`STRUCT`	Object types
Custom reference classes implementing `oracle.sql.ORAData`	`REF`	Reference types
Custom collection classes implementing `oracle.sql.ORAData`	`ARRAY`	Collection types
Custom classes implementing `oracle.sql.ORAData` for `OPAQUE` types (for example, `oracle.xdb.XMLType`)	`OPAQUE`	`OPAQUE` types
Other custom Java classes implementing `oracle.sql.ORAData` (to wrap any `oracle.sql` type)	Any	Any

Table 5–1 (Cont.) Type Mappings for Supported Host Expression Types

Java Type	OracleTypes Definition	Oracle SQL Data Type
SQLJ object Java types (can implement either `SQLData` or `ORAData`)	`JAVA_STRUCT`	SQLJ object SQL types (`JAVA_STRUCT` behind the scenes; automatic conversion to an appropriate Java class)
JAVA TYPES FOR PL/SQL TYPES		
Scalar indexed-by table represented by a Java numeric array or an array of `String`, `oracle.sql.CHAR`, or `oracle.sql.NUMBER`	NA	NA **Note**: There is a `PLSQL_INDEX_TABLE` type, but it does not appear to be used externally.
GLOBALIZATION SUPPORT		
`oracle.sql.NCHAR`	`CHAR`	`CHAR`

`oracle.sql.NString`	`CHAR`
	`VARCHAR`
	`LONGVARCHAR`
		`CHAR`
		`VARCHAR2`
		`LONG`

`oracle.sql.NCLOB`	`CLOB`	`CLOB`
`oracle.sqlj.runtime.NcharCharacterStream`	`LONGVARCHAR`	`LONG`
`oracle.sqlj.runtime.NcharAsciiStream` (Deprecated; use `NcharCharacterStream`.)	`LONGVARCHAR`	`LONG`
`oracle.sqlj.runtime.NcharUnicodeStream` (Deprecated; use `NcharCharacterStream`.)	`LONGVARCHAR`	`LONG`
QUERY RESULT OBJECTS		
`java.sql.ResultSet`	`CURSOR`	`CURSOR`
SQLJ iterator objects	`CURSOR`	`CURSOR`

> **See Also:** *Oracle Database JDBC Developer's Guide and Reference* for more information about Oracle type support.

The following points relate to type support for standard features:

- JDBC and SQLJ do not support Java `char` and `Character` types. Instead, use the Java `String` type to represent character data.
- Do not confuse the supported `java.sql.Date` type with `java.util.Date`, which is not directly supported. The `java.sql.Date` class is a wrapper for `java.util.Date` that enables JDBC to identify the data as a SQL `DATE` and adds formatting and parsing operations to support JDBC escape syntax for date values.
- Remember that all numeric types in Oracle Database 11*g* are stored as `NUMBER`. Although you can specify additional precision when you declare a `NUMBER` during table creation, this precision may be lost when retrieving the data through the Oracle JDBC drivers, depending on the Java type that you use to receive the data. An `oracle.sql.NUMBER` instance would preserve full information.

- The Java wrapper classes, such as `Integer` and `Float`, are useful in cases where `NULL` may be returned by the SQL statement. Primitive types, such as `int` and `float`, cannot contain null values.

  See Also: "NULL-Handling" on page 3-14

- The SQLJ stream classes are required in using streams as host variables.

  See Also: "Support for Streams" on page 5-8

- Weak types cannot be used for `OUT` or `INOUT` parameters. This applies to the `Struct`, `Ref`, and `Array` standard JDBC 2.0 types, as well as to corresponding Oracle extended types.
- A new set of interfaces, in the `oracle.jdbc` package, was first added in the Oracle9*i* JDBC implementation in place of classes of the `oracle.jdbc.driver` package. These interfaces provide a more generic way for users to access Oracle-specific features using Oracle JDBC drivers. Specifically, when creating programs for the middle tier, you should use the `oracle.jdbc` application programming interface (API). However, SQLJ programmers will not typically use these interfaces directly. They are used transparently by the SQLJ run time or in Oracle-specific generated code.

  See Also: "Custom Java Class Interface Specifications" on page 6-5

- For information about SQLJ support for result set and iterator host variables, refer to "Using Iterators and Result Sets as Host Variables" on page 4-45.

The following points relate to Oracle extensions:

- The Oracle SQLJ implementation requires any class that implements `oracle.sql.ORAData` to set the static `_SQL_TYPECODE` parameter according to values defined in the `OracleTypes` class. In some cases, an additional parameter must be set as well, such as `_SQL_NAME` for objects and `_SQL_BASETYPE` for object references. This occurs automatically if you use the Oracle JPublisher utility to generate the class.

  See Also: "Oracle Requirements for Classes Implementing ORAData" on page 6-10

- The `oracle.sql` classes are wrappers for SQL data for each of the Oracle data types. The `ARRAY`, `STRUCT`, `REF`, `BLOB`, and `CLOB` classes correspond to standard JDBC 2.0 interfaces.

  See Also: *Oracle Database JDBC Developer's Guide and Reference* for information about these classes and Oracle extensions

- Custom Java classes can map to Oracle objects, which implement `ORAData` or `SQLData`, references, which implement `ORAData` only, collections, which implement `ORAData` only, `OPAQUE` types, which implement `ORAData` only, or other SQL types for customized handling, which implement `ORAData` only. You can use the Oracle JPublisher utility to automatically generate custom Java classes.

  See Also: "Custom Java Classes" on page 6-4 and "JPublisher and the Creation of Custom Java Classes" on page 6-20

- The Oracle SQLJ implementation has functionality for automatic blank padding when comparing a string to a `CHAR` column value for a `WHERE` clause. Otherwise the string would have to be padded to match the number of characters in the database column. This is available as a SQLJ translator option for Oracle-specific code generation, or as an Oracle customizer option for ISO standard code generation.

  See Also: "CHAR Comparisons with Blank Padding (-fixedchar)" on page 8-46 and "Oracle Customizer CHAR Comparisons with Blank Padding (fixedchar)" on page A-24

- Weak types cannot be used for `OUT` or `INOUT` parameters. This applies to the `STRUCT`, `REF`, and `ARRAY` Oracle extended types and corresponding standard JDBC 2.0 types, as well as to Oracle `OPAQUE` types.
- Using any of the Oracle extensions requires the following:
  - Oracle JDBC driver
  - Oracle-specific code generation or Oracle customization during translation
  - Oracle SQLJ run time when your application runs

Supported Types and Requirements for JDBC 2.0

As indicated in Table 5–1, the Oracle JDBC and SQLJ implementations support JDBC 2.0 types in the standard `java.sql` package. This section lists JDBC 2.0 supported types and related Oracle extensions.

Table 5–2 lists the JDBC 2.0 types supported by the Oracle SQLJ implementation. You can use them wherever you can use the corresponding Oracle extensions, summarized in the table.

The Oracle extensions have been available in prior releases and are still available as well. These `oracle.sql.*` classes provide functionality to wrap raw SQL data.

See Also: *Oracle Database JDBC Developer's Guide and Reference*

Table 5–2 Correlation between Oracle Extensions and JDBC 2.0 Types

JDBC 2.0 Type	Oracle Extension
`java.sql.Blob`	`oracle.sql.BLOB`
`java.sql.Clob`	`oracle.sql.CLOB`
`java.sql.Struct`	`oracle.sql.STRUCT`
`java.sql.Ref`	`oracle.sql.REF`
`java.sql.Array`	`oracle.sql.ARRAY`
`java.sql.SQLData`	NA
NA	`oracle.sql.ORAData` `(_SQL_TYPECODE = OracleTypes.STRUCT)`

`ORAData` functionality is an Oracle-specific alternative to standard `SQLData` functionality for Java support of user-defined types.

> **See Also:** "Custom Java Classes" on page 6-4, "Support for BLOB, CLOB, and BFILE" on page 5-21, and "Support for Weakly Typed Objects, References, and Collections" on page 6-57

The following JDBC 2.0 types are currently *not* supported in the Oracle JDBC and SQLJ implementations:

- `JAVA_OBJECT`: Represents an instance of a Java type in a SQL column.
- `DISTINCT`: A distinct SQL type represented in or retrievable from a basic SQL type. For example, `SHOESIZE --> NUMBER`.

Note: Beginning with Oracle Database 11*g*, the Oracle SQLJ implementation supports the ISO SQLJ feature of allowing array types for iterator columns. You can declare an iterator that uses `java.sql.Array` or `oracle.sql.ARRAY` columns. For example, suppose the following database table is defined:

```
CREATE OR REPLACE TYPE arr_type IS VARRAY(20) OF NUMBER;
CREATE TABLE arr_type (arr_col1 arr_type, arr_col2
                       arr_type);
```

You could define a corresponding iterator type as follows:

```
#sql static iterator MyIter (oracle.sql.ARRAY arr_col1,
                             java.sql.Array arr_col2);
```

Using PL/SQL BOOLEAN, RECORD Types, and TABLE Types

The Oracle SQLJ and JDBC implementations do not support calling arguments or return values of the PL/SQL `BOOLEAN` type or RECORD types.

Support for TABLE Types

The Oracle JDBC drivers support scalar PL/SQL indexed-by tables.

> **See Also:** *Oracle Database JDBC Developer's Guide and Reference*

The Oracle SQLJ implementation simplifies the process of writing and retrieving data in scalar indexed-by tables. The following array types are supported:

- Numeric types: `int[]`, `long[]`, `float[]`, `double[]`, `short[]`, `java.math.BigDecimal[]`, `oracle.sql.NUMBER[]`
- Character types: `java.lang.String[]`, `oracle.sql.CHAR[]`

The following is an example of writing indexed-by table data to the database:

```
int[] vals = {1,2,3};
#sql { call procin(:vals) };
```

The following is an example of retrieving indexed-by table data from the database:

```
oracle.sql.CHAR[] outvals;
#sql { call procout(:OUT outvals/*[111](22)*/) };
```

You must specify the maximum length of the output array being retrieved, using the `[xxx]` syntax inside the `/*...*/` syntax, as shown. Also, for character-like binds, you can optionally include the `(xx)` syntax, as shown, to specify the maximum length of an array element in bytes.

Note: The `oracle.sql.Datum` class is not supported directly. You must use an appropriate subclass, such as `oracle.sql.CHAR` or `oracle.sql.NUMBER`.

Workarounds for Unsupported Types

As a workaround for an unsupported type, you can create wrapper procedures that process the data using supported types. For example, to wrap a stored procedure that uses PL/SQL boolean values, you can create a stored procedure that takes a character or number from JDBC and passes it to the original procedure as `BOOLEAN`, or for an output parameter, accepts a `BOOLEAN` argument from the original procedure and passes it as a `CHAR` or `NUMBER` to JDBC. Similarly, to wrap a stored procedure that uses PL/SQL records, you can create a stored procedure that handles a record in its individual components, such as `CHAR` and `NUMBER`. To wrap a stored procedure that uses PL/SQL TABLE types, you can break the data into components or perhaps use Oracle collection types.

The following is an example of a PL/SQL wrapper procedure `MY_PROC` for a stored procedure `PROC` that takes a `BOOLEAN` as input:

```
PROCEDURE MY_PROC (n NUMBER) IS
BEGIN
   IF n=0
      THEN proc(false);
      ELSE proc(true);
   END IF;
END;

PROCEDURE PROC (b BOOLEAN) IS
BEGIN
...
END;
```

Note: When using these unsupported PL/SQL types in method signatures in PL/SQL packages or SQL objects, consider using the Oracle Database 11*g* JPublisher utility. This facilitates the creation of Java types to call such methods. Refer to "JPublisher and the Creation of Custom Java Classes" on page 6-20 for an overview of JPublisher, and the *Oracle Database JPublisher User's Guide* for more information.

Support for Streams

Standard SQLJ provides the following specialized classes for convenient processing of long data in streams:

- `sqlj.runtime.BinaryStream`
- `sqlj.runtime.CharacterStream`

These stream types can be used for iterator columns to retrieve data from the database or for input host variables to send data to the database. As with Java streams in general, these classes allow the convenience of processing and transferring large data items in manageable chunks.

This section discusses general use of these classes, Oracle SQLJ extended functionality, and stream class methods. It covers the following topics:

> **Note:** Starting from JDBC 2.0, the `CharacterStream` class replaces the `AsciiStream` and `UnicodeStream` classes. `CharacterStream` shelters users from unnecessary logistics regarding encoding.

General Use of SQLJ Streams

Table 5–1 lists the data types you would typically process using these stream classes. To summarize:

- `BinaryStream` is typically used for `LONG RAW` (`Types.LONGVARBINARY`), but might also be used for `RAW` (`Types.BINARY` or `Types.VARBINARY`).
- `CharacterStream` is typically used for `LONG` (`java.sql.Types.LONGVARCHAR`), but might also be used for `VARCHAR2` (`Types.VARCHAR`).

Of course, any use of streams is at your discretion. You can use the SQLJ stream types for host variables to either send or retrieve data.

As Table 5–1 documents, `LONG` and `VARCHAR2` data can also be manifested in Java `String`, while `RAW` and `LONGRAW` data can also be manifested in Java `byte[]` arrays. Also, if your database supports large object types, such as `BLOB` and `CLOB`, then you may find these to be preferable to types like `LONG` and `LONG RAW`, although streams might still be used in extracting data from large objects. The Oracle SQLJ and JDBC implementations support large object types.

See Also: "Support for BLOB, CLOB, and BFILE" on page 5-21

Both SQLJ stream classes are subclasses of standard Java classes, `java.io.InputStream` for `BinaryStream` and `java.io.Reader` for `CharacterStream`, and act as wrappers to provide the functionality required by SQLJ. This functionality is to communicate to SQLJ the type and length of the underlying data so that it can be processed and formatted properly.

Key Aspects of Stream Support Classes

The following abbreviated code illustrates key aspects of the `BinaryStream` class, such as what it extends, constructor signatures, and key method signatures:

```
public class sqlj.runtime.BinaryStream extends sqlj.runtime.StreamWrapper
{   public sqlj.runtime.BinaryStream(java.io.InputStream);
```

```
    public sqlj.runtime.BinaryStream(java.io.InputStream,int);
    public java.io.InputStream getInputStream();
    public int getLength();
    public void setLength(int);
}
```

The following abbreviated code illustrates key aspects of the `CharacterStream` class:

```
public class sqlj.runtime.CharacterStream extends java.io.FilterReader
{   public sqlj.runtime.CharacterStream(java.io.Reader);
    public sqlj.runtime.CharacterStream(java.io.Reader,int);
    public int getLength();
    public java.io.Reader getReader();
    public void setLength(int);
}
```

Note:

- The `int` parameters in the constructors are for data length, in bytes or characters as applicable.
- For any method that takes a `java.io.InputStream` object as input, you can use a `BinaryStream` object instead. Similarly, for any method that takes a `java.io.Reader` object as input, you can use a `CharacterStream` object instead.
- The deprecated `AsciiStream` and `UnicodeStream` classes have the same key aspects and signatures as `BinaryStream`.

Using SQLJ Streams to Send Data

Standard SQLJ enables you to use streams as host variables to update the database. A key point in sending a SQLJ stream to the database is that you must somehow determine the length of the data and specify that length to the constructor of the SQLJ stream.

You can use a SQLJ stream to send data to the database as follows:

1. Determine the length of the data.
2. Create an appropriate standard Java data object for input. For `BinaryStream`, this would be an input stream, an instance of `java.io.InputStream` or some subclass. For `CharacterStream`, this would be a reader object, an instance of `java.io.Reader` or some subclass.
3. Create an instance of the appropriate SQLJ stream class depending on the type of data, passing the data object and length to the constructor.
4. Use the SQLJ stream instance as a host variable in a suitable SQL operation in a SQLJ executable statement.
5. Close the stream.

Note: Although not required, it is recommended that you close the stream after using it.

Updating LONG or LONG RAW from a File

This section illustrates how to create a `CharacterStream` object or a `BinaryStream` object from a `File` object and use it to update the database. The code example at the end uses a `CharacterStream` for a `LONG` column.

In updating a database column from a file, a step is needed to determine the length. You can do this by creating a `java.io.File` object before you create your input stream.

Following are the steps for updating the database from a file:

1. Create a `java.io.File` object from your file. You can specify the file path name to the `File` class constructor.
2. Use the `length()` method of the `File` object to determine the length of the data. This method returns a `long` value, which you must cast to an `int` for input to the SQLJ stream class constructor.

   Note: Before performing this cast, test the `long` value to ensure that it is not too big to fit into an `int` variable. The static constant `MAX_VALUE` in the class `java.lang.Integer` indicates the largest possible Java `int` value.

3. For character data, create a `java.io.FileReader` object from the `File` object. You can pass the `File` object to the `FileReader` constructor.

   For binary data, create a `java.io.FileInputStream` object from the `File` object. You can pass the `File` object to the `FileInputStream` constructor.
4. Create an appropriate SQLJ stream object. This would be a `CharacterStream` object for a character file or a `BinaryStream` object for a binary file. Pass the `FileReader` or `FileInputStream` object, as applicable, and the data length as an `int` to the SQLJ stream class constructor.
5. Use the SQLJ stream object as a host variable in an appropriate SQL operation in a SQLJ executable statement.

The following is an example of writing `LONG` data to the database from a file. Presume you have an HTML file in `/private/mydir/myfile.html` and want to insert the file contents into a `LONG` column, `chardata`, in the `filetable` database table.

```
import java.io.*;
import sqlj.runtime.*;

...
File myfile = new File ("/private/mydir/myfile.html");
int length = (int)myfile.length();     // Must cast long output to int.
FileReader filereader = new FileReader(myfile);
CharacterStream charstream = new CharacterStream(filereader, length);
#sql { INSERT INTO filetable (chardata) VALUES (:charstream) };
charstream.close();
...
```

Updating LONG RAW from a Byte Array

This section illustrates how to create a `BinaryStream` object from a byte array and uses it to update the database.

You must determine the length of the data before updating the database from a byte array. This is more trivial for arrays than for files, though, because all Java arrays have functionality to return the length.

Following are the steps in updating the database from a byte array:

1. Use the `length` functionality of the array to determine the length of the data. This returns an `int`, which is what you will need for the constructor of any of the SQLJ stream classes.
2. Create a `java.io.ByteArrayInputStream` object from your array. You can pass the byte array to the `ByteArrayInputStream` constructor.
3. Create a `BinaryStream` object. Pass the `ByteArrayInputStream` object and data length as an `int` to the `BinaryStream` class constructor.

   The constructor signature is as follows:

   ```
   BinaryStream (InputStream in, int length)
   ```

   You can use an instance of `java.io.InputStream` or of any subclass, such as the `ByteArrayInputStream` class.
4. Use the SQLJ stream object as a host variable in an appropriate SQL operation in a SQLJ executable statement.

The following is an example of writing `LONG RAW` data to the database from a byte array. Presume you have a byte array, `bytearray[]`, and you want to insert its contents into a `LONG RAW` column, `BINDATA`, in the `BINTABLE` database table.

```
import java.io.*;
import sqlj.runtime.*;

...
byte[] bytearray = new byte[100];

(Populate bytearray somehow.)
...
int length = bytearray.length;
ByteArrayInputStream arraystream = new ByteArrayInputStream(bytearray);
BinaryStream binstream = new BinaryStream(arraystream, length);
#sql { INSERT INTO bintable (bindata) VALUES (:binstream) };
binstream.close();
...
```

> **Note:** It is not necessary to use a stream as in this example. Alternatively, you can update the database directly from a byte array.

Retrieving Data into Streams: Precautions

You can also use the SQLJ stream classes to retrieve data, but the logistics of using streams make certain precautions necessary with some database products. When reading long data and writing it to a stream using Oracle Database 11*g* and an Oracle JDBC driver, you must be careful in how you access and process the stream data.

As the Oracle JDBC drivers access data from an iterator row, they must flush any stream item from the communications pipe before accessing the next data item. Even though the stream data is written to a local stream while the iterator row is processed, this stream data will be lost if you do not read it from the local stream before the JDBC

driver accesses the next data item. This is because of the manner in which streams must be processed, which is due to their potentially large size and unknown length.

Therefore, as soon as your Oracle JDBC driver has accessed a stream item and written it to a local stream variable, you must read and process the local stream before anything else is accessed from the iterator.

This is especially problematic in using positional iterators, with their requisite `FETCH INTO` syntax. With each fetch, all columns are read before any are processed. Therefore, there can be only one stream item and it must be the last item accessed.

The precautions you must take can be summarized as follows:

- When using a positional iterator, you can have only one stream column and it must be the last column. As soon as you have fetched each row of the iterator, writing the stream item to a local input stream variable in the process, you must read and process the local stream variable before advancing to the next row of the iterator.
- When using a named iterator, you can have multiple stream columns. However, as you process each iterator row, each time you access a stream field, writing the data to a local stream variable in the process, you must read and process the local stream immediately, before reading anything else from the iterator.

  Furthermore, in processing each row of a named iterator, you must call the column accessor methods in the same order in which the database columns were selected in the query that populated the iterator. As mentioned in the preceding discussion, this is because stream data remains in the communications pipe after the query. If you try to access columns out of order, then the stream data may be skipped over and lost in the course of accessing other columns.

Note:

- Oracle Database 11*g* and the Oracle JDBC drivers do not support use of streams in `SELECT INTO` statements.
- Input streams, by default, do not support `mark` and `reset` methods. If you pass any arbitrary input stream to the constructor, then the `reset` method of `InputStream` class will throw an `IOException`. So, always ensure that the input stream is in the proper state when passed to the `NcharAsciiStream` constructor. For example, reset the stream before passing it to `NcharAsciiStream` if the stream has no more data or if the stream is closed.

Using SQLJ Streams to Retrieve Data

To retrieve data as a stream, standard SQLJ enables you to select data into a named or positional iterator that has a column of the appropriate SQLJ stream type.

This section covers the basic steps in retrieving data into a SQLJ stream using a positional iterator or a named iterator, taking into account the precautions documented in the preceding section.

> **See Also:** "Stream Class Methods" on page 5-15 and "Examples of Retrieving and Processing Stream Data" on page 5-17

Using a SQLJ Stream Column in a Positional Iterator

Use the following steps to retrieve data into a SQLJ stream using a positional iterator:

1. Declare a positional iterator class with the last column being of the appropriate SQLJ stream type.
2. Declare a local variable of your iterator type.
3. Declare a local variable of the appropriate SQLJ stream type. This will be used as a host variable to receive data from each row of the SQLJ stream column of the iterator.
4. Execute a query to populate the iterator you declared in Step 2.
5. Process the iterator as usual. Because the host variables in the `INTO`-list of the `FETCH INTO` statement must be in the same order as the columns of the positional iterator, the local input stream variable is the last host variable in the list.

   See Also: "Using Positional Iterators" on page 4-34

6. In the iterator processing loop, after each iterator row is accessed, immediately read and process the local input stream, storing or printing the stream data as desired.
7. Close the local input stream each time through the iterator processing loop.
8. Close the iterator.

Using SQLJ Stream Columns in a Named Iterator

Use the following steps to retrieve data into one or more SQLJ streams using a named iterator:

1. Declare a named iterator class with one or more columns of appropriate SQLJ stream type.
2. Declare a local variable of your iterator type.
3. Declare a local variable of some input stream or reader type for each SQLJ stream column in the iterator. These will be used to receive data from the stream-column accessor methods. These local stream variables need not be of the SQLJ stream types. They can be standard `java.io.InputStream` or `java.io.Reader`, as applicable.

   Note: The local stream variables need not be of the SQLJ stream types, because the data was already correctly formatted as a result of the iterator columns being of appropriate SQLJ stream types.

4. Execute a query to populate the iterator you declared in Step 2.
5. Process the iterator as usual. In processing each row of the iterator, as each stream-column accessor method returns the stream data, write it to the corresponding local input stream variable you declared in Step 3.

   To ensure that stream data will not be lost, call the column accessor methods in the same order in which columns were selected in the query in Step 4.

   See Also: "Using Positional Iterators" on page 4-34

6. In the iterator processing loop, immediately after calling the accessor method for any stream column and writing the data to a local input stream variable, read and process the local input stream, storing or printing the stream data as desired.
7. Close the local input stream each time through the iterator processing loop.

8. Close the iterator.

Note:

- When you populate a SQLJ stream object with data, the length attribute of the stream will not be meaningful. This attribute is meaningful only when you set it explicitly, either using the `setLength()` method that each SQLJ stream class provides or specifying the length to the constructor.
- Although not required, it is recommended that you close the local input stream each time through the iterator processing loop.

Stream Class Methods

In processing a SQLJ stream column in a named or positional iterator, the local stream variable used to receive the stream data can be either a SQLJ stream type or the standard `java.io.InputStream` or `java.io.Reader` type, as applicable. In either case, standard methods of the input data object are supported.

If the local stream variable is a SQLJ stream type, `BinaryStream` or `CharacterStream`, you have the option of either reading data directly from the SQLJ stream object or retrieving the underlying `InputStream` or `Reader` object and reading data from that.

Note: This is just a matter of preference. The former approach is simpler. However, the latter approach involves more direct and efficient data access.

Binary Stream Methods

The `BinaryStream` class is a subclass of the `sqlj.runtime.StreamWrapper` class. The `StreamWrapper` class provides the following key methods:

- `InputStream getInputStream()`: You can optionally use this method to get the underlying `java.io.InputStream` object. However, this is not required, because you can also process SQLJ stream objects directly.
- `void setLength(int length)`: You can use this to set the `length` attribute of a SQLJ stream object. This is not necessary if you have already set `length` in constructing the stream object, unless you want to change it for some reason.

  The `length` attribute must be set to an appropriate value before you send a SQLJ stream to the database.
- `int getLength()`: This method returns the value of the `length` attribute of a SQLJ stream. This value is meaningful only if you explicitly set it using the stream object constructor or the `setLength()` method. When you retrieve data into a stream, the `length` attribute is not set automatically.

The `sqlj.runtime.StreamWrapper` class is a subclass of the `java.io.FilterInputStream` class, which is a subclass of the `java.io.InputStream` class. The following important methods of the `InputStream` class are supported by the SQLJ `BinaryStream` class as well:

- `int read ()`: Reads the next byte of data from the input stream. The byte of data is returned as an `int` value in the range 0 to 255. If the end of the stream has

already been reached, then the value -1 is returned. This method blocks program execution until one of the following:

  - Input data is available
  - The end of the stream is detected
  - An exception is thrown

- int read (byte b[]): Reads up to b.length bytes of data from the input stream, writing the data into the specified b[] byte array. It returns an int value indicating how many bytes were read, or -1 if the end of the stream has already been reached. This method blocks program execution until input is available.
- int read (byte b[], int off, int len): Reads up to len bytes of data from the input stream, starting at the byte specified by the offset, off, and writing the data into the specified b[] byte array. It returns an int value indicating how many bytes were read, or -1 if the end of the stream has already been reached. This method blocks until input is available.
- long skip (long n): Skips over and discards n bytes of data from the input stream. However, in some circumstances, this method will actually skip a smaller number of bytes. It returns a long value indicating the actual number of bytes skipped.
- void close(): Closes the stream and releases any associated resources.

Character Stream Methods

The CharacterStream class provides the following key methods:

- Reader getReader(): You can optionally use this method to get the underlying java.io.Reader object. However, this is not required, because you can also process SQLJ stream objects directly.
- void setLength(int length): You can use this method to set the length of the stream object.
- int getLength(): You can use this method to get the length of the stream object.

The sqlj.runtime.CharacterStream class is a subclass of the java.io.FilterReader class, which is a subclass of the java.io.Reader class. The following important methods of the Reader class are supported by the SQLJ CharacterStream class as well:

- int read (): Reads the next character of data from the reader. The data is returned as an int value in the range 0 to 65535. If the end of the data has already been reached, then the value -1 is returned. This method blocks program execution until one of the following:

  - Input data is available
  - The end of the data is detected
  - An exception is thrown

- int read (char cbuf[]): Reads characters into an array, writing the data into the specified cbuf[] char array. It returns an int value indicating how many characters were read, or -1 if the end of the data has already been reached. This method blocks program execution until input is available.
- int read (char cbuf[], int off, int len): Reads up to len characters of data from the input, starting at the character specified by the offset, off, and

writing the data into the specified `char[]` char array. It returns an `int` value indicating how many characters were read, or `-1` if the end of the data has already been reached. This method blocks until input is available.

- `long skip (long n)`: Skips over and discards `n` characters of data from the input. However, in some circumstances, this method will actually skip a smaller number of characters. It returns a `long` value indicating the actual number of characters skipped.
- `void close()`: Closes the stream and releases any associated resources.

Examples of Retrieving and Processing Stream Data

This section provides examples of various scenarios of retrieving stream data, as follows:

- Using a `SELECT` statement to select data from a `LONG` column and populate a SQLJ `CharacterStream` column in a named iterator, as shown in Example 5–1
- Using a `SELECT` statement to select data from a `LONG RAW` column and populate a SQLJ `BinaryStream` column in a positional iterator, as shown in Example 5–2

Example 5–1 Selecting LONG Data into CharacterStream Column of Named Iterator

This example selects data from a `LONG` database column, populating a SQLJ `CharacterStream` column in a named iterator.

Assume there is a table named `FILETABLE` with a `VARCHAR2` column called `FILENAME` that contains file names and a `LONG` column called `FILECONTENTS` that contains file contents in character format. The code is as follows:

```
import sqlj.runtime.*;
import java.io.*;
...
#sql iterator MyNamedIter (String filename, CharacterStream filecontents);

...
MyNamedIter namediter = null;
String fname;
CharacterStream charstream;
#sql namediter = { SELECT filename, filecontents FROM filetable };
while (namediter.next()) {
   fname = namediter.filename();
   charstream = namediter.filecontents();
   System.out.println("Contents for file " + fname + ":");
   printStream(charstream);
   charstream.close();
}

namediter.close();
...
public void printStream(Reader in) throws IOException
{
   int character;
   while ((character = in.read()) != -1) {
      System.out.print((char)character);
   }
}
```

Remember that you can pass a SQLJ character stream to any method that takes a standard `java.io.Reader` as an input parameter.

Example 5–2 : Selecting LONG RAW Data into BinaryStream Column of Positional Iterator

This example selects data from a `LONG RAW` column, populating a SQLJ `BinaryStream` column in a positional iterator.

As explained in the preceding section, there can be only one stream column in a positional iterator and it must be the last column. Assume there is a table named `BINTABLE` with a `NUMBER` column called `IDENTIFIER` and a `LONG RAW` column called `BINDATA` that contains binary data associated with the identifier. The code is as follows:

```
import sqlj.runtime.*;
...
#sql iterator MyPosIter (int, BinaryStream);

...
MyPosIter positer = null;
int id=0;
BinaryStream binstream=null;
#sql positer = { SELECT identifier, bindata FROM bintable };
while (true) {
   #sql { FETCH :positer INTO :id, :binstream };
   if (positer.endFetch()) break;

   (...process data as desired...)

   binstream.close();
}
positer.close();
...
```

SQLJ Stream Objects as Output Parameters and Function Return Values

As described in the preceding sections, standard SQLJ supports the use of the `BinaryStream` and `CharacterStream` classes in the `sqlj.runtime` package for retrieval of stream data into iterator columns.

In addition, the Oracle SQLJ implementation enables the following uses of the SQLJ stream types if you use Oracle9*i* Database or later version, an Oracle JDBC driver, Oracle-specific code generation or the Oracle customizer, and the Oracle SQLJ run time:

- They can appear as `OUT` or `INOUT` host variables from a stored procedure or function call.
- They can appear as the return value from a stored function call.

Streams as Stored Procedure Output Parameters

You can use the `BinaryStream` and `CharacterStream` types as the assignment type for a stored procedure or stored function `OUT` or `INOUT` parameter.

Assume the following table definition:

```
CREATE TABLE streamexample (name VARCHAR2 (256), data LONG);
INSERT INTO streamexample (data, name)
   VALUES
   ('000000000011111111112222222222333333333344444444445555555555',
   'StreamExample');
```

Also, presume the following stored procedure definition, which uses the STREAMEXAMPLE table:

```
CREATE OR REPLACE PROCEDURE out_longdata
                            (dataname VARCHAR2, longdata OUT LONG) IS
BEGIN
   SELECT data INTO longdata FROM streamexample WHERE name = dataname;
END out_longdata;
```

The following sample code uses a call to the out_longdata stored procedure to read the long data:

```
import sqlj.runtime.*;

...
CharacterStream data;
#sql { CALL out_longdata('StreamExample', :OUT data) };
int c;
while ((c = data.read ()) != -1)
   System.out.print((char)c);
System.out.flush();
data.close();
...
```

Note: Closing the stream is recommended, but not required.

Streams as Stored Function Results

You can use the BinaryStream and CharacterStream types as the assignment type for a stored function return result.

Assume the same STREAMEXAMPLE table definition as in the preceding stored procedure example. Also, assume the following stored function definition, which uses the STREAMEXAMPLE table:

```
CREATE OR REPLACE FUNCTION get_longdata (dataname VARCHAR2) RETURN long
   IS longdata LONG;
BEGIN
   SELECT data INTO longdata FROM streamexample WHERE name = dataname;
   RETURN longdata;
END get_longdata;
```

The following sample code uses a call to the get_longdata stored function to read the long data:

```
import sqlj.runtime.*;

...
CharacterStream data;
#sql data = { VALUES(get_longdata('StreamExample')) };
int c;
while ((c = data.read ()) != -1)
   System.out.print((char)c);
System.out.flush();
data.close();
...
```

Note: Closing the stream is recommended, but not required.

Support for JDBC 2.0 LOB Types and Oracle Type Extensions

The Oracle SQLJ implementation offers extended functionality for the following JDBC 2.0 and Oracle-specific data types:

- JDBC 2.0 large object (LOB) types (`BLOB` and `CLOB`)
- Oracle `BFILE` type
- Oracle `ROWID` type
- Oracle REF CURSOR types
- Other Oracle Database 11*g* data types, such as `NUMBER` and `RAW`

These data types are supported by classes in the `oracle.sql` package. LOBs and binary files (BFILEs) are handled similarly in many ways, so are discussed together. Additionally, the Oracle SQLJ implementation offers extended support for the standard BigDecimal JDBC type.

JDBC 2.0 functionality for user-defined SQL objects, object references, and collections are also supported.

See Also: Chapter 6, "Objects, Collections, and OPAQUE Types"

Note that using Oracle extensions in your code requires the following:

- Use one of the Oracle JDBC drivers.
- Use Oracle-specific code generation or for ISO code generation, customize the profiles appropriately. The default customizer, `oracle.sqlj.runtime.util.OraCustomizer`, is recommended.
- Use the Oracle SQLJ run time when your application runs.

The Oracle SQLJ run time and an Oracle JDBC driver are required whenever you use the Oracle customizer, even if you do not actually use Oracle extensions in your code.

For Oracle-specific semantics-checking, you must use an appropriate checker. The default checker, `oracle.sqlj.checker.OracleChecker`, acts as a front end and will run the appropriate checker based on your environment. This will be one of the Oracle-specific checkers if you are using an Oracle JDBC driver.

This section covers the following topics:

Package oracle.sql

SQLJ users, as well as JDBC users, should be aware of the `oracle.sql` package, which includes classes to support all the Oracle Database 11*g* data types, such as

`oracle.sql.ROWID`, `oracle.sql.CLOB`, and `oracle.sql.NUMBER`. The `oracle.sql` classes are wrappers for the raw SQL data and provide appropriate mappings and conversion methods to Java formats. An `oracle.sql.*` object contains a binary representation of the corresponding SQL data in the form of a byte array. Each `oracle.sql.*` data type class is a subclass of the `oracle.sql.Datum` class.

For Oracle-specific semantics-checking, you must use an appropriate checker. The default checker, `oracle.sqlj.checker.OracleChecker`, acts as a front end and will run the appropriate checker based on your environment. This will be one of the Oracle-specific checkers if you are using an Oracle JDBC driver.

See Also:

- "Connection Options" on page 8-25
- "Semantics-Checking and Offline-Parsing Options" on page 8-56
- *Oracle Database JDBC Developer's Guide and Reference*

Support for BLOB, CLOB, and BFILE

The Oracle SQLJ and JDBC implementations support JDBC 2.0 LOB types and provide similar support for the Oracle-specific `BFILE` type (read-only binary files stored outside the database). These data types are supported by the following classes:

- `oracle.sql.BLOB`
- `oracle.sql.CLOB`
- `oracle.sql.BFILE`

These classes can be used in Oracle-specific SQLJ applications in the following ways:

- As `IN`, `OUT`, or `INOUT` host variables in executable SQLJ statements and in `INTO`-lists
- As return values from stored function calls
- As column types in iterator declarations

See Also: *Oracle Database JDBC Developer's Guide and Reference* for more information about LOBs and BFILEs and use of supported stream APIs.

You can manipulate LOBs by using methods defined in the `BLOB` and `CLOB` classes, which is recommended, or by using the procedures and functions defined in the `DBMS_LOB` PL/SQL package. All procedures and functions defined in this package can be called by SQLJ programs.

You can manipulate BFILEs by using methods defined in the `BFILE` class, which is recommended, or by using the file-handling routines of the `DBMS_LOB` package.

Using methods of the `BLOB`, `CLOB`, and `BFILE` classes in a Java application is more convenient than using the `DBMS_LOB` package and may also lead to faster execution in some cases.

Note that the type of the chunk being read or written depends on the kind of LOB being manipulated. For example, character large objects (CLOBs) contain character data and, therefore, Java strings are used to hold chunks of data. Binary large objects (BLOBs) contain binary data and, therefore, Java byte arrays are used to hold chunks of data.

> **Note:** The DBMS_LOB package requires a round trip to the server. Methods in the BLOB, CLOB, and BFILE classes may also result in a round trip to the server.

BFILE Class versus DBMS_LOB Functionality for BFILEs

Example 5–3 and Example 5–4 contrast use of the oracle.sql methods with use of the DBMS_LOB package for BFILEs:

Example 5–3 Use of oracle.sql.BFILE File-Handling Methods with BFILE

This example manipulates a BFILE using file-handling methods of the oracle.sql.BFILE class.

```
BFILE openFile (BFILE file) throws SQLException
{
  String dirAlias, name;
  dirAlias = file.getDirAlias();
  name = file.getName();
  System.out.println("name: " + dirAlias + "/" + name);

  if (!file.isFileOpen())
  {
    file.openFile();
  }
  return file;
}
```

The BFILE getDirAlias() and getName() methods construct the full path and file name. The openFile() method opens the file. You cannot manipulate BFILEs until they have been opened.

Example 5–4 Use of DBMS_LOB File-Handling Routines with BFILE

This example manipulates a BFILE using file-handling routines of the DBMS_LOB package.

```
BFILE openFile(BFILE file) throws SQLException
{
   String dirAlias, name;
   #sql { CALL dbms_lob.filegetname(:file, :out dirAlias, :out name) };
   System.out.println("name: " + dirAlias + "/" + name);

   boolean isOpen;
   #sql isOpen = { VALUES(dbms_lob.fileisopen(:file)) };
   if (!isOpen)
   {
      #sql { CALL dbms_lob.fileopen(:inout file) };
   }
   return file;
}
```

The openFile() method prints the name of a file object and then returns an opened version of the file. Note that BFILEs can be manipulated only after being opened with a call to DBMS_LOB.FILEOPEN or equivalent method in the BFILE class.

BLOB and CLOB Classes versus DBMS_LOB Functionality for LOBs

Example 5–5 and Example 5–6 contrast use of the `oracle.sql` methods with use of the `DBMS_LOB` package for BLOBs, and Example 5–7 and Example 5–8 contrast use of the `oracle.sql` methods with use of the `DBMS_LOB` package for CLOBs.

Example 5–5 Example: Use of oracle.sql.CLOB Read Methods with CLOB

This example reads data from a CLOB using methods of the `oracle.sql.CLOB` class.

```
void readFromClob(CLOB clob) throws SQLException
{
  long clobLen, readLen;
  String chunk;

  clobLen = clob.length();

  for (long i = 0; i < clobLen; i+= readLen) {
    chunk = clob.getSubString(i, 10);
    readLen = chunk.length();
    System.out.println("read " + readLen + " chars: " + chunk);
  }
}
```

This method contains a loop that reads from the CLOB and returns a 10-character Java string each time. The loop continues until the entire CLOB has been read.

Example 5–6 Example: Use of DBMS_LOB Read Routines with CLOB

This example uses routines of the `DBMS_LOB` package to read from a CLOB.

```
void readFromClob(CLOB clob) throws SQLException
{
   long clobLen, readLen;
   String chunk;

   #sql clobLen = { VALUES(dbms_lob.getlength(:clob)) };

   for (long i = 1; i <= clobLen; i += readLen) {
       readLen = 10;
       #sql { CALL dbms_lob.read(:clob, :inout readLen, :i, :out chunk) };
       System.out.println("read " + readLen + " chars: " + chunk);
   }
}
```

This method reads the contents of a CLOB in chunks of 10 characters at a time. Note that the chunk host variable is of the `String` type.

Example 5–7 Example: Use of oracle.sql.BLOB Write Routines with BLOB

This example writes data to a BLOB using methods of the `oracle.sql.BLOB` class. Input a BLOB and specified length.

```
void writeToBlob(BLOB blob, long blobLen) throws SQLException
{
  byte[] chunk = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
  long chunkLen = (long)chunk.length;

  for (long i = 0; i < blobLen; i+= chunkLen) {
    if (blobLen < chunkLen) chunkLen = blobLen;
    chunk[0] = (byte)(i+1);
    chunkLen = blob.putBytes(i, chunk);
```

```
  }
}
```

This method goes through a loop that writes to the BLOB in 10-byte chunks until the specified BLOB length has been reached.

Example 5–8 Example: Use of DBMS_LOB Write Routines with BLOB

This example uses routines of the `DBMS_LOB` package to write to a BLOB.

```
void writeToBlob(BLOB blob, long blobLen) throws SQLException
{
   byte[] chunk = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
   long chunkLen = (long)chunk.length;

   for (long i = 1; i <= blobLen; i += chunkLen) {
      if ((blobLen - i + 1) < chunkLen) chunkLen = blobLen - i + 1;
      chunk[0] = (byte)i;
      #sql { CALL dbms_lob.write(:INOUT blob, :chunkLen, :i, :chunk) };
   }
}
```

This method fills the contents of a BLOB in 10-byte chunks. Note that the chunk host variable is of the `byte[]` type.

LOB and BFILE Stored Function Results

Host variables of the `BLOB`, `CLOB`, and `BFILE` type can be assigned to the result of a stored function call. The following example is for a CLOB, but code for BLOBs and BFILEs would be functionally the same.

First, presume the following function definition:

```
CREATE OR REPLACE FUNCTION longer_clob (c1 CLOB, c2 CLOB) RETURN CLOB IS
   result CLOB;
BEGIN
   IF dbms_lob.getLength(c2) > dbms_lob.getLength(c1) THEN
      result := c2;
   ELSE
      result := c1;
   END IF;
   RETURN result;
END longer_clob;
```

The following example uses a CLOB as the assignment type for a return value from the `longer_clob` function:

```
void readFromLongest(CLOB c1, CLOB c2) throws SQLException
{
   CLOB longest;
   #sql longest = { VALUES(longer_clob(:c1, :c2)) };
   readFromClob(longest);
}
```

The `readFromLongest()` method prints the contents of the longer passed CLOB, using the `readFromClob()` method defined previously.

LOB and BFILE Host Variables and SELECT INTO Targets

Host variables of the `BLOB`, `CLOB`, and `BFILE` type can appear in the `INTO`-list of a `SELECT INTO` executable statement. The following example is for a BLOB and CLOB, but code for BFILEs would be functionally the same.

Assume the following table definition:

```
CREATE TABLE basic_lob_table(x VARCHAR2(30), b BLOB, c CLOB);
INSERT INTO basic_lob_table
   VALUES('one', '0101010101010101010101010101', 'onetwothreefour');
INSERT INTO basic_lob_table
   VALUES('two', '02020202020202020202020202', 'twothreefourfivesix');
```

The following example uses a BLOB and a CLOB as host variables that receive data from the table defined, using a `SELECT INTO` statement:

```
...
BLOB blob;
CLOB clob;
#sql { SELECT one.b, two.c INTO :blob, :clob
     FROM basic_lob_table one, basic_lob_table two
     WHERE one.x='one' AND two.x='two' };
#sql { INSERT INTO basic_lob_table VALUES('three', :blob, :clob) };
...
```

This example selects the BLOB from the first row and the CLOB from the second row of `BASIC_LOB_TABLE`. It then inserts a third row into the table using the BLOB and CLOB selected in the previous operation.

LOBs and BFILEs in Iterator Declarations

The `BLOB`, `CLOB`, and `BFILE` types can be used as column types for SQLJ positional and named iterators. Such iterators can be populated as a result of compatible executable SQLJ operations.

Following are sample declarations:

```
#sql iterator NamedLOBIter(CLOB c);
#sql iterator PositionedLOBIter(BLOB);
#sql iterator NamedFILEIter(BFILE bf);
```

LOB and BFILE Host Variables and Named Iterator Results

The following example uses the `BASIC_LOB_TABLE` table and the `readFromLongest()` method defined in previous examples and a CLOB in a named iterator. Similar code could be written for BLOBs and BFILEs.

```
#sql iterator NamedLOBIter(CLOB c);

...
NamedLOBIter iter;
#sql iter = { SELECT c FROM basic_lob_table };
if (iter.next())
   CLOB c1 = iter.c();
if (iter.next())
   CLOB c2 = iter.c();
iter.close();
readFromLongest(c1, c2);
...
```

This example uses an iterator to select two CLOBs from the first two rows of `BASIC_LOB_TABLE`, then prints the larger of the two using the `readFromLongest()` method.

LOB and BFILE Host Variables and Positional Iterator FETCH INTO Targets

Host variables of the BLOB, CLOB, and BFILE type can be used with positional iterators and appear in the INTO-list of the associated FETCH INTO statement if the corresponding column attribute in the iterator is of the identical type.

The following example uses the BASIC_LOB_TABLE table and the writeToBlob() method defined in previous examples. Similar code could be written for CLOBs and BFILEs.

```
#sql iterator PositionedLOBIter(BLOB);

...
PositionedLOBIter iter;
BLOB blob = null;
#sql iter = { SELECT b FROM basic_lob_table };
for (long rowNum = 1; ; rowNum++)
{
    #sql { FETCH :iter INTO :blob };
    if (iter.endFetch()) break;
    writeToBlob(blob, 512*rowNum);
}
iter.close();
...
```

This example calls writeToBlob() for each BLOB in BASIC_LOB_TABLE. Each row writes an additional 512 bytes of data.

Support for Oracle ROWID

The Oracle-specific ROWID type stores the unique address for each row in a database table. The oracle.sql.ROWID class wraps ROWID information and is used to bind and define variables of the ROWID type.

Variables of the oracle.sql.ROWID type can be used in SQLJ applications connecting to Oracle Database 11*g* in the following ways:

- As IN, OUT or INOUT host variables in SQLJ executable statements and in INTO-lists
- As a return value from a stored function call
- As column types in iterator declarations

ROWIDs in Iterator Declarations

You can use oracle.sql.ROWID as a column type for SQLJ positional and named iterators, as shown in the following declarations:

```
#sql iterator NamedRowidIter (String ename, ROWID rowid);

#sql iterator PositionedRowidIter (String, ROWID);
```

ROWID Host Variables and Named-Iterator SELECT Results

You can use ROWID objects as IN, OUT and INOUT parameters in SQLJ executable statements. In addition, you can populate iterators whose columns include ROWID types. This code example uses the preceding example declarations.

```
#sql iterator NamedRowidIter (String ename, ROWID rowid);

...
NamedRowidIter iter;
```

```
ROWID rowid;
#sql iter = { SELECT ename, rowid FROM emp };
while (iter.next())
{
   if (iter.ename().equals("CHUCK TURNER"))
   {
       rowid = iter.rowid();
       #sql { UPDATE emp SET sal = sal + 500 WHERE rowid = :rowid };
   }
}
iter.close();
...
```

This example increases the salary of the employee named Chuck Turner by $500 according to the ROWID.

ROWID Stored Function Results

Consider the following function:

```
CREATE OR REPLACE FUNCTION get_rowid (name VARCHAR2) RETURN ROWID IS
   rid ROWID;
BEGIN
   SELECT rowid INTO rid FROM emp WHERE ename = name;
   RETURN rid;
END get_rowid;
```

Given the preceding stored function, the following example indicates how a `ROWID` object is used as the assignment type for the function return result:

```
ROWID rowid;
#sql rowid = { values(get_rowid('AMY FEINER')) };
#sql { UPDATE emp SET sal = sal + 500 WHERE rowid = :rowid };
```

This example increases the salary of the employee named Amy Feiner by $500 according to the ROWID.

ROWID SELECT INTO Targets

Host variables of the `ROWID` type can appear in the `INTO`-list of a `SELECT INTO` statement.

```
ROWID rowid;
#sql { SELECT rowid INTO :rowid FROM emp WHERE ename='CHUCK TURNER' };
#sql { UPDATE emp SET sal = sal + 500 WHERE rowid = :rowid };
```

This example increases the salary of the employee named Chuck Turner by $500 according to the ROWID.

ROWID Host Variables and Positional Iterator FETCH INTO Targets

Host variables of the `ROWID` type can appear in the `INTO`-list of a `FETCH INTO` statement if the corresponding column attribute in the iterator is of the identical type.

```
#sql iterator PositionedRowidIter (String, ROWID);

...
PositionedRowidIter iter;
ROWID rowid = null;
String ename = null;
#sql iter = { SELECT ename, rowid FROM emp };
while (true)
```

```
{
   #sql { FETCH :iter INTO :ename, :rowid };
   if (iter.endFetch()) break;
   if (ename.equals("CHUCK TURNER"))
   {
       #sql { UPDATE emp SET sal = sal + 500 WHERE rowid = :rowid };
   }
}
iter.close();
...
```

This example is similar to the previous named iterator example, but uses a positional iterator with its customary `FETCH INTO` syntax.

Positioned Update and Delete

In Oracle Database 11*g* release 1 (11.1), SQLJ supports positioned update and delete operations. A positioned update or delete operation can be done using an iterator. The iterator used for positioned update or delete should implement the `sqlj.runtime.ForUpdate` interface. You can use a named iterator, positional iterator, or scrollable iterator.

The following code illustrates a positioned update:

```
...
#sql iterator iter implements sqlj.runtime.ForUpdate(String str)
...
#sql iter = {SELECT ename FROM emp WHERE dpetno=10};
...
while(iter.next())
{
 #sql {UPDATE emp SET sal=sal+5000 WHERE CURRENT OF :iter};
}
...
```

In the preceding code, an iterator `iter` is created and used to update the `emp` table.

You can similarly perform a positioned delete. For example:

```
...
#sql {DELETE FROM emp WHERE CURRENT OF :iter}
...
```

In the preceding example, `iter` is an iterator used to perform positioned delete.

The iterators that can be used with the `WHERE CURRENT OF` clause have the following limitations:

- The query used to populate the iterator should not operate on multiple tables.
- You *cannot* use a PL/SQL procedure returning a `REF CURSOR` with the iterator.
- You *cannot* use an iterator that has been populated from a result set. That is, an iterator populated using the following statement, where `rs` is a result set:

  ```
  #sql iter = {cast :rs}
  ```

for_update Option

If `for_update` option is set at translation time, then `"FOR UPDATE"` is appended to the SELECT statements, which in turn return results into a `ForUpdate` iterator as follows:

```
% sqlj -for_update abc.sqlj

/* abc.sqlj */
#sql iterator  SalByName (double sal, String ename)  implements
sqlj.runtime.ForUpdate;

public class abc {
…..
  void func1()
  {
    SalByName salbn;
    #sql salbn = {select sal, ename from emp };
  }
…..
}
```

Now, "`FOR UPDATE`" is appended to the SELECT statement returning the `ForUpdate` iterator `salbn` in the following way:

```
……
String theSqlTS = "SELECT rowid sjT_rowid,ename, sal FROM emp WHERE ename =  :1
FOR UPDATE";
……
```

Table 5–3 shows the plausible values for the `for_update` option and the corresponding SQL statement for the preceding example:

Table 5–3 Plausible values for the for_update option and the corresponding SQL statement

for_update option	SQLJ Statement	SQL Statement
none	`sqlj -for_update abc.sqlj`	`SELECT rowid sjT_rowid,ename, sal FROM emp WHERE ename = :1 FOR UPDATE`
`nowait`	`sqlj -P-for_update=nowait QueryDemo.sqlj`	`SELECT rowid sjT_rowid,ename, sal FROM emp WHERE ename = :1 FOR UPDATE nowait`
`number`	`sqlj -P-for_update=10 QueryDemo.sqlj`	`SELECT rowid sjT_rowid,ename, sal FROM emp WHERE ename = :1 FOR UPDATE wait 10`

Note: If the application already has `FOR UPDATE` in the select query, then using these new translator options will throw warnings during online check at translation time. If offline parsing is chosen during translation, then errors are not detected at translation time.

Support for Oracle REF CURSOR Types

Oracle PL/SQL and the Oracle SQLJ implementation support the use of cursor variables that represent database cursors.

Overview of REF CURSOR Types

Cursor variables are functionally equivalent to JDBC result sets, essentially encapsulating the results of a query. A cursor variable is often referred to as a REF CURSOR, but `REF CURSOR` itself is a type specifier, and not a type name. Instead, named REF CURSOR types must be specified. The following example shows a REF CURSOR type specification:

```
TYPE EmpCurType IS REF CURSOR;
```

Stored procedures and stored functions can return parameters of Oracle REF CURSOR types. You must use PL/SQL to return a REF CURSOR parameter. You cannot accomplish this using SQL alone. A PL/SQL stored procedure or function can declare a variable of some named REF CURSOR type, execute a `SELECT` statement, and return the results in the REF CURSOR variable.

> **See Also:** *Oracle Database PL/SQL Language Reference*

REF CURSOR Types in SQLJ

In the Oracle SQLJ implementation, a REF CURSOR type can be mapped to iterator columns or host variables of any iterator class type or of the `java.sql.ResultSet` type, but host variables can be `OUT` only. Support for REF CURSOR types can be summarized as follows:

- As result expressions for stored function returns
- As output host expressions for stored procedure or function output parameters
- As output host expressions in `INTO`-lists
- As iterator columns

You can use the SQL `CURSOR` operator for a nested `SELECT` within an outer `SELECT` statement. This is how you can write a REF CURSOR object to an iterator column or `ResultSet` column in an iterator, or write a REF CURSOR object to an iterator host variable or `ResultSet` host variable in an `INTO`-list.

> **See Also:** "Using Iterators and Result Sets as Host Variables" on page 4-37 for examples illustrating the use of implicit REF CURSOR variables, including an example of the `CURSOR` operator.

Note:

- Use the type code `OracleTypes.CURSOR` for REF CURSOR types.
- There is no `oracle.sql` class for REF CURSOR types. Use either `java.sql.ResultSet` or an iterator class. Close the result set or iterator to release resources when you are done processing it.

REF CURSOR Example

The following sample method shows a REF CURSOR type being retrieved from an anonymous block:

```
private static EmpIter refCursInAnonBlock(String name, int no)
  throws java.sql.SQLException {
  EmpIter emps = null;
```

```
  System.out.println("Using anonymous block for ref cursor..");
  #sql { begin
           INSERT INTO emp (ename, empno) VALUES (:name, :no);
           OPEN :out emps FOR SELECT ename, empno FROM emp ORDER BY empno;
         end
       };
  return emps;
}
```

Support for Other Oracle Database 11*g* Data Types

All `oracle.sql` classes can be used for iterator columns or for input, output, or input-output host variables in the same way that any standard Java type can be used. This includes the classes mentioned in the preceding sections and others, such as the `oracle.sql.NUMBER`, `oracle.sql.CHAR`, and `oracle.sql.RAW` classes.

Because the `oracle.sql.*` classes do not require conversion to Java type format, they offer greater efficiency and precision than equivalent Java types. You would have to convert the data to standard Java types, however, to use it with standard Java programs or to display it to end users.

Extended Support for BigDecimal

SQLJ supports `java.math.BigDecimal` in the following situations:

- As host variables in SQLJ executable statements
- As return values from stored function calls
- As iterator column types

Standard SQLJ has the limitation that a value can be retrieved as `BigDecimal` only if that is the JDBC default mapping, which is the case only for numeric and decimal data.

> **See Also:** Table 5–1, " Type Mappings for Supported Host Expression Types"

In the Oracle SQLJ implementation, however, you can map to nondefault types as long as the data type is convertible from numeric and you use Oracle9*i* Database or later version, an Oracle JDBC driver, Oracle-specific code generation or the Oracle customizer, and the Oracle SQLJ run time. The `CHAR`, `VARCHAR2`, `LONG`, and `NUMBER` types are convertible. For example, you can retrieve data from a `CHAR` column into a `BigDecimal` variable. However, to avoid errors, you must be careful that the character data consists only of numbers.

> **Note:** The `BigDecimal` class is in the standard `java.math` package.

6

Objects, Collections, and OPAQUE Types

This chapter discusses how the Oracle SQLJ implementation supports user-defined SQL types. This includes discussion of the Oracle JPublisher utility, which you can use to generate Java classes corresponding to user-defined SQL types. There is also a small section at the end regarding Oracle OPAQUE types.

The chapter consists of the following sections:

Oracle Objects and Collections

This section provides some background conceptual information about Oracle Database 11*g* objects and collections.

> **See Also:** *Oracle Database SQL Language Reference* and *Oracle Database Advanced Application Developer's Guide*.

This section covers the following topics:

Introduction to Objects and Collections

The Oracle SQLJ implementation supports user-defined SQL object types, which are composite data structures, related SQL object reference types, and user-defined SQL

collection types. Oracle objects and collections are composite data structures consisting of individual data elements.

The Oracle SQLJ implementation supports either strongly typed or weakly typed Java representations of object types, reference types, and collection types to use in iterators or host expressions. Strongly typed representations use a custom Java class that maps to a particular object type, reference type, or collection type and must implement either the Java Database Connectivity (JDBC) 2.0 standard `java.sql.SQLData` interface, for object types only, or the Oracle `oracle.sql.ORAData` interface. Either paradigm is supported by the Oracle Database 11*g* JPublisher utility, which you can use to automatically generate custom Java classes.

The term strongly typed is used where a particular Java type is associated with a particular SQL named type or user-defined type. For example, if there is a `PERSON` type, then a corresponding `Person` Java class will be associated with it.

Weakly typed representations use `oracle.sql.STRUCT` for objects, `oracle.sql.REF` for object references, or `oracle.sql.ARRAY` for collections. Alternatively, you can use standard `java.sql.Struct`, `java.sql.Ref`, or `java.sql.Array` objects in a weakly typed scenario.

The term weakly typed is used where a Java type is used in a generic way and can map to multiple SQL named types. The Java class or interface has no special information particular to any SQL type. This is the case for the `oracle.sql.STRUCT`, `oracle.sql.REF`, and `oracle.sql.ARRAY` types and the `java.sql.Struct`, `java.sql.Ref`, and `java.sql.Array` types.

Note that using Oracle extensions in your code requires the following:

- Use one of the Oracle JDBC drivers.
- Use default Oracle-specific code generation or, for ISO code generation, customize the profiles appropriately. For Oracle-specific generated code, no profiles are produced so customization is not applicable. Oracle JDBC application programming interfaces (APIs) are called directly through the generated Java code.

  Note: Oracle recommends the use of the default customizer, `oracle.sqlj.runtime.util.OraCustomizer`.

- Use the Oracle SQLJ run time when your application runs. The Oracle SQLJ run time and an Oracle JDBC driver are required whenever you use the Oracle customizer, even if you do not actually use Oracle extensions in your code.

For Oracle-specific semantics-checking, you must use an appropriate checker. The default checker, `oracle.sqlj.checker.OracleChecker`, acts as a front end and will run the appropriate checker based on your environment. This will be one of the Oracle-specific checkers if you are using an Oracle JDBC driver.

Note: Oracle-specific types for Oracle objects and collections are included in the `oracle.sql` package.

See Also: "Connection Options" on page 8-25 and "Semantics-Checking and Offline-Parsing Options" on page 8-56

Custom Java Class Usage Notes

- This chapter primarily discusses the use of custom Java classes with user-defined types. However, classes implementing `ORAData` can be used for other Oracle SQL types as well. A class implementing `ORAData` can be used to perform any kind of desired processing or conversion in the course of transferring data between SQL and Java.

  See Also: "Additional Uses for ORAData Implementations" on page 6-13

- The `SQLData` interface is intended only for custom object classes. The `ORAData` interface can be used for any custom Java class.

Terminology Notes

- User-defined SQL object types and user-defined SQL collection types are referred to as user-defined types (UDTs).
- Custom Java classes for objects, references, and collections are referred to as custom object classes, custom reference classes, and custom collection classes, respectively.

  See Also: *Oracle Database Object-Relational Developer's Guide* for general information about Oracle object features and functionality

Oracle Object Fundamentals

The Oracle SQL objects are composite data structures that group related data items, such as facts about each employee, into a single data unit. An object type is functionally similar to a Java class. You can populate and use any number of individual objects of a given object type, just as you can instantiate and use individual objects of a Java class.

For example, you can define an object type `EMPLOYEE` that has the attributes `name` of type `CHAR`, `address` of type `CHAR`, `phonenumber` of type `CHAR`, and `employeenumber` of type `NUMBER`.

Oracle objects can also have methods, or stored procedures, associated with the object type. These methods can be either static methods or instance methods and can be implemented either in PL/SQL or Java. Their signatures can include any number of input, output, or input-output parameters. All this depends on how they are initially defined

Oracle Collection Fundamentals

There are two categories of Oracle SQL collections:

- Variable-length arrays (VARRAY types)
- Nested tables (TABLE types)

Both categories are one-dimensional, although the elements can be complex object types. VARRAY types are used for one-dimensional arrays, and nested table types are used for single-column tables within an outer table. A variable of any VARRAY type can be referred to as a VARRAY. A variable of any nested table type can be referred to as a nested table.

A VARRAY, as with any array, is an ordered set of data elements, with each element having an index and all elements being of the same data type. The size of a VARRAY refers to the maximum number of elements. Oracle VARRAYs, as indicated by their

name, are of variable size, but the maximum size of any particular VARRAY type must be specified when the VARRAY type is declared.

A nested table is an unordered set of elements. Nested table elements within a table can themselves be queried in SQL. A nested table, as with any table, is not created with any particular number of rows. This is determined dynamically.

Note: The elements in a VARRAY or the rows in a nested table can be of a user-defined object type, and VARRAY and nested table types can be used for attributes in a user-defined object type. Oracle Database 11*g* supports nesting of collection types. The elements of a VARRAY or rows of a nested table can be of another VARRAY or nested table type, or these elements can be of a user-defined object type that has VARRAY or nested table attributes.

Object and Collection Data Types

In Oracle Database 11*g*, user-defined object and collection definitions function as SQL data type definitions. You can use these data types, as with any other data type, in defining table columns, SQL object attributes, and stored procedure or function parameters. In addition, once you have defined an object type, the related object reference type can be used as any other SQL reference type.

For example, consider the `EMPLOYEE` Oracle object described in the preceding section. Once you have defined this object, it becomes an Oracle data type. You can have a table column of type `EMPLOYEE` just as you can have a table column of type `NUMBER`. Each row in an `EMPLOYEE` column contains a complete `EMPLOYEE` object. You can also have a column type of `REF EMPLOYEE`, consisting of references to `EMPLOYEE` objects.

Similarly, you can define a variable-length array `MYVARR` as `VARRAY(10)` of `NUMBER` and a nested table `NTBL` of `CHAR(20)`. The `MYVARR` and `NTBL` collection types become Oracle data types, and you can have table columns of either type. Each row of a `MYVARR` column consists of an array of up to 10 numbers. Each row of an `NTBL` column consists of 20 characters.

Custom Java Classes

Custom Java classes are first-class types that you can use to read from and write to user-defined SQL types transparently. The purpose of custom Java classes is to provide a way to convert data between SQL and Java and make the data accessible, particularly in supporting objects and collections or if you want to perform custom data conversions.

It is generally advisable to provide custom Java classes for all user-defined types that you use in a SQLJ application. The Oracle JDBC driver will use instances of these classes in converting data, which is more convenient and less error-prone than using the weakly typed `oracle.sql.STRUCT`, `oracle.sql.REF`, and `oracle.sql.ARRAY` classes.

To be used in SQLJ iterators or host expressions, a custom Java class must implement either the `oracle.sql.ORAData` and `oracle.sql.ORADataFactory` interfaces or the standard `java.sql.SQLData` interface. This section provides an overview of these interfaces and custom Java class functionality, covering the following topics:

- Custom Java Class Interface Specifications
- Custom Java Class Support for Object Methods

Custom Java Class Interface Specifications

This section discusses specifications of the `ORAData` and `ORADataFactory` interfaces and the standard `SQLData` interface.

Oracle Database 11*g* includes a set of APIs for Oracle-specific custom Java class functionality for user-defined types: `oracle.sql.ORAData` and `oracle.sql.ORADataFactory`.

The `oracle.sql.CustomDatum` and `oracle.sql.CustomDatumFactory` interfaces used previously for this functionality are deprecated.

ORAData and ORADataFactory Specifications

Oracle provides the `oracle.sql.ORAData` interface and the related `oracle.sql.ORADataFactory` interface to use in mapping and converting Oracle object types, reference types, and collection types to custom Java classes.

Data is sent or retrieved in the form of an `oracle.sql.Datum` object, with the underlying data being in the format of the appropriate `oracle.sql.Datum` subclass, such as `oracle.sql.STRUCT`. This data is still in its SQL format. The `oracle.sql.Datum` object is just a wrapper.

See Also: *Oracle Database JDBC Developer's Guide and Reference*

The `ORAData` interface specifies a `toDatum()` method for data conversion from Java format to SQL format. This method takes as input your connection object and converts data to the appropriate `oracle.sql.*` representation. The connection object is necessary so that the JDBC driver can perform appropriate type checking and type conversions at run time. The `ORAData` and `toDatum()` specification is as follows:

```
interface oracle.sql.ORAData
{
   oracle.sql.Datum toDatum(java.sql.Connection c) throws SQLException;
}
```

The `ORADataFactory` interface specifies a `create()` method that constructs instances of your custom Java class, converting from SQL format to Java format. This method takes as input a `Datum` object containing the data and a type code, such as `OracleTypes.RAW`, indicating the SQL type of the underlying data. It returns an object of your custom Java class, which implements the `ORAData` interface. This object receives its data from the `Datum` object that was input. The `ORADataFactory` and `create()` specification is as follows:

```
interface oracle.sql.ORADataFactory
{
   oracle.sql.ORAData create(oracle.sql.Datum d, int sqlType)
                     throws SQLException;
}
```

To complete the relationship between the `ORAData` and `ORADataFactory` interfaces, you must implement a static `getORADataFactory()` method in any custom Java class that implements the `ORAData` interface. This method returns an object that

implements the ORADataFactory interface and that, therefore, can be used to create instances of your custom Java class. This returned object can itself be an instance of your custom Java class, and its create() method is used by the Oracle JDBC driver to produce further instances of your custom Java class, as necessary.

Note: JPublisher output implements the ORAData interface and its toDatum() method and the ORADataFactory interface and its create() method in a single custom Java class. However, toDatum() and create() are specified in different interfaces to allow the option of implementing them in separate classes. You can have one custom Java class that implements ORAData, its toDatum() method, and the getORADataFactory() method, and have a separate factory class that implements ORADataFactory and its create() method. For purposes of discussion here, however, the assumption is that both interfaces are implemented in a single class.

See Also: "Oracle Requirements for Classes Implementing ORAData" on page 6-8

If you use JPublisher, then specifying -usertypes=oracle will result in JPublisher generating custom Java classes that implement the ORAData and ORADataFactory interfaces and the getORADataFactory() method. For backward compatibility, you have the option of using the JPublisher -compatible option in conjunction with -usertypes=oracle to use the CustomDatum and CustomDatumFactory interfaces instead.

See Also: *Oracle Database JPublisher User's Guide*

ORAData Versus CustomDatum Interfaces

The oracle.jdbc interfaces were introduced in Oracle9*i* Database as replacements for the oracle.jdbc.driver classes. As a result, the oracle.sql.CustomDatum and oracle.sql.CustomDatumFactory interfaces, formerly used to access customized objects, are deprecated in favor of the oracle.sql.ORAData and oracle.sql.ORADataFactory interfaces. Like the CustomDatum interfaces, these can be used as an Oracle-specific alternative to the standard SQLData interface. The CustomDatum interfaces are still supported for backward compatibility.

CustomDatum and CustomDatumFactory have the following definitions:

```
public interface CustomDatum
{
  oracle.sql.Datum toDatum(
    oracle.jdbc.driver.OracleConnection conn
  ) throws SQLException;

public interface CustomDatumFactory
{
  oracle.sql.CustomDatum create(
    oracle.sql.Datum d, int sqlType
    ) throws SQLException;
}
```

The connection conn and type code sqlType are used as described for ORAData and ORADataFactory. Note, however, that CustomDatum uses the Oracle-specific OracleConnection type instead of the standard Connection type.

SQLData Specification

Standard JDBC 2.0 supplies the `java.sql.SQLData` interface to use in mapping and converting structured object types to Java classes. This interface is intended for mapping structured object types only, not object references, collections or arrays, or other SQL types.

The `SQLData` interface is a JDBC 2.0 standard, specifying a `readSQL()` method to read data into a Java object and a `writeSQL()` method to write to the database from a Java object. If you use JPublisher, then specifying `-usertypes=jdbc` will result in JPublisher generating custom Java classes that implement the `SQLData` interface.

For additional information about standard `SQLData` functionality, refer to the Sun Microsystems JDBC 2.0 or later API specification.

> **See Also:** "Requirements for Classes Implementing SQLData" on page 6-10

Custom Java Class Support for Object Methods

Methods of Oracle objects can be invoked from custom Java class wrappers. Whether the underlying stored procedure is written in PL/SQL or is written in Java and published to SQL is invisible to the user.

A Java wrapper method used to invoke a server method requires a connection to communicate with the server. The connection object can be provided as an explicit parameter or can be associated in some other way. For example, as an attribute of your custom Java class. If the connection object used by the wrapper method is a nonstatic attribute, then the wrapper method must be an instance method of the custom Java class in order to have access to the connection. Custom Java classes generated by JPublisher use this technique.

There are also issues regarding output and input-output parameters in methods of Oracle objects. If a stored procedure, that is, a SQL object method, modifies the internal state of one of its arguments, then the actual argument passed to the stored procedure is modified. In Java this is not possible. When a JDBC output parameter is returned from a stored procedure call, it must be stored in a newly created object. The original object identity is lost.

One way to return an output or input-output parameter to the caller is to pass the parameter as an element of an array. If the parameter is input-output, then the wrapper method takes the array element as input. After processing, the wrapper assigns the output to the array element. Custom Java classes generated by JPublisher use this technique, with each output or input-output parameter being passed in a one-element array.

When you use JPublisher, it implements wrapper methods by default. This is true for generated classes implementing either the `SQLData` interface or the `ORAData` interface. To disable this feature, set the JPublisher `-methods` flag to `false`.

> **See Also:** *Oracle Database JPublisher User's Guide*

> **Note:** If you are implementing a custom Java class, then there are various ways that you can implement wrapper methods. Data processing in the server can be done either through the SQL object method directly or by forwarding the object value from the client to the server and then executing the method there. To see how JPublisher implements wrapper methods, and whether this may meet your needs, refer to "JPublisher Implementation of Wrapper Methods" on page 6-33.

Custom Java Class Requirements

Custom Java classes must satisfy certain requirements to be recognized by the Oracle SQLJ translator as valid host variable types and to enable type-checking by the translator.

> **Note:** Custom Java classes for user-defined types are often referred to in this manual as "wrapper classes".

Oracle Requirements for Classes Implementing ORAData

Oracle requirements for `ORAData` implementations are primarily the same for any kind of custom Java class, but vary slightly depending on whether the class is for mapping to objects, object references, collections, or some other SQL type.

These requirements are as follows:

- The class implements the `oracle.sql.ORAData` interface.
- The class implements the `getORADataFactory()` method that returns an `oracle.sql.ORADataFactory` object. The method signature is as follows:

  ```
  public static oracle.sql.ORADataFactory getORADataFactory();
  ```

  If using the deprecated `CustomDatum` interface, then the class implements the `getFactory()` method that returns an `oracle.sql.CustomDatumFactory` object. The method signature is as follows:

  ```
  public static oracle.sql.CustomDatumFactory getFactory();
  ```

- The class has a `String` constant, `_SQL_TYPECODE`, initialized to the `oracle.jdbc.OracleTypes` type code of the `Datum` subclass instance that `toDatum()` returns. The type code is:
  - For custom object classes:

    ```
    public static final int _SQL_TYPECODE = OracleTypes.STRUCT;
    ```

  - For custom reference classes:

    ```
    public static final int _SQL_TYPECODE = OracleTypes.REF;
    ```

  - For custom collection classes:

    ```
    public static final int _SQL_TYPECODE = OracleTypes.ARRAY;
    ```

  For other uses, some other type code might be appropriate. For example, for using a custom Java class to serialize and deserialize Java objects into or out of `RAW` fields, a `_SQL_TYPECODE` of `OracleTypes.RAW` is used.

Note: The `OracleTypes` class simply defines a type code, which is an integer constant, for each Oracle data type. For standard SQL types, the `OracleTypes` entry is identical to the entry in the standard `java.sql.Types` type definitions class.

See Also: "Serialized Java Objects" on page 6-50

- For custom Java classes with `_SQL_TYPECODE` of `STRUCT`, `REF`, or `ARRAY`, that is, for custom Java classes that represent objects, object references, or collections, the class has a constant that indicates the relevant user-defined type name. This is as follows:
  - Custom object classes and custom collection classes must have a `String` constant, `_SQL_NAME`, initialized to the SQL name you declared for the user-defined type, as follows:

    ```
    public static final String _SQL_NAME = UDT name;
    ```

    For example, the custom object class for a user-defined `PERSON` object will have the constant:

    ```
    public static final String _SQL_NAME = "PERSON";
    ```

    The same can be specified along with the schema, if appropriate, as follows:

    ```
    public static final String _SQL_NAME = "SCOTT.PERSON";
    ```

    The custom collection class for a collection of `PERSON` objects, which you have declared as `PERSON_ARRAY`, will have the constant:

    ```
    public static final String _SQL_NAME = "PERSON_ARRAY";
    ```

  - Custom reference classes must have a `String` constant, `_SQL_BASETYPE`, initialized to the SQL name you declared for the user-defined type being referenced, as follows:

    ```
    public static final String _SQL_BASETYPE = UDT name;
    ```

    The custom reference class for `PERSON` references will have the constant:

    ```
    public static final String _SQL_BASETYPE = "PERSON";
    ```

    For other `ORAData` uses, specifying a UDT name is not applicable.

Keep in mind the following usage notes:

- A collection type name reflects the collection type, not the base type. For example, if you have declared a VARRAY or nested table type, `PERSON_ARRAY`, for `PERSON` objects, then the name of the collection type that you specify for the `_SQL_NAME` entry is `PERSON_ARRAY`, not `PERSON`.
- When specifying the SQL type in a `_SQL_NAME` field, if the SQL type was declared in a case-sensitive way (in quotes), then you must specify the SQL name exactly as it was declared, such as `CaseSensitive` or `SCOTT.CaseSensitive`. Note that this differs from usage in a JPublisher input file, where the case-sensitive name must also appear in quotes. If you did *not* declare the SQL type in a case-sensitive way, that is, without no quotes, then you must specify the SQL name in all uppercase, such as `ADDRESS` or `SCOTT.ADDRESS`.

JPublisher automatically generates the value of this field appropriately, according to case-sensitivity and the JPublisher `-omit_schema_names` setting, if applicable.

Requirements for Classes Implementing SQLData

The ISO SQLJ standard outlines requirements for type map definitions for classes implementing the `SQLData` interface. Alternatively, `SQLData` wrapper classes can identify associated SQL object types through the `public static final` fields.

Be aware of the following important points:

- Whether you use a type map or use alternative (nonstandard) `public static final` fields to specify mappings, you must be consistent in your approach. Either use a type map that specifies all relevant mappings so that you do not require the `public static final` fields, or do not use a type map at all and specify all mappings through the `public static final` fields.
- `SQLData`, unlike `ORAData`, is for mapping structured object types only. It is not for object references, collections or arrays, or any other SQL types. If you are not using `ORAData`, then your only choices for mapping object references and collections are the weak `java.sql.Ref` and `java.sql.Array` types, respectively, or `oracle.sql.REF` and `oracle.sql.ARRAY`.
- When specifying the mapping from a SQL type to a Java type, if the SQL type was declared in a case-sensitive way, then you must specify the SQL name exactly as it was declared, such as `CaseSensitive` or `SCOTT.CaseSensitive`. Note that this differs from usage in a JPublisher input file, where the case-sensitive name must also appear in quotes. If you did *not* declare the SQL type in a case-sensitive way, then you must specify the SQL name in all uppercase, such as `ADDRESS` or `SCOTT.ADDRESS`.

Mapping Specified in Type Map Resource

First, consider the mapping representation according to the ISO SQLJ standard. Assume that `Address`, `pack.Person`, and `pack.Manager.InnerPM`, where `InnerPM` is an inner class of `Manager`, are three wrapper classes that implement `java.sql.SQLData`.

Then, you need to consider the following:

- You must use these classes only in statements that use explicit connection context instances of a declared connection context type. For example, assuming that this type is called `SDContext`:

```
Address              a =...;
pack.Person          p =...;
pack.Manager.InnerPM pm =...;
SDContext ctx = new SDContext(url,user,pwd,false);
#sql [ctx] { ... :a ... :p ... :pm ... };
```

- The connection context type must have been declared using the `with` attribute `typeMap` that specifies an associated class implementing `java.util.PropertyResourceBundle`. In the preceding example, `SDContext` may be declared as follows:

```
#sql public static context SDContext with (typeMap="SDMap");
```

- The type map resource must provide the mapping from SQL object types to corresponding Java classes that implement the `java.sql.SQLData` interface. This mapping is specified with entries of the following form:

```
class.java_class_name=STRUCT sql_type_name
```

The `STRUCT` keyword can also be omitted. In the example, the `SDMap.properties` resource file may contain the following entries:

```
class.Address=STRUCT SCOTT.ADDRESS
class.pack.Person=PERSON
class.pack.Manager$InnerPM=STRUCT PRODUCT_MANAGER
```

Although the period (.) separates package and class name, you *must* use the dollar sign ($) to separate an inner class name.

Important: If you used the default Oracle-specific code generation in this example, then any iterator that is used for a statement whose context type is `SDContext` must also have been declared with the same associated type map, `SDMap`, such as in the following example:

```
#sql public static iterator SDIter with (typeMap="SDMap");
...
SDContext sdctx = ...
SDIter sditer;
#sql [sdctx] sditer = { SELECT ...};
```

This is to ensure that proper code is generated for the iterator class.

This mechanism of specifying mappings in a type map resource is more complicated than the nonstandard alternative. Also, it is not possible to associate a type map resource with the default connection context. The advantage is that all the mapping information is placed in a single location, the type map resource. This means that the type mapping in an already compiled application can be easily adjusted at a later time, for example, to accommodate new SQL types and Java wrappers in an expanding SQL-Java type hierarchy.

Be aware of the following:

- You must employ the SQLJ `runtime12` or `runtime12ee` library to use this feature. Type maps are represented as `java.util.Map` objects. These are exposed in the SQLJ run time API and, therefore, *cannot* be supported by the generic run time library.
- You must use the Oracle SQLJ run time and Oracle-specific code generation or profile customization if your `SQLData` wrapper classes occur as `OUT` or `INOUT` parameters in SQLJ statements. This is because the SQL type of such parameters is required for `registerOutParameter()` by the Oracle JDBC driver. Also, for `OUT` parameter type registration, the SQL type is "frozen in" by the type map in effect during translation.
- The SQLJ type map is independent of any JDBC type map you may be using on the underlying connection. Thus, you must be careful when you are mixing SQLJ and JDBC code if both use `SQLData` wrappers. However, you can easily extract the type map in effect on a given SQLJ connection context:

  ```
  ctx.getTypeMap();
  ```

Mapping Specified in Static Field of Wrapper Class

A class that implements `SQLData` can satisfy the following nonstandard requirement:

- The Java class declares the `String` constant `_SQL_NAME`, which defines the name of the SQL type that is being wrapped by the Java class. In the example, the `Address` class would have the following field declaration:

  ```
  public static final String _SQL_NAME="SCOTT.ADDRESS";
  ```

  The following declaration would be in `pack.Person`:

  ```
  public static final String _SQL_NAME="PERSON";
  ```

  And the `pack.Manager.InnerPM` class would have the following:

  ```
  public static final String _SQL_NAME="PRODUCT_MANAGER";
  ```

Note that JPublisher always generates `SQLData` wrapper classes with the `_SQL_NAME` field. However, this field is ignored in SQLJ statements that reference a type map.

Note:

- If a class that implements the `_SQL_NAME` field is used in a SQLJ statement with an explicit connection context type and associated type map, then that type map is used and the `_SQL_NAME` field is ignored. This simplifies migration of existing SQLJ programs to the new ISO SQLJ standard.
- The static SQL-Java type correspondence specified in the `_SQL_NAME` field is independent from any JDBC type map you may be using on the underlying connection. Thus, you must be careful when you are mixing SQLJ and JDBC code if both use `SQLData` wrappers.

Compiling Custom Java Classes

You can include any `.java` files for your custom Java classes, whether `ORAData` or `SQLData` implementations, on the SQLJ command line together with the `.sqlj` files for your application. However, this is not necessary if the SQLJ `-checksource` flag is set to `true`, which is the default, and your classpath includes the directory where the custom Java source is located.

Note: This discussion assumes you are creating `.java` files for your custom objects and collections, not `.sqlj` files. Any `.sqlj` files must be included in the SQLJ command line.

For example, if `ObjectDemo.sqlj` uses the `ADDRESS` and `PERSON` Oracle object types and you have produced custom Java classes for these objects, then you can run SQLJ as follows.

- If `-checksource=true` and the classpath includes the custom Java source location:

  ```
  % sqlj ObjectDemo.sqlj
  ```

- If `-checksource=false` (this is a single wraparound line):

  ```
  % sqlj ObjectDemo.sqlj Address.java AddressRef.java Person.java PersonRef.java
  ```

You also have the choice of using your Java compiler to compile custom `.java` source files directly. If you do this, then you must do it prior to translating `.sqlj` files.

See Also:

- "Source File Name Check (-checkfilename)" on page 8-54
- Chapter 8, "Translator Command Line and Options"

Note: Because `ORAData` implementations rely on Oracle-specific features, SQLJ will report numerous portability warnings if you do not use the `-warn=noportable` translator portability setting, which is the default. For information about the `-warn` flag, refer to "Translator Warnings (-warn)" on page 8-33.

Reading and Writing Custom Data

Through the use of custom Java class instances, the Oracle SQLJ and JDBC implementations allow you to read and write user-defined types as though they are built-in types. Exactly how this is accomplished is transparent to the user.

For the mechanics of how data is read and written, for both `ORAData` implementations and `SQLData` implementations, refer to the *Oracle Database JDBC Developer's Guide and Reference*.

Additional Uses for ORAData Implementations

To this point, discussion of custom Java classes has been for use as one of the following.

- Wrappers for SQL objects: custom object classes, for use with `oracle.sql.STRUCT` instances
- Wrappers for SQL references: custom reference classes, for use with `oracle.sql.REF` instances
- Wrappers for SQL collections: custom collection classes, for use with `oracle.sql.ARRAY` instances

It might be useful, however, to provide custom Java classes to wrap other `oracle.sql.*` types as well, for customized conversions or processing. You can accomplish this with classes that implement `ORAData`, but not `SQLData`, as in the following examples:

- Perform encryption and decryption or validation of data.
- Perform logging of values that have been read or are being written.
- Parse character columns, such as character fields containing URL information, into smaller components.
- Map character strings into numeric constants.
- Map data into more desirable Java formats, such as mapping a `DATE` field to `java.util.Date` format.
- Customize data representation, for example, data in a table column is in feet, but you want it represented in meters after it is selected.
- Serialize and deserialize Java objects, for example, into or out of `RAW` fields.

Note: This sort of functionality is not possible through the `SQLData` interface, as `SQLData` implementations can wrap only structured object types.

See Also: "Serialized Java Objects" on page 6-50

General Use of ORAData: BetterDate.java

This example shows a class that implements the `ORAData` interface to provide a customized representation of Java dates and can be used instead of `java.sql.Date`.

Note: This is not a complete application. There is no `main()` method.

```
import java.util.Date;
import oracle.sql.ORAData;
import oracle.sql.DATE;
import oracle.sql.ORADataFactory;
import oracle.jdbc.OracleTypes;

// a Date class customized for user's preferences:
//      - months are numbers 1..12, not 0..11
//      - years are referred to through four-digit numbers, not two.

public class BetterDate extends java.util.Date
             implements ORAData, ORADataFactory {
  public static final int _SQL_TYPECODE = OracleTypes.DATE;

  String[]monthNames={"JAN", "FEB", "MAR", "APR", "MAY", "JUN",
                      "JUL", "AUG", "SEP", "OCT", "NOV", "DEC"};
  String[]toDigit={"0", "1", "2", "3", "4", "5", "6", "7", "8", "9"};

  static final BetterDate _BetterDateFactory = new BetterDate();

  public static ORADataFactory getORADataFactory() { return _BetterDateFactory;}

  // the current time...
  public BetterDate() {
    super();
  }

  public oracle.sql.Datum toDatum(java.sql.Connection conn) {
    return new DATE(toSQLDate());
  }

  public oracle.sql.ORAData create(oracle.sql.Datum dat, int intx) {
    if (dat==null) return null;
    DATE DAT = ((DATE)dat);
    java.sql.Date jsd = DAT.dateValue();
    return new BetterDate(jsd);
  }

  public java.sql.Date toSQLDate() {
    java.sql.Date retval;
    retval = new java.sql.Date(this.getYear()-1900, this.getMonth()-1,
             this.getDate());
```

```
    return retval;
  }
  public BetterDate(java.sql.Date d) {
    this(d.getYear()+1900, d.getMonth()+1, d.getDate());
  }
  private static int [] deconstructString(String s) {
    int [] retval = new int[3];
    int y,m,d; char temp; int offset;
    StringBuffer sb = new StringBuffer(s);
    temp=sb.charAt(1);
    // figure the day of month
    if (temp < '0' || temp > '9') {
      m = sb.charAt(0)-'0';
      offset=2;
    } else {
      m = (sb.charAt(0)-'0')*10 + (temp-'0');
      offset=3;
    }

    // figure the month
    temp = sb.charAt(offset+1);
    if (temp < '0' || temp > '9') {
      d = sb.charAt(offset)-'0';
      offset+=2;
    } else {
      d = (sb.charAt(offset)-'0')*10 + (temp-'0');
      offset+=3;
    }

    // figure the year, which is either in the format "yy" or "yyyy"
    // (the former assumes the current century)
    if (sb.length() <= (offset+2)) {
      y = (((new BetterDate()).getYear())/100)*100 +
          (sb.charAt(offset)- '0') * 10 +
          (sb.charAt(offset+1)- '0');
    } else {
      y = (sb.charAt(offset)- '0') * 1000 +
          (sb.charAt(offset+1)- '0') * 100 +
          (sb.charAt(offset+2)- '0') * 10 +
          (sb.charAt(offset+3)- '0');
    }
    retval[0]=y;
    retval[1]=m;
    retval[2]=d;
//    System.out.println("Constructing date from string as: "+d+"/"+m+"/"+y);
    return retval;
  }
  private BetterDate(int [] stuff) {
    this(stuff[0], stuff[1], stuff[2]);
  }
  // takes a string in the format: "mm-dd-yyyy" or "mm/dd/yyyy" or
  // "mm-dd-yy" or "mm/dd/yy" (which assumes the current century)
  public BetterDate(String s) {
    this(BetterDate.deconstructString(s));
  }

  // years are as '1990', months from 1..12 (unlike java.util.Date!), date
  // as '1' to '31'
  public BetterDate(int year, int months, int date) {
    super(year-1900,months-1,date);
```

```
  }
  // returns "Date: dd-mon-yyyy"
  public String toString() {
    int yr = getYear();
    return getDate()+"-"+monthNames[getMonth()-1]+"-"+
      toDigit[(yr/1000)%10] +
      toDigit[(yr/100)%10] +
      toDigit[(yr/10)%10] +
      toDigit[yr%10];
//    return "Date: " + getDate() + "-"+getMonth()+"-"+(getYear()%100);
  }
  public BetterDate addDays(int i) {
    if (i==0) return this;
    return new BetterDate(getYear(), getMonth(), getDate()+i);
  }
  public BetterDate addMonths(int i) {
    if (i==0) return this;
    int yr=getYear();
    int mon=getMonth()+i;
    int dat=getDate();
    while(mon<1) {
      --yr;mon+=12;
    }
    return new BetterDate(yr, mon,dat);
  }
  // returns year as in 1996, 2007
  public int getYear() {
    return super.getYear()+1900;
  }
  // returns month as 1..12
  public int getMonth() {
    return super.getMonth()+1;
  }
  public boolean equals(BetterDate sd) {
    return (sd.getDate() == this.getDate() &&
            sd.getMonth() == this.getMonth() &&
            sd.getYear() == this.getYear());
  }
  // subtract the two dates; return the answer in whole years
  // uses the average length of a year, which is 365 days plus
  // a leap year every 4, except 100, except 400 years =
  // = 365 97/400 = 365.2425 days = 31,556,952 seconds
  public double minusInYears(BetterDate sd) {
    // the year (as defined in the preceding text) in milliseconds
    long yearInMillis = 31556952L;
    long diff = myUTC()-sd.myUTC();
    return (((double)diff/(double)yearInMillis)/1000.0);
  }
  public long myUTC() {
    return Date.UTC(getYear()-1900, getMonth()-1, getDate(),0,0,0);
  }

  // returns <0 if this is earlier than sd
  // returns = if this == sd
  // else returns >0
  public int compare(BetterDate sd) {
    if (getYear()!=sd.getYear()) {return getYear()-sd.getYear();}
    if (getMonth()!=sd.getMonth()) {return getMonth()-sd.getMonth();}
    return getDate()-sd.getDate();
  }
```

```
}
```

User-Defined Types

This section contains examples of creating and using user-defined object types and collection types in Oracle Database 11*g*.

Creating Object Types

SQL commands to create object types are of the following form:

```
CREATE TYPE typename AS OBJECT
(
  attrname1    datatype1,
  attrname2    datatype2,
  ...         ...
  attrnameN    datatypeN
);
```

Where *typename* is the desired name of your object type, *attrname1* through *attrnameN* are the desired attribute names, and *datatype1* through *datatypeN* are the attribute data types.

The remainder of this section provides an example of creating user-defined object types in Oracle Database 11*g*.

In this example, the following items are created using SQL:

- Two object types, PERSON and ADDRESS
- A typed table for PERSON objects
- An EMPLOYEES table that includes an ADDRESS column and two columns of PERSON references

The script for creating these items is as follows:

```
/*** Using user-defined types (UDTs) in SQLJ ***/
/
/*** Create ADDRESS UDT ***/
CREATE TYPE ADDRESS AS OBJECT
(
  street        VARCHAR(60),
  city          VARCHAR(30),
  state         CHAR(2),
  zip_code      CHAR(5)
)
/
/*** Create PERSON UDT containing an embedded ADDRESS UDT ***/
CREATE TYPE PERSON AS OBJECT
(
  name    VARCHAR(30),
  ssn     NUMBER,
  addr    ADDRESS
)
/
/*** Create a typed table for PERSON objects ***/
CREATE TABLE persons OF PERSON
/
/*** Create a relational table with two columns that are REFs
     to PERSON objects, as well as a column which is an Address ADT. ***/
```

```
CREATE TABLE  employees
(
  empnumber            INTEGER PRIMARY KEY,
  person_data     REF  PERSON,
  manager         REF  PERSON,
  office_addr          ADDRESS,
  salary               NUMBER
)
/*** Insert some data--2 objects into the persons typed table ***/
INSERT INTO persons VALUES (
            PERSON('Wolfgang Amadeus Mozart', 123456,
               ADDRESS('Am Berg 100', 'Salzburg', 'AT','10424')))
/
INSERT INTO persons VALUES (
            PERSON('Ludwig van Beethoven', 234567,
               ADDRESS('Rheinallee', 'Bonn', 'DE', '69234')))
/
/** Put a row in the employees table **/
INSERT INTO employees (empnumber, office_addr, salary) VALUES (
            1001,
            ADDRESS('500 Oracle Parkway', 'Redwood Shores', 'CA', '94065'),
            50000)
/
/** Set the manager and PERSON REFs for the employee **/
UPDATE employees
   SET manager =
       (SELECT REF(p) FROM persons p WHERE p.name = 'Wolfgang Amadeus Mozart')
/
UPDATE employees
   SET person_data =
       (SELECT REF(p) FROM persons p WHERE p.name = 'Ludwig van Beethoven')
```

Note: Use of a table alias, such as `p` in the example, is a recommended general practice in the Oracle SQL implementation, especially in accessing tables with user-defined types. It is required syntax in some cases where object attributes are accessed. Even when not required, it helps in avoiding ambiguities. Refer to the *Oracle Database SQL Language Reference* for more information about table aliases.

Creating Collection Types

There are two categories of collections

- Variable-length arrays (VARRAYs)
- Nested tables

SQL commands to create VARRAY types are of the following form:

```
CREATE TYPE typename IS VARRAY(n) OF datatype;
```

The *typename* designation is the desired name of your VARRAY type, *n* is the desired maximum number of elements in the array, and *datatype* is the data type of the array elements. For example:

```
CREATE TYPE myvarr IS VARRAY(10) OF INTEGER;
```

SQL commands to create nested table types are of the following form:

```
CREATE TYPE typename AS TABLE OF datatype;
```

The *typename* designation is the desired name of your nested table type and *datatype* is the data type of the table elements. This can be a user-defined type as well as a standard data type. A nested table is limited to one column, although that one column type can be a complex object with multiple attributes. The nested table, as with any database table, can have any number of rows. For example:

```
CREATE TYPE person_array AS TABLE OF person;
```

This command creates a nested table where each row consists of a PERSON object.

The rest of this section provides an example of creating a user-defined collection type, as well as object types, in Oracle Database 11*g*.

The following items are created and populated using SQL:

- Two object types, PARTICIPANT_T and MODULE_T
- A collection type, MODULETBL_T, which is a nested table of MODULE_T objects
- A PROJECTS table that includes a column of PARTICIPANT_T references and a column of MODULETBL_T nested tables
- A collection type PHONE_ARRAY, which is a VARRAY of VARCHAR2(30)
- PERSON and ADDRESS objects (repeating the same definitions used earlier in "Creating Object Types" on page 6-17)
- An EMPLOYEES table, which includes a PHONE_ARRAY column

The script for creating these items is as follows:

```
Rem This is a SQL*Plus script used to create schema to demonstrate collection
Rem manipulation in SQLJ

CREATE TYPE PARTICIPANT_T AS OBJECT (
  empno   NUMBER(4),
  ename   VARCHAR2(20),
  job     VARCHAR2(12),
  mgr     NUMBER(4),
  hiredate DATE,
  sal      NUMBER(7,2),
  deptno   NUMBER(2))
/
SHOW ERRORS
CREATE TYPE MODULE_T  AS OBJECT (
  module_id  NUMBER(4),
  module_name VARCHAR2(20),
  module_owner REF PARTICIPANT_T,
  module_start_date DATE,
  module_duration NUMBER )
/
SHOW ERRORS
CREATE TYPE MODULETBL_T AS TABLE OF MODULE_T;
/
SHOW ERRORS
CREATE TABLE projects (
  id NUMBER(4),
  name VARCHAR(30),
  owner REF PARTICIPANT_T,
  start_date DATE,
  duration NUMBER(3),
  modules  MODULETBL_T  ) NESTED TABLE modules STORE AS modules_tab;
```

```
SHOW ERRORS
CREATE TYPE PHONE_ARRAY IS VARRAY (10) OF varchar2(30)
/

/*** Create ADDRESS UDT ***/
CREATE TYPE ADDRESS AS OBJECT
(
  street        VARCHAR(60),
  city          VARCHAR(30),
  state         CHAR(2),
  zip_code      CHAR(5)
)
/
/*** Create PERSON UDT containing an embedded ADDRESS UDT ***/
CREATE TYPE PERSON AS OBJECT
(
  name    VARCHAR(30),
  ssn     NUMBER,
  addr    ADDRESS
)
/
CREATE TABLE  employees
( empnumber            INTEGER PRIMARY KEY,
  person_data     REF  person,
  manager         REF  person,
  office_addr          address,
  salary               NUMBER,
  phone_nums           phone_array
)
/
```

JPublisher and the Creation of Custom Java Classes

Oracle offers flexibility in how users can customize the mapping of Oracle object types, reference types, and collection types to Java classes in a strongly typed paradigm. Developers have the following choices in creating these custom Java classes:

- Using the Oracle JPublisher utility to automatically generate custom Java classes and using those classes directly without modification
- Using JPublisher to automatically generate custom Java classes and corresponding subclasses, which can subsequently be user-modified for any desired functionality
- Manually coding custom Java classes without using JPublisher, if the classes meet the requirements stated in "Custom Java Class Requirements" on page 6-8

Although you have the option of manually coding your custom Java classes, it is advisable to instead use JPublisher-generated classes directly or modify JPublisher-generated subclasses.

JPublisher can implement either the Oracle `oracle.sql.ORAData` interface or the standard `java.sql.SQLData` interface when it generates a custom object class. If you choose the `ORAData` implementation, then JPublisher will also generate a custom reference class. For compatibility with older JDBC versions, JPublisher can also generate classes that implement the deprecated `oracle.sql.CustomDatum` interface.

The SQLData interface is not intended for custom reference or custom collection classes. If you want your code to be portable, then you have no choice but to use standard weakly typed java.sql.Ref objects to map to references and java.sql.Array objects to map to collections.

This section covers the following topics:

- What JPublisher Produces
- Generating Custom Java Classes
- JPublisher INPUT Files and Properties Files
- Creating Custom Java Classes and Specifying Member Names
- JPublisher Implementation of Wrapper Methods
- JPublisher Custom Java Class Examples
- Extending Classes Generated by JPublisher

See Also:

- *Oracle Database JPublisher User's Guide*
- *Oracle Database JDBC Developer's Guide and Reference*

What JPublisher Produces

When you use JPublisher to generate custom Java classes, you can use either an ORAData implementation, for custom object classes, custom reference classes, or custom collection classes, or a SQLData implementation, for custom object classes only. An ORAData implementation will also implement the ORADataFactory interface, for creating instances of the custom Java class.

This is controlled by how you set the JPublisher -usertypes option. A setting of -usertypes=oracle specifies an ORAData implementation, and a setting of -usertypes=jdbc specifies a SQLData implementation.

ORAData Implementation

When you run JPublisher for a user-defined object type and use the ORAData implementation for your custom object class, JPublisher automatically creates the following:

- A custom object class, typically in a .sqlj source file, to act as a type definition to correspond to your Oracle object type

  This class includes accessor methods for each attribute. For example, getFoo() and setFoo() are the accessor methods for the attribute foo. In addition, JPublisher by default will generate wrapper methods in your class that invoke the associated Oracle object methods executing in the server. However, this can be disabled by setting -methods=false. In this case, JPublisher produces no wrapper methods and generates .java files instead of .sqlj files for custom objects.

- A related custom reference class for object references to your Oracle object type

  This class includes a getValue() method that returns an instance of your custom object class and a setValue() method that updates an object value in the database, taking as input an instance of the custom object class.

  A strongly typed reference class is always generated, regardless of whether the SQL object type uses references.

- Custom classes for any object or collection attributes of the top-level object

  This is necessary so that attributes can be materialized in Java whenever an instance of the top-level class is materialized.

When you run JPublisher for a user-defined collection type, choosing the `ORAData` implementation, JPublisher automatically creates the following:

- A custom collection class to act as a type definition to correspond to your Oracle collection type

  This class includes overloaded `getArray()` and `setArray()` methods to retrieve or update a collection as a whole, a `getElement()` method and `setElement()` method to retrieve or update individual elements of a collection, and additional utility methods.

- A custom object class for the elements, if the elements of the collection are objects

  This is necessary so that object elements can be materialized in Java whenever an instance of the collection is materialized.

JPublisher-generated custom Java classes in any of these categories implement the `ORAData` interface, the `ORADataFactory` interface, and the `getORADataFactory()` method.

Note:

- If you specify the `ORAData` implementation, then the generated classes will use Oracle-specific features and, therefore, will not be portable.
- Although deprecated, JPublisher still supports implementation of the `CustomDatum` interface through the `-compatible` option.

Strongly Typed Object References for ORAData Implementations

For Oracle `ORAData` implementations, JPublisher always generates strongly typed object reference classes as opposed to using the weakly typed `oracle.sql.REF` class. This is to provide greater type safety and to mirror the behavior in SQL, where object references are strongly typed. The strongly typed classes, with names like `PersonRef` for references to `PERSON` objects, are essentially wrappers for the `REF` class.

In these strongly typed `REF` wrappers, there is a `getValue()` method that produces an instance of the SQL object that is referenced, in the form of an instance of the corresponding Java class. (Or, in the case of inheritance, perhaps as an instance of a subclass of the corresponding Java class.) For example, if there is a `PERSON` SQL object type with a corresponding `Person` Java class, then there will also be a `PersonRef` Java class. The `getValue()` method of the `PersonRef` class would return a `Person` instance containing the data for a `PERSON` object in the database.

Whenever a SQL object type has an attribute that is an object reference, the Java class corresponding to the object type would have an attribute that is an instance of a Java class corresponding to the appropriate reference type. For example, if there is a `PERSON` object with a `MANAGER REF` attribute, then the corresponding `Person` Java class will have a `ManagerRef` attribute.

SQLData Implementation

When you run JPublisher for a user-defined object type and choose the `SQLData` implementation for your custom object class, JPublisher will produce a custom object

class to act as a type definition to correspond to your Oracle object type. This class will include the following:

- Accessor methods for each attribute
- Implementations of the `readSQL()` and `writeSQL()` methods of the standard `SQLData` interface
- Wrapper methods that invoke the Oracle object methods executing in the server, unless you specify `-methods=false` when you run JPublisher

Because the `SQLData` interface is intended only for objects, however, and not for references or collections, JPublisher will not generate a custom reference class for references to the Oracle object type. You will have to use standard weakly typed `java.sql.Ref` instances or perhaps `oracle.sql.REF` instances, if you do not require portability. Note that `REF` instances, like custom reference class instances, have Oracle extension methods, `getValue()` and `setValue()`, to read or write instances of the referenced object. Standard `Ref` instances do not have this functionality.

Similarly, because you cannot use a `SQLData` implementation for a custom collection class, you must use standard weakly typed `java.sql.Array` instances or perhaps `oracle.sql.ARRAY` instances, if you do not require portability. `Array` and `ARRAY` instances, like custom collection class instances, have `getArray()` functionality to read the collection as a whole or in part, but do not have the element-level access and writability offered by the `getElement()` and `setElement()` methods of the custom collection class.

Note: The `SQLData` interface is defined in the JDBC specification to be portable. However, if you want the `SQLData` implementation produced by JPublisher to be portable, then you must avoid using any Oracle-specific features and Oracle type mapping, which uses the Oracle-specific `oracle.sql.*` classes.

Generating Custom Java Classes

This section discusses key JPublisher command-line functionality for specifying the user-defined types that you want to map to Java and for specifying object class names, collection class names, attribute type mappings, and wrapper methods. These key points can be summarized as follows:

- Specify the implementation to use, through the JPublisher `-usertypes` option.
- Specify user-defined types to map to Java. You can specify the custom object and custom collection class names for JPublisher to use, or you can accept the default names. Use the JPublisher `-sql`, `-user`, and `-case` options, as appropriate.
- Optionally specify attribute type mappings through the JPublisher `-`*`xxx`*`types` options: `-numbertypes`, `-builtintypes`, and `-lobtypes`.
- Choose whether or not JPublisher will create wrapper methods, in particular for Oracle object methods. Use the JPublisher `-methods` flag, which is enabled by default.

Note: Throughout the remainder of this section, discussion of custom reference classes or custom collection classes is simplified by referring only to `ORAData` implementations.

Choose the Implementation for Generated Classes

Before running JPublisher, consider whether you want the generated classes to implement the Oracle `ORAData` interface or the standard `SQLData` interface. Using `SQLData` will likely make your code more portable, but using `ORAData` offers a number of advantages, including no need for type maps.

Remember the following:

- You must use `ORAData` implementations for custom collection classes. The `SQLData` interface does not support collections.
- Strongly typed reference classes are always generated for `ORAData` custom object class implementations, but not for `SQLData` custom object class implementations. The SQLData interface does not support strongly typed object references. Use the weak `java.sql.Ref` type or `oracle.sql.REF` type instead.

Use the JPublisher `-usertypes` option to specify which interface you want your classes to implement. A setting of `-usertypes=oracle`, which is the default, specifies the `ORAData` interface, while a setting of `-usertypes=jdbc` specifies the `SQLData` interface.

Note: If you have a requirement to implement the `CustomDatum` interface, which was replaced by `ORAData` and deprecated in Oracle9*i* Database, then you can do so with a JPublisher `-compatible` setting of `customdatum`. This, combined with the `-usertypes=oracle` setting, results in generated classes implementing the `CustomDatum` interface. The default is `-compatible=oradata`.

The `-compatible=8i` or `-compatible=both8i` setting also directs JPublisher to use `CustomDatum`, as well as resulting in code generation that is backward compatible with Oracle8*i* versions of JPublisher. Refer to the *Oracle Database JPublisher User's Guide* for more information.

The following JPublisher command-line examples will result in implementation of `ORAData`, `CustomDatum`, and `SQLData`, respectively (assume `%` is a system prompt).

```
% jpub -usertypes=oracle ... <other option settings>

% jpub -usertypes=oracle -compatible=customdatum ... <other option settings>

% jpub -usertypes=jdbc ... <other option settings>
```

JPublisher will ignore a `-compatible=customdatum` or `-compatible=oradata` setting if `-usertypes=jdbc`.

Specify User-Defined Types to Map to Java

In using JPublisher to create custom Java classes, use the `-sql` option to specify the user-defined SQL types that you want to map to Java. You can either specify the custom object class names and custom collection class names, or you can accept the defaults.

The default names of your top-level custom classes, the classes that will correspond to the user-defined type names you specify to the `-sql` option, are identical to the user-defined type names as you enter them on the JPublisher command line. Because SQL names in the database are not case-sensitive by default, you can capitalize them to

ensure that your class names are capitalized according to Java convention. For example, if you want to generate a custom class for `employee` objects, you can run JPublisher as follows:

```
% jpub -sql=Employee ...
```

The default names of other classes, such as for the `home_address` objects that are attributes of `employee` objects, are determined by the JPublisher `-case` option. If you do not set the `-case` option, then it is set to `mixed`. This means that the default for the custom class name is to capitalize the initial character of the corresponding user-defined type name and the initial character of every word unit thereafter. JPublisher interprets underscores (_), dollar signs ($), and any characters that are illegal in Java identifiers as word-unit separators. These characters are discarded in the process.

For example, for Oracle object type `home_address`, JPublisher would create class `HomeAddress` in a `HomeAddress.sqlj` or `.java` source file.

Note:

- Only SQL names that are not case-sensitiveare supported on the JPublisher command line. If a user-defined type was defined in a case-sensitive way in SQL, then you must specify the name in the JPublisher `INPUT` file instead of on the command line and in quotes.
- For backward compatibility to previous versions of JPublisher, the `-types` option is still accepted as an alternative to `-sql`.

On the JPublisher command line, use the following syntax for the `-sql` option:

```
-sql=udt1<:mapclass1><,udt2<:mapclass2>>,...,<udtN<:mapclassN>> ...
```

Note that you can specify multiple actions in a single option setting.

Use the `-user` option to specify the database schema. Following is an example:

```
% jpub -sql=Myobj,mycoll:MyCollClass -user=scott/tiger
```

Note: Do *not* insert a space before or after the comma.

For the `MYOBJ` Oracle object, this command will name it as you typed it, creating `Myobj.sqlj` source to define a `Myobj` class. For the `MYCOLL` Oracle collection, this command will create source `MyCollClass.java` to define a `MyCollClass` class.

You can optionally specify schema names in the `-sql` option, such as in the following example that specifies the `scott` schema:

```
% jpub -sql=scott.Myobj,scott.mycoll:MyCollClass -user=scott/tiger
```

You cannot specify custom reference class names. JPublisher automatically derives them by adding `Ref` to custom object class names. This is relevant to `ORAData` implementations only. For example, if JPublisher produces the `Myobj.sqlj` Java source to define the `Myobj` custom object class, then it will also produce the `MyobjRef.java` Java source to define a `MyobjRef` custom reference class.

> **Note:** When specifying the schema, such as `scott` in the preceding example, this is not incorporated into the custom Java class name.

To create custom Java classes for the object and collection types defined in "User-Defined Types" on page 6-17, you can run JPublisher as follows:

```
% jpub -user=scott/tiger
-sql=Address,Person,Phone_array,Participant_t,Module_t,Moduletbl_t
```

Alternatively, to explicitly specify custom object class and custom collection class names, run it as follows:

```
% jpub -user=scott/tiger -sql=Address,Person,phone_array:PhoneArray,
participant_t:ParticipantT,module_t:ModuleT,moduletbl_t:ModuletblT
```

Note that each of the preceding two examples is a single wraparound command line.

The second example will produce the following Java source files: `Address.sqlj`, `AddressRef.java`, `Person.sqlj`, `PersonRef.java`, `PhoneArray.java`, `ParticipantT.sqlj`, `ParticipantTRef.java`, `ModuleT.sqlj`, `ModuleTRef.java`, and `ModuletblT.java`. Examples of some of these source files are provided in "JPublisher Custom Java Class Examples" on page 6-34.

So that it knows how to populate the custom Java classes, JPublisher connects to the specified schema to determine attributes of your specified object types or elements of your specified collection types.

> **Note:** As an alternative to specifying multiple mappings in a single `-sql` setting, you can use multiple `-sql` options in the same command line. The effect of multiple `-sql` options is cumulative.

If you want to change how JPublisher uses character case in default names for the methods and attributes that it generates, including lower-level custom Java class names for attributes that are objects or collections, then you can accomplish this using the `-case` option. There are four possible settings:

- `-case=mixed` (default)

  The following will be uppercase: the first character of every word unit of a class name, every word unit of an attribute name, and every word unit after the first word unit of a method name. All other characters are in lowercase. JPublisher interprets underscores (_), dollar signs ($), and any characters that are illegal in Java identifiers as word-unit separators. These characters are discarded in the process.

- `-case=same`

  Character case is unchanged from its representation in the database. Underscores and dollar signs are retained, and illegal characters are discarded.

- `-case=upper`

  Lowercase letters are converted to uppercase. Underscores and dollar signs are retained, and illegal characters are discarded.

- `-case=lower`

Uppercase letters are converted to lowercase. Underscores and dollar signs are retained, and illegal characters are discarded.

Note: If you run JPublisher without specifying the user-defined types to map to Java, it will process all user-defined types in the schema. Generated class names, for both your top-level custom classes and any other classes for object attributes or collection elements, will be based on the setting of the `-case` option.

Specify Type Mappings

JPublisher offers several choices for how to map user-defined types and their attribute and element types between SQL and Java.

JPublisher categorizes SQL types into the following groups, with corresponding JPublisher options as noted:

- Numeric types: Anything stored as SQL type `NUMBER`

  Use the JPublisher `-numbertypes` option to specify type-mapping for numeric types.

- Large object (LOB) types: SQL types `BLOB` and `CLOB`

  Use the JPublisher `-lobtypes` option to specify type-mapping for LOB types.

- Built-in types: Anything stored as a SQL type not covered by the preceding categories, for example, `CHAR`, `VARCHAR2`, `LONG`, and `RAW`

  Use the JPublisher `-builtintypes` option to specify type-mapping for built-in types.

JPublisher defines the following type-mapping modes:

- JDBC mapping (setting `jdbc`): Uses standard default mappings between SQL types and Java native types. This setting is valid for the `-numbertypes`, `-lobtypes`, and `-builtintypes` options.
- Oracle mapping (setting `oracle`): Uses corresponding `oracle.sql` types to map to SQL types. This setting is valid for the `-numbertypes`, `-lobtypes`, and `-builtintypes` options.
- Object-JDBC mapping (setting `objectjdbc`): This is an extension of JDBC mapping. Where relevant, object-JDBC mapping uses numeric object types from the standard `java.lang` package, such as `java.lang.Integer`, `Float`, and `Double`, instead of primitive Java types, such as `int`, `float`, and `double`. The `java.lang` types are nullable, but the primitive types are not. This setting is valid for the `-numbertypes` option only.
- `BigDecimal` mapping (setting `bigdecimal`): Uses `java.math.BigDecimal` to map to all numeric attributes. This is appropriate if you are dealing with large numbers, but do not want to map to the `oracle.sql.NUMBER` type. This setting is valid for the `-numbertypes` option only.

Note: Using `BigDecimal` mapping can significantly degrade performance.

If you do not specify mappings for the attribute types of a SQL object type or the element types of a SQL collection type, then JPublisher uses the following defaults:

- For numeric types, object-JDBC mapping is the default mapping.
- For LOB types, Oracle mapping is the default mapping.
- For built-in type types, JDBC mapping is the default mapping.

If you want alternate mappings, then use the `-numbertypes`, `-lobtypes`, and `-builtintypes` options as necessary, depending on the attribute types you have and the mappings you desire.

If an attribute type is itself a SQL object type, then it will be mapped according to the `-usertypes` setting.

> **Note:** If you specify a `SQLData` implementation for the custom object class and want the code to be portable, then you must use portable mappings for the attribute types. The defaults for numeric types and built-in types are portable, but for LOB types you must specify `-lobtypes=jdbc`.

Table 6–1 summarizes JPublisher categories for SQL types, the mapping settings relevant for each category, and the default settings.

Table 6–1 JPublisher SQL Type Categories, Supported Settings, and Defaults

SQL Type Category	JPublisher Mapping Option	Mapping Settings	Default
UDT types	`-usertypes`	`oracle`, `jdbc`	`oracle`
Numeric types	`-numbertypes`	`oracle`, `jdbc`, `objectjdbc`, `bigdecimal`	`objectjdbc`
LOB types	`-lobtypes`	`oracle`, `jdbc`	`oracle`
Built-in types	`-builtintypes`	`oracle`, `jdbc`	`jdbc`

> **Note:** The JPublisher `-mapping` option used in previous releases is deprecated but still supported. For information about how JPublisher converts `-mapping` option settings to settings for the new mapping options, refer to the *Oracle Database JPublisher User's Guide*.

Generate Wrapper Methods

In creating custom object classes to map Oracle objects to Java, the `-methods` option instructs JPublisher whether to include Java wrappers for Oracle object methods. The default `-methods=true` setting generates wrappers and also results in JPublisher generating a `.sqlj` file instead of a `.java` file for a custom object class, unless the underlying SQL object actually has no methods.

Wrapper methods generated by JPublisher are always instance methods, even when the original object methods are static. The following example shows how to set the `-methods` option:

```
% jpub -sql=Myobj,mycoll:MyCollClass -user=scott/tiger -methods=true
```

This will use default naming. The Java method names will be derived in the same fashion as custom Java class names, except that the initial character will be lowercase. For example, by default an object method name of `CALC_SAL` results in a Java

wrapper method of `calcSal()`. Alternatively, you can specify desired Java method names, but this requires use of a JPublisher `INPUT` file.

> **Note:** The `-methods` option has additional uses as well, such as for generating wrapper classes for packages or wrapper methods for package methods. This is beyond the scope of this manual. Refer to the *Oracle Database JPublisher User's Guide* for information.

Regarding Overloaded Methods

If you run JPublisher for an Oracle object that has an overloaded method where multiple signatures have the same corresponding Java signature, then JPublisher will generate a uniquely named method for each signature. It accomplishes this by appending *_n* to function names, where *n* is a number. This is to ensure that no two methods in the generated custom Java class have the same name and signature. For example, consider the SQL functions defined in creating a `MY_TYPE` object type:

```
CREATE OR REPLACE TYPE my_type AS OBJECT
(
   ...

   MEMBER FUNCTION myfunc(x INTEGER)
      RETURN my_return IS
      BEGIN
         ...
      END;

   MEMBER FUNCTION myfunc(y SMALLINT)
      RETURN my_return IS
      BEGIN
         ...
      END;
   ...
);
```

Without precaution, both definitions of `myfunc` result in the following name and signature in Java:

```
myfunc(Integer)
```

This is because both `INTEGER` and `SMALLINT` in SQL map to the Java `Integer` type.

Instead, JPublisher might call one `myfunc_1` and the other `myfunc_2`. The *_n* is unique for each. In simple cases it will likely be `_1`, `_2`, and so on, but it might sometimes be arbitrary, other than being unique for each.

> **Note:** How JPublisher handles overloaded wrapper methods applies to SQL functions created within an object or within a package, but not to top-level functions. Overloading is not allowed at the top level.

Generate Custom Java Classes and Map Alternate Classes

You can use JPublisher to generate a custom Java class but instruct it to map the object type or collection type to an alternative class instead of to the generated class.

A typical scenario is to treat JPublisher-generated classes as superclasses, extend them to add functionality, and map the object types to the subclasses. For example, presume

you have an Oracle object type ADDRESS and want to produce a custom Java class for it that has functionality beyond what is produced by JPublisher. You can use JPublisher to generate a JAddress custom Java class for extending it to produce a MyAddress class. Under this scenario you will add any special functionality to MyAddress and will want JPublisher to map ADDRESS objects to that class, not to the JAddress class. You will also want JPublisher to produce a reference class for MyAddress, not JAddress.

JPublisher has functionality to streamline the process of mapping to alternative classes. Use the following syntax in your -sql option setting:

```
-sql=object_type:generated_class:map_class
```

For the preceding example, use this setting:

```
-sql=ADDRESS:JAddress:MyAddress
```

This generates class JAddress in source file JAddress.sqlj (or possibly .java) but does the following:

- Maps the ADDRESS object type to the MyAddress class, not to the JAddress class. Therefore, if you retrieve an object from the database that has an ADDRESS attribute, then this attribute will be created as an instance of MyAddress in Java. Or, if you retrieve an ADDRESS object directly, then you will retrieve it into a MyAddress instance.
- Creates a MyAddressRef class in MyAddressRef.java, instead of creating a JAddressRef class.
- Creates an initial version of the MyAddress class in a MyAddress.sqlj source file (or possibly MyAddress.java), unless the file already exists, in which case it is not changed.

MyAddress subclasses JAddress. In order to implement the extended functionality for MyAddress, you can start with the JPublisher-generated MyAddress source file, editing it as desired.

For further discussion about extending JPublisher-generated classes (continuing the preceding example), refer to "Extending Classes Generated by JPublisher" on page 6-37.

JPublisher INPUT Files and Properties Files

JPublisher supports the use of special INPUT files and standard properties files to specify type mappings and additional option settings.

Using JPublisher INPUT Files

You can use the JPublisher -input command-line option to specify an INPUT file for JPublisher to use for additional type mappings. SQL in an INPUT file is equivalent to -sql on the command line, and the AS or GENERATE...AS syntax is equivalent to the command-line colon syntax. Use the following syntax, specifying just one mapping per SQL command:

```
SQL udt1 <GENERATE GeneratedClass1> <AS MapClass1>
SQL udt2 <GENERATE GeneratedClass2> <AS MapClass2>
...
```

This generates *GeneratedClass1* and *GeneratedClass2*, but maps *udt1* to *MapClass1* and *udt2* to *MapClass2*.

Note: If a user-defined type was defined in a case-sensitive way in SQL, then you must specify the name in quotes. For example:

```
SQL "CaseSenstiveType" AS CaseSensitiveType
```

If you are also specifying a schema name that is *not* case-sensitive:

```
SQL SCOTT."CaseSensitiveType" AS CaseSensitiveType
```

Alternatively, to also specify a case-sensitive schema name:

```
SQL "Scott"."CaseSensitiveType" AS CaseSensitiveType
```

The `AS` clauses are optional.

Avoid using a period (.) as part of the schema name or type name itself.

INPUT File Example

In the following example, JPublisher will pick up the `-user` option from the command line and go to `INPUT` file `myinput.in` for type mappings.

Command line:

```
% jpub -input=myinput.in -user=scott/tiger
```

Contents of `INPUT` file `myinput.in`:

```
SQL Myobj
SQL mycoll AS MyCollClass
SQL employee GENERATE Employee AS MyEmployee
```

This accomplishes the following:

- User-defined type `MYOBJ` gets the custom object class name `Myobj` because that is how you typed it. JPublisher creates source `Myobj.sqlj` (or possibly `Myobj.java`, if `Myobj` has no methods) and `MyobjRef.java`.
- User-defined type `MYCOLL` is mapped to `MyCollClass`. JPublisher creates a `MyCollClass.java` source file.
- User-defined type `EMPLOYEE` is mapped to the `MyEmployee` class. JPublisher creates source `Employee.sqlj` (or possibly `Employee.java`) and `MyEmployeeRef.java`, as well as an initial version of `MyEmployee.sqlj` (or `.java`) unless the file already exists. If you retrieve an object from the database that has an `EMPLOYEE` attribute, then this attribute would be created as an instance of `MyEmployee` in Java. Or, if you retrieve an `EMPLOYEE` object directly, presumably you will retrieve it into a `MyEmployee` instance. You are responsible for the `MyEmployee` code, but for convenience you can start with the JPublisher-generated `MyEmployee` source file and edit it to implement your specialized functionality for `EMPLOYEE` objects in Java. `MyEmployee` subclasses the `Employee` class.

Using JPublisher Properties Files

You can use the JPublisher `-props` command-line option to specify a properties file for JPublisher to use for additional type mappings and other option settings.

In a properties file, `jpub.` (including the period) is equivalent to the command-line "`-`" (single-dash), and other syntax remains the same. Specify only one option per line.

For type mappings, for example, `jpub.sql` is equivalent to `-sql`. You can specify multiple mappings in a single `jpub.sql` setting. Alternatively, you can use multiple `jpub.sql` options. The effect would be cumulative, as for multiple `-sql` options on the command line.

> **Note:** The behavior of properties files is to ignore any line that does not begin with `jpub.` or `--jpub.` (two dashes followed by `jpub.`). This enables you to use the same file as both a SQL script to create the types and a properties file for JPublisher. If you start each JPublisher statement with "`--`", which indicates a SQL comment, it will be ignored by SQL*Plus. And SQL statements will be ignored by JPublisher.

Properties File Example

In the following example, JPublisher will pick up the `-user` option from the command line and go to the `jpub.properties` properties file for type mappings and the attribute-mapping option.

Command line:

```
% jpub -props=jpub.properties -user=scott/tiger
```

Contents of properties file `jpub.properties`:

```
jpub.sql=Myobj,mycoll:MyCollClass,employee:Employee:MyEmployee
jpub.usertypes=oracle
```

This produces the same results as the preceding input-file example, explicitly specifying the `oracle` mapping setting.

> **Note:** Unlike SQLJ, JPublisher has no default properties file. To use a properties file, you must use the `-props` option.

Creating Custom Java Classes and Specifying Member Names

In generating custom Java classes, you can specify the names of any attributes or methods of the custom class. However, this cannot be specified on the JPublisher command line. You must specify it in a JPublisher `INPUT` file using `TRANSLATE` syntax, as follows:

```
SQL udt <GENERATE GeneratedClass> <AS MapClass> <TRANSLATE membername1 AS
Javaname1> <, membername2 AS Javaname2> ...
```

`TRANSLATE` pairs (*membernameN* `AS` *JavanameN*) are separated by commas.

For example, presume the `EMPLOYEE` Oracle object type has an `ADDRESS` attribute that you want to call `HomeAddress`, and a `GIVE_RAISE` method that you want to call `giveRaise()`. Also presume that you want to generate an `Employee` class but map `EMPLOYEE` objects to a `MyEmployee` class that you will create. (This is not related to specifying member names, but provides a full example of `INPUT` file syntax.)

```
SQL employee GENERATE Employee AS MyEmployee
TRANSLATE address AS HomeAddress, GIVE_RAISE AS giveRaise
```

Note:

- When you specify member names, any member you do not specify will be given the default naming.
- The reason to capitalize the specified attribute, `HomeAddress` instead of `homeAddress`, is that it will be used exactly as specified to name the accessor methods. For example, `getHomeAddress()` follows naming conventions, but `gethomeAddress()` does not.

JPublisher Implementation of Wrapper Methods

This section describes how JPublisher generates wrapper methods and how wrapper method calls are processed at run time.

Generation of Wrapper Methods

The following points describe how JPublisher generates wrapper methods:

- JPublisher-generated wrapper methods are implemented in SQLJ. Therefore, whenever `-methods=true`, the custom object class will be defined in a `.sqlj` file instead of in a `.java` file, assuming the object type defines methods. Run SQLJ to translate the `.sqlj` file.

  Note: Even if the object type does not define methods, you can ensure that a `.sqlj` file is generated by setting `-methods=always`. Refer to the *Oracle Database JPublisher User's Guide* for more information.

- All wrapper methods generated by JPublisher are implemented as instance methods. This is because a database connection is required for you to invoke the corresponding server method. Each instance of a JPublisher-generated custom Java class has a connection associated with it.

Run Time Execution of Wrapper Method Calls

The following points describe what JPublisher-generated Java wrapper methods execute at run time. In this discussion, "Java wrapper method" refers to a method in the custom Java object, while "wrapped SQL method" refers to the SQL object method that is wrapped by the Java wrapper method.

- The custom Java object is converted to a SQL object and passed to the database, where the wrapped SQL method is invoked. After this method invocation, the new value of the SQL object is returned to Java in a new custom Java object, either as a function return from the wrapped SQL method, if the SQL method is a stored procedure, or as an array element in an additional output parameter, if the SQL method is a stored function and there already is a function return.
- Any output or input-output parameter is passed as the element of a one-element array. If the parameter is input-output, then the wrapper method takes the array element as input. After processing, the wrapper assigns the output to the array element.

JPublisher Custom Java Class Examples

This section provides examples of JPublisher-generated `ORAData` implementations for the following user-defined types:

- A custom object class (`Address`, corresponding to the Oracle object type `ADDRESS`) and related custom reference class (`AddressRef`)
- A custom collection class (`ModuletblT`, corresponding to the Oracle collection type `MODULETBL_T`)

Assume that the `-methods` option has its default `true` setting and that the `ADDRESS` type has methods, so that a `.sqlj` file is generated for the `Address` class.

> **See Also:** *Oracle Database JPublisher User's Guide* for examples of JPublisher-generated `SQLData` implementations, as well as further examples of JPublisher-generated `ORAData` implementations.

Custom Object Class: Address.sqlj

Following is an example of the source code that JPublisher generates for a custom object class. Implementation details have been omitted.

In this example, unlike in "Creating Object Types" on page 6-17, assume the Oracle object `ADDRESS` has only the `street` and `zip_code` attributes.

```
package bar;

import java.sql.SQLException;
import java.sql.Connection;
import oracle.jdbc.OracleTypes;
import oracle.sql.ORAData;
import oracle.sql.ORADataFactory;
import oracle.sql.Datum;
import oracle.sql.STRUCT;
import oracle.jpub.MutableStruct;

public class Address implements ORAData, ORADataFactory
{
  public static final String _SQL_NAME = "SCOTT.ADDRESS";
  public static final int _SQL_TYPECODE = OracleTypes.STRUCT;

  public static ORADataFactory getORADataFactory()
  { ... }

  /* constructors */
  public Address()
  { ... }

  public Address(String street, java.math.BigDecimal zip_code)
                throws SQLException
  { ... }

  /* ORAData interface */
  public Datum toDatum(Connection c) throws SQLException
  { ... }

  /* ORADataFactory interface */
  public ORAData create(Datum d, int sqlType) throws SQLException
  { ... }

  /* accessor methods */
```

```
  public String getStreet() throws SQLException
  { ... }

  public void setStreet(String street) throws SQLException
  { ... }


  public java.math.BigDecimal getZipCode() throws SQLException
  { ... }

  public void setZipCode(java.math.BigDecimal zip_code) throws SQLException
  { ... }

}
```

Custom Reference Class: AddressRef.java

Following is an example of the source code that JPublisher generates for a custom reference class to be used for references to `ADDRESS` objects. Implementation details have been omitted.

```
package bar;

import java.sql.SQLException;
import java.sql.Connection;
import oracle.jdbc.OracleTypes;
import oracle.sql.ORAData;
import oracle.sql.ORADataFactory;
import oracle.sql.Datum;
import oracle.sql.REF;
import oracle.sql.STRUCT;

public class AddressRef implements ORAData, ORADataFactory
{
  public static final String _SQL_BASETYPE = "SCOTT.ADDRESS";
  public static final int _SQL_TYPECODE = OracleTypes.REF;

  public static ORADataFactory getORADataFactory()
  { ... }

  /* constructors */
  public AddressRef()
  { ... }

  public static AddressRef(ORAData o) throws SQLException
  { ... }

  /* ORAData interface */
  public Datum toDatum(Connection c) throws SQLException
  { ... }

  /* ORADataFactory interface */
  public ORAData create(Datum d, int sqlType) throws SQLException
  { ... }

  public static AddressRef cast(ORAData o) throws SQLException
  { ... }

  public Address getValue() throws SQLException
  { ... }
```

```
  public void setValue(Address c) throws SQLException
  { ... }
}
```

Custom Collection Class: ModuletblT.java

Following is an example of the source code that JPublisher generates for a custom collection class. Implementation details have been omitted.

```
import java.sql.SQLException;
import java.sql.Connection;
import oracle.jdbc.OracleTypes;
import oracle.sql.ORAData;
import oracle.sql.ORADataFactory;
import oracle.sql.Datum;
import oracle.sql.ARRAY;
import oracle.sql.ArrayDescriptor;
import oracle.jpub.runtime.MutableArray;

public class ModuletblT implements ORAData, ORADataFactory
{
  public static final String _SQL_NAME = "SCOTT.MODULETBL_T";
  public static final int _SQL_TYPECODE = OracleTypes.ARRAY;

  public static ORADataFactory getORADataFactory()
  { ... }

  /* constructors */
  public ModuletblT()
  { ... }

  public ModuletblT(ModuleT[] a)
  { ... }

  /* ORAData interface */
  public Datum toDatum(Connection c) throws SQLException
  { ... }

  /* ORADataFactory interface */
  public ORAData create(Datum d, int sqlType) throws SQLException
  { ... }

  public String getBaseTypeName() throws SQLException
  { ... }

  public int getBaseType() throws SQLException
  { ... }

  public ArrayDescriptor getDescriptor() throws SQLException
  { ... }

  /* array accessor methods */
  public ModuleT[] getArray() throws SQLException
  { ... }

  public void setArray(ModuleT[] a) throws SQLException
  { ... }

  public ModuleT[] getArray(long index, int count) throws SQLException
```

```
  { ... }

  public void setArray(ModuleT[] a, long index) throws SQLException
  { ... }

  public ModuleT getObjectElement(long index) throws SQLException
  { ... }

  public void setElement(ModuleT a, long index) throws SQLException
  { ... }
}
```

Extending Classes Generated by JPublisher

You might want to enhance the functionality of a custom Java class generated by JPublisher by adding methods and transient fields. You can accomplish this by extending the JPublisher-generated class.

For example, suppose you want JPublisher to generate the `JAddress` class from the `ADDRESS` SQL object type. You also want to use a `MyAddress` class to represent `ADDRESS` objects and implement special functionality. The `MyAddress` class must extend `JAddress`.

Another way to enhance the functionality of a JPublisher-generated class is to simply add methods to it. However, adding methods to the generated class is not recommended if you anticipate running JPublisher at some future time to regenerate the class. If you run JPublisher to regenerate a class that you have modified in this way, then you would have to save a copy and manually merge your changes back in.

JPublisher Functionality for Extending Generated Classes

The syntax to have JPublisher generate `JAddress` but map to `MyAddress` is as follows:

```
-sql=ADDRESS:JAddress:MyAddress
```

Or, use the following in an `INPUT` file:

```
SQL ADDRESS GENERATE JAddress AS MyAddress
```

As a result of this, JPublisher will generate the `MyAddressRef` reference class, rather than `JAddressRef`, in `MyAddressRef.java`.

In addition, JPublisher alters the code it generates to implement the following functionality:

- The `MyAddress` class, instead of the `JAddress` class, is used to represent attributes whose SQL type is `ADDRESS`.
- The `MyAddress` class, instead of the `JAddress` class, is used to represent method arguments and function results whose type is `ADDRESS`.
- The `MyAddress` factory, instead of the `JAddress` factory, is used to construct Java objects whose SQL type is `ADDRESS`.

You would presumably use `MyAddress` similarly in any additional code that you write.

At run time, the Oracle JDBC driver will map any occurrences of `ADDRESS` data in the database to `MyAddress` instances, instead of to `JAddress` instances.

Requirements of Extended Classes

By default, JPublisher will create an initial version of the `MyAddress` user subclass in `MyAddress.sqlj`, if the original class uses methods and you are publishing these methods, or `MyAddress.java`, unless the file to be created already exists, in which case it will not be changed. You can edit this file as necessary to add your desired functionality.

`MyAddress` must have a no-argument constructor. The easiest way to construct a properly initialized object is to invoke the constructor of the superclass, either explicitly or implicitly.

As a result of extending the JPublisher-generated class, the subclass will inherit definitions of the `_SQL_NAME` field, which it requires, and the `_SQL_TYPECODE` field.

In addition, one of the following will be true.

- If the JPublisher-generated class implements the `ORAData` and `ORADataFactory` interfaces, then the subclass will inherit this implementation and the necessary `toDatum()` and `create()` functionality of the generated class. The subclass implements a `getORADataFactory()` method that returns an instance of your map class, such as a `MyAddress` object.
- If the JPublisher-generated class implements the `SQLData` interface, then the subclass will inherit this implementation and the necessary `readSQL()` and `writeSQL()` functionality of the generated class.

JPublisher-Generated Custom Object Class: JAddress.sqlj

The code for the JPublisher-generated `JAddress` class, implementing `ORAData` and `ORADataFactory`, is mostly identical to the code shown previously for the `Address` class, with the exception that mentions of `Address` are replaced by mentions of `JAddress`.

JPublisher-Generated Alternate Reference Class: MyAddressRef.java

Continuing the example in the preceding sections, consider code for the JPublisher-generated reference class, `MyAddressRef` (as opposed to `JAddressRef`, because `MyAddress` is the class that `ADDRESS` objects map to). This class also implements `ORAData` and `ORADataFactory`. The implementation is nearly identical to that of `AddressRef.java`, except for the change in class name and the fact that accessor methods use `MyAddress` instances instead of `Address` instances.

Extended Custom Object Class: MyAddress.sqlj

Again continuing the example, here is sample code for a `MyAddress` class that subclasses the JPublisher-generated `JAddress` class. The comments in the code show what is inherited from `JAddress`. Implementation details have been omitted.

```
import java.sql.SQLException;
import oracle.sql.ORAData;
import oracle.sql.ORADataFactory;
import oracle.sql.Datum;
import oracle.sql.STRUCT;
import oracle.jpub.runtime.MutableStruct;

public class MyAddress extends JAddress
{
  /* _SQL_NAME inherited from MyAddress */
  /* _SQL_TYPECODE inherited from MyAddress */

  static _myAddressFactory = new MyAddress();
```

```
  public static ORADataFactory getORADataFactory()
  {
    return _myAddressFactory;
  }

  /* constructor */
  public MyAddress()
  { super(); }

  /* ORAData interface */
  /* toDatum() inherited from JAddress */

  /* ORADataFactory interface */
  public ORAData create(oracle.sql.Datum d, int sqlType) throws SQLException
  { ... }

  /* accessor methods inherited from JAddress */

  /* Additional methods go here.  These additional methods (not shown)
     are the reason that JAddress was extended.
  */
}
```

Strongly Typed Objects and References in SQLJ Executable Statements

The Oracle SQLJ implementation is flexible in how it enables you to use host expressions and iterators in reading or writing object data through strongly typed objects or references.

For iterators, you can use custom object classes as iterator column types. Alternatively, you can have iterator columns that correspond to individual object attributes, similar to extent tables, using column types that appropriately map to the SQL data types of the attributes.

For host expressions, you can use host variables of your custom object class type or custom reference class type. Alternatively, you can use host variables that correspond to object attributes, using variable types that appropriately map to the SQL data types of the attributes.

The remainder of this section provides examples of how to manipulate Oracle objects using custom object classes, custom object class attributes, and custom reference classes for host variables and iterator columns in SQLJ executable statements.

The following two examples operate at the object level:

- Selecting Objects and Object References into Iterator Columns
- Updating an Object

The Inserting an Object Created from Individual Object Attributes example operates at the scalar-attribute level.

The Updating an Object Reference example operates through a reference.

Refer to the Oracle object types `ADDRESS` and `PERSON` in "Creating Object Types" on page 6-17.

Selecting Objects and Object References into Iterator Columns

This example uses a custom Java class and a custom reference class as iterator column types. Presume the following definition of the ADDRESS Oracle object type:

```
CREATE TYPE ADDRESS AS OBJECT
(  street VARCHAR(40),
   zip NUMBER );
```

And the following definition of the EMPADDRS table, which includes an ADDRESS column and an ADDRESS reference column:

```
CREATE TABLE empaddrs
(  name VARCHAR(60),
   home ADDRESS,
   loc REF ADDRESS );
```

Once you use JPublisher or otherwise create a custom Java class, Address, and custom reference class, AddressRef, corresponding to the ADDRESS Oracle object type, you can use Address and AddressRef in a named iterator as follows:

```
#sql iterator EmpIter (String name, Address home, AddressRef loc);

...
EmpIter ecur;
#sql ecur = { SELECT name, home, loc FROM empaddrs };
while (ecur.next()) {
   Address homeAddr = ecur.home();
   // Print out the home address.
   System.out.println ("Name: " + ecur.name() + "\n" +
                       "Home address: " + homeAddr.getStreet() + "    " +
                       homeAddr.getZip());
   // Now update the loc address zip code through the address reference.
   AddressRef homeRef = ecur.loc();
   Address location = homeRef.getValue();
   location.setZip(new BigDecimal(98765));
   homeRef.setValue(location);
   }
...
```

The ecur.home() method call extracts an Address object from the home column of the iterator and assigns it to the homeAddr local variable (for efficiency). The attributes of that object can then be accessed using standard Java dot syntax:

```
homeAddr.getStreet()
```

Use the getValue() and setValue() methods, standard with any JPublisher-generated custom reference class, to manipulate the location address (in this case its zip code).

> **Note:** The remaining examples in this section use the types and tables defined in the SQL script in "Creating Object Types" on page 6-17.

Updating an Object

This example declares and sets an input host variable of the Address Java type to update an ADDRESS object in a column of the employees table. Both before and after

the update, the address is selected into an output host variable of the `Address` type and printed for verification.

```
...
// Updating an object

static void updateObject()
{

   Address addr;
   Address new_addr;
   int empnum = 1001;

   try {
      #sql {
         SELECT office_addr
         INTO :addr
         FROM employees
         WHERE empnumber = :empnum };
      System.out.println("Current office address of employee 1001:");

      printAddressDetails(addr);

      /* Now update the street of address */

      String street ="100 Oracle Parkway";
      addr.setStreet(street);

      /* Put updated object back into the database */

      try {
         #sql {
            UPDATE employees
            SET office_addr = :addr
            WHERE empnumber = :empnum };
         System.out.println
            ("Updated employee 1001 to new address at Oracle Parkway.");

         /* Select new address to verify update */

         try {
            #sql {
               SELECT office_addr
               INTO :new_addr
               FROM employees
               WHERE empnumber = :empnum };

            System.out.println("New office address of employee 1001:");
            printAddressDetails(new_addr);

         } catch (SQLException exn) {
         System.out.println("Verification SELECT failed with "+exn); }

      } catch (SQLException exn) {
      System.out.println("UPDATE failed with "+exn); }

   } catch (SQLException exn) {
   System.out.println("SELECT failed with "+exn); }
}
...
```

Note the use of the `setStreet()` accessor method of the `Address` object. Remember that JPublisher provides such accessor methods for all attributes in any custom Java class that it produces.

This example uses the `printAddressDetails()` utility. The source code for this method is as follows:

```
static void printAddressDetails(Address a) throws SQLException
{

  if (a == null)  {
    System.out.println("No Address available.");
    return;
   }

   String street = ((a.getStreet()==null) ? "NULL street" : a.getStreet()) ;
   String city = (a.getCity()==null) ? "NULL city" : a.getCity();
   String state = (a.getState()==null) ? "NULL state" : a.getState();
   String zip_code = (a.getZipCode()==null) ? "NULL zip" : a.getZipCode();

   System.out.println("Street: '" + street + "'");
   System.out.println("City:   '" + city   + "'");
   System.out.println("State:  '" + state  + "'");
   System.out.println("Zip:    '" + zip_code + "'" );
}
```

Inserting an Object Created from Individual Object Attributes

This example declares and sets input host variables corresponding to attributes of `PERSON` and nested `ADDRESS` objects, then uses these values to insert a new `PERSON` object into the `persons` table in the database.

```
...
// Inserting an object

static void insertObject()
{
   String new_name   = "NEW PERSON";
   int    new_ssn    = 987654;
   String new_street = "NEW STREET";
   String new_city   = "NEW CITY";
   String new_state  = "NS";
   String new_zip    = "NZIP";
  /*
   * Insert a new PERSON object into the persons table
   */
   try {
      #sql {
         INSERT INTO persons
         VALUES (PERSON(:new_name, :new_ssn,
         ADDRESS(:new_street, :new_city, :new_state, :new_zip))) };

      System.out.println("Inserted PERSON object NEW PERSON.");

   } catch (SQLException exn) { System.out.println("INSERT failed with "+exn); }
}
...
```

Updating an Object Reference

This example selects a PERSON reference from the persons table and uses it to update a PERSON reference in the employees table. It uses simple input host variables to check attribute value criteria. The newly updated reference is then used in selecting the PERSON object to which it refers, so that information can be output to the user to verify the change.

```
...
// Updating a REF to an object

static void updateRef()
{
   int empnum = 1001;
   String new_manager = "NEW PERSON";

   System.out.println("Updating manager REF.");
   try {
      #sql {
         UPDATE employees
         SET manager =
            (SELECT REF(p) FROM persons p WHERE p.name = :new_manager)
         WHERE empnumber = :empnum };

      System.out.println("Updated manager of employee 1001. Selecting back");

   } catch (SQLException exn) {
   System.out.println("UPDATE REF failed with "+exn); }

   /* Select manager back to verify the update */
   Person manager;

   try {
      #sql {
         SELECT deref(manager)
         INTO :manager
         FROM employees e
         WHERE empnumber = :empnum };

      System.out.println("Current manager of "+empnum+":");
      printPersonDetails(manager);

   } catch (SQLException exn) {
   System.out.println("SELECT REF failed with "+exn); }

}
...
```

Note: This example uses table alias syntax (p) as discussed previously. Also, the REF syntax is required in selecting a reference through the object to which it refers, and the DEREF syntax is required in selecting an object through a reference. Refer to the *Oracle Database SQL Language Reference* for more information about table aliases, REF, and DEREF.

Strongly Typed Collections in SQLJ Executable Statements

As with strongly typed objects and references, the Oracle SQLJ implementation supports different scenarios for reading and writing data through strongly typed collections, using either iterators or host expressions.

From the perspective of a SQLJ developer, both categories of collections, VARRAY and nested table, are treated essentially the same, but there are some differences in implementation and performance.

The Oracle SQLJ implementation supports syntax choices so that nested tables can be accessed and manipulated either apart from or together with their outer tables. In this section, manipulation of a nested table by itself will be referred to as detail-level manipulation and manipulation of a nested table together with its outer table will be referred to as master-level manipulation.

Most of this section, after a brief discussion of some syntax, focuses on examples of manipulating nested tables, given that their use is somewhat more complicated than that of VARRAYs.

Refer to the `MODULETBL_T` Oracle collection type and related tables and object types defined in "Creating Collection Types" on page 6-18.

> **Note:** In the Oracle SQLJ implementation, VARRAY types and nested table types can be retrieved only in their entirety. This is as opposed to the Oracle SQL implementation, where nested tables can be selectively queried.

This section covers the following topics:

- Accessing Nested Tables: TABLE syntax and CURSOR syntax
- Inserting a Row that Includes a Nested Table
- Selecting a Nested Table into a Host Expression
- Manipulating a Nested Table Using TABLE Syntax
- Selecting Data from a Nested Table Using a Nested Iterator
- Selecting a VARRAY into a Host Expression
- Inserting a Row that Includes a VARRAY

Accessing Nested Tables: TABLE syntax and CURSOR syntax

The Oracle SQLJ implementation supports the use of nested iterators to access data in nested tables. Use the `CURSOR` keyword in the outer `SELECT` statement to encapsulate the inner `SELECT` statement. This is shown in "Selecting Data from a Nested Table Using a Nested Iterator" on page 6-47.

Oracle also supports use of the `TABLE` keyword to manipulate the individual rows of a nested table. This keyword informs Oracle that the column value returned by a subquery is a nested table, as opposed to a scalar value. You must prefix the `TABLE` keyword to a subquery that returns a single column value or an expression that yields a nested table.

The following example shows the use of the `TABLE` syntax:

```
UPDATE TABLE(SELECT a.modules FROM projects a WHERE a.id=555) b
       SET module_owner=
```

```
    (SELECT ref(p) FROM employees p WHERE p.ename= 'Smith')
    WHERE b.module_name = 'Zebra';
```

When you see `TABLE` used as it is here, realize that it is referring to a single nested table that has been selected from a column of an outer table.

> **Note:** This example uses table alias syntax (`a` for `projects`, `b` for the nested table, and `p` for `employees`) as discussed previously.

Inserting a Row that Includes a Nested Table

This example shows an operation that manipulates the master level (outer table) and detail level (nested tables) simultaneously and explicitly. This inserts a row in the `projects` table, where each row includes a nested table of the `MODULETBL_T` type, which contains rows of `MODULE_T` objects.

First, the scalar values are set (`id`, `name`, `start_date`, `duration`), then the nested table values are set. This involves an extra level of abstraction, because the nested table elements are objects with multiple attributes. In setting the nested table values, each attribute value must be set for each `MODULE_T` object in the nested table. Finally, the `owner` values, initially set to `null`, are set in a separate statement.

```
// Insert Nested table details along with master details

  public static void insertProject2(int id)  throws Exception
  {
    System.out.println("Inserting Project with Nested Table details..");
    try {
      #sql { INSERT INTO Projects(id,name,owner,start_date,duration, modules)
             VALUES ( 600, 'Ruby', null, '10-MAY-98',  300,
             moduletbl_t(module_t(6001, 'Setup ', null, '01-JAN-98', 100),
                         module_t(6002, 'BenchMark', null, '05-FEB-98',20) ,
                         module_t(6003, 'Purchase', null, '15-MAR-98', 50),
                         module_t(6004, 'Install', null, '15-MAR-98',44),
                         module_t(6005, 'Launch', null,'12-MAY-98',34))) };
    } catch ( Exception e) {
      System.out.println("Error:insertProject2");
      e.printStackTrace();
    }

    // Assign project owner to this project

    try {
      #sql { UPDATE Projects pr
          SET owner=(SELECT ref(pa) FROM participants pa WHERE pa.empno = 7698)
         WHERE pr.id=600 };
    } catch ( Exception e) {
      System.out.println("Error:insertProject2:update");
      e.printStackTrace();
    }
  }
```

Selecting a Nested Table into a Host Expression

This example presents an operation that works directly at the detail level of the nested table. Recall that `ModuletblT` is a JPublisher-generated custom collection class (`ORAData` implementation) for `MODULETBL_T` nested tables, `ModuleT` is a

JPublisher-generated custom object class for MODULE_T objects, and MODULETBL_T nested tables contain MODULE_T objects.

A nested table of MODULE_T objects is selected from the modules column of the projects table into a ModuletblT host variable.

Following that, the ModuletblT variable (containing the nested table) is passed to a method that accesses its elements through its getArray() method, writing the data to a ModuleT[] array. All custom collection classes generated by JPublisher include a getArray() method. Then each element is copied from the ModuleT[] array into a ModuleT object, and individual attributes are retrieved through accessor methods (getModuleName(), for example) and then printed. All JPublisher-generated custom object classes include such accessor methods.

```
static ModuletblT mymodules=null;
...

public static void getModules2(int projId)
throws Exception
{
  System.out.println("Display modules for project " + projId );

  try {
    #sql {SELECT modules INTO :mymodules
                        FROM projects  WHERE id=:projId };
    showArray(mymodules);
  } catch(Exception e) {
    System.out.println("Error:getModules2");
    e.printStackTrace();
  }
}

public static void showArray(ModuletblT a)
{
  try {
    if ( a == null )
      System.out.println( "The array is null" );
    else {
      System.out.println( "printing ModuleTable array object of size "
                          +a.length());
      ModuleT[] modules = a.getArray();

      for (int i=0;i<modules.length; i++) {
        ModuleT module = modules[i];
        System.out.println("module "+module.getModuleId()+
              ", "+module.getModuleName()+
              ", "+module.getModuleStartDate()+
              ", "+module.getModuleDuration());
      }
    }
  }
  catch( Exception e ) {
    System.out.println("Show Array");
    e.printStackTrace();
  }
}
```

Manipulating a Nested Table Using TABLE Syntax

This example uses `TABLE` syntax to work at the detail level to access and update nested table elements directly, based on master-level criteria.

The `assignModule()` method selects a nested table of `MODULE_T` objects from the `MODULES` column of the `PROJECTS` table, then updates `MODULE_NAME` for a particular row of the nested table. Similarly, the `deleteUnownedModules()` method selects a nested table of `MODULE_T` objects, then deletes any unowned modules in the nested table, where `MODULE_OWNER` is `null`.

These methods use table alias syntax, as discussed previously. In this case, `m` is used for the nested table, and `p` is used for the `participants` table.

```
/* assignModule
   Illustrates accessing the nested table using the TABLE construct
   and updating the nested table row
*/
public static void assignModule(int projId, String moduleName,
                                 String modOwner) throws Exception
{
  System.out.println("Update:Assign '"+moduleName+"' to '"+ modOwner+"'");

  try {
    #sql {UPDATE TABLE(SELECT modules FROM projects WHERE id=:projId) m
          SET m.module_owner=
         (SELECT ref(p) FROM participants p WHERE p.ename= :modOwner)
          WHERE m.module_name = :moduleName };
  } catch(Exception e) {
    System.out.println("Error:insertModules");
    e.printStackTrace();
  }
}

/* deleteUnownedModules
// Demonstrates deletion of the Nested table element
*/

public static void deleteUnownedModules(int projId)
throws Exception
{
  System.out.println("Deleting Unowned Modules for Project " + projId);
  try {
    #sql { DELETE TABLE(SELECT modules FROM projects WHERE id=:projId) m
           WHERE m.module_owner IS NULL };
  } catch(Exception e) {
    System.out.println("Error:deleteUnownedModules");
    e.printStackTrace();
  }
}
```

Selecting Data from a Nested Table Using a Nested Iterator

SQLJ supports the use of nested iterators as a way of accessing nested tables. This requires `CURSOR` syntax, as used in the following example. The code defines a named iterator class, `ModuleIter`, then uses that class as the type for a `modules` column in another named iterator class, `ProjIter`. Inside a populated `ProjIter` instance, each `modules` item is a nested table rendered as a nested iterator.

The CURSOR syntax is part of the nested SELECT statement that populates the nested iterators. Once the data has been selected, it is output to the user through the iterator accessor methods.

This example uses required table alias syntax, as discussed previously. In this case, a for the projects table and b for the nested table.

```
...

//  The Nested Table is accessed using the ModuleIter
//  The ModuleIter is defined as Named Iterator

#sql public static iterator ModuleIter(int moduleId ,
                                       String moduleName ,
                                       String moduleOwner);

// Get the Project Details using the ProjIter defined as
// Named Iterator. Notice the use of ModuleIter:

#sql public static iterator ProjIter(int id,
                                     String name,
                                     String owner,
                                     Date start_date,
                                     ModuleIter modules);

...

public static void listAllProjects() throws SQLException
{
  System.out.println("Listing projects...");

   // Instantiate and initialize the iterators

   ProjIter projs = null;
   ModuleIter  mods = null;
   #sql projs = {SELECT a.id,
                        a.name,
                        initcap(a.owner.ename) as "owner",
                        a.start_date,
                        CURSOR (
                        SELECT b.module_id AS "moduleId",
                               b.module_name AS "moduleName",
                                 initcap(b.module_owner.ename) AS "moduleOwner"
                        FROM TABLE(a.modules) b) AS "modules"
                 FROM projects a };

  // Display Project Details

  while (projs.next()) {
    System.out.println( "\n'" + projs.name() + "' Project Id:"
                + projs.id() + " is owned by " +"'"+ projs.owner() +"'"
                + " start on "
                + projs.start_date());

    // Notice the modules from the ProjIter are assigned to the module
    // iterator variable

    mods = projs.modules();
    System.out.println ("Modules in this Project are : ");

    // Display Module details
```

```
    while(mods.next()) {
      System.out.println ("  "+ mods.moduleId() + " '"+
                                mods.moduleName() + "' owner is '" +
                                mods.moduleOwner()+"'" );
    }                    // end of modules
    mods.close();
  }                      // end of projects
  projs.close();
}
```

Selecting a VARRAY into a Host Expression

This section provides an example of selecting a VARRAY into a host expression. Presume the following SQL definitions:

```
CREATE TYPE PHONE_ARRAY IS VARRAY (10) OF varchar2(30)
/
/*** Create ADDRESS UDT ***/
CREATE TYPE ADDRESS AS OBJECT
(
  street        VARCHAR(60),
  city          VARCHAR(30),
  state         CHAR(2),
  zip_code      CHAR(5)
)
/
/*** Create PERSON UDT containing an embedded ADDRESS UDT ***/
CREATE TYPE PERSON AS OBJECT
(
  name    VARCHAR(30),
  ssn     NUMBER,
  addr    ADDRESS
)
/

CREATE TABLE  employees
( empnumber            INTEGER PRIMARY KEY,
  person_data     REF  person,
  manager         REF  person,
  office_addr          address,
  salary               NUMBER,
  phone_nums           phone_array
)
/
```

And presume that JPublisher is used to create a `PhoneArray` custom collection class to map from the `PHONE_ARRAY` SQL type.

The following method selects a row from this table, placing the data into a host variable of the `PhoneArray` type:

```
private static void selectVarray() throws SQLException
{
  PhoneArray ph;
  #sql {select phone_nums into :ph from employees where empnumber=2001};
  System.out.println(
    "there are "+ph.length()+" phone numbers in the PhoneArray.  They are:");

  String [] pharr = ph.getArray();
```

```
  for (int i=0;i<pharr.length;++i)
    System.out.println(pharr[i]);
}
```

Inserting a Row that Includes a VARRAY

This section provides an example of inserting data from a host expression into a VARRAY, using the same SQL definitions and custom collection class (`PhoneArray`) as in the previous section.

The following methods populate a `PhoneArray` instance and use it as a host variable, inserting its data into a VARRAY in the database:

```
// creates a varray object of PhoneArray and inserts it into a new row
private static void insertVarray() throws SQLException
{
  PhoneArray phForInsert = consUpPhoneArray();
  // clean up from previous demo runs
  #sql {delete from employees where empnumber=2001};
  // insert the PhoneArray object
  #sql {insert into employees (empnumber, phone_nums)
        values(2001, :phForInsert)};
}

private static PhoneArray consUpPhoneArray()
{
  String [] strarr = new String[3];
  strarr[0] = "(510) 555.1111";
  strarr[1] = "(617) 555.2222";
  strarr[2] = "(650) 555.3333";
  return new PhoneArray(strarr);
}
```

Serialized Java Objects

When writing and reading instances of Java objects to or from the database, it is sometimes advantageous to define a SQL object type that corresponds to your Java class and use the mechanisms of mapping custom Java classes described previously. This fully permits SQL queries on your Java objects.

In some cases, however, you may want to store Java objects "as-is" and retrieve them later, using database columns of the `RAW` or `BLOB` type. There are different ways to accomplish this:

- You can map a serializable Java class to `RAW` or `BLOB` columns by using a nonstandard extension to the type map facility or by adding a type code field to the serializable class, so that instances of the serializable class can be stored as `RAW` or `BLOB`.
- You can use the `ORAData` facility to define a serializable wrapper class whose instances can be stored in `RAW` or `BLOB` columns.

Serializing in any of these ways works for any Oracle SQLJ run time library.

This section covers the following topics:

- Serializing Java Classes to RAW and BLOB Columns
- SerializableDatum: an ORAData Implementation

- SerializableDatum in SQLJ Applications
- SerializableDatum (Complete Class)

Serializing Java Classes to RAW and BLOB Columns

If you want to store instances of Java classes directly in `RAW` or `BLOB` columns, then you must meet certain nonstandard requirements to specify the desired SQL-Java mapping. Note that in SQLJ statements the serializable Java objects can be transparently read and written as if they were built-in types.

You have two options in specifying the SQL-Java type mapping:

- Declare a type map in the connection context declaration and use this type map to specify mappings.
- Use the `public static final` field `_SQL_TYPECODE` to specify the mapping.

Defining a Type Map for Serializable Classes

Consider an example where `SAddress`, `pack.SPerson`, and `pack.Manager.InnerSPM`, where `InnerSPM` is an inner class of `Manager`, are serializable Java classes. In other words, these classes implement the `java.io.Serializable` interface.

You must use the classes only in statements that use explicit connection context instances of a declared connection context type, such as `SerContext` in the following example:

```
SAddress               a =...;
pack.SPerson           p =...;
pack.Manager.InnerSPM pm =...;
SerContext ctx = new SerContext(url,user,pwd,false);
#sql [ctx] { ... :a ... :OUT p ... :INOUT pm ... };
```

The following is required:

- The connection context type must have been declared using the `typeMap` attribute of a `with` clause to specify an associated class implementing `java.util.PropertyResourceBundle`. In the example, `SerContext` may be declared as follows.

  ```
  #sql public static context SerContext with (typeMap="SerMap");
  ```

- The type map resource must provide nonstandard mappings from `RAW` or `BLOB` columns to the serializable Java classes. This mapping is specified with entries of the following form, depending on whether the Java class is mapped to a `RAW` or a `BLOB` column:

  ```
  oracle-class.java_class_name=JAVA_OBJECT RAW
  oracle-class.java_class_name=JAVA_OBJECT BLOB
  ```

  The keyword `oracle-class` marks this as an Oracle-specific extension. In the example, the `SerMap.properties` resource file may contain the following entries:

  ```
  oracle-class.SAddress=JAVA_OBJECT RAW
  oracle-class.pack.SPerson=JAVA_OBJECT BLOB
  oracle-class.packManager$InnerSPM=JAVA_OBJECT RAW
  ```

  Although the period (.) separates package and class names, you *must* use the dollar sign ($) to separate an inner class name.

Note that this Oracle-specific extension can be placed in the same type map resource as standard `SQLData` type map entries.

Using Fields to Determine Mapping for Serializable Classes

As an alternative to using a type map for a serializable class, you can use static fields in the serializable class to determine type mapping. You can add either of the following fields to a class that implements the `java.io.Serializable` interface, such as the `SAddress` and `SPerson` classes from the preceding example:

```
public final static int _SQL_TYPECODE = oracle.jdbc.OracleTypes.RAW;

public final static int _SQL_TYPECODE = oracle.jdbc.OracleTypes.BLOB;
```

> **Note:** Using the type map facility supersedes manually adding the `_SQL_TYPECODE` field to the class.

Limitations on Serializing Java Objects

You should be aware of the effect of serialization. If two objects, A and B, share the same object, C, then upon serialization and subsequent deserialization of A and B, each will point to its own clone of the object C. Sharing is broken.

In addition, note that for a given Java class, you can declare only one kind of serialization: either into `RAW` or into `BLOB`. The SQLJ translator can check only that the actual usage conforms to either `RAW` or `BLOB`.

`RAW` columns are limited in size. You might experience run-time errors if the actual size of the serialized Java object exceeds the size of the column.

Column size is much less restrictive for `BLOB` columns. Writing a serialized Java object to a `BLOB` column is supported by the Oracle JDBC Oracle Call Interface (OCI) and Thin drivers. Retrieving a serialized object from a `BLOB` column is supported by all Oracle JDBC drivers since Oracle9*i*.

Finally, treating serialized Java objects this way is an Oracle-specific extension and requires the Oracle SQLJ run time as well as either the default Oracle-specific code generation (`-codegen=oracle` during translation) or, for ISO standard code generation (`-codegen=iso`), Oracle-specific profile customization.

> **Note:** The implementation of this particular serialization mechanism does not use JDBC type maps. The map (to `BLOB` or to `RAW`) is hardcoded in the Oracle profile customization at translation time, or is generated directly into Java code.

SerializableDatum: an ORAData Implementation

"Additional Uses for ORAData Implementations" on page 6-13 includes examples of situations where you might want to define a custom Java class that maps to some `oracle.sql.*` type other than `oracle.sql.STRUCT`, `oracle.sql.REF`, or `oracle.sql.ARRAY`.

An example of such a situation is if you want to serialize and deserialize Java objects into and out of `RAW` fields, with a custom Java class that maps to the `oracle.sql.RAW` type. This could apply equally to `BLOB` fields, with a custom Java class that maps to the `oracle.sql.BLOB` type.

This section presents an example of such an application, creating a class, `SerializableDatum`, that implements the `ORAData` interface and follows the general form of custom Java classes. The example starts with a step-by-step approach to the development of `SerializableDatum`, followed by the complete sample code.

> **Note:** This application uses classes from the `java.io`, `java.sql`, `oracle.sql`, and `oracle.jdbc` packages. The import statements are not shown here.

1. Begin with a skeleton of the class.

```
public class SerializableDatum implements ORAData
{
   // Client methods for constructing and accessing the Java object

   public Datum toDatum(java.sql.Connection c) throws SQLException
   {
      // Implementation of toDatum()
   }

   public static ORADataFactory getORADataFactory()
   {
      return FACTORY;
   }

   private static final ORADataFactory FACTORY =
           // Implementation of an ORADataFactory for SerializableDatum

   // Construction of SerializableDatum from oracle.sql.RAW

   public static final int _SQL_TYPECODE = OracleTypes.RAW;
}
```

   `SerializableDatum` does not implement the `ORADataFactory` interface, but its `getORADataFactory()` method returns a static member that implements this interface.

   The `_SQL_TYPECODE` is set to `OracleTypes.RAW` because this is the data type being read from and written to the database. The SQLJ translator needs this type code information in performing online type-checking to verify compatibility between the user-defined Java type and the SQL type.

2. Define client methods that perform the following:

   - Create a `SerializableDatum` object.
   - Populate a `SerializableDatum` object.
   - Retrieve data from a `SerializableDatum` object.

```
// Client methods for constructing and accessing a SerializableDatum

private Object m_data;
public SerializableDatum()
{
   m_data = null;
}
public void setData(Object data)
{
   m_data = data;
```

```
}
public Object getData()
{
   return m_data;
}
```

3. Implement a `toDatum()` method that serializes data from a `SerializableDatum` object to an `oracle.sql.RAW` object. The implementation of `toDatum()` must return a serialized representation of the object in the `m_data` field as an `oracle.sql.RAW` instance.

```
// Implementation of toDatum()

try {
   ByteArrayOutputStream os = new ByteArrayOutputStream();
   ObjectOutputStream oos = new ObjectOutputStream(os);
   oos.writeObject(m_data);
   oos.close();
   return new RAW(os.toByteArray());
} catch (Exception e) {
  throw new SQLException("SerializableDatum.toDatum: "+e.toString()); }
```

4. Implement data conversion from an `oracle.sql.RAW` object to a `SerializableDatum` object. This step deserializes the data.

```
// Constructing SerializableDatum from oracle.sql.RAW

private SerializableDatum(RAW raw) throws SQLException
{
   try {
      InputStream rawStream = new ByteArrayInputStream(raw.getBytes());
      ObjectInputStream is = new ObjectInputStream(rawStream);
      m_data = is.readObject();
      is.close();
   } catch (Exception e) {
     throw new SQLException("SerializableDatum.create: "+e.toString()); }
}
```

5. Implement an `ORADataFactory`. In this case, it is implemented as an anonymous class.

```
// Implementation of an ORADataFactory for SerializableDatum

new ORADataFactory()
{
   public ORAData create(Datum d, int sqlCode) throws SQLException
   {
      if (sqlCode != _SQL_TYPECODE)
      {
         throw new SQLException
                   ("SerializableDatum: invalid SQL type "+sqlCode);
      }
      return (d==null) ? null : new SerializableDatum((RAW)d);
   }
};
```

SerializableDatum in SQLJ Applications

Given the `SerializableDatum` class created in the preceding section, this section shows how to use an instance of it in a SQLJ application, both as a host variable and as an iterator column.

Presume the following table definition:

```
CREATE TABLE PERSONDATA (NAME VARCHAR2(20) NOT NULL, INFO RAW(2000));
```

SerializableDatum as Host Variable

The following uses a `SerializableDatum` instance as a host variable:

```
...
SerializableDatum pinfo = new SerializableDatum();
pinfo.setData (
   new Object[] {"Some objects", new Integer(51), new Double(1234.27) } );
String pname = "MILLER";
#sql { INSERT INTO persondata VALUES(:pname, :pinfo) };
...
```

SerializableDatum in Iterator Column

Following is an example of using `SerializableDatum` as a named iterator column:

```
#sql iterator PersonIter (SerializableDatum info, String name);

...
PersonIter pcur;
#sql pcur = { SELECT * FROM persondata WHERE info IS NOT NULL };
while (pcur.next())
{
   System.out.println("Name:" + pcur.name() + " Info:" + pcur.info());
}
pcur.close();
...
```

SerializableDatum (Complete Class)

The following is complete code for the `SerializableDatum` class, which was developed in step-by-step fashion in the preceding sections.

```
import java.io.*;
import java.sql.*;
import oracle.sql.*;
import oracle.jdbc.*;

public class SerializableDatum implements ORAData
{
// Client methods for constructing and accessing a SerializableDatum

   private Object m_data;
   public SerializableDatum()
   {
      m_data = null;
   }
   public void setData(Object data)
   {
      m_data = data;
   }
```

```
   public Object getData()
   {
      return m_data;
   }

// Implementation of toDatum()

   public Datum toDatum(Connection c) throws SQLException
   {

      try {
         ByteArrayOutputStream os = new ByteArrayOutputStream();
         ObjectOutputStream oos = new ObjectOutputStream(os);
         oos.writeObject(m_data);
         oos.close();
         return new RAW(os.toByteArray());
      } catch (Exception e) {
        throw new SQLException("SerializableDatum.toDatum: "+e.toString()); }
   }

   public static ORADataFactory getORADataFactory()
   {
      return FACTORY;
   }

// Implementation of an ORADataFactory for SerializableDatum

   private static final ORADataFactory FACTORY =

      new ORADataFactory()
      {
         public ORAData create(Datum d, int sqlCode) throws SQLException
         {
            if (sqlCode != _SQL_TYPECODE)
            {
               throw new SQLException(
                  "SerializableDatum: invalid SQL type "+sqlCode);
            }
            return (d==null) ? null : new SerializableDatum((RAW)d);
         }
      };

// Constructing SerializableDatum from oracle.sql.RAW

   private SerializableDatum(RAW raw) throws SQLException
   {
      try {
         InputStream rawStream = new ByteArrayInputStream(raw.getBytes());
         ObjectInputStream is = new ObjectInputStream(rawStream);
         m_data = is.readObject();
         is.close();
      } catch (Exception e) {
        throw new SQLException("SerializableDatum.create: "+e.toString()); }
   }

   public static final int _SQL_TYPECODE = OracleTypes.RAW;
}
```

Weakly Typed Objects, References, and Collections

Weakly typed objects, references, and collections are supported by SQLJ. Their use is not generally recommended, and there are some specific restrictions, but in some circumstances they can be useful. For example, you might have generic code that can use "any `STRUCT`" or "any `REF`".

This section covers the following topics:

Support for Weakly Typed Objects, References, and Collections

In using Oracle objects, references, or collections in a SQLJ application, you have the option of using generic and weakly typed `java.sql` or `oracle.sql` instances instead of the strongly typed custom object, reference, and collection classes that implement the `ORAData` interface or the strongly typed custom object classes that implement the `SQLData` interface. Note that if you use `SQLData` implementations for your custom object classes, then you will have no choice but to use weakly typed custom reference instances.

The following weak types can be used for iterator columns or host expressions in the Oracle SQLJ implementation:

- `java.sql.Struct` or `oracle.sql.STRUCT` for objects
- `java.sql.Ref` or `oracle.sql.REF` for object references
- `java.sql.Array` or `oracle.sql.ARRAY` for collections

In host expressions, they are supported as follows:

- As input host expressions
- As output host expressions in an `INTO`-list

Using these weak types is not generally recommended, however, as you would lose all the advantages of the strongly typed paradigm that SQLJ offers.

Each attribute in a `STRUCT` object or each element in an `ARRAY` object is stored in an `oracle.sql.Datum` object, with the underlying data being in the form of the appropriate `oracle.sql.*` subtype of `Datum`, such as `oracle.sql.NUMBER` or `oracle.sql.CHAR`. Attributes in a `STRUCT` object are nameless. Because of the generic nature of the `STRUCT` and `ARRAY` classes, SQLJ cannot perform type checking where objects or collections are written to or read from instances of these classes.

It is generally recommended that you use custom Java classes for objects, references, and collections, preferably classes generated by JPublisher.

Restrictions on Weakly Typed Objects, References, and Collections

A weakly typed object (`Struct` or `STRUCT` instance), reference (`Ref` or `REF` instance), or collection (`Array` or `ARRAY` instance) *cannot* be used in host expressions in the following circumstances:

- `IN` parameter if null
- `OUT` or `INOUT` parameter in a stored procedure or function call
- `OUT` parameter in a stored function result expression

They cannot be used in these ways, because there is no way to know the underlying SQL type name, such as `Person`, which is required by the Oracle JDBC driver to materialize an instance of a user-defined type in Java.

Oracle OPAQUE Types

Oracle OPAQUE types are abstract data types. With data implemented as simply a series of bytes, the internal representation is not exposed. Typically an OPAQUE type will be provided by Oracle, not implemented by a customer.

OPAQUE types are similar in some basic ways to object types, with similar concepts of static methods, instances, and instance methods. Typically, only the methods supplied with an OPAQUE type allow you to manipulate the state and internal byte representation. In Java, an OPAQUE type can be represented as `oracle.sql.OPAQUE` or as a custom class implementing the `oracle.sql.ORAData` interface. On the client-side, Java code can be implemented to manipulate the bytes, assuming the byte pattern is known. The Oracle Database 11*g* JPublisher utility can be useful in this way, creating a custom class implementing `ORAData` to allow you to manipulate data without having to make repeated round trips to the database.

See Also: *Oracle Database JPublisher User's Guide*

A key example of an OPAQUE type is `XMLType`, provided with Oracle Database 11*g*. This Oracle-provided type facilitates handling XML data natively in the database.

`SYS.XMLType` offers the following features, exposed through the Java `oracle.xdb.XMLType` class:

- It can be used as the data type of a column in a table or view. `XMLType` can store any content but is designed to optimally store XML content. An instance of it can represent an XML document in SQL.
- It has a SQL API with built-in member functions that operate on XML content. For example, you can use `XMLType` functions to create, query, extract, and index XML data stored in an Oracle Database 11*g* instance.
- It can be used in stored procedures for parameters, return values, and variables.
- Its functionality is also available through APIs provided in PL/SQL, Java, and C (OCI).

See Also: *Oracle XML DB Developer's Guide*

7

Advanced Language Features

This chapter discusses advanced SQLJ language features for use in coding your application. For more basic topics, refer to Chapter 4, "Basic Language Features".

The following topics are discussed:

- Connection Contexts
- Execution Contexts
- Multithreading in SQLJ
- Iterator Class Implementation and Advanced Functionality
- Advanced Transaction Control
- SQLJ and JDBC Interoperability
- Support for Dynamic SQL
- Execution Plan Fixing

Connection Contexts

SQLJ supports the concept of connection contexts, allowing strongly typed connections for use with different sets of SQL entities. You can think of a connection context as being associated with a particular set of SQL entities, such as tables, views, and stored procedures. SQLJ lets you declare additional connection context classes so that you can use each class for connections that use a particular set of SQL entities. Different instances of a single connection context class are not required to use the same physical entities or connect to the same schema, but will at least use sets of entities with the same names and data types.

> **See Also:** "Connection Considerations" on page 3-4 for an overview of connection basics, focusing on situations where you are using just a single set of SQL entities and a single connection context class.

This section covers the following topics:

- Connection Context Concepts
- Connection Context Logistics
- More About Declaring and Using a Connection Context Class
- Example of Multiple Connection Contexts
- Implementation and Functionality of Connection Context Classes
- Using the IMPLEMENTS Clause in Connection Context Declarations

- Semantics-Checking of Your Connection Context Usage
- Standard Data Source Support
- SQLJ-Specific Data Sources
- SQLJ-Specific Connection JavaBeans for JavaServer Pages
- SQLJ Support for Global Transactions

Connection Context Concepts

If your application uses different sets of SQL entities, then you will typically want to declare and use one or more additional connection context classes, as discussed in "Overview of SQLJ Declarations" on page 4-1. Each connection context class can be used for a particular set of interrelated SQL entities, meaning that all the connections you define using a particular connection context class will use tables, views, stored procedures, and so on, which have the same names and use the same data types.

An example of a set of SQL entities is the set of tables and stored procedures used by the Human Resources (HR) department. Perhaps they use the `EMPLOYEES` and `DEPARTMENTS` tables and the `CHANGE_DEPT` and `UPDATE_HEALTH_PLAN` stored procedures. Another set of SQL entities might be the set of tables and procedures used by the Payroll department, perhaps consisting of the `EMPS` table (another table of employees, but different than the one used by HR) and the `GIVE_RAISE` and `CHANGE_WITHHOLDING` stored procedures.

The advantage in tailoring connection context classes to sets of SQL entities is in the degree of online semantics-checking that this allows. Online checking verifies that all the SQL entities appearing in SQLJ statements that use a given connection context class match SQL entities found in the exemplar schema used during translation. An exemplar schema is a database account that SQLJ connects to for online checking of all the SQLJ statements that use a particular connection context class. You provide exemplar schemas to the translator through the SQLJ command-line `-user`, `-password`, and `-url` options. An exemplar schema may or may not be the same account your application will use at run time.

> **See Also:** "Connection Options" on page 8-25

If you have SQLJ statements that use a broad and perhaps unrelated group of SQL entities, but you use only a single connection context class for these statements, then the exemplar schema you provide must be very general. It must contain all the tables, views, and stored procedures used throughout all the statements. Alternatively, if all the SQLJ statements using a given connection context class use a tight, presumably interrelated, set of SQL entities, then you can provide a more specific exemplar schema that enables more thorough and meaningful semantics-checking.

Note:

- Be aware that a connection context class declaration does not define a set of SQL entities to be used with the declared connection context class, and it is permissible to use the same connection context class for connections that use disparate and unrelated sets of entities. How you use your connection context classes is at your discretion. All that limits the SQL entities you can use with a particular connection context class are the set of entities available in the exemplar schema, if you use online semantics-checking during translation, and the set of entities available in the schema you connect to at run time, using instances of the connection context class.
- If you use qualified SQL names in your application, such as `SCOTT.EMP`, which specifies the schema where the entity resides, then the exemplar schema, if you use online checking, and run time schema must have permission to access resources by these fully qualified names.
- It is possible to use a single connection context class, even for connections to databases from different vendors, as long as each schema you connect to has entities that are accessible by the same names and that use compatible data types.

Connection Context Logistics

Declaring a connection context class results in the SQLJ translator defining a class for you in the translator-generated code. In addition to any connection context classes that you declare, there is always the default connection context class:

```
sqlj.runtime.ref.DefaultContext
```

When you construct a connection context instance, specify a particular schema and a particular session and transaction in which SQL operations will execute. You typically accomplish this by specifying a user name, password, and database URL as input to the constructor of the connection context class. The connection context instance manages the set of SQL operations performed during the session.

In each SQLJ statement, you can specify a connection context instance to use. The following example shows basic declaration and use of a connection context class, `MyContext`, to connect to two different schemas. For typical usage, assume these schemas include a set of SQL entities with common names and data types.

```
#sql context MyContext;

...
MyContext mctx1 = new MyContext
  ("jdbc:oracle:thin:@localhost:1521/myservice", "scott", "tiger", false);
MyContext mctx2 =  new MyContext
  ("jdbc:oracle:thin@localhost:1521/myservice", "brian", "mypasswd", false);
```

Note that connection context class constructors specify a boolean auto-commit parameter. In addition, note that you can connect to the same schema with different connection context instances. In the preceding example, both `mctx1` and `mctx2` can specify `scott/tiger` if desired. However, during run time, one connection context instance would not see changes to the database made from the other until the changes are committed. The only exception to this would be if both connection context

instances were created from the same underlying Java Database Connectivity (JDBC) connection instance. One of the constructors of any connection context class takes a JDBC connection instance as input.

More About Declaring and Using a Connection Context Class

This section gives a detailed example of how to declare a connection context class, then define a database connection using an instance of the class.

A connection context class has constructors for opening a connection to a database schema that take any of the following input parameter sets (as with the `DefaultContext` class):

- URL (`String`), user name (`String`), password (`String`), auto-commit (`boolean`)
- URL (`String`), `java.util.Properties` object, auto-commit (`boolean`)
- URL (`String` fully specifying connection and including user name and password), auto-commit setting (`boolean`)
- JDBC connection object (`Connection`)
- SQLJ connection context object

Note:

- When using the constructor that takes a JDBC connection object, do not initialize the connection context instance with a null JDBC connection.
- The auto-commit setting determines whether SQL operations are automatically committed. For more information, refer to "Basic Transaction Control" on page 3-18.
- If a connection context class is declared with a data source `with` clause, then it incorporates a different set of constructors. Refer to "Standard Data Source Support" on page 7-9 for more information.

Declaring the Connection Context Class

The following declaration creates a connection context class:

```
#sql context OrderEntryCtx <implements_clause> <with_clause>;
```

This results in the SQLJ translator generating a class that implements the `sqlj.runtime.ConnectionContext` interface and extends some base class, probably an abstract class, that also implements the `ConnectionContext` interface. This base class would be a feature of the particular SQLJ implementation you are using. The `implements` clause and `with` clause are optional, specifying additional interfaces to implement and variables to define and initialize, respectively.

> **See Also:** "Declaration IMPLEMENTS Clause" on page 4-3 and "Declaration WITH Clause" on page 4-4

The following is an example of what the SQLJ translator generates (with method implementations omitted):

```
class OrderEntryCtx implements sqlj.runtime.ConnectionContext
```

```
      extends ...
{
   public OrderEntryCtx(String url, Properties info, boolean autocommit)
          throws SQLException {...}
   public OrderEntryCtx(String url, boolean autocommit)
          throws SQLException {...}
   public OrderEntryCtx(String url, String user, String password,
          boolean autocommit) throws SQLException {...}
   public OrderEntryCtx(Connection conn) throws SQLException {...}
   public OrderEntryCtx(ConnectionContext other) throws SQLException {...}

   public static OrderEntryCtx getDefaultContext() {...}
   public static void setDefaultContext(OrderEntryCtx ctx) {...}
}
```

Creating a Connection Context Instance

Continuing the preceding example, instantiate the `OrderEntryCtx` class with the following syntax:

```
OrderEntryCtx myOrderConn = new OrderEntryCtx
                            (url, username, password, autocommit);
```

For example:

```
OrderEntryCtx myOrderConn = new OrderEntryCtx
  ("jdbc:oracle:thin:@localhost:1521/myservice", "scott", "tiger", true);
```

This is accomplished in the same way as instantiating the `DefaultContext` class. All connection context classes, including `DefaultContext`, have the same constructor signatures.

Note:

- You typically must register your JDBC driver prior to constructing a connection context instance. Refer to "Driver Selection and Registration for Run Time" on page 3-4.
- If a connection context class is declared with a data source `with` clause, then it incorporates a different set of constructors. Refer to "Standard Data Source Support" on page 7-9 for more information.

Specifying a Connection Context Instance for a SQLJ Clause

Recall that the basic SQLJ statement syntax is as follows:

```
#sql <[<conn><, ><exec>]> { SQL operation };
```

Specify the connection context instance inside square brackets following the `#sql` token. For example, in the following SQLJ statement, the connection context instance is `myOrderConn` from the previous example:

```
#sql [myOrderConn] { UPDATE TAB2 SET COL1 = :w WHERE :v < COL2 };
```

In this way, you can specify an instance of either the `DefaultContext` class or any declared connection context class.

Closing a Connection Context Instance

It is advisable to close all connection context instances when you are done. Each connection context class includes a `close()` method, as discussed for the `DefaultContext` class in "Closing Connections" on page 3-8.

In closing a connection context instance that shares the underlying connection with another connection instance, you might want to keep the underlying connection open.

See Also: "Closing Shared Connections" on page 7-47

Example of Multiple Connection Contexts

The following is an example of a SQLJ application using multiple connection contexts. It implicitly uses an instance of the `DefaultContext` class for one set of SQL entities and an instance of the declared `DeptContext` connection context class for another set of SQL entities.

This example uses the static `Oracle.connect()` method to establish a default connection, then constructs an additional connection by using the static `Oracle.getConnection()` method to pass another `DefaultContext` instance to the `DeptContext` constructor. As previously mentioned, this is just one of several ways you can construct a SQLJ connection context instance.

```
import java.sql.SQLException;
import oracle.sqlj.runtime.Oracle;

// declare a new context class for obtaining departments
#sql context DeptContext;

#sql iterator Employees (String ename, int deptno);

class MultiSchemaDemo
{
  public static void main(String[] args) throws SQLException
  {
    // set the default connection to the URL, user, and password
    // specified in your connect.properties file
    Oracle.connect(MultiSchemaDemo.class, "connect.properties");

    // create a context for querying department info using
    // a second connection
    DeptContext deptCtx =
      new DeptContext(Oracle.getConnection(MultiSchemaDemo.class,
                      "connect.properties"));

    new MultiSchemaDemo().printEmployees(deptCtx);
    deptCtx.close();
  }

  // performs a join on deptno field of two tables accessed from
  // different connections.
  void printEmployees(DeptContext deptCtx) throws SQLException
  {
    // obtain the employees from the default context
    Employees emps;
    #sql emps = { SELECT ename, deptno FROM emp };

    // for each employee, obtain the department name
    // using the dept table connection context
    while (emps.next()) {
```

```
      String dname;
      int deptno = emps.deptno();
      #sql [deptCtx] {
        SELECT dname INTO :dname FROM dept WHERE deptno = :deptno
      };
      System.out.println("employee: " +emps.ename() +
                         ", department: " + dname);
    }
    emps.close();
  }
}
```

Implementation and Functionality of Connection Context Classes

This section discusses how SQLJ implements connection context classes, including the `DefaultContext` class, and what noteworthy methods they contain. As mentioned earlier, the `DefaultContext` class and all generated connection context classes implement the `ConnectionContext` interface.

> **Note:** Extending connection context classes is not permitted in the SQLJ specification and is not supported by the Oracle SQLJ implementation.

ConnectionContext Interface

Each connection context class implements the `sqlj.runtime.ConnectionContext` interface.

Basic methods specified by this interface include the following:

- `close(boolean CLOSE_CONNECTION/KEEP_CONNECTION)`: Releases all resources used in maintaining this connection and closes any open connected profiles. It may close the underlying JDBC connection, depending on whether `CLOSE_CONNECTION` or `KEEP_CONNECTION` is specified. These are static boolean constants of the `ConnectionContext` interface.

  See Also: "Closing Shared Connections" on page 7-47

- `getConnection()`: Returns the underlying JDBC connection object for this connection context instance.
- `getExecutionContext()`: Returns the default `ExecutionContext` instance for this connection context instance.

  See Also: "Execution Contexts" on page 7-24

Additional Connection Context Class Methods

In addition to the methods specified and defined in the `ConnectionContext` interface, each connection context class defines the following methods:

- *`YourCtxClass`* `getDefaultContext()`: This is a static method that returns the default connection context instance for a given connection context class.
- `setDefaultContext(`*`YourCtxClass connctxinstance`*`)`: This is a static method that defines the given connection context instance as the default connection context instance for its class.

Although it is true that you can use an instance of only the `DefaultContext` class as your default connection, it might still be useful to designate an instance of a declared connection context class as the default context for that class, using the `setDefaultContext()` method. Then you could conveniently retrieve it using the `getDefaultContext()` method of the particular class. This would enable you, for example, to specify a connection context instance for a SQLJ executable statement as follows:

```
#sql context MyContext;

...
MyContext myctx1 = new MyContext(url, user, password, autocommit);
...
MyContext.setDefaultContext(myctx1);
...
#sql [MyContext.getDefaultContext()] { SQL operations };
...
```

Additionally, each connection context class defines methods for control of SQLJ statement caching. The following are the static methods:

- `setDefaultStmtCacheSize(int)`
- `int getDefaultStmtCacheSize()`

The following are the instance methods:

- `setStmtCacheSize(int)`
- `int getStmtCacheSize()`

By default, statement caching is enabled.

> **See Also:** "Connection Context Methods for Statement Caching (Oracle-Specific Code)" on page 10-3

Using the IMPLEMENTS Clause in Connection Context Declarations

There may be situations where it is useful to implement an interface in your connection context declarations. For example, you may want to define an interface that exposes just a subset of the functionality of a connection context class. More specifically, you may want a class that has the `getConnection()` functionality, but does not have other functionality of a connection context class.

You can create an interface called `HasConnection`, for example, that specifies a `getConnection()` method, but does not specify other methods found in a connection context class. You can then declare a connection context class but expose only the `getConnection()` functionality by assigning a connection context instance to a variable of the `HasConnection` type, instead of to a variable that has the type of your declared connection context class.

Assuming `HasConnection` is in the `mypackage` package, the declaration will be as follows:

```
#sql public context MyContext implements mypackage.HasConnection;
```

You can then instantiate a connection instance as follows:

```
HasConnection myConn = new MyContext (url, username, password, autocommit);
```

For example:

```
HasConnection myConn = new MyContext
```

```
("jdbc:oracle:thin:@localhost:1521/myservice", "scott", "tiger", true);
```

Semantics-Checking of Your Connection Context Usage

A significant feature of SQLJ is strong typing of connections, with each connection context class typically used for operations on a particular set of interrelated SQL entities. This does not mean that all the connection instances of a single class use the same physical entities. Instead, they use entities that have the same properties, such as names and privileges associated with tables and views, data types of their rows, and names and definitions of stored procedures. This strong typing allows SQLJ semantics-checking to verify during translation that you are using your SQL operations correctly, with respect to your database connections.

To use online semantics-checking during translation, provide a sample schema, which includes an appropriate set of SQL entities, for each connection context class. These sample schemas are referred to as exemplar schemas. Provide exemplar schemas through an appropriate combination of the SQLJ `-user`, `-password`, and `-url` options. Following are two examples, one for the `DefaultContext` class and one for a declared connection context class, where the user, password, and URL are all specified through the -user option:

```
-user=scott/tiger@jdbc:oracle:oci:@
-user@MyContext=scott/tiger@jdbc:oracle:oci:@
```

During semantics-checking, the translator connects to the specified exemplar schema for a particular connection context class and accomplishes the following:

- It examines each SQLJ statement in your code that specifies an instance of the connection context class and checks its SQL operations, such as what tables you access and what stored procedures you use.
- It verifies that entities in the SQL operations match the set of entities existing in the exemplar schema.

It is your responsibility to pick an exemplar schema that represents the run-time schema in appropriate ways. For example, it must have tables, views, stored functions, and stored procedures with names and data types that match what are used in your SQL operations, and with privileges set appropriately.

If no appropriate exemplar schema is available during translation for one of your connection context classes, then it is not necessary to specify SQLJ translator options for that particular connection context class. In that case, SQLJ statements specifying connection objects of that connection context class are semantically checked only to the extent possible.

Note: Remember that the exemplar schema you specify in your translator option settings does not specify the schema to be used at run time. The exemplar schema furnishes the translator only with a set of SQL entities to compare against the entities you use in your SQLJ executable statements.

Standard Data Source Support

The JDBC 2.0 extended application programming interface (API) specifies the use of data sources and Java Naming and Directory Interface (JNDI) as a portable alternative to the `DriverManager` mechanism for obtaining JDBC connections. It permits database connections to be established through a JNDI name lookup. This name is bound to a particular database and schema prior to program run time through a

`javax.sql.DataSource` object, typically installed through a graphical user interface (GUI) JavaBeans deployment tool. The name can be bound to different physical connections without any source code changes simply by rebinding the name in the directory service.

SQLJ uses the same mechanism to create connection context instances in a flexible and portable way. Data sources can also be implemented using a connection pool or distributed transaction service, as defined by the JDBC 2.0 extended API.

See Also: *Oracle Database JDBC Developer's Guide and Reference*

Associating a Connection Context with a Data Source

In SQLJ it is natural to associate a connection context class with a logical schema, in much the same way that a data source name serves as a symbolic name for a JDBC connection. Combine both concepts by adding the data source name to the connection context declaration. For example:

```
#sql context EmpCtx with (dataSource="jdbc/EmpDB");
```

Any connection context class that you declare with a `dataSource` property provides additional constructors. To continue the `EmpCtx` example, the following constructors are provided:

- `EmpCtx()`: Looks up the data source for `jdbc/EmpDB` and then calls the `getConnection()` method on the data source to obtain a connection.
- `EmpCtx(String user, String password)`: Looks up the data source for `jdbc/EmpDB` and calls the `getConnection(user,password)` method on the data source to obtain a connection.
- `EmpCtx(ConnectionContext ctx)`: Delegates to `ctx` to obtain a connection.

Any connection context class declared with a `dataSource` property also omits a number of `DriverManager`-based constructors. Continuing the `EmpCtx` example, the following constructors are omitted:

- `EmpCtx(Connection conn)`
- `EmpCtx(String url, String user, String password, boolean autoCommit)`
- `EmpCtx(String url, boolean autoCommit)`
- `EmpCtx(String url, java.util.Properties info, boolean autoCommit)`
- `EmpCtx(String url, boolean autoCommit)`

Auto-Commit Mode for Data Source Connections

The constructors based on data source, unlike those base on `DriverManager`, do not include an explicit auto-commit parameter. They always use the auto-commit mode defined by the data source.

Data sources are configured to have a default auto-commit mode depending on the deployment scenario. For example, data sources in the server and middle tier typically have auto-commit off. Those on the client may have it on. However, it is also possible to configure data sources with a specific auto-commit setting. This permits data sources to be configured for a particular application and deployment scenario. Contrast this with JDBC URLs that may specify only a single database/driver configuration.

Programs can verify and possibly override the current auto-commit setting with the JDBC connection that underlies their connection context instance.

Note: Be aware of the following points related to the auto-commit status of the connections you establish:

- If you use the `Oracle` class, then auto-commit is off unless you turn it on explicitly.
- If you use `DefaultContext` or a connection context class with `DriverManager`-style constructors, then the auto-commit setting must always be specified explicitly.
- If you use the data source mechanism, then the auto-commit setting is inherited from the underlying data source. In most environments, the data source object originates from JDBC and the auto-commit option is on. To avoid unexpected behavior, always check the auto-commit setting.

Associating a Data Source with the Default Context

If a SQLJ program accesses the default connection context, and the default context has not yet been set, then the SQLJ run time will use the SQLJ default data source to establish its connection. The SQLJ default data source is bound to the JNDI name, `jdbc/defaultDataSource`.

This mechanism provides a portable means to define and install a default JDBC connection for the default SQLJ connection context.

Data Source Support Requirements

For your program to use data sources, you must supply the `javax.sql.*` and `javax.naming.*` packages and an `InitialContext` provider in your Java environment. The latter is required to obtain the JNDI context in which the SQLJ run time can look up the data source object.

All SQLJ run time libraries provided by Oracle support data sources. However, if you use the `runtime12ee` library you must have `javax.sql.*` and `javax.naming.*` in your classpath in order for the run time to load. By contrast, the other run time libraries use reflection to retrieve `DataSource` objects.

SQLJ-Specific Data Sources

The Oracle SQLJ implementation provides SQLJ-specific data source support in the `runtime12ee` library. Currently, SQLJ-specific data sources can be used in client-side or middle-tier applications, but not inside the server.

SQLJ-specific data sources extend JDBC data source functionality with methods that return SQLJ connection context instances. This enables a SQLJ developer to manage connection contexts just as a JDBC developer manages connections. In general, each SQLJ-specific data source interface or class is based on a corresponding standard JDBC data source interface or Oracle data source class.

SQLJ Data Source Interfaces

The `sqlj.runtime.ConnectionContextFactory` interface acts as a base interface for SQLJ data source functionality. It is implemented by a set of more specialized Oracle data source interfaces that add support for features such as connection pooling, connection caching, or distributed transactions.

The `ConnectionContextFactory` interface specifies the following methods to return SQLJ connection context instances:

- `DefaultContext getDefaultContext()`
- `DefaultContext getDefaultContext(boolean autoCommit)`
- `DefaultContext getDefaultContext(String user, String password)`
- `DefaultContext getDefaultContext(String user, String password, boolean autoCommit)`
- `ConnectionContext getContext(Class aContextClass)`
- `ConnectionContext getContext(Class aContextClass, boolean autoCommit)`
- `ConnectionContext getContext(Class aContextClass, String user, String password)`
- `ConnectionContext getContext(Class aContextClass, String user, String password, boolean autoCommit)`

The `getDefaultContext` methods return a `sqlj.runtime.ref.DefaultContext` instance for the SQLJ default context. The `getContext()` methods return a `sqlj.runtime.ConnectionContext` instance. Specifically, it returns an instance of a user-declared connection context class that is specified in the method call.

For both `getDefaultContext()` and `getContext()`, there are signatures that enable you to specify connection parameters for the JDBC connection that underlies the connection context instance: the auto-commit setting, user and password settings, or all three. If you do not specify the user and password, then they are obtained from the underlying data source that generates the connection. If you do not specify an auto-commit setting, then the default is `false` unless it was explicitly set to `true` for the underlying data source.

Each Oracle data source interface that implements `ConnectionContextFactory` also implements a standard JDBC data source interface to specify methods for the appropriate functionality, such as for basic data sources, connection pooling data sources, or distributed transaction (XA) data sources. Oracle has implemented the `SqljDataSource`, `SqljConnectionPoolDataSource`, and `SqljXADataSource` interfaces, located in the `sqlj.runtime` package and specified as follows:

- `interface SqljDataSource extends javax.sql.DataSource, ConnectionContextFactory { }`
- `interface SqljDataSource extends javax.sql.ConnectionPoolDataSource, ConnectionContextFactory { }`
- `interface SqljXADataSource extends javax.sql.XADataSource, ConnectionContextFactory { }`

SQLJ Data Source Classes

Oracle provides SQLJ-specific counterparts for the following JDBC data source classes: `OracleDataSource`, `OracleConnectionPoolDataSource`, `OracleXADataSource`, `OracleConnectionCacheImpl`, `OracleXAConnectionCacheImpl`, and `OracleOCIConnectionPool`.

See Also: *Oracle Database JDBC Developer's Guide and Reference*

Oracle SQLJ-specific data source classes are located in two packages: `oracle.sqlj.runtime` and `oracle.sqlj.runtime.client`.

The `oracle.sqlj.runtime` package includes the following:

- `class OracleSqljDataSource extends oracle.jdbc.pool.OracleDataSource implements ConnectionContextFactory`

  Note: The `OracleSqljDataSource` class implements the `java.io.Serializable` interface. It is therefore serializable and can be used in clustered environments, such as Oracle9*i* Application Server Containers for J2EE (OC4J).

- `class OracleSqljConnectionPoolDataSource extends oracle.jdbc.pool.OracleConnectionPoolDataSource implements ConnectionContextFactory`
- `abstract class OracleSqljXADataSource extends oracle.jdbc.xa.OracleXADataSource implements ConnectionContextFactory`
- `class OracleSqljOCIConnectionPool extends oracle.jdbc.pool.OracleOCIConnectionPool implements ConnectionContextFactory`

  Note:

  - If you are using `OracleSqljConnectionCacheImpl`, then you need to replace it with `OracleSqljDataSource`.
  - If you are using `OracleSqljXAConnectionCacheImpl`, then you need to replace it with `OracleSqljXADataSource`.

The `oracle.sqlj.runtime.client` package includes the following:

- `class OracleSqljXADataSource extends oracle.jdbc.xa.client.OracleXADataSource implements ConnectionContextFactory`

You can use these classes in place of the corresponding JDBC classes that they extend. They include the `getDefaultContext()` and `getContext()` methods. When you call these methods, the following steps take place for you:

1. A new logical JDBC connection is acquired from the present data source.
2. A connection context instance is created from the logical connection and returned.

Examples: Using SQLJ Data Sources

When used in middle-tier environments, SQLJ-specific data sources, like JDBC data sources, are bound to JNDI locations. You can do the binding explicitly, as in the following example:

```
//Initialize the data source
SqljXADataSource sqljDS = new OracleSqljXADataSource();
sqljDS.setUser("scott");
sqljDS.setPassword("tiger");
sqljDS.setServerName("myserver");
```

```
sqljDS.setDatabaseName("ORCL");
sqljDS.setDataSourceName("jdbc/OracleSqljXADS");

//Bind the data source to JNDI
Context ctx = new InitialContext();
ctx.bind("jdbc/OracleSqljXADS");
```

In a middle-tier OC4J environment, another alternative is to instantiate data sources and bind them to JNDI through settings in the `j2ee/home/config/data-sources.xml` file. For example, the following `<data-source>` element in that file creates an `OracleSqljXADataSource` instance and binds it to the JNDI location, `jdbc/OracleSqljXADS`:

```
<data-source
     class="oracle.sqlj.runtime.OracleSqljXADataSource"
     name="jdbc/OracleSqljXADS"
     location="jdbc/OracleSqljXADS"
     xa-location="jdbc/OracleSqljXADS/xa"
     username="scott"
     password="tiger"
     url="jdbc:oracle:thin:@myhost:1521/myservice"
/>
```

See Also: *Oracle Application Server Containers for J2EE Services Guide*

A SQLJ-specific data source bound to a JNDI location can be looked up and used in creating connection context instances. The following code segment uses information from the preceding `<data-source>` element to create connection context instances, a `DefaultContext` instance and an instance of a user-declared `MyCtx` class, respectively:

```
sqlj.runtime.SqljDataSource sqljDS;
InitialContext initCtx = new InitialContext();
sqljDS = (sqlj.runtime.SqljDataSource)initCtx.lookup("jdbc/OracleSqljXADS");
// getDefaultContext
DefaultContext ctx = sqljDS.getDefaultContext();
// getContext
/* Declare MyCtx connection context class. You could optionally use a "with"
   clause to specify  any desired connection parameters not available
   through the underlying data source.
*/
#sql public static context MyCtx;
MyCtx ctx = (MyCtx) sqljDS.getContext(MyCtx.class);
```

SQLJ-Specific Connection JavaBeans for JavaServer Pages

Oracle has implemented a set of JavaBeans for database connections from within Java Server Pages (JSP) pages. The original beans, `ConnBean` and `ConnCacheBean` in `oracle.jsp.dbutil`, are documented in the *Oracle Application Server Containers for J2EE JSP Tag Libraries and Utilities Reference*.

The Oracle SQLJ implementation provides the following extensions of these JavaBeans in the `runtime12ee` library for use in SQLJ JSP pages:

- `oracle.sqlj.runtime.SqljConnBean`
- `oracle.sqlj.runtime.SqljConnCacheBean`

ConnBean and ConnCacheBean include methods that return JDBC connection objects. SqljConnBean and SqljConnCacheBean extend this functionality to support a bean property called ContextClass of type String and to return SQLJ connection context instances.

> **Note:** The SqljConnBean class implements the java.io.Serializable interface. It is therefore serializable and can be used in clustered environments, such as OC4J.

SqljConnBean and SqljConnCacheBean provide the following methods:

- `void setContextClass(String contextClassName)`
- `String getContextClass()`
- `DefaultContext getDefaultContext()`
- `ConnectionContext getContext()`

The ContextClass property specifies the name of a user-declared connection context class, if you are not using DefaultContext. You can set this property through the setContextClass() method.

To retrieve a connection context instance, use getDefaultContext() or getContext(), as appropriate. The former returns a sqlj.runtime.ref.DefaultContext instance, and the latter returns a sqlj.runtime.ConnectionContext instance, specifically, an instance of the class specified in the ContextClass property (by default, DefaultContext).

However, note that the getDefaultContext() and getContext() methods are implemented differently between SqljConnBean and SqljConnCacheBean.

Behavior of SqljConnBean (Simple Connections)

A SqljConnBean instance can wrap only one logical JDBC connection and one SQLJ connection context instance at any given time.

The first getDefaultContext() or getContext() method call will create and return a connection context instance based on the underlying JDBC connection. This connection context instance will also be stored in the SqljConnBean instance.

Once a connection context instance has been created and stored, the behavior of subsequent getDefaultContext() or getContext() calls will depend on the type of the stored connection context and, for getContext(), on the connection context type specified in the ContextClass property, as follows:

- For subsequent getDefaultContext() calls:
  - If the stored connection context instance is a DefaultContext instance: The method will keep returning that instance.
  - If the stored connection context instance is *not* a DefaultContext instance: The method will close the stored connection context instance and reuse the underlying JDBC connection to create and return a new connection context as a DefaultContext instance (regardless of the previous connection context type). This becomes the new connection context instance stored in the SqljConnBean instance.
- For subsequent getContext() calls:

- If the stored connection context instance is of the same type as that specified by the `ContextClass` property: The method will keep returning that instance.
- If the stored connection context instance is *not* of the same type as that specified by `ContextClass`: The method will close the stored connection context instance and reuse the underlying JDBC connection to create and return a new connection context instance, an instance of what is specified in `ContextClass`. This becomes the new connection context instance stored in the `SqljConnBean` instance.

Note: When `SqljConnBean` closes a connection context instance, it does so with the `KEEP_CONNECTION` setting, leaving the underlying JDBC connection intact. Refer to "Closing Shared Connections" on page 7-47 for related information.

Behavior of SqljConnCacheBean (Connection Caching)

Unlike with `SqljConnBean`, the `SqljConnCacheBean` JavaBean creates and returns a new connection context instance, based on a new logical JDBC connection, for each invocation of `getDefaultContext()` or `getContext()`. The connection context type will be `DefaultContext` for a `getDefaultContext()` call or the type specified in the `ContextClass` property for a `getContext()` call.

`SqljConnCacheBean` does not store the connection context instances it creates.

Example: SQLJ JSP Page Using SqljConnCacheBean

The following program, `SQLJSelectInto.sqljsp`, demonstrates the use of `SqljConnCacheBean`, its `ContextClass` bean property, and its `getContext()` method:

```
<%@ page language="sqlj"
         import="java.sql.*, oracle.sqlj.runtime.SqljConnCacheBean" %>
<jsp:useBean id="cbean" class="oracle.sqlj.runtime.SqljConnCacheBean"
             scope="session">
     <jsp:setProperty name="cbean" property="User" value="scott"/>
     <jsp:setProperty name="cbean" property="Password" value="tiger"/>
     <jsp:setProperty name="cbean" property="URL"
                      value="jdbc:oracle:thin:@myhost:1521/myservice"/>
     <jsp:setProperty name="cbean" property="ContextClass"
                      value="sqlj.runtime.ref.DefaultContext"/>
</jsp:useBean>
<HTML>
<HEAD> <TITLE> The SQLJSelectInto JSP  </TITLE> </HEAD>
<BODY BGCOLOR=white>
<% String empno = request.getParameter("empno");
   if (empno != null) { %>
      <H3> Employee # <%=empno %> Details: </H3>
      <% String ename = null;  double sal = 0.0;  String hireDate = null;
         StringBuffer sb = new StringBuffer();
         sqlj.runtime.ref.DefaultContext ctx=null;
         try {
           // Make the Connection
           ctx = (sqlj.runtime.ref.DefaultContext) cbean.getContext();
         } catch (SQLException e) {
         }
          try {
             #sql [ctx] { SELECT ename, sal, TO_CHAR(hiredate, 'DD-MON-YYYY')
```

```
                            INTO :ename, :sal, :hireDate
                            FROM scott.emp WHERE UPPER(empno) = UPPER(:empno)
                };
                sb.append("<BLOCKQUOTE><BIG><B><PRE>\n");
                sb.append("Name       : " + ename + "\n");
                sb.append("Salary     : " + sal + "\n");
                sb.append("Date hired : " + hireDate);
                sb.append("</PRE></B></BIG></BLOCKQUOTE>");
            } catch (java.sql.SQLException e) {
                sb.append("<P> SQL error: <PRE> " + e + " </PRE> </P>\n");
            } finally {
                if (ctx!= null) ctx.close();
            }
        %>
       <H3><%=sb.toString()%></H3>
<%}
%>
<B>Enter an employee number:</B>
<FORM METHOD=get>
<INPUT TYPE="text" NAME="empno" SIZE=10>
<INPUT TYPE="submit" VALUE="Ask Oracle");
</FORM>
</BODY>
</HTML>
```

> **Note:** This example uses the `ContextClass` property for illustrative purposes. However, be aware that `DefaultContext` is the default value anyway and if you want to use `DefaultContext`, then the value of `ContextClass` is irrelevant, if you use `getDefaultContext()` instead of `getContext()`.

SQLJ Support for Global Transactions

A distributed transaction, sometimes referred to as a global transaction, is a set of two or more related transactions that must be managed in a coordinated way. The transactions that constitute a distributed transaction might be in the same database, but more typically are in different databases and often in different locations. Each individual transaction of a distributed transaction is referred to as a transaction branch.

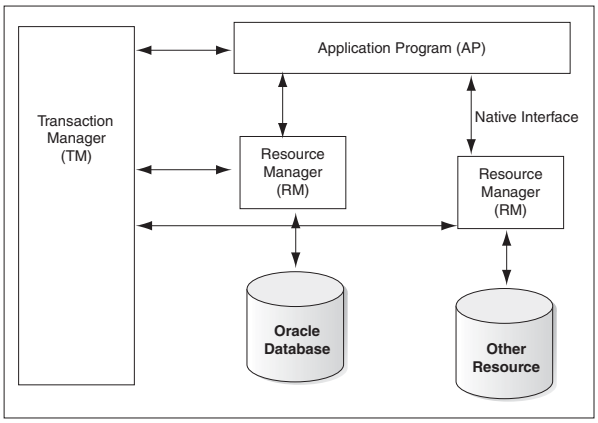
The X/Open Distributed Transaction Processing (DTP) architecture defines a standard architecture that enables multiple but related transactions belonging to the same resource manager or different resource managers to work as a single unit. It coordinates the work between an application program (AP) and a resource manager (RM) into global transactions. Either all the transactions are committed or rolled back.

The Oracle XA library is an external interface that enables transaction managers other than the Oracle server to coordinate global transactions. XA library use supports non-Oracle resource managers, in distributed transactions. This is particularly useful in transactions between several databases and resources. The implementation of the Oracle XA library conforms to the X/Open Distributed Transaction Processing (DTP) software architecture's XA interface specification. The Oracle XA Library is installed as part of Oracle Database Enterprise Edition.

Note:

- JDBC provides several classes and interfaces to support XA. The OracleXADataSource implements the XADataSource interface. The OracleXADatasource is a factory for XA connections. For more information refer to *Oracle Database JDBC Developer's Guide and Reference*.
- This document clearly specifies the methods supported by SQLJ to form a Connection Context in a XA application. To form the connection context, SQLJ uses the JDBC connection formed from the OracleXADataSource.

Figure 7–1



Following is an example of Distributed Transaction Processing (DTP) model:

The **transaction manager** is an external middle tier component residing outside Oracle Database. It provides an API for specifying the boundaries of the transaction and manages commit and recovery. The TM implements a two-phase commit engine to provide an *all-or-none* semantics across distributed RMs.

A **resource manager** controls a shared, recoverable resource that can be returned to a consistent state after a failure. For example, Oracle is a resource manager.

The `javax.sql.XADataSource` interface outlines standard functionality of XA data sources. An XA data source is a factory for XA connections. Oracle JDBC implements the XADataSource interface though the OracleXADatasource class. The `getConnection( )` method of the OracleXADatasource class returns an XA connection to the underlying data source. In SQLJ, connections to the database can be obtained through the `DefaultContext` class or the `ConnectionContext` class. For multiple connections that use different SQL entities, it is advantageous to use connection context declarations to define additional connection context classes.

The code snippet shows how to create an XADatasource first and then a JDBC connection from the datasource through the following steps:

- Start XA Resource1
- Start XA Resource2
- Perform DML operations with the first Connection object
- End XA Resource1

- End XA Resource2
- Prepare Resource1
- Prepare Resource2
- Commit 1
- Commit 2

> **Note:** The following is not a complete example and contains only relevant codes to create and use an XADatasource.

Example: Creating an XADatasource and using it to create a JDBC connection

```
import javax.sql.*;
import javax.transaction.*;
import javax.transaction.xa.*;
...
import oracle.jdbc.driver.*;
import oracle.jdbc.xa.OracleXid;
import oracle.jdbc.xa.OracleXAException;
import oracle.jdbc.xa.client.*;
……
#sql context MyContext;
#sql iterator Iterator2 (String job_id, String job_title);
#sql iterator Iterator3 (String region_id, String region_name);
……
class XA3mod{
public static void main (String args [])throws SQLException{
try{
/*create an XADataSource instance*/
OracleXADataSource oxds = new OracleXADataSource();
oxds.setURL(url);
oxds.setUser("hr");
oxds.setPassword("hr");
/*get an XA connection to the underlying data source*/
javax.sql.XAConnection pc1  = oxds.getXAConnection();
/*use the same data source */
javax.sql.XAConnection pc2  = oxds.getXAConnection();
/*get the Physical Connections*/
java.sql.Connection conn1 = pc1.getConnection();
java.sql.Connection conn2 = pc2.getConnection();
/*an application may access data through multiple database connections. Each
database connection is enlisted
with the transaction manager as a transactional resource. The transaction manager
obtains an XAResource
 for each connection participating in a global transaction */
XAResource oxar1 = pc1.getXAResource();
XAResource oxar2 = pc2.getXAResource();
/*create the Xids With the Same Global Ids. The Xid interface is a Java mapping of
the X/Open transaction
identifier XID structure*/
Xid xid1 = createXid(1);
Xid xid2 = createXid(2);
/*start the Resources. This would start work on behalf of a transaction branch
specified in xid1 and xid2.
The transaction manager uses the start method to associate the global transaction
with the resource,
and it uses the end method to disassociate the transaction from the resource */
oxar1.start (xid1, XAResource.TMNOFLAGS);
```

```
oxar2.start (xid2, XAResource.TMNOFLAGS);
/*Do something with conn1 */
DoSomeWork (conn1);
/*END both the branches */
xar1.end(xid1, XAResource.TMSUCCESS);
xar2.end(xid2, XAResource.TMSUCCESS);
/*Prepare the RMs. The Oracle XA library interface follows the two-phase commit
protocol. Preparing the transactions
is the first step in this protocol. The two phase commit protocol is explained in
detail in the glossary section. */
int prp1 =  oxar1.prepare (xid1);
int prp2 =  oxar2.prepare (xid2);
boolean do_commit = true;
if(!((prp1==XAResource.XA_OK)||(prp1==XAResource.XA_RDONLY)))
            do_commit = false;
if(!((prp2==XAResource.XA_OK)||(prp2==XAResource.XA_RDONLY)))
            do_commit = false;
/*issue a commit on all transactions only if all the transactions completed
without and errors. Rollback even
if a single transaction failed.*/
if (prp1 == XAResource.XA_OK)
           if (do_commit)
           oxar1.commit (xid1, false);
           else
              oxar1.rollback (xid1);
if (prp2 == XAResource.XA_OK)
           if (do_commit)
              oxar2.commit (xid2, false);
           else
              oxar2.rollback (xid2);
/* close connections */
conn1.close(); conn1 = null;
conn2.close(); conn2 = null;
pc1.close();   pc1 = null;
pc2.close();   pc2 = null;
} catch (XAException xae){
if (xae instanceof OracleXAException) {
System.out.println("XA Error is " + ((OracleXAException)xae).getXAError());
System.out.println("SQL Error is " +((OracleXAException)xae).getOracleError());
}
}
} //end class
```

The following examples explain the different SQLJ methods that can accept a JDBC connection obtained from an OracleXADatasource.

Using Oracle.connect( ) method with a JDBC connection obtained from an XA Datasource:

```
private static void DoSomeWork (java.sql.Connection conn) throws SQLException{
String chr = "XA_CERT";
Oracle.connect(conn);
#sql  {insert into xa_test values (1,:chr)};
try{
     Iterator3 iter = null;
     #sql iter = {SELECT id,name FROM xa_test};
     while (iter.next( )){
     System.out.print(iter.id());
     System.out.print(" ");
     System.out.println(iter.name());
```

```
     }
}
catch (Exception e){
     System.out.println(e);
     e.printStackTrace();
     }
}
```

Using Oracle.getConnection( ) method with a JDBC connection obtained from an XA Datasource

```
private static void DoSomeWork (java.sql.Connection conn) throws SQLException{
String chr = "XA_CERT";
DefaultContext  ctx = Oracle.getConnection(conn);
#sql [ctx]  {insert into xa_test values (1,:chr)};
try{
    Iterator3 iter = null;
    #sql [ctx] iter = {SELECT id,name FROM xa_test};
    while (iter.next( )){
    System.out.print(iter.id());
    System.out.print(" ");
    System.out.println(iter.name());
    }
}
catch (Exception e){
    System.out.println(e);
    e.printStackTrace();
}
}
```

Using DefaultContext Constructor with a JDBC connection obtained from an XA Datasource

```
private static void DoSomeWork (java.sql.Connection conn) throws SQLException{
String chr = "XA_CERT";
DefaultContext  ctx = new DefaultContext(conn)
#sql [ctx]  {insert into xa_test values (1,:chr)};
try{
Iterator3 iter = null;
#sql [ctx] iter = {SELECT id,name FROM xa_test};
while (iter.next( )){
System.out.print(iter.id());
System.out.print(" ");
System.out.println(iter.name());
}
}
catch (Exception e){
System.out.println(e);
e.printStackTrace();
}
}
```

Using DefaultContext Constructor by passing a ConnectionContext to it. The ConnectionContext is created through the JDBC connection obtained from an XA Datasource

```
private static void DoSomeWork (java.sql.Connection conn) throws SQLException{
String chr = "XA_CERT";
```

```
MyContext myctx1= new MyContext (conn);
DefaultContext  ctx = new DefaultContext(myctx1);
#sql [ctx]  {insert into xa_test values (1,:chr)};
try{
    Iterator3 iter = null;
    #sql [ctx] iter = {SELECT id,name FROM xa_test};
    while (iter.next( )){
    System.out.print(iter.id());
    System.out.print(" ");
    System.out.println(iter.name());
    }
}
catch (Exception e){
    System.out.println(e);
    e.printStackTrace();
}
}
```

Using Oracle.connect( ) method by passing a ConnectionContext to it. The ConnectionContext is created through the JDBC connection obtained from an XA Datasource

```
private static void DoSomeWork (java.sql.Connection conn) throws SQLException{
String chr = "XA_CERT";
MyContext myctx1= new MyContext (conn);
Oracle.connect(myctx1);
#sql {insert into xa_test values (1,:chr)};
try{
    Iterator3 iter = null;
    #sql iter = {SELECT id,name FROM xa_test};
    while (iter.next( )){
    System.out.print(iter.id());
    System.out.print(" ");
    System.out.println(iter.name());
    }
}
catch (Exception e){
    System.out.println(e);
    e.printStackTrace();
    }
}
```

Using Oracle. getConnection( ) method by passing a ConnectionContext to it. The ConnectionContext is created through the JDBC connection obtained from an XA Datasource

```
private static void DoSomeWork (java.sql.Connection conn) throws SQLException{
String chr = "XA_CERT";
MyContext myctx1= new MyContext (conn);
DefaultContext  ctx = Oracle.getConnection(myctx1);
#sql [ctx]  {insert into xa_test values (1,:chr)};
try{
    Iterator3 iter = null;
    #sql [ctx] iter = {SELECT id,name FROM xa_test};
    while (iter.next( )){
    System.out.print(iter.id());
    System.out.print(" ");
    System.out.println(iter.name());
    }
```

```
}
catch (Exception e){
    System.out.println(e);
    e.printStackTrace();
}
}
```

The setDefaultContext( ) method of the DefaultContext class can also be used to set a context which was created through the JDBC connection obtained from an XA Datasource

```
DefaultContext.setDefaultContext(ctx);
```

Using the ConnectionContext constructor by passing a JDBC connection obtained from an XA Datasource

```
private static void DoSomeWork (java.sql.Connection conn) throws SQLException{
String chr = "XA_CERT";
MyContext myctx1= new MyContext (conn);
#sql [myctx1]  {insert into xa_test values (1,:chr)};
try{
    Iterator3 iter = null;
    #sql [myctx1] iter = {SELECT id,name FROM xa_test};
    while (iter.next( )){
    System.out.print(iter.id());
    System.out.print(" ");
    System.out.println(iter.name());
    }
}
catch (Exception e){
    System.out.println(e);
    e.printStackTrace();
}
}
```

Using the ConnectionContext constructor by passing a ConnectionContext to it. The ConnectionContext is created through the JDBC connection obtained from an XA Datasource

```
private static void DoSomeWork (java.sql.Connection conn) throws SQLException{
String chr = "XA_CERT";
MyContext myctx= new MyContext (conn);
MyContext myctx1= new MyContext (myctx);
#sql [myctx1]  {insert into xa_test values (1,:chr)};
try{
    Iterator3 iter = null;
    #sql [myctx1] iter = {SELECT id,name FROM xa_test};
    while (iter.next( )){
    System.out.print(iter.id());
    System.out.print(" ");
    System.out.println(iter.name());
    }
}
catch (Exception e){
    System.out.println(e);
    e.printStackTrace();
}
}
```

Execution Contexts

An execution context is an instance of the `sqlj.runtime.ExecutionContext` class and provides a context in which SQL operations are executed. An execution context instance is associated either implicitly or explicitly with each SQL operation in your SQLJ application.

The `ExecutionContext` class contains methods for the following features:

- Execution control operations modify the semantics of subsequent SQL operations.
- Execution status operations describe the results of the most recent SQL operation.
- Execution cancellation operations terminate the SQL operation that is currently executing.
- Update-batching operations enable and disable update batching, set the batch limit, and get update counts.
- Savepoint operations set a savepoint, roll back to a savepoint, and release a savepoint.
- Closure operations close the execution context instance to avoid resource leakage.

> **Note:** There is only one execution context class, unlike connection context classes where you declare additional classes as desired. Every execution context is an instance of the `ExecutionContext` class. So while the term connection context usually refers to a class that you have declared, the term execution context always refers to an instance of the `ExecutionContext` class. This document specifies connection context class, connection context instance, and execution context instance to avoid confusion.

This section covers the following topics:

Relation of Execution Contexts to Connection Contexts

Each connection context instance implicitly has its own default execution context instance, which you can retrieve by using the `getExecutionContext()` method of the connection context instance.

A single execution context instance will be sufficient for a connection context instance except in the following circumstances:

- You are using multiple threads with a single connection context instance.

  When using multithreading, each thread must have its own execution context instance.

- You want to use different SQL execution control operations on different SQLJ statements that use the same connection context instance.
- You want to retain different sets of SQL status information from multiple SQL operations that use the same connection context instance.

  As you execute successive SQL operations that use the same execution context instance, the status information from each operation overwrites the status information from the previous operation.

Although execution context instances might appear to be associated with connection context instances (given that each connection context instance has a default execution context instance, and you can specify a connection context instance and an execution context instance together for a particular SQLJ statement), they actually operate independently. You can use different execution context instances in statements that use the same connection context instance, and vice versa.

For example, it is useful to use multiple execution context instances with a single connection context instance if you use multithreading, with a separate execution context instance for each thread. And you can use multiple connection context instances with a single explicit execution context instance if your program is single-threaded and you want the same set of SQL control parameters to apply to all the connection context instances.

See Also: "Execution Context Methods" on page 7-26

To use different execution context instances with a single connection context instance, you must create additional instances of the `ExecutionContext` class and specify them appropriately with your SQLJ statements.

Creating and Specifying Execution Context Instances

To use an execution context instance other than the default with a given connection context instance, you must construct another execution context instance. There are no input parameters for the `ExectionContext` constructor. For example:

```
ExecutionContext myExecCtx = new ExecutionContext();
```

You can then specify this execution context instance for use with any particular SQLJ statement, much as you would specify a connection context instance. The general syntax is as follows:

```
#sql [<conn_context><, ><exec_context>] { SQL operation };
```

For example, if you also declare and instantiate a connection context class, `MyConnCtxClass`, and create an instance, `myConnCtx`, then you can use the following statement:

```
#sql [myConnCtx, myExecCtx] { DELETE FROM emp WHERE sal > 30000 };
```

You can subsequently use different execution context instances with `myConnCtx` or different connection context instances with `myExecCtx`.

You can optionally specify an execution context instance while using the default connection context instance, as follows:

```
#sql [myExecCtx] { DELETE FROM emp WHERE sal > 30000 };
```

Note:

- If you specify a connection context instance without an execution context instance, then the default execution context instance of that connection context instance is used.
- If you specify an execution context instance without a connection context instance, then the execution context instance is used with the default connection context instance of your application.
- If you specify no connection context instance and no execution context instance, then SQLJ uses the default connection and its default execution context instance.

Execution Context Synchronization

`ExecutionContext` methods are all `synchronized` methods. Therefore, for ISO standard code generation, anytime a statement tries to use an execution context instance already in use, the second statement will be blocked until the first statement completes.

In a client application, this typically involves multithreading situations. A thread that tries to use an execution context instance currently in use by another thread will be blocked. To avoid such blockage, you must specify a separate execution context instance for each thread that you use.

See Also: "Multithreading in SQLJ" on page 7-31

The preceding discussion does not apply for default Oracle-specific code generation. For performance reasons, SQLJ performs no additional synchronization against `ExecutionContext` instances for Oracle-specific generated code. Therefore, you are responsible for ensuring that the same execution context instance will not be used by more than one thread. If multiple threads use the same execution context, then your application, rather than blocking, will experience errors such as incorrect results or `NullPointer` exceptions.

Another exception to the discussion is for recursion, which is encountered only in the server. Multiple SQLJ statements in the same thread are allowed to simultaneously use the same execution context instance if this situation results from recursive calls. An example of this is where a SQLJ stored procedure or function has a call to another SQLJ stored procedure or function. If both use the default execution context instance, as is typical, then the SQLJ statements in the second procedure will use this execution context while the SQLJ call statement from the first procedure is also still using it. This is allowed.

See Also: "Recursive SQLJ Calls in the Server" on page 11-18

Execution Context Methods

The following sections list public methods of the `ExecutionContext` class and provide an example:

- Status Methods
- Control Methods
- Cancellation Method

Status Methods

Use the following methods of an execution context instance to obtain status information about the most recent SQL operation that completed using that instance:

- `SQLWarning getWarnings()`: Returns a `java.sql.SQLWarning` object containing the first warning reported by the most recent SQL operation that completed using this execution context instance. Warnings are returned in a chain. Use the `getWarnings()` method of the execution context instance to get the first warning, then use the `getNextWarning()` method of each `SQLWarning` object to get the next warning. The chain contains all warnings generated during the execution of the SQL operation.
- `int getUpdateCount()`: Except when update batching is enabled, this returns an `int` value specifying the number of rows updated by the last SQL operation that completed using this execution context instance. Zero (`0`) is returned if the last SQL operation was not a data manipulation language (DML) statement. The `QUERY_COUNT` constant is returned, if the last SQL operation produced an iterator or result set. The `EXCEPTION_COUNT` constant is returned, if the last SQL operation terminated before completing execution or if no operation has yet been attempted using this execution context instance.

  For batch-enabled applications, the value returned by `getUpdateCount()` would be one of several batch-related constant values: `NEW_BATCH_COUNT`, `ADD_BATCH_COUNT`, or `EXEC_BATCH_COUNT`.

  See Also: "Execution Context Update Counts" on page 10-12

Control Methods

Use the following methods of an execution context instance to control the operation of future SQL operations executed using that instance (operations that have not yet started):

- `int getMaxFieldSize()`: Returns an `int` value specifying the maximum amount of data (in bytes) that would be returned from a SQL operation subsequently, using this execution context instance. This applies only to columns of the `BINARY`, `VARBINARY`, `LONGVARBINARY`, `CHAR`, `VARCHAR`, or `LONGVARCHAR` type.

  By default this parameter is set to `0`, meaning there is no size limit.
- `setMaxFieldSize(int)`: Takes an `int` value as input to modify the maximum field-size.
- `int getMaxRows()`: Returns an `int` value specifying the maximum number of rows that can be contained by any SQLJ iterator or JDBC result set created using this execution context instance. If the limit is exceeded, then the excess rows are silently dropped without any error report or warning.

  By default, this parameter is set to `0`, meaning there is no row limit.
- `setMaxRows(int)`: Takes an `int` value as input to modify the maximum row value.

- `int getQueryTimeout()`: Returns an `int` value specifying the timeout interval, in seconds, for any SQL operation that uses this execution context instance. If a SQL operation exceeds this limit, then a SQL exception is thrown.

  By default, this parameter is set to `0`, meaning there is no query timeout limit.

- `setQueryTimeout(int)`: Takes an `int` value as input to modify the query timeout limit.

- `int getFetchSize()`: Retrieves the number of rows that is the current fetch size for iterator objects generated from this `ExecutionContext` object. If this `ExecutionContext` object has not set a fetch size by calling `setFetchSize()`, then the value returned is `0`. If this `ExecutionContext` object has set a non-negative fetch size by calling the method `setFetchSize()`, then the return value is the fetch size specified on `setFetchSize()`.

- `setFetchSize(int)`: Gives the SQLJ run time a hint as to the number of rows that should be fetched when more rows are needed. The number of rows specified affects only iterator objects created using this `ExecutionContext` object. Specifying zero means that an implementation-dependent default value will be used for the fetch size.

- `int getFetchDirection()`: Retrieves the default direction for fetching data, for scrollable iterator objects that are generated from this `ExecutionContext` object. If this `ExecutionContext` object has not set a fetch direction by calling the method `setFetchDirection()`, then the return value is `FETCH_FORWARD`.

- `setFetchDirection(int)`: Gives the SQLJ run time a hint as to the direction in which rows of scrollable iterator objects are processed. The hint applies only to scrollable iterator objects that are created using this `ExecutionContext` object. The default value is:

  ```
  sqlj.runtime.ResultSetIterator.FETCH_FORWARD.
  ```

  This method throws a `SQLException` if the given direction is not one of `FETCH_FORWARD`, `FETCH_REVERSE`, or `FETCH_UNKNOWN` (`int` constants).

Cancellation Method

Use the following method to cancel SQL operations in a multithreading environment or to cancel a pending statement batch if update batching is enabled:

- `cancel()`: In a multithreading environment, use this method in one thread to cancel a SQL operation currently executing in another thread. It cancels the most recent operation that has started but not completed, using this execution context instance. This method has no effect if no statement is currently being executed using this execution context instance.

  In a batch-enabled environment, use this to cancel a pending statement batch. The batch is emptied, and none of the statements in the batch are executed. After you cancel a batch, the next batchable statement encountered will be added to a new batch.

  See Also: "Canceling a Batch" on page 10-12

Update Batching Methods

Use the following methods to control update batching if you want your application to use that performance enhancement feature:

- `int[] executeBatch()`: Executes the pending statement batch, returning an array of `int` update counts.

> **See Also:** "Execution Context Update Counts" on page 10-12, "Explicit and Implicit Batch Execution" on page 10-10, and "Error Conditions During Batch Execution" on page 10-16

- `int getBatchLimit()`: Returns an `int` value indicating the current batch limit. If there is a batch limit, then a pending batch is implicitly executed once it contains that number of statements.

  By default, the batch limit is set to the `ExecutionContext` static constant value `UNLIMITED_BATCH`, meaning there is no batch limit.

  > **See Also:** "Setting a Batch Limit" on page 10-13

- `int[] getBatchUpdateCounts()`: Returns an array of `int` update counts for the last batch executed. This method is useful in situations where the batch was executed implicitly.

  > **See Also:** "Execution Context Update Counts" on page 10-12

- `boolean isBatching()`: Returns a boolean value indicating whether update batching is enabled.

  This does not indicate whether there is currently a pending batch, but you can use the `getUpdateCount()` method to see whether a batch has been newly created, added to, or executed.

- `setBatching(boolean)`: Takes a boolean value to enable update batching.

  Update batching is disabled by default.

  > **See Also:** "Enabling and Disabling Statement Caching (Oracle-Specific Code)" on page 10-5

- `setBatchLimit(int)`: Takes a positive, nonzero `int` value as input to set the current batch limit. Two special values you can assign are `UNLIMITED_BATCH`, which means there is no limit, and `AUTO_BATCH`, which lets the SQLJ run time dynamically determine a batch limit.

  > **See Also:** "Update Batching" on page 10-9

Savepoint Methods

The Oracle SQLJ implementation supports JDBC 3.0 savepoints. Savepoints are stored in the `ExecutionContext` instance, and the following public methods exist to support the SQLJ savepoint statements:

- `Object oracleSetSavepoint(ConnectionContextImpl, String)`

  Register a savepoint and return the savepoint as an `Object` instance. This method takes the connection context as an instance of the `sqlj.runtime.ref.ConnectionContextImpl` class and a string that specifies the savepoint name.

  The Oracle SQLJ implementation instantiates a savepoint as an instance of the `oracle.jdbc.OracleSavepoint` class, which extends the `java.sql.Savepoint` interface. The `Savepoint` interface requires Java Development Kit (JDK) 1.4. Oracle uses an `Object` instance for this method to avoid the JDK 1.4 requirement.

- `void oracleRollbackToSavepoint (ConnectionContextImpl, Object)`

  Roll back changes to the specified savepoint. This method takes the connection context as an instance of `ConnectionContextImpl` and the savepoint as an `Object` instance.

- `void oracleReleaseSavepoint(ConnectionContextImpl, Object)`

  Release the specified savepoint. This method takes the connection context as an instance of `ConnectionContextImpl` and the savepoint as an `Object` instance.

You will generally use SQLJ savepoint statements instead of using these methods directly.

Close Method

The Oracle SQLJ implementation provides extended functionality with a `close()` method for the `ExecutionContext` class:

- `close()`: To avoid resource leakage, use this method if the following circumstances are all true:
  - You are using the Oracle-specific code generation.
  - You explicitly created and used the `ExecutionContext` instance, instead of using the default instance available through the connection context instance.
  - You are *not* issuing SQLJ rollback or commit statements explicitly using the `ExecutionContext` instance:

    ```
    #sql [ec] { COMMIT };
    #sql [ec] { ROLLBACK };
    ```

  - You are *not* calling `executeBatch()` on the `ExecutionContext` instance.

  Under this set of circumstances, a batchable statement might remain open on the `ExecutionContext` instance and over time you may run out of database cursors. To avoid this, use the `close()` method as in the following example:

```
Execution Context ec = new ExecutionContext();
...
try {
   ...
   #sql [ec] { SQL operation };
   ...
} finally { ec.close(); }
```

> **Note:** When an execution context instance is associated with a connection context instance, instead of being declared explicitly, then closing the connection context instance, with or without closing the underlying JDBC connection, will automatically close any statement remaining on the execution context instance.

Example: Using ExecutionContext Methods

The following code demonstrates the use of some `ExecutionContext` methods:

```
ExecutionContext execCtx =
   DefaultContext.getDefaultContext().getExecutionContext();
```

```
// Wait only 3 seconds for operations to complete
execCtx.setQueryTimeout(3);

// delete using execution context of default connection context
#sql { DELETE FROM emp WHERE sal > 10000 };

System.out.println
     ("removed " + execCtx.getUpdateCount() + " employees");
```

Relation of Execution Contexts to Multithreading

Do not use multiple threads with a single execution context. If you do, and two SQLJ statements try to use the same execution context simultaneously, then the second statement will be blocked until the first statement completes. Furthermore, status information from the first operation will likely be overwritten before it can be retrieved.

Therefore, if you are using multiple threads with a single connection context instance, then you should take the following steps:

1. Instantiate a unique execution context instance for use with each thread.
2. Specify execution contexts with your `#sql` statements so that each thread uses its own execution context.

If you are using a different connection context instance with each thread, then no instantiation and specification of execution context instances is necessary, because each connection context instance implicitly has its own default execution context instance.

Note: For performance reasons, SQLJ performs no additional synchronization against `ExecutionContext` instances for Oracle-specific generated code. Therefore, you are responsible for ensuring that the same execution context instance will not be used by more than one thread. If multiple threads use the same execution context, then your application, rather than blocking, will experience errors such as incorrect results or `NullPointer` exceptions.

Multithreading in SQLJ

This section discusses SQLJ support and requirements for multithreading and the relation between multithreading and execution context instances.

You can use SQLJ in writing multithreaded applications. However, any use of multithreading in your SQLJ application is subject to the limitations of your JDBC driver or proprietary database access vehicle. This includes any synchronization limitations.

You are required to use a different execution context instance for each thread. You can accomplish this in one of two ways:

- Specify connection context instances for your SQLJ statements such that a different connection context instance is used for each thread. Each connection context instance automatically has its own default execution context instance.
- If you are using the same connection context instance with multiple threads, then declare additional execution context instances and specify execution context

instances for your SQLJ statements such that a different execution context instance is used for each thread.

> **See Also:** "Specifying Connection Context Instances and Execution Context Instances" on page 4-10

If you are using one of the Oracle JDBC drivers, then multiple threads can use the same connection context instance, if desired, as long as different execution context instances are specified and there are no synchronization requirements directly visible to you. However, note that data access is sequential. Only one thread is accessing data at any given time. Synchronization refers to the control flow of the various stages of the SQL operations executing through your threads. For example, each statement can bind input parameters, then execute, and then bind output parameters. With some JDBC drivers, special care must be taken not to intermingle these stages.

For ISO standard code generation, if a thread attempts to execute a SQL operation that uses an execution context that is in use by another operation, then the thread is blocked until the current operation completes. If an execution context were shared between threads, then the results of a SQL operation performed by one thread would be visible in the other thread. If both threads were executing SQL operations, then a race condition might occur. The results of an execution in one thread might be overwritten by the results of an execution in the other thread before the first thread had processed the original results. This is why multiple threads are not allowed to share an execution context instance.

> **Note:** The preceding paragraph does not apply if you use default Oracle-specific code generation. For performance reasons, SQLJ performs no additional synchronization against `ExecutionContext` instances for Oracle-specific generated code. Therefore, you are responsible for ensuring that the same execution context instance will not be used by more than one thread. If multiple threads use the same execution context, then your application, rather than blocking, will experience errors such as incorrect results or `NullPointer` exceptions.

Multithreading: MultiThreadDemo.sqlj

The following is an example of a SQLJ application using multithreading. A `ROLLBACK` operation is executed before closing the connection, so the data is not permanently altered.

```
import java.sql.SQLException;
import java.util.Random;
import sqlj.runtime.ExecutionContext;
import oracle.sqlj.runtime.Oracle;
/**
  Each instance of MultiThreadDemo is a thread that gives all employees
  a raise of some ammount when run.  The main program creates two such
  instances and computes the net raise after both threads have completed.
  **/
class MultiThreadDemo extends Thread
{
  double raise;
  static Random randomizer = new Random();

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

```
    try {
      // set the default connection to the URL, user, and password
      // specified in your connect.properties file
      Oracle.connect(MultiThreadDemo.class, "connect.properties");
      double avgStart = calcAvgSal();
      MultiThreadDemo t1 = new MultiThreadDemo(250.50);
      MultiThreadDemo t2 = new MultiThreadDemo(150.50);
      t1.start();
      t2.start();
      t1.join();
      t2.join();
      double avgEnd = calcAvgSal();
      System.out.println("average salary change: " + (avgEnd - avgStart));
    } catch (Exception e) {
      System.err.println("Error running the example: " + e);
    }
    try { #sql { ROLLBACK }; Oracle.close(); } catch (SQLException e) { }
  }
  static double calcAvgSal() throws SQLException
  {
    double avg;
    #sql { SELECT AVG(sal) INTO :avg FROM emp };
    return avg;
  }
  MultiThreadDemo(double raise)
  {
    this.raise = raise;
  }
  public void run()
  {
    // Since all threads will be using the same default connection
    // context, each run uses an explicit execution context instance to
    // avoid conflict during execution
    try {
      delay();
      ExecutionContext execCtx = new ExecutionContext();
      #sql [execCtx] { UPDATE EMP SET sal = sal + :raise };
      int updateCount = execCtx.getUpdateCount();
      System.out.println("Gave raise of " + raise + " to " +
                         updateCount + " employees");
    } catch (SQLException e) {
      System.err.println("error updating employees: " + e);
    }
  }
  // delay is used to introduce some randomness into the execution order
  private void delay()
  {
    try {
      sleep((long)Math.abs(randomizer.nextInt()/10000000));
    } catch (InterruptedException e) {}
  }
}
```

Iterator Class Implementation and Advanced Functionality

This section discusses how iterator classes are implemented and what additional functionality is available beyond the essential methods. The following topics are covered:

Implementation and Functionality of Iterator Classes

Any named iterator class you declare will be generated by the SQLJ translator to implement the `sqlj.runtime.NamedIterator` interface. Classes implementing the `NamedIterator` interface have functionality that maps iterator columns to database columns by name, not by position.

Any positional iterator class you declare will be generated by the SQLJ translator to implement the `sqlj.runtime.PositionedIterator` interface. Classes implementing the `PositionedIterator` interface have functionality that maps iterator columns to database columns by position, not by name.

Both the `NamedIterator` interface and the `PositionedIterator` interface, and therefore all generated SQLJ iterator classes as well, implement or extend the `sqlj.runtime.ResultSetIterator` interface.

The `ResultSetIterator` interface specifies the following methods for all SQLJ iterators:

- `close()`: Closes the iterator.
- `ResultSet getResultSet()`: Extracts the underlying JDBC result set from the iterator.
- `boolean isClosed()`: Determines if the iterator has been closed.
- `boolean next()`: Moves to the next row of the iterator, returning `true` if there is a valid next row to go to.

The `PositionedIterator` interface adds the following method specification for positional iterators:

- `boolean endFetch()`: Determines if you have reached the last row of a positional iterator.

Use the `next()` method to advance through the rows of a named iterator and accessor methods to retrieve the data. The SQLJ generation of a named iterator class defines an accessor method for each iterator column, where each method name is identical to the corresponding column name. For example, if you declare a `name` column, then a `name()` method will be generated.

Use a `FETCH INTO` statement together with the `endFetch()` method to advance through the rows of a positional iterator and retrieve the data. A `FETCH INTO` statement implicitly calls the `next()` method. Do not explicitly use the `next()` method in a positional iterator unless you are using the special `FETCH CURRENT` syntax. The `FETCH INTO` statement also implicitly calls accessor methods that are named according to iterator column numbers. The SQLJ generation of a positional iterator class defines an accessor method for each iterator column, where each method name corresponds to the column position.

See Also: "FETCH CURRENT Syntax: from JDBC Result Sets to SQLJ Iterators" on page 7-40

Use the `close()` method to close any iterator once you are done with it. The `getResultSet()` method is central to SQLJ-JDBC interoperability.

See Also: "SQLJ Iterator and JDBC Result Set Interoperability" on page 7-48

Note: Alternatively, you can use a `ResultSetIterator` instance or a `ScrollableResultSetIterator` instance directly as a weakly typed iterator. (`ScrollableResultSetIterator` extends `ResultSetIterator`.) This is convenient if you are interested only in converting it to a JDBC result set and you do not need named or positional iterator functionality. You can also access it through SQLJ `FETCH CURRENT` syntax.

Using the IMPLEMENTS Clause in Iterator Declarations

There may be situations where it will be useful to implement an interface in your iterator declaration. For example, you may have an iterator class where you want to restrict access to one or more columns. A named iterator class generated by SQLJ has an accessor method for each column in the iterator. If you want to restrict access to certain columns, you can create an interface with only a subset of the accessor methods, then expose instances of the interface type to the user instead of exposing instances of the iterator class type.

For example, assume you are creating a named iterator of employee data, with columns `ENAME` (employee name), `EMPNO` (employee number), and `SAL` (salary). Accomplish this as follows:

```
#sql iterator EmpIter (String ename, int empno, float sal);
```

This generates a class `EmpIter` with `ename()`, `empno()`, and `sal()` accessor methods.

Assume, though, that you want to prevent access to the `SAL` column. You can create an `EmpIterIntfc` interface that has `ename()` and `empno()` methods, but no `sal()` method. Then you can use the following iterator declaration instead of the preceding declaration (presuming `EmpIterIntfc` is in the `mypackage` package):

```
#sql iterator EmpIter implements mypackage.EmpIterIntfc
     (String emame, int empno, float sal);
```

Then if you code your application so that users can access data only through `EmpIterIntfc` instances, then they will not have access to the `SAL` column.

Support for Extending Iterator Classes

SQLJ supports the ability to extend iterator classes. This feature can be very useful in allowing you to add functionality to your queries and query results.

The one key requirement of an iterator subclass is that you must supply a public constructor that takes an instance of `sqlj.runtime.RTResultSet` as input. The SQLJ run time will call this constructor in assigning query results to an instance of your subclass. Beyond that, you provide functionality as you choose.

You can continue to use functionality of the original iterator class (the superclass of your subclass). For example, you can advance through query results by calling the `super.next()` method.

Result Set Iterators

You may have situations where you do not require the strongly typed functionality of a SQLJ iterator.

For such circumstances, you can directly use instances of the `sqlj.runtime.ResultSetIterator` type to receive query data, so that you are not required to declare a named or positional iterator class. Alternatively, you can use the `sqlj.runtime.ScrollableResultSetIterator` type, which extends `ResultSetIterator`. This enables you to use SQLJ scrollable iterator functionality. In using a result set iterator instead of a strongly typed iterator, you are trading the strong type-checking of the SQLJ `SELECT` operation for the convenience of not having to declare an iterator class.

The `ResultSetIterator` interface underlies all named and positional iterator classes and specifies the `getResultSet()` and `close()` methods. If you want to use SQLJ to process a result set iterator instance, then use a `ScrollableResultSetIterator` instance and the `FETCH CURRENT` syntax.

> **See Also:** "FETCH CURRENT Syntax: from JDBC Result Sets to SQLJ Iterators" on page 7-40

If you want to use JDBC to process a result set iterator instance, you can use its `getResultSet()` method and then process the underlying result set that you retrieve. If you process a result set iterator through its underlying result set, you should close the result set iterator, not the result set, when you are finished. Closing the result set iterator will also close the result set, but closing the result set will not close the result set iterator.

> **See Also:** "Using and Converting Weakly Typed Iterators (ResultSetIterator)" on page 7-50

Note: The Oracle SQLJ implementation supports result set iterators for use as host expressions and to represent cursors in `FETCH` statements. This functionality was not supported prior to Oracle9*i* Database.

Scrollable Iterators

The ISO standard for SQLJ supports scrollable iterators, with functionality being patterned after the JDBC 2.0 specification for scrollable JDBC result sets. The Oracle SQLJ implementation supports this functionality.

> **See Also:** *Oracle Database JDBC Developer's Guide and Reference*

Declaring Scrollable Iterators

To characterize an iterator as scrollable, add the following clause to the iterator declaration:

```
implements sqlj.runtime.Scrollable
```

This instructs the SQLJ translator to generate an iterator that implements the `Scrollable` interface. Following is an example of a declaration of a named, scrollable iterator:

```
#sql public static MyScrIter implements sqlj.runtime.Scrollable
                             (String ename, int empno);
```

The code that the SQLJ translator generates for the `MyScrIter` class will automatically support all the methods of the `Scrollable` interface.

Scrollable Iterator Sensitivity

You can declare scrollable iterators, like scrollable result sets, to have sensitivity to changes to the underlying data. By default, scrollable iterators in the Oracle SQLJ implementation have a `sensitivity` setting of `INSENSITIVE`, meaning they do not detect any such changes in the underlying data. However, you can use a `with` clause to alter this setting. The following example expands an earlier example to specify sensitivity:

```
#sql public static MyScrIter implements sqlj.runtime.Scrollable
                            with (sensitivity=SENSITIVE)
                            (String ename, int empno);
```

Note: The `implements` clause must precede the `with` clause.

The SQLJ standard also allows a setting of `ASENSITIVE`, but in the Oracle implementation this is undefined. Setting `sensitivity` to `ASENSITIVE` results instead in the default setting, `INSENSITIVE`, being used.

Given the preceding declaration, `MyScrIter` instances will be sensitive to data changes, subject to factors such as the fetch size window.

See Also: *Oracle Database JDBC Developer's Guide and Reference* for information about scrollable result sets

The Scrollable Interface

This section documents some key methods of the `sqlj.runtime.Scrollable` interface.

You can provide hints about the fetch direction to scrollable iterators. The following methods are defined on scrollable iterators as well as on execution contexts. Use an `ExecutionContext` instance to provide the default direction to be used in creation of scrollable iterators.

- `setFetchDirection(int)`: Gives the SQLJ run time a hint as to the direction in which rows are processed. The direction should be one of `sqlj.runtime.ResultSetIterator.FETCH_FORWARD`, `FETCH_REVERSE`, or `FETCH_UNKNOWN`.

  If you do not specify a value for the direction on the `ExecutionContext`, then `FETCH_FORWARD` will be used as a default.

- `int getFetchDirection()`: Retrieves the current direction for fetching rows of data (one of the integer constants described in the previous point).

There are also a number of scrollable iterator methods that will return information about the current position of the iterator object in the underlying result set. All these methods will return `false` whenever the result set underlying the iterator contains no rows:

- `boolean isBeforeFirst()`: Indicates whether the iterator object is before the first row in the result set.

- `boolean isFirst()`: Indicates whether the iterator object is on the first row of the result set.
- `boolean isLast()`: Indicates whether the iterator object is on the last row of the result set. Note that calling the `isLast()` method may be expensive, because the JDBC driver may have to fetch ahead one row to determine whether the current row is the last row in the result set.
- `boolean isAfterLast()`: Indicates whether the iterator object is after the last row in the result set.

Note: Additional methods for navigation, also defined in the `Scrollable` interface, are available as well.

Scrollable Named Iterators

Named iterators use navigation methods, defined in the `Scrollable` interface, to move through the rows of a result set. As described earlier in this manual, nonscrollable iterators have only the following method for navigation:

- `boolean next()`: Moves the iterator object to the next row in the result set.

See Also: "Using Named Iterators" on page 4-31

Additional navigation methods are available for scrollable named iterators. These methods function similarly to the `next()` method. In that they try to position the iterator on an actual row of the result set. They return `true` if the iterator ends up on a valid row and `false` if it does not. Additionally, if you attempt to position the iterator object before the first row or after the last row in the result set, this leaves the iterator object in the "before first" or "after last" position, respectively.

The following methods are supported:

- `boolean previous()`: Moves the iterator object to the previous row in the result set.
- `boolean first()`: Moves the iterator object to the first row in the result set.
- `boolean last()`: Moves the iterator object to the last row in the result set.
- `boolean absolute(int)`: Moves the iterator object to the given row number in the result set. The first row is row 1, the second is row 2, and so on. If the given row number is negative, then the iterator object moves to a row position relative to the end of the result set. For example, calling `absolute(-1)` positions the iterator object on the last row, `absolute(-2)` indicates the next-to-last row, and so on.
- `boolean relative(int)`: Moves the iterator object a relative number of rows, either positive or negative from the current position. Calling `relative(0)` is valid, but does not change the iterator position.
- `void beforeFirst()`: Moves the iterator object to the front of the result set, before the first row. This has no effect if the result set contains no rows.
- `void afterLast()`: Moves the iterator object to the end of the result set, after the last row. This has no effect if the result set contains no rows.

Note: The `beforeFirst()` and `afterLast()` methods return `void`, because they never place the iterator object on an actual row of the result set.

Scrollable Positional Iterators

General `FETCH` syntax for positional iterators was described earlier, in "Using Positional Iterators" on page 4-34. For example:

```
#sql { FETCH :iter INTO :x, :y, :z };
```

This is actually an abbreviated version of the following syntax:

```
#sql { FETCH NEXT FROM :iter INTO :x, :y, :z  };
```

This suggests the pattern for alternatively moving to the previous, first, or last row in the result set. Unfortunately, JDBC 2.0, after which the movement methods were modeled, uses `previous()`. The `FETCH` syntax, which is patterned after SQL, employs `PRIOR`. In case you forget this inconsistency, the Oracle Database 11*g* SQLJ translator will also accept `FETCH PREVIOUS`.

The syntax are:

```
#sql { FETCH PRIOR FROM :iter INTO :x, :y, :z  };
#sql { FETCH FIRST FROM :iter INTO :x, :y, :z  };
#sql { FETCH LAST FROM :iter INTO :x, :y, :z  };
```

There is also syntax to pass a numeric value for absolute or relative movements, to move to a particular (absolute) row, or to move forward or backward from the current position. The syntax are:

```
#sql { FETCH ABSOLUTE :n FROM :iter INTO :x, :y, :z  };
#sql { FETCH RELATIVE :n FROM :iter INTO :x, :y, :z  };
```

Note: In all of the preceding cases, the iterator `endFetch()` method returns `true` whenever the `FETCH` fails to move to a valid row and retrieve values.

Note that you *must* use a host expression to specify the movement. You cannot simply use a constant for the numeric value. Thus, instead of the following:

```
#sql { FETCH RELATIVE 0 FROM :iter INTO :x, :y, :z };
```

You must write the following:

```
#sql { FETCH RELATIVE :(0) FROM :iter INTO :x, :y, :z  };
```

Incidentally, this command leaves the position of the iterator unchanged. If the iterator is on a valid row, then the command just populates the variables.

Note: Alternatively, you can navigate through a scrollable positional iterator through a combination of the navigation methods and the `FETCH CURRENT` syntax.

FETCH CURRENT Syntax: from JDBC Result Sets to SQLJ Iterators

Consider a situation where you have an existing JDBC program that you want to rewrite in SQLJ with as little modification as possible.

Your JDBC result set will use only movement methods, such as `next()`, `previous()`, `absolute()`, and so on. You can immediately model this in SQLJ through a named iterator. However, this also implies that all columns of the SQL result set must have a proper name. In practice, many columns of the result set, if not all, will require introduction of alias names. This is unacceptable if the query text is to remain untouched.

The alternative, to avoid change to the query source, is to define a positional iterator type for the result set. However, this approach forces changes to the control-flow logic of the program. Consider the following JDBC code sample:

```
ResultSet rs = ... // execute ...query...;
while (rs.next()) {
   x := rs.getXxx(1); y:=rs.getXxx(2);
   ...process...
}
```

This translates along the following lines to SQLJ:

```
MyIter iter;
#sql iter = { ...query... };
while(true) {
   #sql { FETCH :iter INTO :x, :y };
   if (iter.endFetch()) break;
   ...process...
}
```

The transformations to the program logic will become even more difficult when considering arbitrary movements on scrollable iterators. Because positional iterators implement all the movement commands of named iterators, it is possible to exploit this and use `RELATIVE :(0)` to populate variables from the iterator:

```
MyIter iter;
#sql iter = { ...query... };
while (iter.next()) {
   #sql { FETCH RELATIVE :(0) FROM :iter INTO :x, :y };
   ...process...
}
```

Now, you can preserve both the original query and the original program logic. Unfortunately, there still is one drawback to this approach. The `MyIter` iterator type must implement the `Scrollable` interface, even if this property is not really needed. To address this, the Oracle SQLJ implementation supports the following syntax extension:

```
#sql { FETCH CURRENT FROM :iter INTO :x, :y, :z  };
```

Given this syntax, you can rewrite the JDBC example in SQLJ for scrollable as well as nonscrollable iterators:

```
AnyIterator ai;
#sql ai = { ...query... };
while (ai.next()) {
   #sql { FETCH CURRENT FROM :ai INTO :x, :y };
   ...process...
}
```

Scrollable Result Set Iterators

Support in the Oracle SQLJ implementation for weakly typed result set iterators includes a scrollable result set iterator type:

```
package sqlj.runtime;
public interface ScrollableResultSetIterator
                 extends ResultSetIterator
                 implements Scrollable
{ }
```

Because this type extends `sqlj.runtime.ResultSetIterator`, it supports the methods described in "Result Set Iterators" on page 7-36.

Because it also implements the `sqlj.runtime.Scrollable` interface, it supports the methods described in "The Scrollable Interface" on page 7-37 and "Scrollable Named Iterators" on page 7-38.

Furthermore, scrollable result set iterators support the `FETCH CURRENT` syntax described in "FETCH CURRENT Syntax: from JDBC Result Sets to SQLJ Iterators" on page 7-40.

Consider the following JDBC code:

```
Statement st = conn.createStatement("SELECT ename, empid FROM emp");
ResultSet rs = st.executeQuery();
while (rs.next()) {
   x = rs.getString(1);
   y = rs.getInt(2);
}
rs.close();
```

You can use a SQLJ result set iterator in writing equivalent code, as follows:

```
sqlj.runtime.ResultSetIterator rsi;
#sql rsi = { SELECT ename, empid FROM emp };
while (rsi.next()) {
   #sql { FETCH CURRENT FROM :rsi INTO :x, :y };
}
rsi.close();
```

To take advantage of scrollability features, you could also write the following code:

```
sqlj.runtime.ScrollableResultSetIterator srsi;
#sql srsi = { SELECT ename, empid FROM emp };
srsi.afterLast();
while (srsi.previous()) {
   #sql { FETCH CURRENT FROM :srsi INTO :x, :y };
}
srsi.close();
```

Advanced Transaction Control

SQLJ supports the SQL `SET TRANSACTION` statement to specify the access mode and isolation level of any given transaction. Standard SQLJ supports `READ ONLY` and `READ WRITE` access mode settings, but the Oracle JDBC implementation does not support `READ ONLY`. However, you can set permissions to have the same effect. Supported settings for isolation level are `SERIALIZABLE`, `READ COMMITTED`, `READ UNCOMMITTED`, and `REPEATABLE READ`. However, the Oracle SQL implementation does not support `READ UNCOMMITTED` or `REPEATABLE READ`.

READ WRITE is the default access mode in both standard SQL and the Oracle SQL implementation. READ COMMITTED is the default isolation level in the Oracle SQL implementation. SERIALIZABLE is the default in standard SQL.

The following sections provide details:

- SET TRANSACTION Syntax
- Access Mode Settings
- Isolation Level Settings
- Using JDBC Connection Class Methods

See Also: "Basic Transaction Control" on page 3-18

SET TRANSACTION Syntax

The SQLJ SET TRANSACTION statement has the following syntax:

```
#sql { SET TRANSACTION <access_mode>, <ISOLATION LEVEL isolation_level> };
```

If you do not specify a connection context instance, then the statement applies to the default connection. If you use SET TRANSACTION, then it must be the first statement in a transaction, preceding any DML statements. In other words, the first statement since your connection to the database or your most recent COMMIT or ROLLBACK.

In standard SQLJ, any access mode or isolation level you set will remain in effect across transactions until you explicitly reset it at the beginning of a subsequent transaction. In a standard SQLJ SET TRANSACTION statement, you can optionally specify the isolation level first or only the access mode or only the isolation level. Following are some examples:

```
#sql { SET TRANSACTION READ WRITE };

#sql { SET TRANSACTION ISOLATION LEVEL SERIALIZABLE };

#sql { SET TRANSACTION READ WRITE, ISOLATION LEVEL SERIALIZABLE };

#sql { SET TRANSACTION ISOLATION LEVEL READ COMMITTED, READ WRITE };
```

You can also specify a particular connection context instance for a SET TRANSACTION statement, as opposed to having it apply to the default connection:

```
#sql [myCtxt] { SET TRANSACTION ISOLATION LEVEL SERIALIZABLE };
```

Note that in SQLJ, both the access mode and the isolation level can be set in a single SET TRANSACTION statement. This is not true in other Oracle SQL tools, such as Server Manager or SQL*Plus, where a single statement can set one or the other, but not both.

Access Mode Settings

The READ WRITE and READ ONLY access mode settings, where supported, have the following functionality:

- READ WRITE (default): In a READ WRITE transaction, you are not allowed to update the database. SELECT, INSERT, UPDATE, and DELETE are all legal.
- READ ONLY (also supported by the Oracle JDBC implementation): In a READ ONLY transaction, you are not allowed to update the database. SELECT is legal, but INSERT, UPDATE, DELETE, and SELECT FOR UPDATE are not.

Isolation Level Settings

The `READ COMMITTED`, `SERIALIZABLE`, `READ UNCOMMITTED`, and `REPEATABLE READ` isolation level settings, where supported, have the following functionality:

- `READ UNCOMMITTED`: Dirty reads, nonrepeatable reads, and phantom reads are all allowed.
- `READ COMMITTED` (default for Oracle10*g*): Dirty reads are prevented, and nonrepeatable reads and phantom reads are allowed. If the transaction contains DML statements that require row locks held by other transactions, then any of the statements will block until the row lock it needs is released by the other transaction.
- `REPEATABLE READ`: Dirty reads and nonrepeatable reads are prevented, and phantom reads are allowed.
- `SERIALIZABLE`: Dirty reads, nonrepeatable reads, and phantom reads are all prevented. Any DML statements in the transaction cannot update any resource that might have had changes committed after the transaction began. Such DML statements will fail.

A dirty read occurs when transaction B accesses a row that was updated by transaction A, but transaction A later rolls back the updates. As a result, transaction B sees data that was never actually committed to the database.

A nonrepeatable read occurs when transaction A retrieves a row, transaction B subsequently updates the row, and transaction A later retrieves the same row again. Transaction A retrieves the same row twice but sees different data.

A phantom read occurs when transaction A retrieves a set of rows satisfying a given condition, transaction B subsequently inserts or updates a row such that the row now meets the condition in transaction A, and transaction A later repeats the conditional retrieval. Transaction A now sees an additional row. This row is referred to as a phantom.

You can think of the four isolation level settings being in a progression:

```
SERIALIZABLE > REPEATABLE READ > READ COMMITTED > READ UNCOMMITTED
```

If a desired setting is unavailable to you, such as `REPEATABLE READ` or `READ UNCOMMITTED` if you use Oracle Database 11*g*, use a greater setting (one further to the left) to ensure having at least the level of isolation that you want.

See Also: *Oracle Database Advanced Application Developer's Guide*

Using JDBC Connection Class Methods

You can optionally access and set the access mode and isolation level of a transaction, using methods of the underlying JDBC connection instance of your connection context instance. SQLJ code using these JDBC methods is not portable, however.

Following are the `Connection` class methods for access mode and isolation level settings:

- `abstract int getTransactionIsolation()`: Returns the current transaction isolation level as one of the following constant values:

  ```
  TRANSACTION_NONE
  TRANSACTION_READ_COMMITTED
  TRANSACTION_SERIALIZABLE
  ```

```
TRANSACTION_READ_UNCOMMITTED
TRANSACTION_REPEATABLE_READ
```

- `abstract void setTransactionIsolation(int)`: Sets the transaction isolation level, taking as input one of the preceding constant values.
- `abstract boolean isReadOnly()`: Returns `true` if the transaction is `READ ONLY`. Returns `false` if the transaction is `READ WRITE`.
- `abstract void setReadOnly(boolean)`: Sets the transaction access mode to `READ ONLY` if `true` is input. Sets the access mode to `READ WRITE` if `false` is input.

SQLJ and JDBC Interoperability

SQLJ statements are typically used for static SQL operations. Oracle Database 11*g* has extensions to support dynamic SQL as well, but another alternative is to use JDBC code within your SQLJ application for dynamic operations, which would be more portable. And there might be additional scenarios where using JDBC code in your SQLJ application might be useful or even required. Because of this, SQLJ enables you to use SQLJ and JDBC statements concurrently and provides interoperability between SQLJ and JDBC constructs.

Two kinds of interactions between SQLJ and JDBC are particularly useful:

- Between SQLJ connection contexts and JDBC connections
- Between SQLJ iterators and JDBC result sets

See Also: *Oracle Database JDBC Developer's Guide and Reference*

This section covers the following topics:

- SQLJ Connection Context and JDBC Connection Interoperability
- SQLJ Iterator and JDBC Result Set Interoperability

SQLJ Connection Context and JDBC Connection Interoperability

SQLJ enables you to convert, in either direction, between SQLJ connection context instances and JDBC connection instances.

Note: When converting between a SQLJ connection context and a JDBC connection, bear in mind that the two objects are sharing the same underlying physical connection.

Converting from Connection Contexts to JDBC Connections

If you want to perform a JDBC operation through a database connection that you have established in SQLJ (for example, if your application calls a library routine that returns a JDBC connection object), then you must convert the SQLJ connection context instance to a JDBC connection instance.

Any connection context instance in a SQLJ application, whether an instance of the `sqlj.runtime.ref.DefaultContext` class or of a declared connection context class, contains an underlying JDBC connection instance and a `getConnection()` method that returns that JDBC connection instance. Use the JDBC connection instance to create JDBC statement objects if you want to use JDBC operations.

Following is an example of how to use the `getConnection()` method.

```
import java.sql.*;

...
DefaultContext ctx = new DefaultContext
      ("jdbc:oracle:thin:@localhost:1521/myservice", "scott", "tiger", true);
...
(SQLJ operations through SQLJ ctx connection context instance)
...
Connection conn = ctx.getConnection();
...
(JDBC operations through JDBC conn connection instance)
...
```

The connection context instance can be an instance of the `DefaultContext` class or of any connection context class that you have declared.

To retrieve the underlying JDBC connection of your default SQLJ connection, you can use `getConnection()` directly from a `DefaultContext.getDefaultContext()` call, where `getDefaultContext()` returns a `DefaultContext` instance that you had previously initialized as your default connection and `getConnection()` returns its underlying JDBC connection instance. In this case, because you do not have to use the `DefaultContext` instance explicitly, you can also use the `Oracle.connect()` method. This method implicitly creates the instance and makes it the default connection.

> **See Also:** "Connection Considerations" on page 3-4 and "More About the Oracle Class" on page 3-9

Following is an example:

```
import java.sql.*;

...
Connection conn = Oracle.connect
   ("jdbc:oracle:thin:@localhost:1521/myservice",
    "scott", "tiger").getConnection();
...
(JDBC operations through JDBC conn connection instance)
...
```

Example: JDBC and SQLJ Connection Interoperability for Dynamic SQL

Following is a sample method that uses the underlying JDBC connection instance of the default SQLJ connection context instance to perform dynamic SQL operations in JDBC. The dynamic operations are performed using JDBC `java.sql.Connection`, `java.sql.PreparedStatement`, and `java.sql.ResultSet` objects. Alternatively, you can use Oracle SQLJ extensions for dynamic SQL operations.

> **See Also:** *Oracle Database JDBC Developer's Guide and Reference* and "Support for Dynamic SQL" on page 7-50

```
import java.sql.*;

public static void projectsDue(boolean dueThisMonth) throws SQLException {

   // Get JDBC connection from previously initialized SQLJ DefaultContext.
   Connection conn = DefaultContext.getDefaultContext().getConnection();
```

```
   String query = "SELECT name, start_date + duration " +
                  "FROM projects WHERE start_date + duration >= sysdate";
   if (dueThisMonth)
      query += " AND to_char(start_date + duration, 'fmMonth') " +
               " = to_char(sysdate, 'fmMonth') ";

   PreparedStatement pstmt = conn.prepareStatement(query);
   ResultSet rs = pstmt.executeQuery();
   while (rs.next()) {
      System.out.println("Project: " + rs.getString(1) + " Deadline: " +
                         rs.getDate(2));
   }
   rs.close();
   pstmt.close();
}
```

For a rework of this example using SQLJ dynamic SQL functionality with `FETCH` functionality from a result set iterator, refer to Example 5: Dynamic SQL with FETCH from Result Set Iterator on page 7-54.

Converting from JDBC Connections to Connection Contexts

If you initiate a connection as a JDBC `Connection` instance but later want to use it as a SQLJ connection context instance (for example, if you want to use it in a context expression to specify the connection to use for a SQLJ executable statement), you can convert the JDBC connection instance to a SQLJ connection context instance.

The `DefaultContext` class and all declared connection context classes have a constructor that takes a JDBC connection instance as input and constructs a SQLJ connection context instance.

For example, presume you instantiated and defined the JDBC connection instance `conn` and want to use the same connection for an instance of a declared SQLJ connection context class `MyContext`. You can do this as follows:

```
...
#sql context MyContext;
...
MyContext myctx = new MyContext(conn);
...
```

About Shared Connections

A SQLJ connection context instance and the associated JDBC connection instance share the same underlying physical connection. As a result, the following is true:

- When you get a JDBC connection instance from a SQLJ connection context instance (using the connection context `getConnection()` method), the `Connection` instance inherits the state of the connection context instance. Among other things, the `Connection` instance will retain the auto-commit setting of the connection context instance.
- When you construct a SQLJ connection context instance from a JDBC connection instance (using the connection context constructor that takes a connection instance as input), the connection context instance inherits the state of the `Connection` instance. Among other things, the connection context instance will retain the auto-commit setting of the `Connection` instance. By default, a JDBC connection instance has an auto-commit setting of `true`, but you can alter this through the `setAutoCommit()` method of the `Connection` instance.

- Given a SQLJ connection context instance and associated JDBC connection instance, calls to methods that alter session state in one instance will also affect the other instance, because it is actually the underlying shared session that is being altered.
- Because there is just a single underlying physical connection, there is also a single underlying set of transactions. A `COMMIT` or `ROLLBACK` operation in one connection instance will affect any other connection instances that share the same underlying connection.

> **Note:** It is also possible for multiple SQLJ connection context instances to be created from the same JDBC connection instance and, therefore, to share the same underlying physical connection. This might be useful, for example, if you want to share the same set of transactions between program modules. The preceding notes apply to this situation as well.

Closing Shared Connections

When you get a JDBC connection instance from a SQLJ connection context instance (using the `getConnection()` method) or you create a SQLJ connection context instance from a JDBC connection instance (using the connection context constructor), you must close only the connection context instance. By default, calling the `close()` method of a connection context instance closes the associated JDBC connection instance and the underlying physical connection, thereby freeing all resources associated with the connection.

If you want to close a SQLJ connection context instance *without* closing the associated JDBC connection instance (if, for example, the `Connection` instance is being used elsewhere, either directly or by another connection context instance), then you can specify the boolean constant `KEEP_CONNECTION` to the `close()` method, as follows (assume a connection context instance `ctx`):

```
ctx.close(ConnectionContext.KEEP_CONNECTION);
```

If you do not specify `KEEP_CONNECTION`, then the associated JDBC connection instance is closed by default. You can also specify this explicitly:

```
ctx.close(ConnectionContext.CLOSE_CONNECTION);
```

`KEEP_CONNECTION` and `CLOSE_CONNECTION` are static constants of the `sqlj.runtime.ConnectionContext` interface.

If you close only the JDBC connection instance, this will *not* close the associated SQLJ connection context instance. The underlying physical connection would be closed, but the resources of the connection context instance would not be freed until garbage collection.

Note:

- If the same underlying JDBC connection is shared by multiple connection context instances, then use `KEEP_CONNECTION` when closing all but the last remaining open connection context instance.
- An error message will be issued if you try to close a connection context instance whose underlying JDBC connection has already been closed, or if you try to close the underlying connection when it has already been closed. If you encounter this, then verify that the JDBC connection is not being closed independently by JDBC code and all preceding `close()` calls on SQLJ connection context instances that use the underlying connection use the `KEEP_CONNECTION` parameter.

SQLJ Iterator and JDBC Result Set Interoperability

SQLJ enables you to convert in either direction between SQLJ iterators and JDBC result sets. For situations where you are selecting data in a SQLJ statement but do not care about strongly typed iterator functionality, SQLJ also supports a weakly typed iterator, which you can convert to a JDBC result set.

Converting from Result Sets to Named or Positional Iterators

There are a number of situations where you might find yourself manipulating JDBC result sets. For example, another package might be implemented in JDBC and provide access to data only through result sets or might require `ResultSetMetaData` information because it is a routine written generically for any type of result set. Or your SQLJ application might invoke a stored procedure that returns a JDBC result set.

If the dynamic result set has a known structure, it is typically desirable to manipulate it as an iterator to use the strongly typed paradigm that iterators offer.

In SQLJ, you can populate a named or positional iterator object by converting an existing JDBC result set object. This can be thought of as casting a result set to an iterator, and the syntax reflects this as follows:

```
#sql iter = { CAST :rs };
```

This binds the result set object, `rs`, into the SQLJ executable statement, converts the result set, and populates the iterator, `iter`, with the result set data.

Following is an example. Assume `myEmpQuery()` is a static Java function in a class called `RSClass`, with a predefined query that returns a JDBC result set object:

```
import java.sql.*;
...
#sql public iterator MyIterator (String ename, float sal);
...
ResultSet rs;
MyIterator iter;
...
rs = RSClass.myEmpQuery();
#sql iter = { CAST :rs };
...
(process iterator)
...
iter.close();
...
```

This example could have used a positional iterator instead of a named iterator. The functionality is identical.

The following rules apply when converting a JDBC result set to a SQLJ iterator and processing the data:

- To convert to a positional iterator, the result set and iterator must have the same number of columns and the types must map correctly.
- To convert to a named iterator, the result set must have at least as many columns as the iterator and all columns of the iterator must be matched by name and type. If the result set and iterator do not have the same number of columns, then the SQLJ translator will generate a warning unless you use the `-warn=nostrict` option setting.
- The result set being cast must implement the `java.sql.ResultSet` interface. The class `oracle.jdbc.OracleResultSet` implements this interface, as does any standard result set class.
- The iterator receiving the cast must be an instance of an iterator class that was declared as `public`.
- Do not access data from the result set, either before or after the conversion. Access data from the iterator only.
- When you are finished, close the iterator, not the result set. Closing the iterator will also close the result set, but closing the result set will not close the iterator. When interoperating with JDBC, always close the SQLJ entity.

Converting from Named or Positional Iterators to Result Sets

You might also encounter situations where you want to define a query using SQLJ but ultimately need a result set.

> **Note:** SQLJ offers more natural and concise syntax, but perhaps you want to do dynamic processing of the results, or perhaps you want to use an existing Java method that takes a result set as input.

So that you can convert iterators to result sets, every SQLJ iterator class, whether named or positional, is generated with a `getResultSet()` method. This method can be used to return the underlying JDBC result set object of an iterator object.

Following is an example showing use of the `getResultSet()` method:

```
import java.sql.*;

#sql public iterator MyIterator (String ename, float sal);

...
MyIterator iter;
...
#sql iter = { SELECT * FROM emp };
ResultSet rs = iter.getResultSet();
...
(process result set)
...
iter.close();
...
```

The following rules apply when converting a SQLJ iterator to a JDBC result set and processing the data.

- When writing iterator data to a result set, you should access data only through the result set. Do not attempt to directly access the iterator, either before or after the conversion.
- When you finish, close the original iterator, not the result set. Closing the iterator will also close the result set, but closing the result set will not close the iterator. When interoperating with JDBC, always close the SQLJ entity.

Using and Converting Weakly Typed Iterators (ResultSetIterator)

You might have a situation similar to what is discussed in "Converting from Named or Positional Iterators to Result Sets" on page 7-49, but where you do not require the strongly typed functionality of the iterator. All you might care about is being able to use SQLJ syntax for the query and then processing the data dynamically from a result set. For such circumstances, you can directly use the `sqlj.runtime.ResultSetIterator` type to receive query data.

In using SQLJ statements and `ResultSetIterator` functionality instead of using JDBC statements and standard result set functionality, you enable yourself to use the more concise `SELECT` syntax of SQLJ.

Following is an example of how to use and convert a weakly typed result set iterator:

```
import sqlj.runtime.*;
import java.sql.*;

...
ResultSetIterator rsiter;
...
#sql rsiter = { SELECT * FROM table };
ResultSet rs = rsiter.getResultSet();
...
(process result set)
...
rsiter.close();
...
```

> **Note:** The Oracle SQLJ implementation permits navigation through a result set iterator using the `next()` method and `FETCH CURRENT` syntax. Furthermore, for scrollable result set iterators, additional navigation methods are supported.

Support for Dynamic SQL

The Oracle SQLJ implementation includes extensions to support dynamic SQL, operations that are not predefined and can change in real time. Dynamic SQL expressions embedded in SQLJ statements are referred to as meta bind expressions.

> **Note:** Using JDBC code is still an option for dynamic SQL in Oracle Database 11*g* and might be preferable if code portability is a concern, but SQLJ support for dynamic SQL permits use of SQLJ as a single, simplified API for data access.

This section covers the following topics:

Meta Bind Expressions

Meta bind expressions are used for dynamic SQL in SQLJ statements, where otherwise static SQL clauses would appear. A meta bind expression contains a Java identifier of `String` type or a string-valued Java expression that is interpreted at run time. In addition, so that SQLJ can perform online semantics-checking, a meta bind expression can optionally include static SQL replacement code to be used for checking during translation.

Meta Bind Expressions: General Usage and Restrictions

You can use a meta bind expression in place of any of the following:

- Table name
- Column name in a `SELECT` statement (without the column alias, if specified)
- All or part of a `WHERE` clause condition
- Role, schema, catalog, or package name in a data definition language (DDL) or DML statement
- SQL literal value or SQL expression

Be aware of the following restrictions on meta bind expressions, enforced to ensure that the SQLJ translator can properly determine the nature of the SQL operation and can perform syntactic analysis of the SQLJ statement as a whole:

- A meta bind expression cannot be the first noncomment of the SQL operation within a SQLJ statement.
- A meta bind expression cannot contain the `INTO` token of a SQLJ `SELECT INTO` statement and cannot expand to become the `INTO`-list of a `SELECT INTO` statement.
- A meta bind expression cannot appear in any of the following kinds of SQL/SQLJ instructions or clauses: `CALL`, `VALUES`, `PSM SET`, `COMMIT`, `ROLLBACK`, `FETCH INTO`, or `CAST`.

Meta Bind Expressions: Syntax and Behavior

Following is the general syntax for meta bind expressions:

```
:{ Java_bind_expression }
```

or:

```
:{ Java_bind_expression :: SQL_replacement_code }
```

Note that spaces are optional. There can be multiple meta bind expressions within the SQL instructions of a SQLJ statement.

Java Bind Expression

A Java bind expression can be either of the following:

- Java identifier of the `String` type
- Java expression that evaluates to a character string

Java bind expressions within meta bind expressions are subject to standard Java lexing rules and have syntax similar to that of SQLJ host expressions. However, unlike host expressions, Java bind expressions within meta bind expressions are not enclosed within parentheses. This is because, if there is SQL replacement code, then the `::` token acts as a separator between the Java bind expression and the SQL code. If there is no SQL replacement code, then the closing braces (}) acts as a terminator. In either case, there is no ambiguity.

> **Note:** There can be no mode specifiers, `IN`, `OUT`, or `INOUT`, within a Java bind expression or between `:` and `{` of the meta bind expression.

SQL Replacement Code

A SQL replacement code clause consists of a sequence of zero or more SQL tokens, with the following requirements and restrictions:

- It is subject to SQL lexing rules.
- Braces ({ }) must occur in matching pairs (with the exception of those that are part of a SQL comment, constant, or identifier).
- There can be no SQLJ host expressions or nested meta bind expressions within the SQL instructions.

> **Note:** It is permissible for the SQL replacement code to be empty.

Translation-Time Behavior

Whenever there is SQL replacement code (even if only an empty string) in a meta bind expression, then the meta bind expression is replaced by the SQL code during translation. The purpose of SQL replacement code is to enable the SQLJ translator to perform online semantics-checking.

If any meta bind expression within a SQLJ statement has no SQL replacement code clause, then the SQLJ translator cannot perform online semantics-checking on the statement. It is only checked syntactically.

Run Time Behavior

At run time, each meta bind expression is replaced by the evaluation of its Java bind expression. If a Java bind expression evaluates to `null`, then the dynamic SQL statement as a whole becomes undefined.

SQLJ Dynamic SQL Examples

This section provides examples of dynamic SQL usage in SQLJ code.

Example 1

```
...
int x = 10;
int y = x + 10;
int z = y + 10;
String table = "new_Emp";
#sql { INSERT INTO :{table :: emp} VALUES (:x, :y, :z) };
...
```

During translation, the SQL operation becomes:

```
INSERT INTO emp VALUES (10, 20, 30);
```

SQLJ can perform online semantics-checking against a schema that has an `emp` table. Perhaps `new_Emp` only exists in the run-time schema and is not created until the application executes.

During run time, the SQL operation becomes:

```
INSERT INTO new_Emp VALUES (10, 20, 30);
```

Example 2

```
...
String table = "new_Emp";
String query = "ename LIKE 'S%' AND sal>1000";
#sql myIter = { SELECT * FROM :{table :: emp2}
                         WHERE :{query :: ename='SCOTT'} };
...
```

During translation, the SQL operation becomes:

```
SELECT * FROM emp2 WHERE ename='SCOTT';
```

SQLJ can perform online semantics-checking against a schema that has an `emp2` table.

During run time, the SQL operation becomes:

```
SELECT * FROM new_Emp WHERE ename LIKE 'S%' AND sal>1000;
```

Example 3

```
...
double raise = 1.12;
String col = "comm";
String whereQuery = "WHERE "+col+" IS NOT null";
for (int i=0; i<5; i++)
{
   #sql { UPDATE :{"emp"+i :: emp}
          SET :{col :: sal} = :{col :: sal} * :raise :{whereQuery ::} };
}
...
```

During translation, the SQL operation becomes:

```
UPDATE emp SET sal = sal * 1.12;
```

SQLJ can perform online semantics-checking against a schema that has an `emp` table. There is no `WHERE` clause during translation, because the SQL replacement code is empty.

During run time, the SQL operation is executed five times, becoming:

```
UPDATE emp0 SET comm = comm * 1.12 WHERE comm IS NOT null;
UPDATE emp1 SET comm = comm * 1.12 WHERE comm IS NOT null;
UPDATE emp2 SET comm = comm * 1.12 WHERE comm IS NOT null;
UPDATE emp3 SET comm = comm * 1.12 WHERE comm IS NOT null;
UPDATE emp4 SET comm = comm * 1.12 WHERE comm IS NOT null;
```

Example 4

```
...
double raise = 1.12;
String col = "comm";
String whereQuery = "WHERE "+col+" IS NOT null";
for (int i=0; i<10; i++)
{
   #sql { UPDATE :{"emp"+i}
          SET :{col :: sal} = :{col :: sal} * :raise :{whereQuery ::} };
}
...
```

The run-time behaviors of Example 3 and Example 4 are identical. However, a difference occurs during translation, where SQLJ cannot perform online semantics-checking for Example 4, because there is no SQL replacement code for the first meta bind expression, `:{"emp"+i}`.

Example 5: Dynamic SQL with FETCH from Result Set Iterator

This example is a rework of "Example: JDBC and SQLJ Connection Interoperability for Dynamic SQL" on page 7-45, using SQLJ statements instead of JDBC statements. This example also uses `FETCH CURRENT` functionality from a result set iterator.

```
import java.sql.*;

public static void projectsDue(boolean dueThisMonth) throws SQLException {

   ResultSetIterator rsi;
   String andClause = (dueThisMonth) ?
                      " AND to_char(start_date + duration, 'fmMonth' ) "
                      + " = to_char(sysdate, 'fmMonth') "
                      : "";
   #sql rsi = { SELECT name, start_date + duration FROM projects
                WHERE start_date + duration >= sysdate :{andClause :: } };
   while (rsi.next())
   {
      String name = null;
      java.sql.Date deadline = null;
      #sql { FETCH CURRENT FROM :rsi INTO :name, :deadline };
      System.out.println("Project: " + name + "Deadline: " + deadline);
   }
   rsi.close();
}
```

Execution Plan Fixing

If you run the risk of any performance changes in the application due to change in the environment, then you may use the outline feature of Oracle. An outline is implemented as a set of optimizer hints that are associated with the SQL statement. If the use of the outline is enabled for the statement, Oracle automatically considers the stored hints and tries to generate an execution plan in accordance with those hints. You can group outlines into categories, that is, whether they are default or as specified by the client, and control the category of outlines Oracle uses to simplify outline administration and deployment. The hints in the outlines are used during the execution of respective statements if you have set `USE_STORED_OUTLINES` to the category name or to `TRUE`.

> **See Also:** *Oracle Database SQL Language Reference* for more information about outlines.

When you translate the file with the new outline option set to `true` or the category name, then:

1. A separate SQL file is created containing the `CREATE OUTLINE` statements for all the SQL statements present in the input SQLJ file.
2. A log file containing the SQL statements, outline name, outline SQL statement, outline category, and status information is generated.
3. If you specify the `-runoutline option`, then the SQL file generated is run at the end of successful translation of the input file.

SQL statements that can be used to create outlines are:

- SELECT
- DELETE
- UPDATE
- INSERT ... SELECT
- CREATE TABLE ... AS SELECT

You have the following restrictions on creating outlines:

- You cannot create outlines on MERGE statements.
- You cannot create outlines on a multi-table INSERT statement.
- The SQL statement in the outline cannot include any DML operation on a remote object.

Options to Generate Outlines

> **Note:** The outline options are valid *only* if online checking is done.

Consider the SQLJ program `abc.sqlj` contains the following code snippet:

```
{
#sql iter = {SELECT * FROM emp WHERE empno=:var;}
#sql iter1 = {SELECT * FROM dept};
}
```

Compile the SQLJ program as:

```
%sqlj -url=jdbc:oracle:oci8:@ -user=scott/tiger -outline=abccat abc.sqlj
```

The generated SQL file `abc_sqlj.sql` for the above SQLJ code snippet looks as follows:

```
CREATE OR REPLACE OUTLINE abccat_abc_sqlj_0001 FOR CATEGORY abccat ON SELECT *
FROM emp WHERE empno=:B1 /* abccat_abc_sqlj_0001 */;

CREATE OR REPLACE OUTLINE abccat_abc_sqlj_0002 FOR CATEGORY abccat ON SELECT *
FROM dept /* abccat_abc_sqlj_0002 */;
```

The option `-outline` generates two files at the end of successful translation: a SQL file and a LOG file. The generated SQL file name has the following format:

```
<filename>_<filetype>.sql
```

For example, the generated SQL file for filename `abc.sqlj` is `abc_sqlj.sql`.

The format of the unique identifier used as outline name and comment is:

```
<categoryname >_<filename>_<filetype>_<sequence no.>
```

where, the sequence number is a four-digit sequence number ranging from 0001 to 9999. If the SQLJ program contains more than 9999 SQL statements, then you get the "`Max sequence number exceeded for outlines`" error. For example, the format of the unique identifier generated for `abc.sqlj` is `abccat_abc_sqlj_0001`, where, `abccat` is the name of the category.

Note: The same comment is added to the SQLs in the generated java or class file that is used at runtime.

If you set outline to `true`, then the default category will be used to store the outlines:

```
%sqlj -user=scott/tiger -url=jdbc:oracle:oci8:@ -outline=true abc.sqlj
```

In this case, the generated SQL file `abc_sqlj.sql` looks as follows:

```
CREATE OR REPLACE OUTLINE default_abc_sqlj_0001 ON SELECT * FROM emp WHERE
empno=:B1 /* default_abc_sqlj_0001 */;

CREATE OR REPLACE OUTLINE default_abc_sqlj_0002 ON SELECT * FROM dept /*
default_abc_sqlj_0002 */;
```

You can use the following command to set the outline name to a particular prefix:

```
%sqlj -user=scott/tiger -url=jdbc:oracle:oci8:@ -outline=abccat
-outlineprefix=pref1 abc.sqlj
```

In this case, the generated SQL file `abc_sqlj.sql` looks as follows:

```
CREATE OR REPLACE OUTLINE pref1_0001 FOR CATEGORY abccat ON SELECT * FROM emp
WHERE empno=:B1 /* pref1_0001 */';

CREATE OR REPLACE OUTLINE pref1_0002 FOR CATEGORY abccat ON SELECT * FROM dept /*
pref1_0002 */';
```

Note: To translate multiple files with the `outlineprefix` option, you can do the following:

```
%sqlj -outline=abccat -outlineprefix=pref1,pref2,pref3 abc.sqlj
def.sqlj fgh.sqlj
```

Currently, the upper limit on the length of the outline name is 30 bytes. Hence, if the generated outline name exceeds 30 bytes, a SQLJ error "`Outline name exceeds maximum limit. Use -outlineprefix option`" is thrown. In such cases, if you want to use the `-outline` option, you need to call `-outlineprefix` option as shown in the preceding example. If you want database server to generate the outline names instead of the SQLJ generated outline names, then you can set the `-outlineprefix` option to `none`. For example:

```
%sqlj -user=scott/tiger -url=jdbc:oracle:oci8:@ -outline=abccat
```

```
-outlineprefix=none abc.sqlj
```

In this case, the generated SQL file `abc_sqlj.sql` looks as follows:

```
CREATE OR REPLACE OUTLINE FOR CATEGORY abccat ON SELECT * FROM emp WHERE empno=:B1
/* abccat_abc_sqlj_0001 */';

CREATE OR REPLACE OUTLINE FOR CATEGORY abccat ON SELECT * FROM dept /*
abccat_abc_sqlj_0002 */';
```

If you want to translate multiple files with the `-outlineprefix` option, then you can use the following command:

```
%sqlj -user=scott/tiger -url=jdbc:oracle:oci8:@ -outline=abccat
-outlineprefix=pref1,pref2 abc.sqlj def.sqlj
```

If the SQLJ file is part of a package and you have not specified the `-outlineprefix` option, then the package name is appended to the outline name and is added to the comment. For example, if `abc.sqlj` is part of `xyz.def.fgh` package, then generated SQL file `abc_sqlj.sql`, for the command `%sqlj -url=jdbc:oracle:oci8:@ -user=scott/tiger -outline=abccat abc.sqlj` looks as follows:

```
CREATE OR REPLACE OUTLINE abccat_xyz$def$fgh$abc_sqlj_0001 FOR CATEGORY abccat ON
SELECT * FROM emp WHERE empno=:B1 /* abccat_xyz$def$fgh$abc_sqlj_0001 */;

CREATE OR REPLACE OUTLINE abccat_xyz$def$fgh$abc_sqlj_0002 FOR CATEGORY abccat ON
SELECT * FROM dept /* abccat_xyz$def$fgh$abc_sqlj_0002 */;
```

If you want the generated SQL file to be executed by the translator at the end of successful translation, then you can set the `runoutline` option to `true`. By default it is `false`. For example:

```
%sqlj -user=scott/tiger -url=jdbc:oracle:oci8:@ -outline=default -runoutline=true
abc.sqlj
```

Now, if you want to retrieve the outline name for exporting or for modifying the plan of the SQL code, then you can retrieve the same from the `OL$` table, either manually or by using a tool. You can use the comment in the SQL query to search for the appropriate SQL statement to identify the outline name because the comment uniquely identifies the SQL statement.

Table 7–1 shows all the options and values you can pass to the translator for generating outlines.

Table 7–1 Table showing the options and values for generating outlines

Option Name	Option Value
`-outline`	`true`<category name>
`-outlineprefix`	`none`<prefix name>|`none`<prefix name 1, prefix name 2,…>
`-runoutline`	`true|false`

> **See Also:** *Oracle Database SQL Language Reference* for more information about outlines.

Generated LOG File Name

The format of the generated file name is:

```
<filename>_<filetype>.log
```

For example, on translating `abc.sqlj`, the generated log file is `abc_sqlj.log`.

Generated LOG File Format

Suppose, you have the following code snippet:

```
#sql iter = {SELECT * FROM emp WHERE empno=:var };
#sql iter1 = {SELECT * FROM dept };
```

The generated log file for the preceding code snippet is as follows:

```
CATERGORY abccat
Source SQL_1
SELECT * FROM emp WHERE empno=:B1
OUTLINE NAME
abccat_abc_sqlj_0001
OUTLINE SQL_1
CREATE OR REPLACE OUTLINE abccat_abc_sqlj_0001 FOR CATEGORY abccat ON SELECT *
FROM emp WHERE empno = :B1 /* abccat_abc_sqlj_0001 */
STATUS success
Source SQL_2
SELECT * FROM dept
OUTLINE NAME
abccat_abc_sqlj_0002
OUTLINE SQL_2
CREATE OR REPLACE OUTLINE abccat_abc_sqlj_2 FOR abccat ON SELECT * FROM dept /*
abccat_abc_sqlj_2 */
STATUS fail
```

In the preceding example of the generated log file format:

- Category means the category of the outline to be generated
- Source means the SQL statements for which outline is to be generated
- Outline Name is the name of the outline to be generated
- Status is the execution status of the SQL statements used as the source. If the execution is successful, then status is `success`. Otherwise, it is `fail`.

Configuration Files

You can set the different command-line options in the configuration file as follows:

sqlj.outline

- Parameter Name: outline
- Parameter Type: String
- Allowable Values: {true | category_name}
- Default Value: true
- Description: Indicates that outline SQL file needs to be generated for the SQL statements and it should be in:
  - DEFAULT category if the value is default, that is, `true`
  - The category name if *category_name* is mentioned

  Outline SQL file is not generated if this option is not used.

- Dependency on other parameters: Online check should be full when this option is turned on

sqlj.runoutline

- Parameter Name: runoutline
- Parameter Type: boolean
- Allowable Values: {true | false}
- Default Value: false
- Description: If `runoutline=true` then the generated SQL file should be executed by the translator at the end of successful translation.
- Dependency on other parameters: Online check should be full when this option is turned on, and the outline option should be set.

sqlj.outlineprefix

- Parameter Name: outlineprefix
- Parameter Type: String
- Allowable Values: {prefix name}, none
- Description: If this option is set, the outline name in the generated SQL is set to `<prefix>_<seqno>`. When this option is set to any value apart from `none` in the properties file, only one SQLJ file may be passed to the translator. If the option is set to `none`, outline name is generated by the system when the `create outline` statement is executed in the server. Also, you may pass multiple files to the translator when `-outlineprefix` is set to none.
- Dependency on other parameters: Online check should be full when this option is turned on, and the outline option should be set.

8

Translator Command Line and Options

Once you have written your source code, you must translate it using the SQLJ translator. This chapter covers the SQLJ translator command line options and properties files. The following topics are covered:

Translator Command Line and Properties Files

The `sqlj` script invokes a Java virtual machine (JVM) and passes the class name of the SQLJ translator, `sqlj.tools.Sqlj`, to the JVM. The JVM invokes the translator and performs operations such as parsing the command line and properties files. For simplicity, running the script is referred to as running SQLJ, and its command line is referred to as the SQLJ command line.

The typical general syntax for the command line is as follows:

```
% sqlj <optionlist> filelist
```

The *`optionlist`* is a list of SQLJ option settings separated by spaces. There are also prefixes to mark options to pass to the Java interpreter, compiler, and customizer.

The *`filelist`* is the list of files, delimited by spaces, to be processed by the SQLJ translator. The files can be `.sqlj`, `.java`, `.ser`, or `.jar` files. The `*` wildcard entry can be used in file names. For example, `Foo*.sqlj` would find `Foo1.sqlj`, `Foo2.sqlj`, and `Foobar.sqlj`.

Note:

- All options need *not* precede the file list. Options may appear anywhere in the command line and are processed in order.
- All command-line options apply to all files being translated. It is not possible to have file-specific option settings.

Do *not* include `.class` files in the file list, but ensure that your classpath is set so that the SQLJ translator can find any classes it must have for type resolution of variables in your SQLJ source files. If the `-checksource` flag is enabled, which is the default setting, then the SQLJ translator can also find classes it needs in uncompiled `.java` files in the classpath.

See Also: "Source Check for Type Resolution (-checksource)" on page 8-54

Note: If you run the script by entering only `sqlj`, you will receive a synopsis of the most frequently used SQLJ options. In fact, this is true whenever you run the script without specifying any files to process. This is equivalent to using the `-help` flag setting.

This section covers the following topics:

- SQLJ Options, Flags, and Prefixes
- Command-Line Syntax and Operations
- Properties Files for Option Settings
- SQLJ_OPTIONS Environment Variable for Option Settings
- Order of Precedence of Option Settings

SQLJ Options, Flags, and Prefixes

This section discusses options supported by the SQLJ translator. Boolean options are referred to as flags. Prefixes used to pass options to the JVM, which the SQLJ script invokes, and to the Java compiler and SQLJ profile customizer, which the JVM invokes, are also listed.

Summary of SQLJ Options

Table 8–1 lists options supported by the SQLJ translator, categorized as follows:

- Flags and options listed as Basic are discussed in "Basic Translator Options" on page 8-16.
- Flags, options, and prefixes listed as Advanced are discussed in "Advanced Translator Options" on page 8-47.
- Flags and options listed as Environment are discussed in "Translator Support and Options for Alternative Environments" on page 8-62. These flags and options are for use of a nonstandard JVM, compiler, or customizer.
- Options with a category of `javac` are `javac` compiler options that SQLJ recognizes directly, without the compiler prefix. They are passed to the Java compiler, typically `javac`, and some also affect SQLJ translator settings. These options are discussed in "Option Support for javac" on page 8-7.

Table 8–1 SQLJ Translator Options

Option	Description	Default	Category
`-bind-by-identifier`	Flag to treat multiple appearances of the same host variable in a given SQLJ statement as a single bind occurrence.	`false`	Advanced
`-C`	Prefix that marks options to pass to the Java compiler.	NA	Advanced
`-cache`	Enables caching of online semantics-checking results (to reduce trips to database).	`false`	Advanced

Table 8–1 (Cont.) SQLJ Translator Options

Option	Description	Default	Category
`-checkfilename`	Specifies whether a warning is issued during translation if a source file name does not correspond to the name of the public class (if any) defined there.	`true`	Environment
`-checksource`	Instructs SQLJ type resolution to examine source files in addition to class files in certain circumstances.	`true`	Advanced
`-classpath`	Specifies the classpath to the JVM and Java compiler; also passed to `javac`. Use this on the command line only.	None	Basic
`-codegen`	Specifies mode of code generation: `oracle` for Oracle-specific code generation with direct Oracle Java Database Connectivity (JDBC) calls; `iso` for ISO standard SQLJ code generation.	`oracle`	Basic
`-compile`	Enables or disables the Java compilation step, either for `.java` files generated during the current SQLJ run or for previously generated or other `.java` files specified on the command line.	`true`	Advanced
`-compiler-executable`	Specifies the Java compiler to use.	`javac`	Environment
`-compiler-encoding-flag`	Instructs SQLJ whether to pass the `-encoding` setting, if set, to the Java compiler.	`true`	Environment
`-compiler-output-file`	Specifies a file to which the Java compiler output should be written. If this option is not set, then SQLJ assumes that compiler output goes to standard output.	None	Environment
`-compiler-pipe-output-flag`	Instructs SQLJ whether to set the `javac.pipe.output` system property, which determines whether the Java compiler prints errors and messages to `STDOUT` instead of `STDERR`.	`true`	Environment
`-components`	Specifies the components (packages and classes) to instrument for use with Oracle Dynamic Monitoring Service (DMS). This assumes instrumentation is enabled through the `-instrument` option. Use `all` to instrument all components being translated.	`all`	Basic
`-d`	Specifies the output directory for `.ser` profile files, if applicable, generated by SQLJ, and `.class` files generated by the compiler; also passed to `javac`.	Empty (Use directory of `.java` files to place generated `.class` files; use directory of `.sqlj` files to place generated `.ser` files.)	Basic
`-default-customizer`	Determines the profile customizer to use. Specify a class name.	`oracle.sqlj.runtime.util.OraCustomizer`	Environment
`-default-url-prefix`	Sets the default prefix for URL settings.	`jdbc:oracle:thin:`	Basic

Table 8–1 (Cont.) SQLJ Translator Options

Option	Description	Default	Category
`-depend`	Passed to `javac`; enables `-checksource`. This option requires the `-C` compiler prefix if set in a properties file.	NA	`javac`
`-deprecation`	Passed to `javac` only. This option requires the `-C` compiler prefix if set in a properties file.	NA	`javac`
`-dir`	Sets the output directory for SQLJ-generated `.java` files.	Empty (Use directory of `.sqlj` input file.)	Basic
`-driver`	Determines the JDBC driver class to register. Specify a class name or comma-delimited list of class names.	oracle.jdbc.OracleDriver	Basic
`-encoding`	Specifies the encoding that SQLJ and the compiler will use in globalization support; also passed to `javac`. You can use `-e` on the command line.	JVM `file.encoding` setting	Basic
`-explain`	Flag to request cause and action information to be displayed with translator error messages.	`false`	Basic
`-fixedchar`	Flag to account for blank padding when binding a string into a `WHERE` clause for comparison with `CHAR` data.	`false`	Basic
`-g`	Passed to `javac`; enables `-linemap`. This option requires the `-C` compiler prefix if set in a properties file.	NA	`javac`

`-help`
`-help-long`
`-help-alias`
	Flags to display different levels of information about SQLJ option names, descriptions, and current values. Use these on the command line only. You can use `-h` instead of `-help`.	Disabled	Basic

`-instrument`	Specifies whether to instrument translated files for use with Oracle DMS.	`false`	Basic
`-jdblinemap`	Variant of `-linemap` option for use with the Sun Microsystems `jdb` debugger.	`false`	Basic
`-J`	Prefix that marks options to pass to the JVM. Use this on the command line only.	NA	Advanced
`-linemap`	Enables mapping of line numbers between the generated Java class file and the original SQLJ code.	`false`	Basic
`-n`	Instructs the `sqlj` script to echo the full command line as it would be passed to the SQLJ translator, including settings in `SQLJ_OPTIONS`, without having the translator execute it. This is equivalent to `-vm=echo`. Use this on the command line only.	Disabled	Basic
`-ncharconv`	Performs bind to `NCHAR` columns for `String` host variables.	`false`	Basic
`-nowarn`	Passed to `javac`; sets `-warn=none`. This option requires the `-C` compiler prefix if set in a properties file.	NA	`javac`

Table 8–1 (Cont.) SQLJ Translator Options

Option	Description	Default	Category
`-O`	Passed to `javac`; disables `-linemap`. This option requires the `-C` compiler prefix if set in a properties file.	NA	`javac`
`-offline`	Determines the offline checker to use for semantics-checking. Specify a list of fully qualified class names.	oracle.sqlj.checker. OracleChecker	Advanced
`-online`	Determines the online checker to use for semantics-checking. Specify a fully qualified class name. (You must also set `-user` to enable online checking.)	oracle.sqlj.checker. OracleChecker	Advanced
`-optcols`	Enables iterator column type and size definitions to optimize performance. It is used directly by the translator for Oracle-specific code generation, or forwarded to the Oracle customizer along with user, password, and URL settings for ISO code generation.	`false`	Basic
`-optparams`	Enables parameter size definitions to optimize JDBC resource allocation (used with `-optparamdefaults`). This is used directly by the translator for Oracle-specific code generation, or forwarded to the Oracle customizer for ISO code generation.	`false`	Basic
`-optparamdefaults`	Sets parameter size defaults for particular data types (used with `-optparams`). This is used directly by the translator for Oracle-specific code generation, or forwarded to the Oracle customizer for ISO code generation.	`false`	Basic
`-P`	Prefix that marks options to pass to the SQLJ profile customizer.	NA	Advanced
`-parse`	Option to enable the offline SQL parser. Possible settings: `both`, `online-only`, `offline-only`, `none`, or the name of a Java class that implements an alternative parser. **Note**: Some settings for this option will also disable online semantics-checking, overriding the effect of the `-user` option.	`both`	Advanced
`-passes`	Instructs the `sqlj` script to run SQLJ in two separate passes, with compilation in between. Use this on the command line only.	`false`	Environment
`-password`	Sets the user password for the database connection for online semantics-checking. You can use `-p` on the command line.	None	Basic
`-profile`	For ISO code generation, enables or disables the profile customization step for profile files generated during the current SQLJ run.	`true`	Advanced

Table 8–1 (Cont.) SQLJ Translator Options

Option	Description	Default	Category
`-props`	Specifies a properties file, an alternative to the command line for setting options. (The `sqlj.properties` is also still read.) Use this on the command line only.	None	Basic
`-ser2class`	For ISO code generation, instructs SQLJ to translate generated `.ser` profiles to `.class` files.	`false`	Advanced
`-status`	Requests SQLJ to display status messages as it runs. Instead of `-status`, you can use `-v` on the command line.	`false`	Basic
`-url`	Sets the URL for the database connection for online semantics-checking.	`jdbc:oracle:oci:@`	Basic
`-user`	Enables online semantics-checking and sets the user name (and optionally password and URL) for the database connection. You can use `-u` on the command line.	None (no online semantics-checking)	Basic
`-verbose`	Passed to `javac`; enables `-status`. This requires the `-C` compiler prefix if set in a properties file.	NA	`javac`
`-version` `-version-long`	Flag to display different levels of SQLJ and JDBC driver version information. Use these settings on the command line only.	Disabled	Basic
`-vm`	Specifies the JVM to use for running the SQLJ translator. Use this on the command line only.	`java`	Environment

`-warn`	Comma-delimited list of flags to enable or disable different SQLJ warnings. Individual flags are `cast/nocast`
	`precision/noprecision`,
	`nulls/nonulls`,
	`portable/noportable`,
	`strict/nostrict`, and
	`verbose/noverbose`. The global flag is `all/none`.
		`cast`
		`precision`
		`nulls`
		`noportable`
		`strict`
		`noverbose`
			Basic

Notes Regarding Options, Flags, and Prefixes

Keep the following in mind:

- Flags, options, and prefixes listed as command line only in the Description column of the preceding table *cannot* be set in a properties file.
- The names of command-line options, including options passed elsewhere, are case-sensitive and usually all lowercase. Option values are usually case-sensitive as well.
- Several options, as indicated in Table 8–1, accept alternative syntax if specified on the command line, to support compatibility with the Oracle `loadjava` utility.
- Most SQLJ options can also be set in a properties file.
- The `SQLJ_OPTIONS` environment variable can be used in addition to, or instead of, the command line for setting options.
- In this document, boolean flags are usually discussed as being `true` or `false`, but they can also be enabled or disabled by setting them to `yes/no`, `on/off`, or `1/0`.

See Also: "Command-Line Syntax and Operations" on page 8-9

Notes Regarding the -password Option

You can choose one of the following ways to provide the password, ensuring that it is not being intercepted by other users through utilities such as `ps`:

- Omit the `-password` argument. In this case, you will be prompted on the command line to enter the password. Then the password argument will not be visible to the operating system.
- Place the password setting into a properties file, and instruct the SQLJ translator to use this properties file. Thus it is possible to script SQLJ translation without exposing the password to the operating system.
- Use SQLJ under JDeveloper. This does not expose the password to the operating system.

Options for loadjava Compatibility

For compatibility with the `loadjava` utility used to load Java and SQLJ applications into an Oracle Database 11*g* instance, the following alternative syntax is recognized for the indicated options when specified on the command line:

- `-e` (for `-encoding`)
- `-h` (for `-help`)
- `-p` (for `-password`)
- `-u` (for `-user`)
- `-v` (for verbose message output; equivalent to `-status`)

To maintain full consistency with `loadjava` syntax, you can use a space instead of equal sign (=) in setting these options, as in the following example:

```
-u scott/tiger -v -e SJIS
```

Note: This alternative option syntax is recognized only on the command line or in the `SQLJ_OPTIONS` environment variable, not in properties files.

See Also: *Oracle Database Java Developer's Guide* for general information about the `loadjava` utility

Option Support for javac

SQLJ supports option settings for `javac`, the Java compiler supplied with the Sun Microsystems Java Development Kit (JDK), in the following ways:

- Some `javac` options that take values are combined into SQLJ options (`-classpath`, `-d`, `-encoding`).
- For other `javac` options that take values, special processing has been implemented to correctly pass the value to the compiler (`-bootclasspath`, `-extdirs`, `-target`). These require a compiler prefix. They have no effect on SQLJ operation.

- Flags for `javac` are recognized on the command line without a `compiler` prefix (`-depend`, `-deprecation`, `-g`, `-nowarn`, `-O`, `-verbose`). Some of these flags affect SQLJ translator flag settings as well.

This is summarized in Table 8–2. All of these options can be set on the SQLJ command line or in a properties file, though sometimes a compiler prefix is required, as noted in the table.

Note:

- By default, `javac` compiles classes against the bootstrap and extension classes of the platform with which it was shipped. But `javac` also supports cross-compiling classes against bootstrap and extension classes of a different Java platform. The `javac -bootclasspath` and `-extdirs` options are for use in cross-compiling (JDK 1.5.x).
- By default, `javac` generates `.class` files that are compatible with the JDK version from which `javac` was obtained. Use the `-target` option to alter this.

Table 8–2 SQLJ Support for javac Options

Command-Line Option (with -C Prefix if Noted)	Description	Relationship to SQLJ
`-C-bootclasspath`	Instructs `javac` to cross-compile against the specified set of bootstrap classes.	None
`-classpath`	Sets the classpath for `javac` and the JVM.	This is also a SQLJ option.
`-d`	Sets the output directory for `.class` files and SQLJ profile files.	This is also a SQLJ option
`-depend`	Instructs `javac` to compile out-of-date files recursively.	Enables the SQLJ `-checksource` option.
`-deprecation`	Instructs `javac` to output source locations where deprecated APIs are used.	None
`-encoding`	Sets the encoding for both `SQLJ` and `javac`.	This is also a SQLJ option.
`-C-extdirs`	Instructs `javac` to cross-compile against the specified extension directories.	None
`-g`	Generates `javac` debugging information.	Enables the `SQLJ -linemap` option.
`-nowarn`	Instructs `javac` to generate no warnings.	Sets the `SQLJ` option `-warn=none`.
`-O`	Instructs `javac` to optimize.	Disables the `SQLJ -linemap` option.
`-C-target`	Instructs `javac` to generate `.class` files to work only on JVMs of the specified JDK version level or higher.	None
`-verbose`	Instructs `javac` to produce real-time status messages.	Enables the `SQLJ -status` option.

Refer to `javac` documentation for additional information about `javac` option settings and functionality.

Syntax Notes for javac Options

Keep the following in mind regarding the `javac` options syntax:

- If you want to set different classpath values for the Java compiler and for the JVM that runs SQLJ, then you must use separate settings, one with a `-C` prefix and one with a `-J` prefix. Otherwise, no prefix is required.
- Do not use the `-C` prefix to specify the `-d` or `-encoding` compiler options. Note that this also means that SQLJ and the compiler use the same settings for `-d` and `-encoding`.
- You can optionally use the `-C` prefix for `-depend`, `-deprecation`, `-g`, `-nowarn`, `-O`, and `-verbose`.
- All `javac` options, aside from those that are also SQLJ options (`-classpath`, `-d`, and `-encoding`) require the `compile.` prefix if you set them in a properties file.
- For consistency, it is advisable to use an equal sign (=) for options that take values, but a space also works when using a compiler prefix (`-C` on the command line or `compile.` in a properties file).

Example

The following example, which is a single wraparound command line, uses the `-C-bootclasspath`, `-C-extdirs`, and `-C-target options`.

```
% sqlj -vm=/usr/local/packages/jdk1.5.2/bin/java
       -compiler-executable=/usr/local/packages/jdk1.5.2/bin/javac
       -C-bootclasspath=/usr/local/packages/jdk1.5.1/jre/lib/rt.jar
       -C-extdirs="" -C-target=1.1.8 Demo.sqlj
```

Profile Customizer Options

Profile customizer options, that is, options for the customizer harness front end, the default Oracle customizer, and special customizers for debugging and deployment-time semantics-checking, are documented in "Customization and Specialized Customizers" on page A-1. This is relevant for ISO standard code generation only (`-codegen=iso`).

Command-Line Syntax and Operations

The general sequence of events triggered by running the script `sqlj` was discussed in "SQLJ Translation Steps" on page 2-5. This section will add some operational details to that discussion, as part of this overview of the command line.

Use of Command-Line Arguments

Recall the typical general syntax for the command line:

```
% sqlj <optionlist> filelist
```

When the `sqlj` script invokes a JVM, it passes all of its command-line arguments to the JVM, which later passes them elsewhere, such as to the Java compiler or profile customizer, as appropriate.

Use an equal sign (=) to specify option and flag settings, although for simplicity you do not have to specify `=true` to turn on a flag. Typing the flag name alone will suffice.

However, you must specify `=false` to turn a flag off. A flag will not toggle from its previous value. For example:

`-linemap=true` or just `-linemap` to enable line-mapping

`-linemap=false` to disable line-mapping

> **Note:** If the same option appears more than once on the command line or in the properties file, then the last value is used.

Arguments from the Option List

Option list arguments are used in the following ways:

- Options not designated by the `-J`, `-C`, or `-P` prefixes are SQLJ options (except for directly supported compiler options) and are passed to the SQLJ translator as the JVM invokes it.
- Options designated by the `-J` prefix are JVM options and are used by the JVM directly. Such options must be specified on the command line or in the `SQLJ_OPTIONS` environment variable. As with translator options, use an equal sign (=) in setting the option, such as:

  ```
  -J-Djavac.pipe.output=true
  ```

  If you want to set different classpath values for the Java compiler and for the JVM that runs SQLJ, you must use separate settings, one with a `-C` prefix and one with a `-J` prefix.
- Options designated by the `-C` prefix are Java compiler options and are passed to the compiler as the JVM invokes it. Compiler options taking values require special support, which has been implemented for `javac` options. You can use an equal sign for these, as follows (though a space also works):

  ```
  -C-bootclasspath=/usr/local/packages/jdk1.5.1/jre/lib/rt.jar
  ```

- Options designated by the `-P` prefix are SQLJ profile customizer options and are passed to the customizer as the JVM invokes it (relevant only for ISO standard code generation, `-codegen=iso`). As with translator options, use an equal sign (=) in setting the option, such as:

  ```
  -P-user=scott/tiger
  ```

  Any profile customization other than what SQLJ performs automatically is considered an advanced feature.

  > **See Also:** Appendix A, "Customization and Specialized Customizers"

Arguments from the File List

The SQLJ front end parses the file list, processes wildcard characters, and expands file names. By default, files are processed as follows:

- The `.sqlj` files are processed by the SQLJ translator, Java compiler, and SQLJ profile customizer (profile customizer for `-codegen=iso` only).
- The `.java` files are processed by the Java compiler and are also used by the SQLJ translator for type resolution.
- The `.ser` profiles and `.jar` files are processed only by the profile customizer (relevant only for `-codegen=iso`).

Note that you can specify `.sqlj` files together with `.java` files on the command line, or you can specify `.ser` files together with `.jar` files, but you cannot mix the two categories. If you have `.sqlj` files and `.java` files with interdependencies, each requiring access to code in the others, then enter them all on the command line for a single execution of SQLJ. You *cannot* specify them for separate executions of SQLJ, because then SQLJ would be unable to resolve all the types.

> **Note:** As an alternative to entering `.java` file names on the command line, you can enable the `-checksource` option and then ensure that the `.java` files are in the classpath.

Processing to Avoid Source Conflicts

The SQLJ translator takes steps to try to prevent having multiple source files define the same class in the same location. If your command-line file list includes multiple references to the same `.sqlj` or `.java` file, then all but the first reference are discarded from the command line. In addition, if you list a `.java` and `.sqlj` file with the same base name and in the same location without using the `-dir` option, then only the `.sqlj` file is processed. This processing also applies to wildcard characters.

Consider the following command-line examples, where `%` is the system prompt. Assume that your current directory is `/myhome/mypackage`, which contains the files `Foo.sqlj` and `Foo.java`:

```
% sqlj Foo.sqlj /myhome/mypackage/Foo.sqlj
```

These both refer to the same file, so the translator discards `/myhome/mypackage/Foo.sqlj` from the command line.

```
% sqlj Foo.sqlj Foo.java
```

The translator discards `Foo.java` from the command line. Otherwise, this command line would result in the translator writing and reading from `Foo.java` in the same execution.

```
% sqlj Foo.*
```

Again, the translator discards `Foo.java` from the command line. Otherwise, the translator would find both `Foo.sqlj` and `Foo.java`, which again would cause it to write and read from `Foo.java` in the same execution.

```
% sqlj -dir=outdir -d=outclasses Foo.sqlj Foo.java
```

This is fine, because the generated `Foo.java` will be in the `outdir` subdirectory, while the `Foo.java` being read is in the `/myhome/mypackage` directory. Presuming that `Foo.java` and `Foo.sqlj` define classes in different packages, the `.class` files created by Java compilation will be placed in different subdirectories under the `outclasses` directory hierarchy.

This processing of the command line means that you can, for example, type the following command and have it execute without difficulty (with file references being automatically discarded as necessary):

```
% sqlj *.sqlj *.java
```

This is convenient in many situations.

Command-Line Example and Results

The following is a sample command line, where % is the system prompt. This example uses some advanced concepts more fully explained later in this chapter, but is presented in the interest of showing a complete example of command-line syntax.

```
% sqlj -J-Duser.language=ja  -warn=none -J-prof -encoding=SJIS *Bar.sqlj Foo*.java
```

The `sqlj` script invokes a JVM, passes it the class name of the SQLJ translator, then passes it the command-line arguments. The JVM passes the SQLJ options to the translator and compiler. If there are any options for the JVM, as designated by `-J`, then the script passes them to the JVM ahead of the translator class file name (just as you would type Java options prior to typing the class file name if you were invoking Java manually). There is no customization in this example, because it uses the default Oracle-specific code generation.

After these steps are completed, the results are equivalent to the user having typed the following (presuming `SushiBar.sqlj`, `DiveBar.sqlj`, `FooBar.java`, and `FooBaz.java` were all in the current directory):

```
% java -Duser.language=ja -prof sqlj.tools.Sqlj -warn=none -encoding=SJIS
SushiBar.sqlj DiveBar.sqlj FooBar.java FooBaz.java
```

Note that this is one wraparound command line.

> **See Also:** "Options to Pass to the Java Virtual Machine (-J)" on page 8-48

Echoing the Command Line without Executing

You can use the SQLJ `-n` option (or, alternatively, `-vm=echo`) to echo the command line that the `sqlj` script would construct and pass to the SQLJ translator, without executing it. This includes settings in the `SQLJ_OPTIONS` environment variable as well as on the command line, but does not include settings in properties files.

> **See Also:** "Command Line Echo without Execution (-n)" on page 8-20

Properties Files for Option Settings

You can use properties files, instead of the command line, to set options for the SQLJ translator, Java compiler, and SQLJ profile customizer.

In addition, if your Java compiler will be running in a separate JVM and you want to specify options to this JVM regarding operation of the compiler, then you can use properties files to supply such options. Such options are passed to the JVM at the time the compiler is run, after the SQLJ translation step. However, it is typical to pass options to the JVM of the compiler by using the command-line `-C-J` prefix.

You *cannot* use properties files to set the following SQLJ options, flags, and prefixes:

- `-classpath`
- `-help`, `-help-long`, `-help-alias`, `-C-help`, `-P-help`
- `-J`
- `-n`
- `-passes`
- `-props`
- `-version`, `-version-long`

- `-vm`

It is not possible to use properties files to specify options to the JVM, for example, because properties files are read after the JVM is invoked.

Also note that in properties files you cannot use option abbreviations recognized on the command line for compatibility with `loadjava` (`-e`, `-h`, `-p`, `-u`, `-v`).

> **Note:** Discussion of SQLJ properties files applies only to client-side SQLJ, not server-side SQLJ. There is a different mechanism for specifying options to SQLJ in the server, and only a small subset of options are supported. For information, refer to "Option Support in the Server Embedded Translator" on page 11-12.

Properties File Syntax

Option settings in a properties file are placed one per line. Lines with SQLJ options, compiler options, and customizer options can be interspersed. They are parsed by the SQLJ front end and processed appropriately.

Syntax for the different kinds of options is as follows:

- Each SQLJ option is prefixed by `sqlj.` (including the period) instead of an initial hyphen. Only options that start with this prefix are passed to the SQLJ translator. For example:

  ```
  sqlj.warn=none
  sqlj.linemap=true
  ```

- Each Java compiler option is prefixed by `compile.` (including the period) instead of `-C-`. Options that start with this prefix are passed to the Java compiler. For example:

  ```
  compile.verbose
  compile.bootclasspath=/usr/local/packages/jdk1.5.1/jre/lib/rt.jar
  ```

- General profile customization options, which apply regardless of the particular customizer you are using, are prefixed by `profile.` (including the period) instead of `-P-`. Only options that start with this prefix are passed to the profile customizer. For example:

  ```
  profile.backup
  profile.user=scott/tiger
  ```

  You can also specify options to a particular customizer by using `profile.C` as follows:

  ```
  profile.Csummary
  profile.Coptparamdefaults=VAR%(50),LONG%(500),RAW_TYPE()
  ```

  Any profile customization other than the default Oracle customization is considered an advanced feature.

  > **See Also:** Appendix A, "Customization and Specialized Customizers"

- Comment lines start with a pound sign (`#`). For example:

  ```
  # Comment line.
  ```

- Blank lines are also permitted.

As on the command line, a flag can be enabled or disabled in a properties file with `true/false`, `on/off`, `1/0`, or `yes/no`. A flag can also be enabled simply by entering it without a setting, such as the following:

```
sqlj.linemap
```

> **Note:** For consistency, it is best to always use the equal sign (=) in a properties file for options that take values, even though there are some circumstances where a space also works.

Properties File: Simple Example

The following are sample properties file entries:

```
# Set user and JDBC driver
sqlj.user=scott
sqlj.driver=oracle.jdbc.OracleDriver

# Turn on the compiler verbose option
compile.verbose
```

These entries are equivalent to having the following on the SQLJ command line:

```
% sqlj -user=scott -driver=oracle.jdbc.OracleDriver -C-verbose
```

Properties File: Nondefault Connection Context Classes

Following is a sample properties file that specifies settings for a connection context class, `SourceContext`, that you declared:

```
# JDBC driver
sqlj.driver=oracle.jdbc.OracleDriver

# Oracle 9.2 on spock.natdecsys.com
sqlj.user@SourceContext=sde
sqlj.password@SourceContext=fornow
sqlj.url@SourceContext=jdbc:oracle:thin:@207.67.155.3:1521/myservice

# Warning settings
sqlj.warn=all

# Cache
sqlj.cache=on
```

Default Properties Files

Regardless of whether a properties file is specified on the SQLJ command line, the SQLJ front end looks for files named `sqlj.properties`. It looks for them in the Java home directory, the user home directory, and the current directory, in that order. It processes each `sqlj.properties` file it finds, overriding previously set options as it encounters new ones. Thus, options set in the `sqlj.properties` file in the current directory override those set in the `sqlj.properties` file in the user home directory or Java home directory.

See Also: "Order of Precedence of Option Settings" on page 8-15

SQLJ_OPTIONS Environment Variable for Option Settings

The Oracle SQLJ implementation supports an environment variable called SQLJ_OPTIONS as an alternative to the command line for setting SQLJ options. Any option referred to as command line only, meaning it cannot be set in a properties file, can also be set using the SQLJ_OPTIONS variable.

You can use the SQLJ_OPTIONS variable to set any SQLJ option, but it is intended especially for option settings to be passed to the JVM. And it is particularly useful for command-line-only options, such as -classpath, that you use repeatedly with the same setting.

Following is an example of a SQLJ_OPTIONS setting:

```
-vm=jview -J-verbose
```

When you use SQLJ_OPTIONS, SQLJ effectively inserts the SQLJ_OPTIONS settings, in order, at the beginning of the SQLJ command line, prior to any other command-line option settings.

Note: Generally, syntax in SQLJ_OPTIONS is the same as on the command line, but this may depend on your operating system. There can be operating system specific restrictions. For example, on Microsoft Windows 95 you use the Environment tab in the System control panel. Additionally, because Windows 95 does not support the equal sign (=) in variable settings, SQLJ supports the use of # instead of = in setting SQLJ_OPTIONS. Refer to your operating system documentation.

Order of Precedence of Option Settings

SQLJ takes option settings in the following order:

1. Sets options to default settings, where applicable.
2. Looks for a sqlj.properties file in the Java home directory. If it finds one, it sets options as specified there.
3. Looks for a sqlj.properties file in the user home directory. If it finds one, it sets options as specified there.
4. Looks for a sqlj.properties file in the current directory. If it finds one, it sets options as specified there.
5. It looks for option settings in the SQLJ_OPTIONS environment variable and effectively prepends them to the beginning of the command line. It sets options as specified in SQLJ_OPTIONS.
6. It looks for option settings on the command line and sets options as specified there. As SQLJ processes the command line, it looks in any file specified by the -props option and sets options as specified there.

Note:

- At each step, SQLJ overrides any previous settings for any given option.
- In the `sqlj.properties` files, SQLJ reads option settings from top to bottom, with later entries taking precedence over earlier entries.
- If there is a properties file specified by the `-props` option on the command line, SQLJ effectively inserts the option settings of the file into the position on the command line where the `-props` option was specified.
- SQLJ reads options on the command line, with options from a `-props` file inserted, in order from left to right. Any later (right-hand) setting takes precedence over earlier (left-hand) settings.

Example

Presume SQLJ is run as follows:

```
% sqlj -user=scott -props=myprops.properties -dir=/home/java
```

And presume the file `myprops.properties` is in the current directory and contains the following entries:

```
sqlj.user=tony
sqlj.dir=/home/myjava
```

These settings are processed as if they were inserted into the command line where the `-props` option was specified. Therefore, the `tony` entry takes precedence over the `scott` entry for the `user` option, but the `/home/java` entry takes precedence over the `/home/myjava` entry for the `dir` option.

Basic Translator Options

This section documents the syntax and functionality of the basic flags and options you can specify in running SQLJ. These options enable you to run in a fairly standard mode of operation. For options that can also be specified in a properties file, that syntax is noted as well.

This section covers the following topics:

See Also:

- "Properties Files for Option Settings" on page 8-12
- "Advanced Translator Options" on page 8-47
- "Translator Support and Options for Alternative Environments" on page 8-62

Basic Options for the Command Line Only

The following basic options can be specified only on the SQLJ command line or, equivalently, in the `SQLJ_OPTIONS` environment variable:

- `-props`
- `-classpath`
- `-help`, `-help-long`, `-help-alias`, `-P-help`, `-C-help`
- `-version`, `-version-long`
- `-n`

These options *cannot* be specified in properties files. The command-line-only flags (`-help`, `-version`, and `-n`) do *not* support `=true` syntax. Enable them by typing only the flag name, as follows:

```
sqlj -version-long
```

Note: Additionally, there are advanced options, flags, and prefixes that can be set only on the command line or in `SQLJ_OPTIONS`: `-J`, `-passes`, and `-vm`.

Input Properties File (-props)

The `-props` option specifies a properties file from which SQLJ can read option settings. The command-line syntax is as follows:

```
-props=filename
```

For example:

```
-props=myprops.properties
```

Classpath for Java Virtual Machine and Compiler (-classpath)

For compatibility with the syntax of most JVMs and compilers, SQLJ recognizes the `-classpath` option if it is specified on the command line. In setting this option, you can use either a space, as with most JVMs or compilers, or the equal sign (=), as with other SQLJ options. The following examples (both for a UNIX environment) demonstrate this:

```
-classpath
.:$ORACLE_HOME/jdbc/lib/ojdbc5.jar:$ORACLE_HOME/sqlj/lib/translator.jar:$ORACLE_HO
ME/sqlj/lib/runtime12.jar

-classpath=
.:$ORACLE_HOME/jdbc/lib/ojdbc5.jar:$ORACLE_HOME/sqlj/lib/translator.jar:$ORACLE_HO
ME/sqlj/lib/runtime12.jar
```

The `-classpath` option sets the Java classpath for both the JVM and the Java compiler. If you do not want to use the same classpath for both, then set them separately using the SQLJ `-J` and `-C` prefixes.

> **See Also:** "Prefixes that Pass Option Settings to Other Executables" on page 8-48

> **Note:** As with other options described in this chapter, if you use = in setting the `-classpath` option, then it is stripped out when the option string is passed to the JVM and compiler, because JVMs and compilers do not support the = syntax in their option settings.

The command-line syntax is as follows

```
sqlj -classpath=class_path
```

For example:

```
sqlj
-classpath=$ORACLE_HOME/jdbc/lib/ojdbc5.jar:$ORACLE_HOME/sqlj/lib/translator.jar:$
ORACLE_HOME/sqlj/lib/runtime2.jar
```

SQLJ Option Information (-help)

The following settings of the `-help` flag, specified on the command line, instruct SQLJ to display varying levels of information about SQLJ options:

- `-help`
- `-help-long`
- `-help-alias`

You can enable this option by typing the desired setting on the command line as in the following examples:

```
% sqlj -help

% sqlj -help-long

% sqlj -help-alias
```

No input-file translation is performed when you use the `-help` flag in any of these forms, even if you include file names and other options on the command line as well. SQLJ assumes that you either want to run the translator or you want help, but not both.

You can also receive information about the profile customizer or Java compiler, requesting help through the `-P` and `-C` prefixes, as in the following examples. As with the `-help` flag, no translation is performed if you request customizer or compiler help.

```
% sqlj -P-help

% sqlj -C-help
```

As with other command-line-only flags, `-help` (as well as `-P-help` and `-C-help`) does *not* support `=true` syntax. Enable it by typing only the desired flag setting.

Note:

- For compatibility with the `loadjava` utility, `-h` is recognized as equivalent to `-help` when specified on the command line.
- You can use multiple `-help` flag settings on the same command line, including `-P-help` and `-C-help`.
- Although `-P` and `-C` settings can generally be set in properties files, `-P-help` and `-C-help` are for only the command line.
- Help is also provided if you run SQLJ without specifying any files to process. This is equivalent to using the `-help` setting.

The most basic level of help is achieved by specifying the `-help` setting. This provides the following:

- A synopsis of the most frequently used SQLJ options
- A listing of the additional `-help` flag settings available

The `-help-long` setting provides a complete list of SQLJ option information, including the following for each option:

- Option name
- Option type (the Java type that the option takes as input, such as `int` or `String`)
- Description
- Current value
- How the current value was set (from the command line, from a properties file, or by default)

Note: It is often useful to include other option settings on the command line with a `-help-long` option, especially with complex options, such as `-warn`, or combinations of options, so that you can see what option settings resulted from your actions.

The `-help-alias` setting provides a synopsis of the command-line abbreviations supported for compatibility with the `loadjava` utility.

The command-line syntax is as follows:

```
sqlj help_flag_settings
```

For example:

```
sqlj -help
sqlj -help -help-alias
sqlj -help-long
sqlj -warn=none,null -help-long
sqlj -help-alias
```

By default, these settings are disabled.

SQLJ Version Number (-version)

The following settings of the `-version` flag, specified on the command line, instruct SQLJ to display varying levels of information about SQLJ and JDBC driver versions:

- `-version`
- `-version-long`

You can enable this option by typing the desired setting on the command line as in the following examples:

```
% sqlj -version

% sqlj -version-long
```

No input-file translation is performed when you use the `-version` option, even if you include file names and other options on the command line. SQLJ assumes that you either want to run the translator or you want version information, but not both. Properties files and anything else you type on the command line are ignored. As with other command-line-only flags, `-version` does *not* support the `=true` syntax. Enable it by typing only the flag name.

The `-version` setting displays the SQLJ release number, as follows:

```
sqlj -version
Oracle SQLJ Release 10.2.0.1.0 Production
Copyright  1997, 2005, Oracle Corporation. All Rights Reserved.
```

The `-version-long` setting displays information about the SQLJ and SQLJ run time library release, the JDBC driver release number if one can be found, and the Java environment. For example, if an Oracle JDBC driver is used, this option would display something as follows:

```
sqlj -version-long
Oracle SQLJ Release 10.2.0.1.0 Production
Copyright  1997, 2005, Oracle Corporation. All Rights Reserved.
JDBC version: Oracle JDBC driver version 10.2 (10.2.0.1.0)
Java version: 1.2 (1.2.2)
```

This flag offers a good way to check your SQLJ installation and the JDBC and JDK versions you are using. The command-line syntax is as follows:

```
sqlj version_flag_settings
```

For example:

```
sqlj -version
sqlj -version -version-long
sqlj -version-long
```

By default, these settings are disabled.

Command Line Echo without Execution (-n)

The `-n` flag, specified on the command line, instructs the `sqlj` script to construct the full command line that would be passed to the SQLJ translator, including any `SQLJ_OPTIONS` settings, and echo it to the user without having the SQLJ translator execute it. This includes capturing and echoing the name of the JVM that would be launched to execute the SQLJ translator and echoing the full class name of the translator. This does *not* include settings from properties files.

This is useful in displaying the following:

- The fully expanded form of any options you abbreviated, such as `-u` and other abbreviations supported for `loadjava` compatibility.

- The order in which options would be placed when the overall command string is constructed and passed to the translator.
- Possible conflicts between SQLJ_OPTIONS settings and command-line settings.

The -n option can appear anywhere on the command line or in the SQLJ_OPTIONS variable. As with other command-line-only flags, -n does *not* support the =true syntax. Enable it by typing only the flag name.

Consider a sample scenario. You have the following setting for SQLJ_OPTIONS:

```
-user=scott/tiger@jdbc:oracle:thin:@ -classpath=/myclasses/bin
```

You enter the following command line:

```
% sqlj -n -e SJIS myapp.sqlj
```

You would see the following echo:

```
java -classpath /myclasses/bin sqlj.tools.Sqlj
-user=scott/tiger@jdbc:oracle:thin:@ -C-classpath=/myclasses/bin
-encoding=SJIS myapp.sqlj
```

Note that this is all one wraparound line.

Note:

- As an alternative to -n, you can use the -vm=echo setting.
- Another effective way to check option settings is to use the -help-long flag. This displays current settings for all options, including other options you set on the command line as well as settings in properties files and in SQLJ_OPTIONS.

The command-line syntax is as follows:

```
-n
```

For example:

```
-n
```

By default, this setting is disabled.

Options for Output Files and Directories

The -encoding option specifies encoding for SQLJ input and output source files. The following options specify where SQLJ output files are placed:

- -d
- -dir

Encoding for Input and Output Source Files (-encoding)

The -encoding option specifies the encoding to be applied to .sqlj and .java input files and .java generated files for globalization support. For compatibility with javac, you can use either a space or equal sign (=) in setting this option on the command line, as in the following examples:

```
-encoding=SJIS
```

```
-encoding SJIS
```

However, if setting `sqlj.encoding` in a properties file, then use =, not a space.

When this option is specified, it is also passed to the Java compiler, unless the `-compiler-encoding-flag` is off, which uses it to specify encoding for `.java` files processed by the compiler.

Note the following:

- As with the `-classpath` and `-d` options, if you do use an = in setting the `-encoding` option, then it is stripped out when the option string is passed to the JVM and compiler. This is because JVMs and compilers do not support the = syntax in their option settings.
- For compatibility with the `loadjava` utility, `-e` is recognized as equivalent to `-encoding` when specified on the command line.
- The `-encoding` option does not apply to Java properties files, such as `sqlj.properties` and `connect.properties`. Properties files always use the encoding `8859_1`. This is a feature of Java in general, not SQLJ in particular. However, you can use Unicode escape sequences in a properties file. You can use the `native2ascii` utility to create escape sequences for a natively encoded file.

See Also: "Using native2ascii for Source File Encoding" on page 9-22

The command-line syntax is as follows:

```
-encoding=Java_character_encoding
```

For example:

```
-encoding=SJIS
```

The syntax for a properties file entry for this option is as follows:

```
sqlj.encoding=Java_character_encoding
```

For example

```
sqlj.encoding=SJIS
```

By default, this option is set to the JVM system property `file.encoding`.

Output Directory for Generated .ser and .class Files (-d)

The `-d` option specifies the root output directory for profiles generated by the SQLJ translator (relevant for ISO standard code generation, `-codegen=iso`), and is also passed to the Java compiler to specify the root output directory for `.class` files generated by the compiler. Whether profiles are generated as `.ser` files (default) or `.class` files (if the `-ser2class` option is enabled) is irrelevant for placement through the `-d` option.

Whenever a directory is specified, the output files are generated under this directory according to the package name, if applicable. For example, if you have source files in the `a.b.c` package and specify directory, `/mydir`, output files will be placed in the `/mydir/a/b/c` directory. If you specify a relative directory path, then this will be from your current directory.

For compatibility with `javac`, you can use either a space or = in setting this option on the command line, as in the following examples (both of which make `/root` the root directory for generated profile files):

```
-d=/root

-d /root
```

However, if setting `-d` in a properties file, then use `=`, not a space. For example:

```
sqlj.d=/root
```

If your current directory is `/root/home/mydir` and you set the `-d` option to the relative directory path, `mysubdir/myothersubdir`, as follows, then `/root/home/mydir/mysubdir/myothersubdir` will be the root directory for the generated profile files:

```
-d=mysubdir/myothersubdir
```

You can also use standard syntax, such as a period for the current directory or two periods to go up a level, as follows:

```
-d=.

-d=../paralleldir
```

If the `-d` option is empty or not specified, then a generated `.class` file is placed in the same directory as the corresponding `.java` file, which is according to the `-dir` option for a `.java` file generated by SQLJ, and a generated `.ser` file is placed in the same directory as the corresponding `.sqlj` file.

Note:

- You can specifically set `-d` to be empty (to override settings in a properties file, for example) as follows:

  ```
  -d=
  ```

- Throughout this discussion, slash (`/`) was used as the file separator. However, it is important to note that in specifying this or similar options, you must actually use the file separator of your operating system, as specified in the `file.separator` system property of your JVM.
- As with the `-classpath` and `-encoding` options, if you do use an equal sign (`=`) in setting the `-d` option, then it is stripped out when the option string is passed to the JVM and compiler. This is because JVMs and compilers do not support the `=` syntax in their option settings.

The command-line syntax is as follows:

```
-d=directory_path
```

For example:

```
-d=/topleveldir/mydir
```

The syntax for a properties file entry for this option is as follows:

```
sqlj.d=directory_path
```

For example:

```
sqlj.d=/topleveldir/mydir
```

This option does not have any default value.

Output Directory for Generated .java Files (-dir)

The `-dir` option specifies the root directory for `.java` files generated by the SQLJ translator. Whenever a directory is specified, the output files are generated under this directory according to the package name, if applicable. For example, if you have source files in the `a.b.c` package and specify directory, `/mydir`, then output files will be placed in the `/mydir/a/b/c` directory. If you specify a relative directory path, then it will be from your current directory.

A simple example is as follows, which will make `/root` the root directory for generated `.java` files:

```
-dir=/root
```

Consider that your current directory is `/root/home/mydir` and you set the `-dir` option to the relative directory path `mysubdir/myothersubdir` as follows:

```
-dir=mysubdir/myothersubdir
```

Then `/root/home/mydir/mysubdir/myothersubdir` will be the root directory for generated `.java` files.

You can also use standard syntax, such as a period for the current directory or two periods to go up a level, as follows:

```
-dir=.

-dir=../paralleldir
```

If the `-dir` option is not specified, then files are generated under the same directory as the original `.sqlj` source file (*not* under the current directory). If you specifically want the output directory to be the same as your `.sqlj` source directory (perhaps overriding other `-dir` settings, such as in properties files), then you can use the `-dir` option as follows:

```
-dir=
```

> **Note:** If you specify the `-dir` option but not the `-d` option, then generated `.class` files will also be placed in the directory specified by `-dir`, but generated `.ser` files will be placed in the directory of the `.sqlj` file.

The command-line syntax is as follows:

```
-dir=directory_path
```

For example:

```
-dir=/topleveldir/mydir
```

The syntax for a properties file entry for this option is as follows:

```
sqlj.dir=directory_path
```

For example:

```
sqlj.dir=/topleveldir/mydir
```

This option does not have any default value.

Connection Options

You can use the following options for the database connection for online semantics-checking:

- `-user`
- `-password`
- `-url`
- `-default-url-prefix`
- `-driver`

There is no requirement for the SQLJ translator to connect to the same database or schema as the application does at run time. The connection information in application source code can be independent of the connection information in the SQLJ options. In fact, the deployment environment might be unavailable during development and testing.

Online Semantics-Checking and User Name (-user)

Simple semantics-checking not involving a database connection is referred to as offline checking. The more thorough semantics-checking requiring a connection is referred to as online checking. Online checking offers one of the prime advantages of the SQLJ strong-typing paradigm, namely that type incompatibilities that would usually result in run-time SQL exceptions are caught during translation, before users ever run the application.

The `-user` option enables online semantics-checking and specifies the user name (schema name) for the exemplar schema, which is the sample database schema that you provide to the translator for it to use in performing the checking. You can also use the `-user` option to specify the password and URL, as opposed to using the `-password` and `-url` options separately.

Note that there is no other flag to enable or disable online semantics-checking. SQLJ enables or disables it according to the presence or absence of the `-user` option.

Note:

- Some settings of the SQLJ `-parse` option will disable online semantics-checking, overriding the effect of the `-user` option.
- For compatibility with the `loadjava` utility, `-u` is recognized as equivalent to `-user` when specified on the command line.
- User names cannot contain the characters `/` or `@`.
- You are allowed to use a space instead of = in a user name setting on the command line, as in the following examples:

```
-user scott/tiger
-user@CtxClass scott/tiger
-u scott/tiger
-u@CtxClass scott/tiger
```

- If a password contains the character `@`, then you cannot set the password through the `-user` option. You must use separate `-user` and `-password` settings.
- If your login name is a member of the `DBA` group, you may have special privilege to connect as `SYSDBA` to the `SYS` schema. In this case, you can specify the user name `SYS` or `INTERNAL`.
- For ISO code generation, the translator `-user` setting is forwarded to the profile customizer, but can be overridden by the customizer `user` setting.

The most basic usage of the `-user` option is as follows:

```
-user=scott
```

When you are using only the default connection or other instances of the `DefaultContext` class, such a setting will apply to all your SQLJ executable statements. This example results in online checking against the `scott` schema.

You can also specify the password, URL, or both along with the user name, using syntax as in the following examples (with `/` preceding the password and `@` preceding the URL):

```
-user=scott/tiger

-user=scott@jdbc:oracle:oci:@

-user=scott/tiger@jdbc:oracle:oci:@
```

Otherwise, the URL can be specified through the `-url` option, and the password can be specified interactively or through the `-password` option.

You can disable online semantics-checking by setting the `-user` option to an empty string, as follows:

```
-user=
```

Again, when you are using only the default connection or other instances of the `DefaultContext` class, this will apply to all your SQLJ executable statements.

Disabling online semantics-checking is useful, for example, if you have online checking enabled in a properties file but want to override that on the command line, or

have it enabled in the default properties file but want to override that in a user-specified properties file, specified using the `-props` option.

There is also a special user name, `URL.CONNECT`, which you can use when the URL specifies the user and password as well as the other details of the connection.

> **See Also:** "Connection URL for Online Semantics-Checking (-url)" on page 8-30

If you declare and use additional connection context classes in your application, then you can specify `-user` settings for the testing of SQLJ executable statements that use instances of those classes. Specify a user name for online checking against a particular connection context class, for example, `CtxClass`, as follows:

```
-user@CtxClass=scott
```

This results in online checking against the `scott` schema for any of your SQLJ executable statements that specify a connection context instance of `CtxClass`.

As with the default connection context class, you can also specify the password or URL in your `-user` setting for a particular connection context class, as in the following example:

```
-user@CtxClass=scott/tiger@jdbc:oracle:oci:@
```

The `CtxClass` connection context class must be declared in your source code or previously compiled into a `.class` file.

Use the `-user` option separately for each connection context class for which you want to enable online checking and set a user name. These settings have no influence on each other. For example:

```
-user@CtxClass1=user1 -user@CtxClass2=user2 -user@CtxClass3=user3
```

When you are using multiple connection context classes in your application, a `-user` setting that does not specify a class will apply to the `DefaultContext` class as well as to all classes for which you do not otherwise specify a `-user` setting. Presumably, though, you will specify a `-user` setting for each connection context class, given that different connection context classes are typically intended for use with different sets of SQL objects.

Consider a situation where you have declared connection context classes `CtxClass1`, `CtxClass2`, and `CtxClass3` and you set `-user` as follows:

```
-user@CtxClass2=scott/tiger -user=bill/lion
```

Any statement in your application that uses an instance of `CtxClass2` will be checked against the `scott` schema. Any statement that uses an instance of `DefaultContext`, `CtxClass1`, or `CtxClass3` will be checked against the `bill` schema.

In addition, once you enable online checking by setting the `-user` option, you can disable online checking for a particular connection context by setting the `-user` option again with an empty user name for that connection context. For example, consider the following setting:

```
-user@CtxClass2=
```

This disables online semantics-checking for any SQLJ executable statements that specify a connection object that is an instance of `CtxClass2`.

You can disable online semantics-checking for the default connection context class and any other connection context classes for which you do not specify a user name as follows:

```
-user=
```

The general command-line syntax for this option is as follows:

```
-user<@conn_context_class>=username</password><@url>
```

For example:

```
-user=scott
-user=scott/tiger
-user=scott@jdbc:oracle:oci:@
-user=scott/tiger@jdbc:oracle:oci:@
-user=
-user=URL.CONNECT
-user@CtxClass=scott/tiger
-user@CtxClass=
```

The syntax for a properties file entry for this option is as follows:

```
sqlj.user<@conn _context_class>=username</password><@url>
```

For example:

```
sqlj.user=scott
sqlj.user=scott/tiger
sqlj.user=scott@jdbc:oracle:oci:@
sqlj.user=scott/tiger@jdbc:oracle:oci:@
sqlj.user=
sqlj.user=URL.CONNECT
sqlj.user@CtxClass=scott/tiger
sqlj.user@CtxClass=
```

This option does not have a default value. By default, there is no online-semantics checking.

> **Note:** Be aware of the difference in format between specifying user, password, and URL in the `user` option and specifying them in the `-url` option. In the `-url` option, the user name and password are included in the URL, immediately following the JDBC driver type. In the `-user` option they precede the URL.

User Password for Online Semantics-Checking (-password)

The `-password` option specifies the user password for the database connection for online semantics-checking. For the `-password` setting to be meaningful, the `-user` option must also be set.

You can also specify the password as part of the `-user` option setting. Do not use the `-password` option for a connection context class if you have already set its password in the `-user` option, which takes precedence.

For the most part, functionality of the `-password` option parallels that of the `-user` option. That is, if your application uses only the default connection or other instances of `DefaultContext`, then the following will set the password for the schema to be used in checking all of your SQLJ statements:

```
-password=tiger
```

If you declare and use additional connection context classes, `CtxClass1` for example, then you will presumably use the `-user` option to specify additional exemplar schemas to use in testing statements that use those connection context classes. Similarly, use the `-password` option to specify passwords for those schemas, as in the following example:

```
-password@CtxClass1=tiger
```

A connection context class without a password setting, either through the `-password` setting or the `-user` setting, uses the password setting for the default connection context class. If you set no password for the default connection context class, then SQLJ prompts you interactively for that password. If you also set no password for a user-defined connection context class, then SQLJ prompts you interactively for that password as well. An exception to this discussion is where user name `URL.CONNECT` is used. In this case, user name and password are determined from the string specified in the `-url` setting and any setting of the `-password` option is ignored.

You can specifically set an empty password to override other settings of the `-password` option, such as in a properties file, and be prompted interactively. You can do this for the `DefaultContext` class or any particular connection context class, as in the following examples:

```
-password=

-password@CtxClass1=
```

If you actually want to use an empty password to log in, specify `EMPTY.PASSWORD` as in the following examples:

```
-password=EMPTY.PASSWORD

-password@CtxClass2=EMPTY.PASSWORD
```

However Oracle Database 11*g* does not permit an empty password.

Note:

- When specified on the command line, `-p` is recognized as equivalent to `-password`.
- You are allowed to use a space instead of = in a password setting on the command line, as in the following examples:

  ```
  -password tiger
  -password@CtxClass tiger
  -p tiger
  -p@CtxClass tiger
  ```

- For ISO code generation, the translator `-password` setting is forwarded to the profile customizer, but can be overridden by the customizer `password` setting.

The command-line syntax for this option is as follows:

```
-password<@conn_context_class>=user_password
```

For example:

```
-password=tiger
```

```
-password=
-password=EMPTY.PASSWORD
-password@CtxClass=tiger
```

The syntax for a properties file entry for this option is as follows:

```
sqlj.password<@conn_context_class>=user_password
```

For example:

```
sqlj.password=tiger
sqlj.password=
sqlj.password=EMPTY.PASSWORD
sqlj.password@CtxClass=tiger
```

This option does not have a default value. Either the password for `DefaultContext` is used or the user is prompted.

Connection URL for Online Semantics-Checking (-url)

The `-url` option specifies a URL for establishing a database connection for online semantics-checking. As necessary, the URL can include a host name, port number, and database service name (or SID, which is deprecated in Oracle Database 11*g*).

You can also specify the URL as part of the `-user` option setting. Do not use the `-url` option for a connection context class if you have already set its URL in the `-user` option, which takes precedence.

For the most part, functionality of the `-url` option parallels that of the `-user` option. That is, if your application uses only the default connection or other instances of `DefaultContext`, then the following example would set the URL to use for the connection for checking all your SQLJ statements:

```
-url=jdbc:oracle:oci:@
```

Alternatively, to include the host name, port number, and service name:

```
-url=jdbc:oracle:thin:@myhost:1521/myservice
```

If you do not begin a URL setting with `jdbc:`, then the setting is assumed to be of the form *`host:port/servicename`* and, by default, is automatically prefixed with the following:

```
jdbc:oracle:thin:@
```

A `-url` setting of `localhost:1521/myservice` would result in the following URL:

```
jdbc:oracle:thin:@localhost:1521/myservice
```

You can remove or alter this default prefix with the `-default-url-prefix` option.

See Also: "Default URL Prefix (-default-url-prefix)" on page 8-32

You can specify the user and password in the `-url` setting, instead of in the `-user` and `-password` settings. In such a case, set `-user` to `URL.CONNECT`, as follows:

```
-url=jdbc:oracle:oci:scott/tiger@ -user=URL.CONNECT
```

If you declare and use additional connection context classes, `CtxClass1` for example, you will presumably specify additional exemplar schemas to use in testing statements that use those connection context classes. You can use the `-url` option to specify URLs for those schemas, as in the following example:

```
-url@CtxClass1=jdbc:oracle:oci:@
```

Any connection context class without a URL setting, either through the -url setting or the -user setting, uses the URL setting for the default connection context class, presuming a URL has been set for the default context class.

Note:

- Remember that any connection context class with a URL setting must also have a user name setting for online checking to occur.
- You are allowed to use a space instead of = in a URL setting on the command line, as in the following examples:

  ```
  -url jdbc:oracle:oci:@
  -url@CtxClass jdbc:oracle:oci:@
  ```

- For ISO code generation, the translator -url setting is forwarded to the profile customizer, but can be overridden by the customizer url setting.

The command-line syntax for this option is as follows:

```
-url<@conn_context_class>=URL
```

For example:

```
-url=jdbc:oracle:oci:@
-url=jdbc:oracle:thin:@hostname:1521/myservice
-url=jdbc:oracle:oci:scott/tiger@
-url=hostname:1521/myservice
-url@CtxClass=jdbc:oracle:oci:@
```

The syntax for a properties file entry for this option is as follows:

```
sqlj.url<@conn_context_class>=URL
```

For example:

```
sqlj.url=jdbc:oracle:oci:@
sqlj.url=jdbc:oracle:thin:@hostname:1521/myservice
sqlj.url=jdbc:oracle:oci:scott/tiger@
sqlj.url=hostname:1521/myservice
sqlj.url@CtxClass=jdbc:oracle:oci:@
```

The default value for this option is:

```
jdbc:oracle:oci:@
```

Note: Be aware of the difference in format between specifying user, password, and URL in the -user option and specifying them in the -url option. In the -url option, the user name and password are included in the URL, immediately following the JDBC driver type. In the -user option, they precede the URL.

Default URL Prefix (-default-url-prefix)

Use the `-default-url-prefix` option to alter or remove the default prefix. The following is the default prefix for any URL setting you specify that does not already start with `jdbc:`

```
jdbc:oracle:thin:@
```

This enables you to use a shorthand in specifying a URL setting, either in the `-user` option or the `-url` option. It is permissible to specify only the host, port, and service name (or SID, which is deprecated) of the database. As an example, presume you set a URL as follows:

```
-url=myhost:1521/myservice

-user=scott/tiger@myhost:1521/myservice
```

By default, the URL will be interpreted to be the following:

```
jdbc:oracle:thin:@myhost:1521/myservice
```

If you specify a full URL that starts with `jdbc:`, then the default prefix will not be used.

However, if you want your URL settings to default to the JDBC Oracle Call Interface (OCI) driver, for example, instead of the JDBC Thin driver, then set the default prefix as follows:

```
-default-url-prefix=jdbc:oracle:oci:@
```

If you do not want any prefix, then set the `-default-url-prefix` option to an empty string, as follows:

```
-default-url-prefix=
```

The command-line syntax for this option is as follows:

```
-default-url-prefix=url_prefix
```

For example

```
-default-url-prefix=jdbc:oracle:oci:@
-default-url-prefix=
```

The syntax for a properties file entry for this option is as follows:

```
sqlj.default-url-prefix=url_prefix
```

For example:

```
sqlj.default-url-prefix=jdbc:oracle:oci:@
sqlj.default-url-prefix=
```

The default value for this option is:

```
jdbc:oracle:thin:@
```

JDBC Drivers to Register for Online Semantics-Checking (-driver)

The `-driver` option specifies the JDBC driver class to register for interpreting JDBC connection URLs for online semantics-checking. Specify a driver class or comma-delimited list of classes. The default, `OracleDriver`, supports the Oracle JDBC OCI, JDBC Thin, and server-side JDBC drivers for use with Oracle Database 11*g*.

The command-line syntax for this option is as follows:

```
-driver=driver1<,driver2,driver3,...>
```

For example:

```
-driver=oracle.jdbc.OracleDriver
-driver=oracle.jdbc.OracleDriver,sun.jdbc.odbc.JdbcOdbcDriver
```

The syntax for a properties file entry for this option is as follows:

```
sqlj.driver=driver1<,driver2,driver3,...>
```

For example:

```
sqlj.driver=oracle.jdbc.OracleDriver
sqlj.driver=oracle.jdbc.OracleDriver,sun.jdbc.odbc.JdbcOdbcDriver
```

The default value for this option is:

```
oracle.jdbc.OracleDriver
```

Options for Reporting and Line-Mapping

The following options specify what types of conditions SQLJ should monitor, whether to generate real-time error and status messages and whether to include cause and action information with translator error messages:

- `-warn`
- `-status`
- `-explain`

The following options enable line-mapping from the generated Java `.class` file back to the `.sqlj` source file, so that you can trace run-time errors back to the appropriate location in your original source code:

- `-linemap`
- `-jdblinemap`

Use `-jdblinemap` in conjunction with the Sun Microsystems `jdb` debugger. Otherwise, use `-linemap`.

Translator Warnings (-warn)

There are various warnings and informational messages that the SQLJ translator can display as dictated by conditions it encounters during the translation. The `-warn` option consists of a set of flags that specify which of those warnings and messages should be displayed, in other words, which conditions should be monitored and which should be ignored. All the flags for this option must be combined into a single, comma-delimited string.

Table 8–3 lists the conditions that can be tested, what the `true` and `false` flag values are for each condition, what a `true` flag value means, and which value is the default.

Table 8–3 Tests and Flags for SQLJ Warnings

Test and Flag Functions	TRUE/FALSE Values
Test for requirement of subtypes of declared object type in an inheritance hierarchy: Enable `cast` to receive warnings when usage of SQL object types in a SQL inheritance hierarchy requires that subtypes of a declared type must be passed at run time.	`cast` (default) `nocast`
Data precision test: Enable `precision` to receive warnings if there was a possible loss of precision when moving values from database columns to Java host variables.	`precision` (default) `noprecision`
Conversion loss test for nullable data: Enable `nulls` to receive warnings if there was possible conversion loss when moving nullable columns or nullable Java types from database columns to Java host variables.	`nulls` (default) `nonulls`
Portability test: Enable `portable` to check SQLJ clauses for portability and receive warnings if there are nonportable clauses. (Where nonportable refers to the use of extensions to the SQLJ standard, such as vendor-specific types or features.)	`portable` `noportable` (default)
Strict matching test for named iterators: Enable `strict` to instruct SQLJ to require that the number of columns selected from the database must equal the number of columns in the named iterator being populated. A warning is issued for any column in the database cursor for which there is no corresponding column in the iterator. The `nostrict` setting allows more (but not fewer) columns in the database cursor. Unmatched columns are ignored.	`strict` (default) `nostrict`
Translation-time informational messages: Enable `verbose` to provide additional informational messages about the translation process, such as what database connections were made for online checking.	`verbose` `noverbose` (default)
Global enabling or disabling of warnings: Use `all` or `none` to enable or disable all warnings.	`all` `none`

The `verbose/noverbose` flag works differently from the others. It does not enable a particular test but enables output of general informational messages about the semantics-checking.

> **Note:** Do not confuse `-warn=verbose` with the `-status` flag. The `-status` flag provides real-time informational messages about all aspects of SQLJ translation: translation, semantics-checking, compilation, and profile customization, if applicable. The `-warn=verbose` flag results in additional reporting about the translation phase only.

The global `all/none` flag takes priority over default settings. You can use it to enable or disable all flags, or to serve as an initialization to ensure that all flags are off before you turn selected flags on, or all flags are on before you turn selected flags off.

The `all` setting is equivalent to the following:

```
cast,precision,nulls,portable,strict,verbose
```

And the `none` setting is equivalent to the following:

```
nocast,noprecision,nonulls,noportable,nostrict,noverbose
```

There is no default for `all/none`. There are only defaults for individual flags.

Following are some examples:

- Use the following sequence to ensure that only the `nulls` flag is on:

  ```
  -warn=none,nulls
  ```

- The following sequence will have the same result, because the `verbose` setting will be overridden:

```
-warn=verbose,none,nulls
```

- Use the following to ensure that everything except the portability flag is on:

  ```
  -warn=all,noportable
  ```

- This sequence will have the same result, because the `nonulls` setting will be overridden:

  ```
  -warn=nonulls,all,noportable
  ```

Other than placement of the `all/none` flag, the order in which flags appear in a `-warn` setting is unimportant, except in the case of conflicting settings. If there are conflicts, such as in `-warn=portable,noportable`, then the last (right-most) setting is used.

Separate settings of the `-warn` option in properties files and on the command line are *not* cumulative. Only the last setting is processed. In the following example, the `-warn=portable` setting is ignored. That flag and all other flags besides `nulls/nonulls` are set according to their defaults:

```
-warn=portable -warn=nonulls
```

> **Note:** The cast, precision, nullability, and strictness tests are part of online semantics-checking and require a database connection.

The command-line syntax for this option is as follows:

```
-warn=comma-delimited_list_of_flags
```

For example:

```
-warn=none,nulls,precision
```

The syntax for a properties file entry for this option is as follows:

```
sqlj.warn=comma-delimited_list_of_flags
```

For example:

```
sqlj.warn=none,nulls,precision
```

The default value for this option is as follows:

```
cast,precision,nulls,noportable,strict,noverbose
```

Real-Time Status Messages (-status)

The `-status` flag instructs SQLJ to display additional status messages throughout all aspects of the SQLJ process: translation, semantics-checking, compilation, and customization. Messages are displayed as each file is processed and at each stage of the SQLJ operation.

Note:

- Do not confuse `-warn=verbose` with the `-status` flag. The `-status` flag provides real-time informational messages about all aspects of SQLJ translation. The `-warn=verbose` flag results in additional reporting about the translation phase only.
- For compatibility with the `loadjava` utility, `-v` is recognized as equivalent to `-status` when specified on the command line.

The command-line syntax for this option is as follows:

```
-status<=true|false>
```

For example:

```
-status
```

The syntax for a properties file entry for this option is as follows:

```
sqlj.status<=true|false>
```

For example:

```
sqlj.status
```

The default value for this option is:

```
false
```

Cause and Action for Translator Errors (-explain)

The `-explain` flag instructs the SQLJ translator to include cause and action information, as available, with translator error message output for the first occurrence of each error.

The command-line syntax for this option is as follows:

```
-explain<=true|false>
```

For example:

```
-explain
```

The syntax for a properties file entry for this option is as follows:

```
sqlj.explain<=true|false>
```

For example:

```
sqlj.explain
```

The default value for this option is:

```
false
```

Line-Mapping to SQLJ Source File (-linemap)

The `-linemap` flag instructs SQLJ to map line numbers from a SQLJ source code file to locations in the corresponding `.class` file. This will be the `.class` file created during compilation of the `.java` file generated by the SQLJ translator. As a result,

when Java run-time errors occur, the line number reported by the JVM is the line number in the SQLJ source code, making it much easier to debug.

Usually, the instructions in a `.class` file map to source code lines in the corresponding `.java` file. This would be of limited use to SQLJ developers, though, as they would still need to map line numbers in the generated `.java` file to line numbers in their original `.sqlj` file.

The SQLJ translator modifies the `.class` file to implement the `-linemap` option, replacing line numbers and the file name from the generated `.java` file with corresponding line numbers and the file name from the original `.sqlj` file. This process is known as instrumenting the class file.

In performing this, SQLJ takes the following into account:

- The `-d` option setting, which determines the root directory for `.class` files
- The `-dir` option setting, which determines the root directory for generated `.java` files

---

Note:

- If you are processing a `.sqlj` file and the compilation step is skipped due to error, then no line-mapping can be performed either, because no `.class` file is available for mapping.
- When the Java compiler is invoked from SQLJ, it always reports compilation errors using line numbers of the original `.sqlj` source file, not the generated `.java` file. No option needs to be set for this mapping.
- Anonymous classes in a `.sqlj` file will not be instrumented.
- If you are using the Sun Microsystems `jdb` debugger, then use the `-jdblinemap` option instead of the `-linemap` option.

The command-line syntax for this option is as follows:

```
-linemap<=true|false>
```

For example:

```
-linemap
```

The syntax for a properties file entry for this option is as follows:

```
sqlj.linemap<=true|false>
```

For example:

```
sqlj.linemap
```

The default value for this option is:

```
false
```

Line-Mapping to SQLJ Source File for jdb Debugger (-jdblinemap)

This option is equivalent to the `-linemap` option, but you should use it instead of `-linemap` if you are using the Sun Microsystems `jdb` debugger. This is because `jdb` can access only source files with a `.java` file name extension.

With the `-jdblinemap` setting, SQLJ does the following:

- Overwrites the contents of the `.java` file generated by the translator with the contents of the original `.sqlj` file
- Preserves the `.java` file name, instead of the `.sqlj` file name, in the generated `.class` file

In this way, the SQLJ source code is accessible to `jdb`.

The command-line syntax for this option is as follows:

```
-jdblinemap<=true|false>
```

For example:

```
-jdblinemap
```

The syntax for a properties file entry for this option is as follows:

```
sqlj.jdblinemap<=true|false>
```

For example:

```
sqlj.jdblinemap
```

The default value for this option is:

```
false
```

Options for DMS

The Oracle SQLJ implementation provides translator front-end options to support DMS:

- `-instrument`: Enable instrumentation and designate a name for the application (the collective of the components being translated).
- `-components`: Specify the components (packages and classes) to be instrumented.

> **See Also:** "SQLJ Support for Oracle Performance Monitoring" on page 10-20

Instrumentation for DMS (-instrument)

Use the SQLJ `-instrument` option to enable instrumentation and specify an application name. In this context, the term application refers to all the SQLJ and Java components specified for translation in the SQLJ command line.

Possible settings of the `-instrument` option are as follows:

- *`application_name`*: To enable instrumentation and use the specified application name, optionally prefixed with a package name in the standard Java dot syntax. Use a slash (`/`), with no spaces, between the package name and the application name.
- `true`: To enable instrumentation and use the default application name, `defaultApp`.
- `false` (default): To disable instrumentation.

If instrumentation is enabled, a SQLJ DMS properties file is created. Its name and location are according to the `-instrument` setting, starting from the current directory,

according to the package name and also according to any setting of the SQLJ `-d` option. If no application name is specified, as is the case with the setting `true`, then the properties file is named `sqlmonitor.properties` in the current directory.

As a simple example, a setting of `-instrument=myapp` will result in creation of the properties file, `myapp.properties`, in the current directory.

Now consider the following example, for an application name of `stock` and the package `com.acme`:

```
% sqlj -instrument=com.acme/stock Stock.sqlj Trading.sqlj
```

In this case, the properties file `./com/acme/stock.properties` is created.

Now consider the following example:

```
% sqlj -instrument=com.acme/stock -d /home Stock.sqlj Trading.sqlj
```

In this case, because of the `-d` option, the file `/home/com/acme/stock.properties` is created.

You can also set the `-instrument` option in `sqlj.properties` as follows:

```
sqlj.instrument=com.acme/stock
```

> **Note:** A setting of `-instrument` is equivalent to `-instrument=true`.

The command-line syntax for this option is as follows:

`-instrument<=true|false|`*`application_name`*`>`

For example:

```
-instrument=com.acme/stock
```

The syntax for a properties file entry for this option is as follows:

`sqlj.instrument<=true|false|`*`application_name`*`>`

For example:

```
sqlj.instrument=com.acme/stock
```

The default value for this option is:

```
false
```

Components to Instrument for DMS (-components)

When instrumentation is enabled through the `-instrument` option, use the `-components` option to specify the components to be instrumented for DMS monitoring. This is a subset of the components being translated, typically most or all of them to allow flexibility in what you can monitor during run time. At run time, instrumented components are monitored according to what is specified in the SQLJ DMS properties file.

> **See Also:** "SQLJ Run Time Commands and Properties File Settings for DMS" on page 10-23

Note that any components that are not instrumented during translation cannot be monitored during run time, regardless of what is specified in the properties file.

The `-components` option supports either of the following settings:

- *`list_of_components`*: A comma-delimited list of packages or classes to instrument
- `all` (default): Specification to instrument all components being translated

For the list of components, you can specify fully qualified class names, using the standard Java dot syntax, or you can specify package names to instrument all classes in the packages.

For example, to instrument the `Stock` and `Trading` classes:

```
% sqlj ... -components=com.acme.Stock,com.acme.Trading
```

Alternatively, here is an equivalent specification in the `sqlj.properties` file:

```
sqlj.components=com.acme.Stock,com.acme.Trading
```

The command-line syntax for this option is as follows:

```
-components=all|list_of_components
```

For example:

```
-components=com.acme.Stock,com.acme.Trading
```

The syntax for a properties file entry for this option is as follows:

```
sqlj.components=all|list_of_components
```

For example:

```
sqlj.components=com.acme.Stock,com.acme.Trading
```

The default value for this option is:

```
all
```

Options for Code Generation, Optimizations, and CHAR Comparisons

By default, the Oracle SQLJ implementation uses Oracle-specific code generation, which generates Oracle JDBC code directly, as an alternative to ISO standard SQLJ code generation. With Oracle-specific code generation, no profiles are generated, and the SQLJ run time is largely bypassed during code execution.

Because profile customization is not applicable with Oracle-specific code generation, some generally useful optimization options, formerly available only through the Oracle customizer, are now available directly through the SQLJ translator.

There is also an option for `CHAR` comparisons in a `WHERE` clause, accounting for any blank padding in the column. This option is also available as either a translator option (for Oracle-specific code generation) or an Oracle customizer option (for ISO standard code generation).

This section describes the following code generation, optimization, and `CHAR` comparison and bind options:

- `-codegen`
- `-optcols`

- -optparams
- -optparamdefaults
- -fixedchar
- -ncharconv

Code Generation (-codegen)

The Oracle SQLJ implementation can either generate Oracle-specific JDBC code directly or generate ISO standard code that calls the SQLJ run time, which in turn calls JDBC. With Oracle-specific code generation, there are no profile files and the SQLJ run time is largely bypassed during program execution.

Use the SQLJ translator -codegen option if you want to specify code generation according to the ISO standard (the default in earlier releases), as follows:

```
-codegen=iso
```

The default is Oracle-specific SQLJ code generation, but you can also explicitly specify this as follows:

```
-codegen=oracle
```

> **Note:** When codegen=iso, translator settings for -user, -password, -url, -optparams, -optparamdefaults, and -fixedchar are forwarded to the profile customizer as well. However, if you want to override these settings for customization, particularly for -user, -password, and -url, then you can do so by setting the customizer options directly.

The command-line syntax for this option is as follows:

```
-codegen=iso|oracle
```

For example:

```
-codegen=iso
```

The syntax for as properties file entry for this option is as follows:

```
sqlj.codegen=iso|oracle
```

For example:

```
sqlj.codegen=iso
```

The default value for this option is:

```
oracle
```

Column Definitions (-optcols)

Use the SQLJ translator -optcols flag to instruct the translator to determine types and sizes of iterator or result set columns. This enables registration of the columns with the Oracle JDBC driver when your application runs, saving round trips to the database, depending on the particular driver implementation. Specifically, this is effective for the JDBC Thin driver and positional iterators.

See Also: "Column Definitions" on page 10-16

Note: This translator option is equivalent to the `optcols` Oracle customizer option and was created for the default Oracle-specific code generation scenario, where there are no profiles. But it is also applicable for ISO standard code generation. In this case, setting the translator option will automatically set the customizer option as well.

You can enable or disable this flag on the SQLJ command line or in a properties file.

Enable it on the command line as follows:

```
-optcols
```

or:

```
-optcols=true
```

This flag is disabled by default, but you can also disable it explicitly. Disable it on the command line as follows:

```
-optcols=false
```

Column definitions require a database connection for examination of the columns of tables being queried, so the SQLJ translator `-user`, `-password`, and `-url` options must also be set appropriately. For example:

```
% sqlj -user=scott/tiger@jdbc:oracle:oci:@ -optcols MyApp.sqlj
```

Note:

- Because definitions are created for all columns that you select, it is advisable in your SQL operations to explicitly select the columns you will use, rather than using the `SELECT *` syntax, if you may not actually use all the columns selected. A situation where you select more than you need exposes you to a greater risk of run-time errors, if any changes were made to the table between customization and run time, especially when you have customized with column definitions. You may want to translate with the SQLJ `-warn=strict` flag set, which will warn you if additional (unwanted) columns will be selected by your query.
- Column definitions are not possible for any iterator or result set that includes one or more object or collection columns.
- An error will be generated if you enable the `-optcols` option without setting the user name, password, and URL for a database connection.
- The translator does not have to connect to the same schema or even the same database that your application will connect to at run time, but the relevant columns will have to be in the same order and of identical types and sizes to avoid run-time errors.

The command-line syntax for this option is as follows:

```
-optcols<=true|false>
```

For example:

```
-optcols
```

The syntax for a properties file entry for this option is as follows:

```
sqlj.optcols<=true|false>
```

For example:

```
sqlj.optcols
```

The default value for this option is:

```
false
```

Parameter Definitions (-optparams)

Use the SQLJ translator `-optparams` flag to enable parameter size definitions. If this flag is enabled, SQLJ will register your input and output parameters to optimize JDBC resource allocations according to sizes you specify, with the following precedence:

1. Size specified in a source code hint, if any
2. Default size, if any, specified for the corresponding data type in the `-optparamdefaults` option setting

If there is no source code hint or default data type size for a given host variable, then resource allocation is left to JDBC.

See Also: "Column Definitions" on page 10-16

Note: This translator option is equivalent to the `optparams` Oracle customizer option. It was created for the default Oracle-specific code generation scenario, where there are no profiles. But it is also applicable for ISO standard code generation. In this case, setting the translator option will automatically set the customizer option as well.

You can enable or disable the `-optparams` flag on the command line or in a SQLJ properties file.

Enable it on the command line as follows:

```
-optparams
```

or:

```
-optparams=true
```

This flag is disabled by default, but you can also disable it explicitly. Disable it on the command line as follows:

```
-optparams=false
```

> **Note:** Unlike the `-optcols` option, the `-optparams` option does not require a database connection, because you are providing the size specifications yourself.

Following is a command-line example (omitting a setting for the `-optparamdefaults` option):

```
% sqlj -optparams -optparamdefaults=defaults_string MyApp.sqlj
```

The command-line syntax for this option is as follows:

```
-optparams<=true|false>
```

For example:

```
-optparams
```

The syntax for a properties file entry for this option is as follows:

```
sqlj.optparams<=true|false>
```

For example:

```
sqlj.optparams
```

The default value for this option is:

```
false
```

Parameter Default Size (-optparamdefaults)

If you enable the `-optparams` option to set parameter sizes, then use the `-optparamdefaults` option as desired to set default sizes for specified data types. If `-optparams` is *not* enabled, then any `-optparamdefaults` setting is ignored.

If a host variable has a source code hint to specify its size, then that takes precedence over the corresponding data type default size set with this option. If there is no source code hint or corresponding data type default size for a particular host variable, then resource allocation for that variable is determined by the JDBC driver, just as it would be if `-optparams` were not enabled.

There is no requirement to use the `-optparamdefaults` option, although it is typically used whenever `-optparams` is enabled. If `-optparams` is enabled and there are no default size settings, then resources are allocated either according to source code hints, if any, or according to the JDBC driver.

> **See Also:** "Parameter Size Definitions" on page 10-17

> **Note:** This translator option is equivalent to the `optparamdefaults` Oracle customizer option. It was created for the default Oracle-specific code generation scenario, where there are no profiles. But it is also applicable for ISO standard code generation. In this case, setting the translator option will automatically set the customizer option as well.

You can set the `-optparamdefaults` flag on the command line or in a SQLJ properties file.

Set it on the command line as follows:

```
-optparamdefaults=datatype1(size1),datatype2(size2),...
```

All sizes are in bytes. Do not include any white space. Use empty parentheses for a null setting.

For example, the following will set sizes of 30 bytes for VARCHAR2 and 1000 bytes for RAW, and will specify a null size setting for CHAR. So, for any host variable corresponding to the CHAR data type, if there is no source code hint, then the JDBC driver is left to allocate the resources.

```
-optparamdefaults=VARCHAR2(30),RAW(1000),CHAR()
```

The -optparamdefaults option recognizes the following data type names:

- CHAR
- VARCHAR, VARCHAR2 (synonymous)
- LONG, LONGVARCHAR (synonymous)
- BINARY, RAW (synonymous)
- VARBINARY
- LONGVARBINARY, LONGRAW (synonymous)

The -optparamdefaults option also recognizes group names and wildcards, as follows:

- CHAR_TYPE covers CHAR, VARCHAR/VARCHAR2, and LONG/LONGVARCHAR.
- RAW_TYPE covers BINARY/RAW, VARBINARY, and LONGVARBINARY/LONGRAW.
- A percent sign (%) by itself covers all recognized data types or appended to a partial name, covers a subset of data types. For example, VAR% includes all data types that start with "VAR".

The -optparamdefaults setting is processed from left to right. When using group names or wildcards, you can override a group setting for particular data types.

The following example sets a general default size of 50 bytes, overrides that with a setting of 500 bytes for raw types, then overrides the raw type group setting with a null setting for VARBINARY (leaving that to JDBC for corresponding host variables with no source code hints):

```
-optparamdefaults=%(50),RAW_TYPE(500),VARBINARY()
```

Following is a command-line example, including the -optparams setting as well:

```
% sqlj -optparams -optparamdefaults=CHAR_TYPE(50),RAW_TYPE(500),CHAR(10)
MyApp.sqlj
```

> **Note:** If at run time the actual size exceeds the registered size of any parameter, then run-time errors will occur.

The command-line syntax for this option is as follows:

```
-optparamdefaults=defaults_string
```

For example:

```
-optparamdefaults=VAR%(50),LONG%(500),RAW_TYPE()
```

The syntax for a properties file entry for this option is as follows:

```
sqlj.optparamdefaults=defaults_string
```

For example

```
sqlj.optparamdefaults=VAR%(50),LONG%(500),RAW_TYPE()
```

The default value for this option is:

```
null
```

CHAR Comparisons with Blank Padding (-fixedchar)

Set this flag to `true` to account for blank padding in `CHAR` database columns when binding character strings for `WHERE` clause comparisons. This way, for example, "mystring" would compare positively against "mystring ".

This functionality uses the JDBC `setFixedCHAR()` method, an Oracle extension to take padding into account. The standard JDBC `setString()` method does not account for blank padding.

Following is an example of `-fixedchar` usage:

```
% sqlj -fixedchar MyProgram.sqlj AnotherProg.java ...
```

Note:

- This translator option is equivalent to the `fixedchar` Oracle customizer option. It was created for the default Oracle-specific code generation scenario, where there are no profiles. But it is also applicable for ISO standard code generation. In this case, setting the translator option will automatically set the customizer option as well.
- In `CHAR` or `VARCHAR2` columns, the Oracle SQL implementation treats the values `NULL` and "" (empty string) synonymously. Unfortunately, however, while you can insert the string "", you cannot successfully compare against it without using `IS NULL` syntax. Using `-fixedchar` functionality does not resolve this issue.

The command-line syntax for this option is as follows:

```
-fixedchar<=true|false>
```

For example:

```
-fixedchar
```

The syntax for a properties file entry for this option is as follows:

```
sqlj.fixedchar<=true|false>
```

For example:

```
sqlj.fixedchar
```

The default value for this option is:

```
false
```

NCHAR Bind (-ncharconv)

Set this option if you want to use `String` host variables to bind to `NCHAR` columns. This option specifies that the `SetFormOfUse` method should be used in the generated code for all binds to character columns. You need to translate the SQLJ file with is option as follows:

```
% sqlj -ncharconv MyApp.sqlj AnotherApp.java ...
```

This option is supported by both `codegen=oracle` and `codegen=iso`.

Note:

- When the SQLJ file is compiled with the `-ncharconv` option, the `setFormOfUse` method is used in the generated code for `codegen=oracle`. For `codegen=iso`, this option information is passed to the Oracle SQLJ run time, which internally uses `SetFormOfUse` for bind at run time.
- This translator option is *not* available in database releases prior to Oracle Database 10*g* Release 2 (10.2).

The command-line syntax for this option is as follows:

```
-ncharconv<=true|false>
```

For example:

```
-ncharconv
```

The syntax for a properties file entry for this option is as follows:

```
sqlj.ncharconv<=true|false>
```

For example:

```
sqlj.ncharconv
```

The default value for this option is:

```
false
```

Advanced Translator Options

This section documents the syntax and functionality of the advanced flags and options you can specify in running SQLJ, as well as prefixes used to pass options to the JVM, Java compiler, or SQLJ profile customizer. These options enable you to exercise any of the specialized features of the Oracle SQLJ implementation. For options that can also be specified in a properties file, that syntax is noted as well.

This section covers the following topics:

- Prefixes that Pass Option Settings to Other Executables
- Flags for Special Processing
- Semantics-Checking and Offline-Parsing Options

Prefixes that Pass Option Settings to Other Executables

The following flags mark options to be passed to the Java interpreter, Java compiler, and SQLJ profile customizer:

- `-J` (mark options for the Java interpreter)
- `-C` (mark options for the Java compiler)
- `-P` (mark options for the profile customizer, for ISO code generation only)

Options to Pass to the Java Virtual Machine (-J)

The `-J` prefix, specified on the command line, marks options to be passed to the JVM from which SQLJ was invoked. This prefix immediately precedes a JVM option, with no spaces in between. After stripping off the `-J` prefix, the `sqlj` script passes the Java option to the JVM. For example:

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

After stripping the `-J` prefix, the `sqlj` script passes the `-Duser.language=ja` argument as is to the JVM. In the Sun Microsystems JDK, the

`-Duser.language=ja` flag sets the `user.language` system property to the value `ja` (Japanese), but specific flags are dependent on the actual Java executable you are using and are not interpreted or acted upon by the `sqlj` script in any way.

You cannot pass options to the JVM from a properties file, because properties files are read after the JVM is invoked.

Note:

- While it is not possible to use a properties file to pass options directly to the JVM in which the SQLJ translator runs, it *is* possible to use the `SQLJ_OPTIONS` environment variable for this purpose. It is also possible, if applicable, to use a properties file to pass options to the JVM in which the Java compiler runs.
- The JVM `file.encoding` setting does not apply to Java properties files. Properties files always use the encoding `8859_1`. This is a feature of Java in general, not SQLJ in particular. However, you can use Unicode escape sequences in a properties file. You can use the `native2ascii` utility to determine escape sequences.

The command-line syntax for this option is as follows:

```
-J-Java_option
```

For example:

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

Options to Pass to the Java Compiler (-C)

The `-C` prefix marks options to pass to the Java compiler invoked from the `sqlj` script. This prefix immediately precedes a Java compiler option, with no spaces in between. After stripping off the `-C` prefix, the `sqlj` script passes the compiler option to the Java compiler. For example:

```
-C-nowarn
```

After stripping the -C prefix, the sqlj script passes the -nowarn argument as is to the compiler.

Generally, compiler options are passed without change, but when you use an equal sign (=) to set a compiler option that takes a value, such as for -bootclasspath, -extdirs, or -target, the equal sign is stripped out when the option is passed to the compiler. Consider the following example:

```
% sqlj -C-bootclasspath=/usr/local/packages/jdk1.5.1/jre/lib/rt.jar myfile.sqlj
```

Also note that if the Java compiler runs in its own JVM, then you can pass options to that JVM through the compiler. Accomplish this by prefixing the JVM option with -C-J with no spaces between this prefix combination and the option. For example:

```
-C-J-Duser.language=de
```

Observe the following restrictions in using the -C prefix:

- Do not use -C-encoding to specify encoding of .java files processed by the Java compiler. Instead, use the SQLJ -encoding option, which specifies encoding of .sqlj files processed by SQLJ and .java files generated by SQLJ, and is also passed to the compiler. This ensures that .sqlj files and .java files receive the same encoding.
- Do not use -C-d to specify an output directory for .class files. Instead, use the SQLJ -d option, which specifies the output directory for generated profile files (.ser), and is also passed to the Java compiler. This will ensure that .class files and .ser files are in the same directory.

Note:

- If you specify compiler options but disable compilation (-compile=false), then the compiler options are silently ignored.
- The compiler help option (-C-help, presuming your compiler supports -help) can be specified only on the command line or in the SQLJ_OPTIONS variable, not in a properties file. As with the SQLJ -help option, no translation will be done. This is true even if you also specify files to process. SQLJ assumes that you want help or you want translation, but not both.

The command-line syntax for this option is as follows:

```
-C-Java_compiler_option
```

For example:

```
-C-nowarn
```

The syntax for a properties file entry for this option is as follows:

```
compile.Java_compiler_option
```

For example:

```
compile.nowarn
```

Options to Pass to the Profile Customizer (-P)

During the customization phase (relevant only for ISO standard code generation), the `sqlj` script invokes a front-end customizer harness, which coordinates the customization and runs your particular customizer. The `-P` prefix marks options for customization, as follows:

- Use `-P` by itself to pass generic options to the customizer harness that apply regardless of the customizer.
- Use `-P-C` to pass vendor-specific options to the particular customizer you are using.

The `-P` and `-P-C` prefixes immediately precede a customizer option, with no spaces in between. After stripping off the prefix, the `sqlj` script passes the customizer option as is to the profile customizer.

One use of the `-P` prefix is to override the default customizer determined by the SQLJ `-default-customizer` option, as follows:

```
-P-customizer=your_customizer_class
```

Example of a generic customizer option:

```
-P-backup
```

The `-backup` flag is a generic customizer option to backup the previous customization before generating a new one.

Following is an example of a vendor-specific customizer option (in this case, Oracle-specific):

```
-P-Csummary
```

The `summary` flag is an Oracle customizer option that prints a summary of the customizations performed.

Note:

- There is no hyphen between `-P-C` and a vendor-specific customizer option. With other prefixes and prefix combinations, there *is* a hyphen between the prefix and the option.
- The customizer help option (`-P-help`) can be specified only on the command line or in the `SQLJ_OPTIONS` variable, not in a properties file. As with the SQLJ `-help` option, no translation will be done. This is true even if you also specify files to process. SQLJ assumes that you want help or you want translation, but not both.
- For ISO code generation, if you specify customization options but turn off customization for `.sqlj` files (and have no `.ser` files on the command line), then the customization options are silently ignored.
- The `-P` prefix is not applicable for the default Oracle-specific code generation, where no profiles are produced and so no customization is performed.

The command-line syntax for this option is as follows:

```
-P-<C>profile_customizer_option
```

For example:

```
-P-driver=oracle.jdbc.OracleDriver
-P-Csummary
```

The syntax for a properties file entry for this option is as follows:

```
profile.<C>profile_customizer_option
```

For example:

```
profile.driver=oracle.jdbc.OracleDriver
profile.Csummary
```

Flags for Special Processing

The `.sqlj` files are typically processed by the SQLJ translator, the Java compiler, and, for ISO code generation, the SQLJ profile customizer. The following flags limit this processing, directing the SQLJ startup script to skip the indicated process:

- `-compile`
- `-profile`

The `-ser2class` flag, for ISO code generation, directs SQLJ to convert profiles from serialized resource (`.ser`) files to class files after customization.

The `-checksource` flag instructs SQLJ type resolution, in certain circumstances, to examine source files as well as class files or files specified on the SQLJ command line.

The `-bind-by-identifier` flag specifies that SQLJ treat multiple appearances of the same host variable in a given SQLJ statement as a single bind occurrence.

Compilation Flag (-compile)

The `-compile` flag enables or disables processing of `.java` files by the compiler. This applies both to generated `.java` files and to `.java` files specified on the command line. This flag is useful, for example, if you want to compile `.java` files later using a compiler other than `javac`. The flag is `true` by default. Setting it to `false` disables compilation.

When you process a `.sqlj` file with `-compile=false`, you are responsible for compiling and customizing it later as necessary.

Setting `-compile=false` also implicitly sets `-profile=false`. In other words, whenever `-compile` is `false`, both compilation and customization are skipped. If you set `-compile=false` and `-profile=true`, then your `-profile` setting is ignored.

Note: There are situations where it is sensible for `-compile` to be set to `false` even when `.java` files must be accessed for type resolution. You may do this, for example, if you are translating a `.sqlj` file and want to specify one or more `.java` files on the command line for type resolution during translation, but want to compile all your `.java` files later using a particular compiler.

Note, however, that the `-checksource` option can simplify the type resolution process by eliminating the need to enter `.java` files for resolution on the SQLJ command line.

The command-line syntax for this option is as follows:

```
-compile<=true|false>
```

For example:

```
-compile=false
```

The syntax for a properties file entry for this option is as follows:

```
sqlj.compile<=true|false>
```

For example:

```
sqlj.compile=false
```

The default value for this option is:

```
true
```

Profile Customization Flag (-profile)

For ISO code generation, the `-profile` flag enables or disables processing of generated profile files (`.ser`) by the SQLJ profile customizer. However, this applies only to `.ser` files generated by the SQLJ translator from `.sqlj` files that you specify on the current command line. It does not apply to previously generated `.ser` files (or to `.jar` files) that you specify on the command line. The flag is `true` by default. Setting it to `false` disables customization.

This option acts differently than the `-compile` option for files specified on the command line. Any `.ser` and `.jar` files specified on the command line are still customized if `-profile=false`. However, `.java` files specified on the command line are *not* compiled if `-compile=false`. The reason for this is that you may want other operations, such as line mapping, to be performed on a `.java` file. There are, however, no other operations that can be performed on a `.ser` or `.jar` file specified on the command line.

When you process a `.sqlj` file with `-profile=false`, you are responsible for customizing it later, as necessary.

Note:

- Set this option to `false` if you do not want your application to require the Oracle SQLJ run time and an Oracle JDBC driver when it runs. Or accomplish this by specifying a nondefault customizer, using the `-default-customizer` option. If no customization is performed, then the generic SQLJ run time will be used when your application runs.
- Setting `-compile=false` also implicitly sets `-profile=false`. In other words, whenever `-compile` is `false`, both compilation and customization are skipped. If you set `-compile=false` and `-profile=true`, then your `-profile` setting is ignored.
- This option is not applicable for the default Oracle-specific code generation, where no profiles are produced and so no customization is performed.

The command-line syntax for this option is as follows:

```
-profile<=true|false>
```

For example:

```
-profile=false
```

The syntax for a properties file entry for this option is as follows:

```
sqlj.profile<=true|false>
```

For example:

```
sqlj.profile=false
```

The default value for this option is:

```
true
```

Conversion of .ser File to .class File (-ser2class)

With ISO standard SQLJ code generation, the `-ser2class` flag instructs SQLJ to convert generated `.ser` files to `.class` files. This is necessary if you are using SQLJ to create an applet that will be run from a browser that does not support resource file names with the `.ser` suffix.

This also simplifies the naming of schema objects for your profiles in situations where you are translating a SQLJ program on a client and then loading classes and resource files into the server. Loaded class schema objects have a simpler naming convention than loaded resource schema objects.

> **See Also:** "Naming of Loaded Class and Resource Schema Objects" on page 11-7

The conversion is performed after profile customization so that it includes your customizations. The base names of converted files are identical to those of the original files. The only difference in the file name is `.ser` being replaced by `.class`. For example, consider the following:

```
Foo_SJProfile0.ser
```

This is converted to:

```
Foo_SJProfile0.class
```

Note:

- The original `.ser` file is not saved.
- Once a profile has been converted to a `.class` file, it cannot be further customized. You would have to delete the `.class` file and rerun SQLJ to recreate the profile.
- Where encoding is necessary, the `-ser2class` option always uses `8859_1` encoding, ignoring the SQLJ `-encoding` setting.
- If you use the default Oracle-specific code generation, then no profiles are produced and the `-ser2class` option does not apply.

The command-line syntax for this option is as follows:

```
-ser2class<=true|false>
```

For example:

```
-ser2class
```

The syntax for a properties file entry for this option is as follows:

```
sqlj.ser2class<=true|false>
```

For example

```
sqlj.ser2class
```

The default value for this option is:

```
false
```

Source Check for Type Resolution (-checksource)

It may not be sufficient for the SQLJ type resolution process to examine only class files in the classpath and class or source files specified on the SQLJ command line. The `-checksource` flag instructs SQLJ to also examine source files in the classpath under the following circumstances:

- If a class file cannot be found for a required class, but a source file can be found
- If a source file has a more recent modification date than its corresponding class file

Note: This applies only to Java types that appear in `#sql` statements, not elsewhere in your Java code. Therefore, you should always explicitly provide the names of any required `.sqlj` files on the SQLJ command line.

The command-line syntax for this option is as follows:

```
-checksource<=true|false>
```

For example:

```
-checksource=false
```

The syntax for a properties file entry for this option is as follows:

```
sqlj.checksource=<=true|false>
```

For example:

```
sqlj.checksource=false
```

The default value for this option is:

```
true
```

Binding Host Expressions by Identifier (-bind-by-identifier)

In keeping with the SQLJ standard, the Oracle implementation by default creates a unique name for each host-variable bind reference in a statement, even if there are multiple occurrences of the same host variable. The SQLJ standard is based on JDBC, and JDBC does not make provisions for binding the same variable into different positions. Instead, each bind position (identified by `?`) is bound to an individual value.

In some situations this causes errors, such as in the following example:

```
#sql emps = { SELECT substr(ename, 1, :bind_var), sum(sal) FROM emp
              GROUP BY substr(ename, 1, :bind_var) };
```

Because separate bind reference names are created for the two occurrences of `bind_var`, this results in a SQL exception at run time. When the differing bind names are detected, the SQL engine concludes that the `GROUP BY` clause is not part of the `SELECT`-list.

To avoid such problems, Oracle extends standard functionality with the `-bind-by-identifier` flag. A setting of `true` results in all bind occurrences of the same identifier in a given SQLJ statement or PL/SQL block being treated as a single bind occurrence. A SQLJ statement with four bind operations, `:x`, `:x`, `:y`, `:x`, would be bound as `:1`, `:1`, `:2`, `:1` instead of `:1`, `:2`, `:3`, `:4`.

In the preceding example, both bindings would be as `substr(ename, 1, :1)` instead of as `substr(ename, 1, :1)` and `substr(ename, 1, :2)`.

> **Note:** The `-bind-by-identifier` flag applies only to host expressions that are simple host variables.

The command-line syntax for this option is as follows:

```
-bind-by-identifier<=true|false>
```

For example:

```
-bind-by-identifier
```

The syntax for a properties file entry for this option is as follows:

```
sqlj.bind-by-identifier=<=true|false>
```

For example:

```
sqlj.bind-by-identifier
```

The default value for this option is:

```
false
```

Semantics-Checking and Offline-Parsing Options

The following options specify characteristics of online and offline semantics-checking and offline parsing:

- `-offline`
- `-online`
- `-cache`
- `-parse`

Description of these options is preceded by two introductory discussions:

- A discussion of `OracleChecker` (the default front-end class for semantics-checking) and an introduction to the Oracle semantics-checkers
- A comparison of online semantics-checking versus offline parsing

> **Note:** As described in "Online Semantics-Checking and User Name (-user)" on page 8-25, online semantics-checking is enabled by setting the translator `-user` option. However, the setting of the `-parse` option, which is used to enable or disable offline parsing, can override this. See "Offline Parser (-parse)" on page 8-61.

Semantics-Checkers and the OracleChecker Front End (default checker)

Oracle supplies Oracle-specific offline checkers, a generic offline checker, Oracle-specific online checkers, and a generic online checker. The generic checkers assume you use only standard SQL92 and standard JDBC features. Oracle recommends that you use the Oracle-specific checkers when using Oracle Database.

The default checker, which is satisfactory in the great majority of circumstances, is `oracle.sqlj.checker.OracleChecker` for both online and offline checking. This class acts as a front end and runs the appropriate semantics-checker, depending on your environment and whether you choose offline or online checking.

For Oracle, there is the Oracle8 checker for Oracle 10*g*, Oracle9*i*, and Oracle8*i* types, for both online and offline checking (as used in the corresponding JDBC implementations).

Online Checking with Oracle Database and JDBC Driver

If you are using Oracle Database and Oracle JDBC driver with online checking, then `OracleChecker` will choose a checker based on the lower of your database version and JDBC driver version. Table 8–4 summarizes the choices for the possible combinations of database version and driver version, and also notes any other Oracle checkers that would be legal.

Table 8–4 Oracle Online Semantics-Checkers Chosen by OracleChecker

Database Release	JDBC Release	Chosen Online Checker	Other Legal Online Checkers
Oracle10*g*, 9*i*, or 8*i*	Oracle11*g*	Oracle8JdbcChecker	Oracle8To7JdbcChecker

Offline Checking with Oracle JDBC Driver

If you are using an Oracle JDBC driver with offline checking, then `OracleChecker` chooses a checker based on your JDBC driver version. Table 8–5 summarizes the possible choices.

Table 8–5 Oracle Offline Semantics-Checkers Chosen by OracleChecker

JDBC Release	Chosen Offline Checker	Other Legal Offline Checkers
Oracle11*g*	Oracle8OfflineChecker	No

Online Semantics-Checking Versus Offline Parsing

The Oracle SQLJ implementation supports a feature known as offline parsing that offers a limited alternative to online semantics-checking. Offline parsing does not use a database connection, so cannot perform verification of operations against the database schema, but does offer syntax-checking of all SQL and PL/SQL statements. (Prior to Oracle9*i*, syntax-checking was not possible without a database connection.)

Table 8–6 provides a comparative summary of what offline parsing and online semantics-checking offer.

Table 8–6 Feature Comparison: Offline Parsing Versus Online Semantics-Checking

Feature	By Offline Parsing?	By Online Checking?
Verify data manipulation language (DML), SELECT, and PL/SQL syntax.	Yes	Yes
Verify data definition language (DDL) syntax.	Yes	No
Verify DML, SELECT, and PL/SQL semantics (comparison against database schema).	No	Yes
Verify DDL semantics (comparison against database schema).	No	No

Online checking offers the primary advantage of verifying SQL and PL/SQL operations against the database schema. This includes verifying that column types match SQL operations and verifying the existence of called stored procedures. It requires a database connection during translation, however, which may be problematic in some circumstances. It also performs no verification of DDL operations.

Offline parsing offers the advantage of SQL syntax-checking without a database connection during translation, and also includes DDL operations in its syntax verifications.

Note that neither mode performs DDL semantics-checking against the database schema.

Note:

- If both offline parsing and online checking are enabled, some types of errors will be reported twice.
- Problems detected by either the offline parser or the online checker are reported at a warning or advisory level, not a fatal level.
- Do not confuse offline parsing with offline semantics-checking. Offline checking consists of basic semantics-checking steps that always occur, regardless of whether online checking is enabled and regardless of whether offline parsing is enabled: analysis of the types of Java expressions in your SQLJ executable statements, and categorization of embedded SQL operations according to keyword, such as `SELECT`.
- Compatibility of data corresponding to weakly typed host expressions is never checked.
- Mode compatibility of expressions in PL/SQL anonymous blocks is never checked.

Offline Semantics-Checker (-offline)

The `-offline` option specifies a Java class that implements the semantics-checking component of SQLJ for offline checking. With offline checking, there is no database connection. Only SQL syntax and usage of Java types is checked. Note that offline checking is neither enabled nor disabled by the `-offline` option. Offline checking runs only when online checking does not, either because online checking is not enabled or because the database connection cannot be established.

You can specify different offline checkers for different connection contexts, with a limit of one checker per context. Do *not* list multiple offline checkers for one connection context. The default `OracleChecker`, a front-end class, will serve your needs unless you want to specify a particular checker that would not be chosen by `OracleChecker`.

The following example shows how to select the Oracle8 offline checker for a particular connection context (`CtxClass`):

```
-offline@CtxClass=oracle.sqlj.checker.Oracle8OfflineChecker
```

This results in SQLJ using `oracle.sqlj.checker.Oracle8OfflineChecker` for offline checking of any of your SQLJ executable statements that specify a connection object that is a `CtxClass` instance.

The `CtxClass` connection context class must be declared in your source code or previously compiled into a `.class` file. (See "Connection Contexts" on page 7-1 for more information.)

Use the `-offline` option separately for each connection context offline checker you want to specify; these settings have no influence on each other. For example:

```
-offline@CtxClass2=oracle.sqlj.checker.Oracle8OfflineChecker
-offline@CtxClass3=sqlj.semantics.OfflineChecker
```

To specify the offline checker for the default connection context and any other connection contexts for which you do not specify an offline checker:

```
-offline=oracle.sqlj.checker.Oracle8OfflineChecker
```

Any connection context without an offline checker setting uses the offline checker setting of the default connection context, presuming an offline checker has been set for the default context.

The command-line syntax for this option is as follow:

```
-offline<@conn_context_class>=checker_class
```

For example:

```
-offline=oracle.sqlj.checker.Oracle8OfflineChecker
-offline@CtxClass=oracle.sqlj.checker.Oracle8OfflineChecker
```

The syntax for a properties file entry for this option is as follows:

```
sqlj.offline<@conn_context_class>=checker_class
```

For example:

```
sqlj.offline=oracle.sqlj.checker.Oracle8OfflineChecker
sqlj.offline@CtxClass=oracle.sqlj.checker.Oracle8OfflineChecker
```

The default value for this option is:

```
oracle.sqlj.checker.OracleChecker
```

Online Semantics-Checker (-online)

The `-online` option specifies a Java class or list of classes that implement the online semantics-checking component of SQLJ. This involves connecting to a database. Note that online checking is not enabled by the `-online` option. You must enable it through the `-user` option. The `-password`, `-url`, and `-driver` options must be set appropriately as well.

> **Note:** Some settings of the SQLJ `-parse` option will disable online semantics-checking, overriding the effect of the `-user` option.

You can specify different online checkers for different connection contexts, and you can list multiple checkers (separated by commas) for any given context. In cases where multiple checkers are listed for a single context, SQLJ uses the first checker (reading from left to right in the list) that accepts the database connection established for online checking. At analysis time, a connection is passed to each online checker and the checker decides whether it recognizes the database.

The default `OracleChecker`, a front-end class, will serve your needs unless you want to specify a particular checker that would not be chosen by `OracleChecker`.

The following example shows how to select the Oracle8 online checker for the `DefaultContext` class and any other connection context classes without a specified setting:

```
-online=oracle.sqlj.checker.Oracle8JdbcChecker
```

To specify a list of drivers and allow the proper class to be selected depending on what kind of database is being accessed:

```
-online=oracle.sqlj.checker.Oracle8JdbcChecker,sqlj.semantics.JdbcChecker
```

With this specification, if connection is made to Oracle Database, then SQLJ uses the `oracle.sqlj.checker.Oracle8JdbcChecker` semantics-checker. If connection is made to any other kind of database, then SQLJ uses the generic `sqlj.semantics.JdbcChecker` semantics-checker. This is similar functionally to what the default `OracleChecker`.

To specify the online checker for a particular connection context (`CtxClass`):

```
-online@CtxClass=oracle.sqlj.checker.Oracle8JdbcChecker
```

This results in the use of `oracle.sqlj.checker.Oracle8JdbcChecker` for online checking of any of your SQLJ executable statements that specify a connection object that is an instance of `CtxClass`, presuming you enable online checking for `CtxClass`.

The `CtxClass` connection context class must be declared in your source code or previously compiled into a `.class` file.

Use the `-online` option separately for each connection context online checker you want to specify. These settings have no influence on each other:

```
-online@CtxClass2=oracle.sqlj.checker.Oracle8JdbcChecker
-online@CtxClass3=sqlj.semantics.JdbcChecker
```

Any connection context without an online checker setting uses the online checker setting of the default connection context.

The command-line syntax for this option is as follows:

```
-online<@conn_context_class>=checker_class(list)
```

For example:

```
-online=oracle.sqlj.checker.Oracle8JdbcChecker
-online=oracle.sqlj.checker.Oracle8JdbcChecker,sqlj.semantics.JdbcChecker
-online@CtxClass=oracle.sqlj.checker.Oracle8JdbcChecker
```

The syntax for a properties file entry for this option is as follows:

```
sqlj.online<@conn_context_class>=checker_class(list)
```

For example:

```
sqlj.online=oracle.sqlj.checker.Oracle8JdbcChecker
sqlj.online=oracle.sqlj.checker.Oracle8JdbcChecker,sqlj.semantics.JdbcChecker
sqlj.online@CtxClass=oracle.sqlj.checker.Oracle8JdbcChecker
```

The default value for this option is:

```
oracle.sqlj.checker.OracleChecker
```

Caching of Online Semantics-Checker Results (-cache)

Use the `-cache` option to enable caching of the results generated by the online checker. This avoids additional database connections during subsequent SQLJ translation runs. The analysis results are cached in a file, `SQLChecker.cache`, that is placed in your current directory. The cache contains serialized representations of all SQL statements successfully translated (translated without error or warning messages), including all statement parameters, return types, translator settings, and modes.

The cache is cumulative and continues to grow through successive invocations of the SQLJ translator. Delete the `SQLChecker.cache` file to empty the cache.

The command-line syntax for this option is as follows:

```
-cache<=true|false>
```

For example:

```
-cache
```

The syntax for a properties file entry for this option is as follows:

```
sqlj.cache<=true|false>
```

For example:

```
sqlj.cache
```

The default value for this option is:

```
false
```

Offline Parser (-parse)

Use the `-parse` option to enable offline parsing. This feature is a complement to online semantics-checking, offering SQL and PL/SQL syntax-checking (but not verification against the schema) without a database connection during translation. Offline parsing also checks syntax for DDL statements, which online checking does not.

Also be aware that the setting of the `-parse` option can override the enabling of online checking by the `-user` option. Possible `-parse` settings are as follows:

- `both` (default): Enable the offline parser and allow online checking. In this case, online checking is determined by the `-user` option.
- `online-only`: Disable the offline parser and allow online checking. Again, online checking is determined by the `-user` option.
- `offline-only`: Enable the offline parser and disallow online checking. This overrides any `-user` option setting that would otherwise enable online checking.
- `none`: Disable the offline parser and disallow online checking, Again, this overrides any `-user` option setting that would otherwise enable online checking.
- *`parserclassname`*: Specify the name of a Java class that implements an alternative SQL parser. The class must implement the `sqlj.framework.checker.SimpleChecker` interface. This setting enables the specified parser, and only that parser is used for SQL-checking. The standard offline parser and online checking are both disabled.

The `offline-only` and `none` settings are offered for completeness, but are not typical modes of operation. It is best to let the `-user` option determine online checking. It is also not typical to specify your own parser.

> **Note:** In modes where both offline parsing and online checking are enabled, there may be duplicate reporting of some problems.

The command-line syntax for this option is as follows:

```
-parse=both|online-only|offline-only|none|parserclassname
```

For example:

```
-parse=online-only
```

The syntax for a properties file entry for this option is as follows:

```
sqlj.parse=both|online-only|offline-only|none|parserclassname
```

For example:

```
sqlj.parse=online-only
```

The default value for this option is:

```
both
```

Translator Support and Options for Alternative Environments

By default, the Oracle Database 11*g* SQLJ implementation is configured to run under the Sun Microsystems JDK 1.5.x and to use the Sun Microsystems compiler `javac`. These are not requirements, however. You can configure SQLJ to work with alternative JVMs or compilers. To do so, you must supply SQLJ with the following information:

- The name of the JVM to use (`-vm` option)
- The name of the Java compiler to use (`-compiler-executable` option)
- Any settings the compiler requires

A set of SQLJ options enables you to provide this information. SQLJ also defaults to the Oracle profile customizer, but can work with alternative customizers as well.

> **Note:** Be aware of the limitations of any operating system and environment you use. In particular, the complete, expanded SQLJ command line must not exceed the maximum command-line size. Consult your operating system documentation.

This section covers the following topics:

- Java and Compiler Options
- Customization Options

Java and Compiler Options

The following options relate to the operation of the JVM and Java compiler:

- `-vm` (to specify the JVM, on the command line only)
- `-compiler-executable` (to specify the Java compiler)
- `-compiler-encoding-flag`
- `-compiler-output-file`
- `-compiler-pipe-output-flag`

Some compilers, such as the standard `javac`, require a Java source file name to match the name of the public class, if any, defined there. Therefore, by default the SQLJ

translator verifies that this is true. However, you can use the -checkfilename option to instruct SQLJ not to verify this.

For some JVM and compiler configurations, there may be problems with the way SQLJ usually invokes the compiler. You can use the -passes option to alleviate this by breaking SQLJ processing into a two-pass process. You can also pass options directly to the particular JVM or compiler you use, through the -J and -C prefixes.

> **Note:** The -vm option, -passes option, and -J prefix cannot be used in a properties file. You can set them on the command line or, more conveniently, in the SQLJ_OPTIONS environment variable.

Name of the Java Virtual Machine (-vm)

Use the -vm option if you want to specify a particular JVM for SQLJ to use. Otherwise, SQLJ uses the standard java from the Sun Microsystems JDK. You cannot set this option in a properties file, because properties files are read after the JVM is invoked.

If you do not specify a directory path along with the name of the JVM executable file, then SQLJ looks for the executable according to the setting of your operating system PATH variable.

> **Note:** Special functionality of this option, -vm=echo, is supported. This is equivalent to the -n option, instructing the sqlj script to construct the full command line that would be passed to the SQLJ translator, and echo it to the user without having the translator execute it.

The command-line syntax for this option is as follows:

```
-vm=JVM_path+name
```

For example:

```
-vm=/myjavadir/myjavavm
```

The default value is:

```
java
```

Name of the Java Compiler (-compiler-executable)

Use the -compiler-executable option if you want to specify a particular Java compiler for SQLJ to use. Otherwise SQLJ, uses the standard javac from the Sun Microsystems JDK.

If you do not specify a directory path along with the name of the compiler executable file, then SQLJ looks for the executable according to the setting of your operating system PATH variable.

The following is required of any Java compiler that you use:

- It can write error and status information to the standard output device (for example, STDOUT on a UNIX system) or to a file, as directed by the -compiler-output-file option.
- It will understand the SQLJ -d option, which determines the root directory for class files.

- It must return a nonzero exit code to the operating system whenever a compilation error occurs.
- The line information that it provides in any errors or messages must be in one of the following formats (items in <> brackets being optional):
  - Sun Microsystems `javac` format

    filename`.java:`*line*`<.`*column*`><-`*line*`<.`*column*`>>`

    Example: `myfile.java:15: Illegal character: '\u01234'`
  - Microsoft `jvc` format

    filename`.java(`*line*`,`*column*`)`

    Example: `myfile.java(15,7) Illegal character: '\u01234'`

As always, SQLJ processes compiler line information so that it refers to line numbers in the original `.sqlj` file, not in the produced `.java` file.

Note: For a compiler that does not support an `-encoding` option, disable the `-compiler-encoding-flag`.

The command-line syntax for this option is as follows:

```
-compiler-executable=Java_compiler_path+name
```

For example:

```
-compiler-executable=/myjavadir/myjavac
```

The syntax for a properties file entry for this option is as follows:

```
sqlj.compiler-executable=Java_compiler_path+name
```

For example:

```
sqlj.compiler-executable=myjavac
```

The default value is:

```
javac
```

Compiler Encoding Support (-compiler-encoding-flag)

When you use the `-encoding` option to specify an encoding character set for SQLJ to use, SQLJ passes this to the Java compiler for the compiler to use as well. Set the `-compiler-encoding-flag` to `false` if you do not want SQLJ to pass the character encoding to the compiler. For example, if you are using a compiler other than `javac` and it does not support an `-encoding` option by that name.

The command-line syntax for this option is as follows:

```
-compiler-encoding-flag<=true|false>
```

For example:

```
-compiler-encoding-flag=false
```

The syntax for a properties file entry for this option is as follows:

```
sqlj.compiler-encoding-flag<=true|false>
```

For example:

```
sqlj.compiler-encoding-flag=false
```

The default value is:

```
true
```

Compiler Output File (-compiler-output-file)

If you have instructed the Java compiler to write its results to a file, then use the `-compiler-output-file` option to make SQLJ aware of the file name. Otherwise, SQLJ assumes that the compiler writes to the standard output device, such as `STDOUT` on a UNIX system. As appropriate, specify an absolute path or a relative path from the current directory.

> **Note:** You cannot use this option if you enable `-passes`, which requires output to `STDOUT`.

The command-line syntax for this option is as follows:

```
-compiler-output-file=output_file_path+name
```

For example:

```
-compiler-output-file=/myjavadir/mycmploutput
```

The syntax for a properties file entry for this option is as follows:

```
sqlj.compiler-output-file=output_file_path+name
```

For example:

```
sqlj.compiler-output-file=/myjavadir/mycmploutput
```

This option does not have a default value.

Source File Name Check (-checkfilename)

It is generally advisable for the source file name to always match the name of the public class defined or, if there is no public class, the name of the first class defined. For example, public class `MyPublicClass` should be defined in a `MyPublicClass.sqlj` source file.

The `-checkfilename` flag instructs SQLJ whether to verify that the SQLJ source file name matches the name of the public class, if any, defined there. Some compilers, such as the standard `javac`, require this to be the case, while others do not.

To maximize portability of your code, this flag should be enabled, which it is by default.

> **Note:** If you are translating in the server, where there is no equivalent naming requirement, there is no `-checkfilename` option and the translator executes no such check.

The command-line syntax for this option is as follows:

```
-checkfilename<=true|false>
```

For example:

```
-checkfilename=false
```

The syntax for a properties file entry for this option is as follows:

```
sqlj.checkfilename<=true|false>
```

For example:

```
sqlj.checkfilename=false
```

The default value is:

```
true
```

SQLJ Two-Pass Execution (-passes)

By default, the following occurs when you invoke the `sqlj` script:

1. The `sqlj` script invokes your JVM, which runs the SQLJ translator.
2. The translator completes the semantics-checking and translation of your `.sqlj` files, generating translated `.java` files.
3. The translator invokes your Java compiler, which compiles the generated `.java` files.
4. The translator processes the compiler output.
5. If any profile files were generated, then the translator invokes a profile customizer to customize them.

For some JVM and compiler configurations, however, the compiler invocation in Step 3 might not return, in which case your translation will suspend.

If you encounter this situation, the solution is to instruct SQLJ to run in two passes, with the compilation step in between. To accomplish this, you must enable the two-pass execution flag as follows:

```
-passes
```

The `-passes` option must be specified on the command line or, equivalently, in the `SQLJ_OPTIONS` environment variable. It cannot be specified in a properties file.

Note:

- If you enable `-passes`, then compiler output must go to `STDOUT`. Therefore, leave `-compiler-pipe-output-flag` enabled, which is its default. In addition, you cannot use the `-compiler-output-file` option, which would result in writing to a file instead of to `STDOUT`.
- Like other command-line-only flags (`-help`, `-version`, `-n`), the `-passes` flag does not support `=true` syntax.

With `-passes` enabled, the following occurs when you invoke the `sqlj` script:

1. The `sqlj` script invokes your JVM, which runs the SQLJ translator for its first pass.

2. The translator completes the semantics-checking and translation of your `.sqlj` files, generating translated `.java` files.
3. The JVM is terminated.
4. The `sqlj` script invokes the Java compiler, which compiles the generated `.java` files.
5. The `sqlj` script invokes your JVM again, which runs the SQLJ translator for its second pass.
6. The translator processes compiler output.
7. If any profile files were generated, the JVM runs your profile customizer to customize them.

With this sequence, you circumvent any problems the JVM might have in invoking the Java compiler.

The command-line syntax for this option is as follows:

```
-passes
```

For example:

```
-passes
```

The default value is:

```
off
```

Customization Options

The following options relate to the customization of your SQLJ profiles, if applicable:

- `-default-customizer`
- Options passed directly to the customizer

> **Note:** If you use the default Oracle-specific code generation, then SQLJ generates no profiles and, therefore, performs no customization. In that case, these options do not apply.

Default Profile Customizer (-default-customizer)

Use the `-default-customizer` option to instruct SQLJ to use a profile customizer other than the default, which is:

```
oracle.sqlj.runtime.util.OraCustomizer
```

In particular, use this option if you are not using Oracle Database. This option takes a fully qualified Java class name as its argument.

> **Note:** You can override this option with the `-P-customizer` option in your SQLJ command line or properties file.

The command-line syntax for this option is as follows:

```
-default-customizer=customizer_classname
```

For example:

```
-default-customizer=sqlj.myutil.MyCustomizer
```

The syntax for a properties file entry for this option is as follows:

```
sqlj.default-customizer=customizer_classname
```

For example:

```
sqlj.default-customizer=sqlj.myutil.MyCustomizer
```

The default value for this option is:

```
oracle.sqlj.runtime.util.OraCustomizer
```

Note: When you use Oracle Database and ISO code generation, Oracle recommends that you use the default `OraCustomizer` for your profile customization.

Options Passed Directly to the Customizer

As with the JVM and compiler, you can pass options directly to the profile customizer harness using a prefix, in this case `-P`.

> **See Also:** "Options to Pass to the Profile Customizer (-P)" on page 8-50

Details about these options, both general customization options and Oracle-specific customizer options, are covered in "Customization Options and Choosing a Customizer" on page A-6.

9

Translator and Run Time Functionality

This chapter discusses internal operations and functionality of the Oracle SQLJ translator and run time.

The following topics are covered:

Internal Translator Operations

The following topics summarize the operations executed by the SQLJ translator during a translation:

Java and SQLJ Code-Parsing and Syntax-Checking

In this first phase of SQLJ translation, a SQLJ parser and a Java parser are used to process all the source code and check syntax. As the SQLJ translator parses the `.sqlj` file, it invokes a Java parser to check the syntax of Java statements and a SQLJ parser to check the syntax of SQLJ constructs (anything preceded by `#sql`). The SQLJ parser also invokes the Java parser to check the syntax of Java host variables and expressions within SQLJ executable statements.

The SQLJ parser checks the grammar of SQLJ constructs according to the SQLJ language specification. However, it does *not* check the grammar of the embedded SQL operations. SQL syntax is not checked until the semantics-checking or offline parsing step.

This syntax-check will look for errors like missing semi-colons, mismatched curly braces, and obvious type mismatches, such as multiplying a number by a string. If the parsers find any syntax errors or type mismatches during this phase, then the translation is aborted and the errors are reported to the user.

SQL Semantics-Checking and Offline Parsing

Once the SQLJ and Java application source code is verified as syntactically correct, the translator enters into the semantics-checking phase and invokes a SQL semantics-checker or a SQL offline parser or both, according to SQLJ option settings.

Setting the `-user` option enables online checking, and setting the `-password` and `-url` options specifies the database connection, if the password and URL were not specified in the `-user` option. The `-offline` or `-online` option specifies which checker to use. The default, typically sufficient, is a checker front end called `OracleChecker` that chooses the most appropriate checker, according to whether you have enabled online checking and which Java Database Connectivity (JDBC) driver you are using.

The `-parse` option, `true` by default, is for enabling the offline parser, which offers a way to verify SQL and PL/SQL syntax (but not data types against database columns) without necessitating a database connection during translation. Note that some settings of the `-parse` option will override the `-user` option and disable online checking.

> **See Also:** "Connection Options" on page 8-25 and "Semantics-Checking and Offline-Parsing Options" on page 8-56

Note: For ISO code generation, semantics-checking can also be performed on a profile that was produced during a previous execution of the SQLJ translator. Refer to "SQLCheckerCustomizer for Profile Semantics-Checking" on page A-30.

The following tasks are always performed during semantics-checking, regardless of the status of online checking or offline parsing:

1. SQLJ analyzes the types of Java expressions in your SQLJ executable statements.

   This includes examining the SQLJ source files being translated, any `.java` files entered on the command line, and any imported Java classes whose `.class` files or `.java` files can be found through the classpath. SQLJ examines whether and how stream types are used in `SELECT` or `CAST` statements, what Java types are used in iterator columns or `INTO`-lists, what Java types are used as input host variables, and what Java types are used as output host variables.

   SQLJ also processes `FETCH`, `CAST`, `CALL`, `SET TRANSACTION`, `VALUES`, and `SET` statements syntactically.

   Any Java expression in a SQLJ executable statement must have a Java type valid for the given situation and usage. For example, consider the following statement:

   ```
   #sql [myCtx] { UPDATE ... };
   ```

   The `myCtx` variable, which might be used to specify a connection context instance or execution context instance for this statement, must actually resolve to a SQLJ connection context type or execution context type.

   Now consider the following example:

   ```
   #sql { UPDATE emp SET sal = :newSal };
   ```

   If `newSal` is a variable, as opposed to a field, then an error is generated if `newSal` was not previously declared. In any case, an error is generated if it cannot be

assigned to a valid Java type or its Java type cannot be used in a SQL statement (for example, `java.util.Vector`).

> **Note:** Semantics-checking of Java types is performed only for Java expressions within SQLJ executable statements. Such errors in your standard Java statements will *not* be detected until compilation by the Java compiler.

2. SQLJ tries to categorize your embedded SQL operations. Each operation must have a recognizable keyword, such as `SELECT` or `INSERT`, so that SQLJ knows what kind of operation it is. For example, the following statement will generate an error:

   ```
   #sql { foo };
   ```

3. If either online checking or offline parsing (or both) is enabled, then SQLJ analyzes and verifies the syntax of embedded SQL and PL/SQL operations.
4. If either online checking or offline parsing (or both) is enabled, then SQLJ checks the types of Java expressions in SQLJ executable statements against SQL types of corresponding columns in the database and SQL types of corresponding arguments and return variables of stored procedures and functions.

   In the process of doing this, SQLJ verifies that the SQL entities used in your SQLJ executable statements, such as tables, views, and stored procedures, actually exist in the database. SQLJ also checks nullability of database columns whose data is being selected into iterator columns of Java primitive types, which cannot process null data. However, nullability is not checked for stored procedure and function output parameters and return values.

Code Generation

For the `.sqlj` application source file, the SQLJ translator generates a `.java` file and, for ISO standard SQLJ code generation, at least one profile (either in `.ser` or `.class` files). The `.java` contains your translated application source code, class definitions for any private iterators and connection contexts you declared, and, for ISO code, a profile-keys class definition generated and used internally by SQLJ.

> **Note:** No profiles or profile-keys class are generated if you use the default Oracle-specific code generation mode.
>
> With ISO code generation, there are no profiles or profile-keys class if you do not use any SQLJ executable statements in your code.

Generated Application Code in .java File

For the default Oracle-specific code generation, the generated `.java` file for your application contains direct calls to the Oracle JDBC driver in place of the original SQLJ executable statements. There are also calls to an Oracle-specific SQLJ run time. For ISO standard SQLJ code generation, SQLJ executable statements are replaced by calls to the SQLJ run time, which in turn contains calls to the JDBC driver.

For convenience, generated `.java` files also include a comment for each of your `#sql` statements, repeating the statement in its entirety for reference. The generated `.java` file will have the same base name as the input `.sqlj` file, which would be the name of the public class defined in the `.sqlj` file or the first class defined if there are no public

classes. For example, `Foo.sqlj` defines the `Foo` class. The `Foo.java` source file will be generated by the translator.

The location of the generated `.java` file depends on whether and how you set the SQLJ `-dir` option. By default, the `.java` file will be placed in the directory of the `.sqlj` input file.

> **See Also:** "Output Directory for Generated .java Files (-dir)" on page 8-24

Generated Profile-Keys Class in .java File (ISO Code Generation)

If you use ISO standard SQLJ code generation, SQLJ generates a profile-keys class that it uses internally during run time to load and access the serialized profile. This class contains mapping information between the SQLJ run time calls in your translated application and the SQL operations placed in the serialized profile. It also contains methods to access the serialized profile.

> **Note:** If you use the default Oracle-specific code generation, no profiles or profile-keys classes are generated.

The profile-keys class is defined in the same `.java` output file that has your translated application source code, with a class name based on the base name of your `.sqlj` source file as follows:

```
Basename_SJProfileKeys
```

For example, translating `Foo.sqlj` defines the following profile-keys class in the generated `.java` file:

```
Foo_SJProfileKeys
```

If your application is in a package, this is reflected appropriately. For example, translating `Foo.sqlj` in the `a.b` package defines the following class:

```
a.b.Foo_SJProfileKeys
```

Generated Profiles in .ser or .class Files (ISO Code Generation)

If you use ISO standard SQLJ code generation, SQLJ generates profiles that it uses to store information about the SQL operations found in the input file. A profile is generated for each connection context class that you use in your application. It describes the operations to be performed using instances of the associated connection context class, such as SQL operations to execute, tables to access, and stored procedures and functions to call.

> **Note:** If you use the default Oracle-specific code generation, then information about the SQL operations is embedded in the generated code, which calls the Oracle JDBC driver directly. In this case, SQLJ does not generate profiles.

Profiles are generated in `.ser` serialized resource files. However, if you enable the SQLJ `-ser2class` option, then the profiles are automatically converted to `.class` files as part of the translation. In this case, no further customization of the profile is possible. You would have to delete the `.class` file and rerun the SQLJ translator to regenerate the profile.

Profile base names are generated similarly to the profile-keys class name. They are fully qualified with the package name, followed by the `.sqlj` file base name, followed by the string:

```
_SJProfilen
```

Where *n* is a unique number, starting with 0, for each profile generated for a particular `.sqlj` input file.

Again using the example of the `Foo.sqlj` input file, if two profiles are generated, then they will have the following base names (presuming no package):

```
Foo_SJProfile0
Foo_SJProfile1
```

If `Foo.sqlj` is in the `a.b` package, then the profile base names will be:

```
a.b.Foo_SJProfile0
a.b.Foo_SJProfile1
```

Physically, a profile exists as a Java serialized object contained within a resource file. Resource files containing profiles use the `.ser` extension and are named according to the base name of the profile (excluding package names). Resource files for the two previously mentioned profiles will be named as follows:

```
Foo_SJProfile0.ser
Foo_SJProfile1.ser
```

Or, they will be named `Foo_SJProfile0.class` and `Foo_SJProfile1.class` if you enable the `-ser2class` option. If you choose this option, then the conversion to `.class` takes place *after* the customization step.

> **See Also:** "Conversion of .ser File to .class File (-ser2class)" on page 8-53

The location of these files depends on how the SQLJ `-d` option is set, which determines where all generated `.ser` and `.class` files are placed.

> **See Also:** "Output Directory for Generated .ser and .class Files (-d)" on page 8-22

More About Generated Calls to SQLJ Run Time

When `#sql` statements are replaced by calls to the JDBC driver (for Oracle-specific code generation) or to the SQLJ run time (for ISO standard SQLJ code generation), these calls implement the steps in Table 9–1.

Table 9–1 Steps for Generated Calls, ISO Standard Versus Oracle-Specific

Steps for ISO Standard Code Generation	Steps for Oracle Code Generation
Get a SQLJ statement object, using information stored in the associated profile entry.	Get an Oracle JDBC statement object.
Bind inputs into the statement, using `setXXX()` methods of the statement object.	Bind inputs using Oracle JDBC statement methods and, if necessary, register output parameters.
Execute the statement, using the `executeUpdate()` or `executeQuery()` method of the statement object.	Execute the Oracle statement.

Table 9–1 (Cont.) Steps for Generated Calls, ISO Standard Versus Oracle-Specific

Steps for ISO Standard Code Generation	Steps for Oracle Code Generation
Create iterator instances, if applicable.	Create iterator instances, if applicable.
Retrieve outputs from the statement, using `getXXX()` methods of the statement object.	Retrieve outputs from the statement using appropriate JDBC getter methods.
Close the SQLJ statement object (by default, recycling it through the SQLJ statement cache).	Close the JDBC statement object (by default, recycling it through the JDBC statement cache).

A SQLJ run time uses SQLJ statement objects that are similar to JDBC statement objects, although a particular implementation of SQLJ might or might not use JDBC statement classes directly. SQLJ statement classes add functionality particular to SQLJ. For example:

- Standard SQLJ statement objects raise a SQL exception if a null value from the database is to be output to a primitive Java type, such as `int` or `float`, which cannot take null values.
- Oracle SQLJ statement objects allow user-defined object and collection types to be passed to or retrieved from Oracle Database.

Java Compilation

After code generation, SQLJ invokes the Java compiler to compile the generated `.java` file. This produces a `.class` file for each class you defined in your application, including iterator and connection context declarations, as well as a `.class` file for the generated profile-keys class if you use ISO code generation (and presuming your application uses SQLJ executable statements). Any `.java` files you specified directly on the SQLJ command line (for type-resolution, for example) are compiled at this time as well.

In the example used in "Code Generation" on page 9-3, the following `.class` files would be produced in the appropriate directory (given package information in the source code):

- `Foo.class`
- `Foo_SJProfileKeys.class` (ISO code generation only)
- A `.class` file for each additional class you defined in `Foo.sqlj`
- A `.class` file for each iterator and connection context class you declared in `Foo.sqlj` (whether public or private)

To ensure that `.class` files and profiles (if any, whether `.ser` or `.class`) will be located in the same directory, SQLJ passes its `-d` option to the Java compiler. If the `-d` option is not set, then `.class` files and profiles are placed in the same directory as the generated `.java` file, which is placed according to the `-dir` option setting.

In addition, so that SQLJ and the Java compiler will use the same encoding, SQLJ passes its `-encoding` option to the Java compiler unless the SQLJ `-compiler-encoding-flag` is turned off. If the `-encoding` option is not set, then SQLJ and the compiler will use the setting in the Java virtual machine (JVM) `file.encoding` property.

By default, SQLJ invokes the standard `javac` compiler of the Sun Microsystems Java Development Kit (JDK), but other compilers can be used instead. You can request that an alternative Java compiler be used by setting the SQLJ `-compiler-executable` option.

> **Note:** If you are using the SQLJ `-encoding` option but using a compiler that does not have an `-encoding` option, then turn off the SQLJ `-compiler-encoding-flag`. Otherwise, SQLJ will attempt to pass the `-encoding` option to the compiler.

For information about SQLJ support for compiler options and compiler-related SQLJ options, refer to the following:

- "Option Support for javac" on page 8-7
- "Output Directory for Generated .ser and .class Files (-d)" on page 8-22
- "Encoding for Input and Output Source Files (-encoding)" on page 8-21
- "Options to Pass to the Java Compiler (-C)" on page 8-48
- "Compilation Flag (-compile)" on page 8-51
- "Compiler Encoding Support (-compiler-encoding-flag)" on page 8-64
- "Name of the Java Compiler (-compiler-executable)" on page 8-63
- "Compiler Output File (-compiler-output-file)" on page 8-65

Profile Customization (ISO Code Generation)

After Java compilation, if you are using ISO standard code generation, then the generated profiles containing information about your embedded SQL instructions are customized, so that your application can work efficiently with your database and use vendor-specific extensions.

> **Note:** If you use the default Oracle-specific code generation, then SQLJ produces no profiles and skips the customization step. Your code will support Oracle-specific features through direct calls to Oracle JDBC application programming interfaces (APIs).

If you want to check for the options already set on the customizer, then you can make use of the following command:

```
% sqlj -P-print *.ser
```

For more information about profile print option, refer to "Specialized Customizer: Profile Print Option (print)" on page A-16.

To accomplish customization, SQLJ invokes a front end called the customizer harness, which is a Java class that functions as a command-line utility. The harness, in turn, invokes a particular customizer, either the default Oracle customizer or a customizer that you specify through SQLJ option settings.

During customization, profiles are updated in the following ways:

- To enable your application to use any vendor-specific database types or features, if applicable
- To tailor the profiles so that your application is as efficient as possible in using features of the relevant database environment

Without customization, you can access and use only standard JDBC types.

For example, the Oracle customizer can update a profile to support a SQL `PERSON` type that you had defined. You could then use `PERSON` as you would any other supported data type.

You must also customize with the Oracle customizer to use any of the `oracle.sql` type extensions.

Note: Be aware of the following regarding profile customization:

- The Oracle SQLJ run time and an Oracle JDBC driver will be required by your application whenever you use the Oracle customizer during translation, even if you do not use Oracle extensions in your code.
- The generic SQLJ run time will be used if your application has no customizations, or none suitable for the connection.
- You can customize previously created profiles by specifying `.ser` files, or `.jar` files containing `.ser` files, on the command line. But you cannot do this in the same running of SQLJ where translations are taking place. You can specify `.ser`/`.jar` files to be customized or `.sqlj`/`.java` files to be translated, compiled, and customized, but not both categories. For more information about how `.jar` files are used, refer to "JAR Files for Profiles" on page A-29.

For more information about profile customization, refer to Appendix A, "Customization and Specialized Customizers".

Also see the following for information about SQLJ options related to profile customization:

- "Default Profile Customizer (-default-customizer)" on page 8-67
- "Options to Pass to the Profile Customizer (-P)" on page 8-50
- "Profile Customization Flag (-profile)" on page 8-52
- "Customization Options and Choosing a Customizer" on page A-6

Functionality of Translator Errors, Messages, and Exit Codes

This section provides an overview of SQLJ translator messages and exit codes. It covers the following topics:

- Translator Error, Warning, and Information Messages
- Translator Status Messages
- Translator Exit Codes

Translator Error, Warning, and Information Messages

There are three major levels of SQLJ messages that you may encounter during the translation phase: error, warning, and information. Warning messages can be further broken down into two levels: nonsuppressible and suppressible. Therefore, there are four message categories (in order of seriousness):

1. Errors
2. Nonsuppressible warnings

3. Suppressible warnings
4. Information

You can control suppressible warnings and information by using the SQLJ `-warn` option.

Error messages, prefixed by `Error:`, indicate that one of the following has been encountered:

- A condition that would prevent compilation (for example, the source file contains a public class whose name does not match the base file name)
- A condition that would result in a run-time error if the code were executed (for example, the code attempts to fetch a `VARCHAR` into a `java.util.Vector`, using an Oracle JDBC driver)

If errors are encountered during SQLJ translation, then no output is produced and compilation and customization are not executed.

Nonsuppressible warning messages, prefixed by `Warning:`, indicate that one of the following has been encountered:

- A condition that would probably, but not necessarily, result in a run-time error if the code were executed (for example, a `SELECT` statement whose output is not assigned to anything)
- A condition that compromises the ability of SQLJ to verify run-time aspects of your source code (for example, not being able to connect to the database you specify for online checking)
- A condition that presumably resulted from a coding error or oversight

SQLJ translation will complete if a nonsuppressible warning is encountered, but you should analyze the problem and determine if it should be fixed before running the application. If online checking is specified but cannot be completed, then offline checking is performed instead.

> **Note:** For logistical reasons, the parser that the SQLJ translator uses to analyze SQL operations is not the same top-level SQL parser that will be used at run time. Therefore, errors might occasionally be detected during translation that will not actually cause problems when your application runs. Accordingly, such errors are reported as nonsuppressible warnings, rather than fatal errors.

Suppressible warning messages, also prefixed by `Warning:`, indicate that there is a problem with a particular aspect of your application, such as portability. An example of this is using an Oracle-specific type, such as `oracle.sql.NUMBER`, to read from or write to Oracle Database 11*g*.

Informational or status messages prefixed by `Info:` do not indicate an error condition. They merely provide additional information about what occurred during the translation phase.

Suppressible warning and status messages can be suppressed by using the various `-warn` option flags:

- `cast/nocast`: The `nocast` setting suppresses warnings about possible run-time errors when trying to cast an object type instance to an instance of a subtype.

- `precision/noprecision`: The `noprecision` setting suppresses warnings regarding possible loss of data precision during conversion.
- `nulls/nonulls`: The `nonulls` setting suppresses warnings about possible run-time errors due to nullable columns or types.
- `portable/noportable`: The `noportable` setting suppresses warnings regarding SQLJ code that uses Oracle-specific features or might otherwise be nonstandard and, therefore, not portable to other environments.
- `strict/nostrict`: The `nostrict` setting suppresses warnings issued if there are fewer columns in a named iterator than in the selected data that is to populate the iterator.
- `verbose/noverbose`: The `noverbose` setting suppresses status messages that are merely informational and do not indicate error or warning conditions.

See Also: "Translator Warnings (-warn)" on page 8-33

If you receive warnings during your SQLJ translation, then you can try running the translator again with `-warn=none` to see if any of the warnings are of the more serious (nonsuppressible) variety.

Table 9–2 summarizes the categories of error and status messages generated by the SQLJ translator.

Table 9–2 SQLJ Translator Error Message Categories

Message Category	Prefix	Indicates	Suppressed By
Error	`Error:`	Fatal error that will cause compilation failure or run-time failure (translation aborted)	NA
Nonsuppressible warning	`Warning`:	Condition that prevents proper translation or might cause run-time failure (translation completed)	NA
Suppressible warning	`Warning`:	Problem regarding a particular aspect of your application (translation completed)	`-warn` option flags: `nocast` `noprecision` `nonulls` `noportable` `nostrict`
Informational/status message	`Info`:	Information regarding the translation process	`-warn` option flag: `noverbose`

Translator Status Messages

In addition to the error, warning, and information messages, SQLJ can produce status messages throughout all phases of SQLJ operation: translation, compilation, and customization. Status messages are produced as each file is processed and at each phase of SQLJ operation.

You can control status messages by using the SQLJ `-status` option.

See Also: "Real-Time Status Messages (-status)" on page 8-35

Translator Exit Codes

The following exit codes are returned by the SQLJ translator to the operating system upon completion:

- `0`: No error in execution
- `1`: Error in SQLJ execution
- `2`: Error in Java compilation
- `3`: Error in profile customization
- `4`: Error in class instrumentation, the optional mapping of line numbers from your `.sqlj` source file to the resulting `.class` file
- `5`: Error in `ser2class` conversion, the optional conversion of profile files from `.ser` files to `.class` files

---

Note:

- If you issue the `-help` or `-version` option, then the SQLJ exit code is `0`.
- If you run SQLJ without specifying any files to process, then SQLJ issues help output and returns exit code `1`.

SQLJ Run Time

This section presents information about the Oracle SQLJ run time, which is a thin layer of pure Java code that runs above the JDBC driver.

If you use the default Oracle-specific code generation, then the SQLJ run time layer becomes even thinner, with a run time subset being used in conjunction with an Oracle JDBC driver. Most of the run time functionality is compiled directly into Oracle JDBC calls. You cannot use a non-Oracle JDBC driver.

> **See Also:** "Oracle-Specific Code Generation (No Profiles)" on page 3-28

When SQLJ translates SQLJ source code using ISO standard code generation, embedded SQL commands in your Java application are replaced by calls to the SQLJ run time. Run time classes act as wrappers for equivalent JDBC classes, providing special SQLJ functionality. When the end user runs the application, the SQLJ run time acts as an intermediary, reading information about your SQL operations from your profile and passing instructions along to the JDBC driver.

Generally speaking, however, a SQLJ run time can be implemented to use any JDBC driver or vendor-proprietary means of accessing the database. The Oracle SQLJ run time requires a JDBC driver, but can use any standard JDBC driver. To use Oracle-specific data types and features, however, you must use an Oracle JDBC driver. For the purposes of this document, it is generally assumed that you are using Oracle Database and one of the Oracle JDBC drivers.

> **Note:** For ISO standard SQLJ code generation, the Oracle SQLJ run time and an Oracle JDBC driver will be required by your application whenever you use the Oracle customizer during translation, even if you do not use Oracle extensions in your code. The generic SQLJ run time will be used if your application has no customizations, or none suitable for the connection.

SQLJ Run Time Packages

The Oracle SQLJ run time includes packages you will likely import and use directly, and others that are used only indirectly.

> **Note:** These packages are included in the run time libraries `runtime12`, `runtime12ee`, and `runtime`.

Packages Used Directly

The packages containing classes that you can import and use directly in your application are:

- `sqlj.runtime`

  This package includes the `ExecutionContext` class, `ConnectionContext` interface, `ConnectionContextFactory` interface, `ResultSetIterator` interface, `ScrollableResultSetIterator` interface, and wrapper classes for streams (`BinaryStream` and `CharacterStream`, as well as the deprecated `AsciiStream` and `UnicodeStream`).

  Interfaces and abstract classes in this package are implemented by classes in the `sqlj.runtime.ref` or `oracle.sqlj.runtime` package or by classes generated by the SQLJ translator.

- `sqlj.runtime.ref`

  The classes in this package implement interfaces and abstract classes in the `sqlj.runtime` package. You will likely use the `sqlj.runtime.ref.DefaultContext` class, which is used to specify your default connection and create default connection context instances. The other classes in this package are used internally by SQLJ in defining classes during code generation, such as iterator classes and connection context classes that you declare in your SQLJ code.

- `oracle.sqlj.runtime`

  This package contains the `Oracle` class that you can use to instantiate the `DefaultContext` class and establish your default connection. It also contains Oracle-specific run time classes used by the Oracle implementation of SQLJ, including functionality to convert to and from Oracle type extensions.

  > **Note:** Packages whose names begin with `oracle` are for Oracle-specific SQLJ features.

Packages Used Indirectly

The packages containing classes that are for internal use by SQLJ are:

- `sqlj.runtime.profile`

This package contains interfaces and abstract classes that define what SQLJ profiles look like (applicable only for ISO standard code generation). This includes the `EntryInfo` and `TypeInfo` classes. Each entry in a profile is described by an `EntryInfo` object, where a profile entry corresponds to a SQL operation in your application. Each parameter in a profile entry is described by a `TypeInfo` object.

The interfaces and classes in this package are implemented by classes in the `sqlj.runtime.profile.ref` package.

- `sqlj.runtime.profile.ref`

  This package contains classes that implement the interfaces and abstract classes of the `sqlj.runtime.profile` package and are used internally by the SQLJ translator in defining profiles (for ISO standard code generation only). It also provides the default JDBC-based run time implementation.

- `sqlj.runtime.error`

  This package, used internally by SQLJ, contains resource files for all generic (not Oracle-specific) error messages that can be generated by the SQLJ translator.

- `oracle.sqlj.runtime.error`

  This package, used internally by SQLJ, contains resource files for all Oracle-specific error messages that can be generated by the SQLJ translator.

Categories of Run-Time Errors

Run-time errors can be generated by any of the following:

- SQLJ run time
- JDBC driver
- RDBMS

In any of these cases, a SQL exception is generated as an instance of the `java.sql.SQLException` class, or as a subclass, such as `sqlj.runtime.SQLNullException`.

Depending on where the error came from, there might be meaningful information you can retrieve from an exception using the `getSQLState()`, `getErrorCode()`, and `getMessage()` methods. SQLJ errors, for example, include meaningful SQL states and messages.

See Also: "Retrieving SQL States and Error Codes" on page 3-17

If errors are generated by the Oracle JDBC driver or RDBMS at run time, look at the prefix and consult the appropriate documentation:

- *Oracle Database JDBC Developer's Guide and Reference* for JDBC errors
- Oracle error message documentation for RDBMS errors (see "Related Documents" on page xvi)

Globalization Support in the Translator and Run Time

The Oracle SQLJ implementation uses the Java built-in capabilities for globalization support. This section discusses the following:

- Basics of SQLJ support for globalization and native character encoding, starting with background information covering some of the implementation details of character encoding and language support in the Oracle implementation
- Options available through the SQLJ command line that enable you to adjust your Oracle Globalization Support configuration
- Extended Oracle globalization support
- Relevant manipulation outside of SQLJ for globalization support

> **Note:** Some prior knowledge of Oracle Globalization Support is assumed, particularly regarding character encoding and locales. For information, refer to the *Oracle Database Globalization Support Guide*.

This section covers the following topics:

Character Encoding and Language Support

There are two main areas of SQLJ globalization support:

- Character encoding

  There are three parts to this:

  – Character encoding for reading and generating source files during SQLJ translation

  – Character encoding for generating error and status messages during SQLJ translation

  – Character encoding for generating error and status messages when the application runs

- Language support

  This determines which translations of error and status message lists are used when SQLJ outputs messages to the user, either during SQLJ translation or at SQLJ run time.

Globalization support at run time is transparent to the user, presuming your SQLJ source code and SQL character data use only characters that are within the database character set. SQL character data is transparently mapped into and out of Unicode.

Note that for multi-language applications, it is advisable to use one of the following options:

- Use a database whose character set supports Unicode.
- Even if your database character set does not support Unicode, specify that the national language character set supports Unicode. (Refer to the *Oracle Database Globalization Support Guide*.) In this case, you will typically use the SQLJ Unicode character types described in "SQLJ Extended Globalization Support" on page 9-18.

Note:

- The SQLJ translator fully supports Unicode 2.0 and Java Unicode escape sequences. However, the SQLJ command-line utility does not support Unicode escape sequences. You can use only native characters supported by the operating system. Command-line options requiring Unicode escape sequences can be entered in a SQLJ properties file instead, because properties files do support Unicode escape sequences.
- Encoding and conversion of characters in your embedded SQL operations and characters read or written to the database, are handled by JDBC directly. SQLJ does not play a role in this. If online semantics-checking is enabled during translation, however, then you will be warned if there are characters within the text of your SQL data manipulation language (DML) operations that might not be convertible to the database character set.
- For information about JDBC globalization support functionality, refer to the *Oracle Database JDBC Developer's Guide and Reference*.

Overview of Character Encoding

The character encoding setting for source files tells SQLJ two things:

- How source code is represented in `.sqlj` and `.java` input files that the SQLJ translator must read
- How SQLJ should represent source code in `.java` output files that it generates

By default, SQLJ uses the encoding indicated by the JVM `file.encoding` property. If your source files use other encodings, then you must indicate this to SQLJ so that appropriate conversion can be performed.

Use the SQLJ `-encoding` option to accomplish this. SQLJ also passes the `-encoding` setting to the compiler for it to use in reading `.java` files, unless the SQLJ `-compiler-encoding-flag` is off.

Note: Do not alter the `file.encoding` system property to specify encodings for source files. This might impact other aspects of your Java operation and might offer only a limited number of encodings, depending on platform or operating system considerations.

The system character-encoding setting also determines how SQLJ error and status messages are represented when output to the user, either during translation or during run time when the end user is running the application. This is set according to the `file.encoding` property and is unaffected by the SQLJ `-encoding` option.

For source file encoding, you can use the `-encoding` option to specify any character encoding supported by your Java environment. If you are using the Sun Microsystems JDK, then these are listed in the `native2ascii` documentation, which you can find at the following Web site:

`http://java.sun.com/j2se/1.3/docs/tooldocs/solaris/native2ascii.html`

Dozens of encodings are supported by the Sun Microsystems JDK. These include `8859_1` through `8859_9` (ISO Latin-1 through ISO Latin-9), `JIS` (Japanese), `SJIS` (shift-JIS, Japanese), and `UTF8`.

Character Encoding Notes

Be aware of the following:

- A character that is not representable in the encoding used, for either messages or source files, can always be represented as a Java Unicode escape sequence. This is of the form `\uHHHH`, where each H is a hexadecimal digit.
- As a `.sqlj` source file is read and processed during translation, error messages quote source locations based on character position (not byte position) in the input encoding.
- Encoding settings, either set through the SQLJ `-encoding` option or the Java `file.encoding` setting, do not apply to Java properties files, such as `sqlj.properties` and `connect.properties`. Properties files always use the encoding `8859_1`. This is a feature of Java in general, not SQLJ in particular. However, you can use Unicode escape sequences in a properties file. You can use the `native2ascii` utility to determine escape sequences.

See Also: "Using native2ascii for Source File Encoding" on page 9-22

Overview of Language Support

SQLJ error and status reporting, either during translation or during run time, uses the Java locale setting in the JVM `user.language` property. Users typically do not have to alter this setting.

Language support is implemented through message resources that use key/value pairs. For example, where an English-language resource has a key/value pair of `"OkKey", "Okay"`, a German-language resource has a key/value pair of `"OkKey", "Gut"`. The locale setting determines the message resources used.

SQLJ supports locale settings of `en` (English), `de` (German), `fr` (French), and `ja` (Japanese).

Note: Java locale settings can support country and variant extensions in addition to language extensions. For example, consider `ErrorMessages_de_CH_var1`, where `CH` is the Swiss country extension of German and `var1` is an additional variant. SQLJ, however, currently supports only language extensions (`de` in this example), ignoring country and variant extensions.

SQLJ and Java Settings for Character Encoding and Language Support

The Oracle SQLJ implementation provides syntax that enables you to set the following:

- The character encoding used by the SQLJ translator and Java compiler in representing source code

  Use the SQLJ `-encoding` option.

- The character encoding used by the SQLJ translator and run time in representing error and status messages

  Use the SQLJ `-J` prefix to set the Java `file.encoding` property.

- The locale used by the SQLJ translator and run time for error and status messages

  Use the SQLJ `-J` prefix to set the Java `user.language` property.

Setting Character Encoding for Source Code

Use the SQLJ `-encoding` option to determine the character encoding used in representing `.sqlj` files read by the translator, `.java` files generated by the translator, and `.java` files read by the compiler. The option setting is passed by SQLJ to the compiler, unless the SQLJ `-compiler-encoding-flag` is off.

This option can be set on the command line or `SQLJ_OPTIONS` environment variable, as in the following example:

```
-encoding=SJIS
```

Alternatively, you can set it in a SQLJ properties file, as follows:

```
sqlj.encoding=SJIS
```

If the encoding option is not set, then both the translator and compiler will use the encoding specified in the JVM `file.encoding` property. This can also be set through the SQLJ command line.

> **See Also:** "Encoding for Input and Output Source Files (-encoding)" on page 8-21 and "Compiler Encoding Support (-compiler-encoding-flag)" on page 8-21

Note: If your `-encoding` is to be set routinely to the same value, then it is most convenient to specify it in a properties file, as in the second example.

Setting Character Encoding and Locale for SQLJ Messages

Character encoding and locale for SQLJ error and status messages produced, during both translation and run time, are determined by the Java `file.encoding` and `user.language` properties. Although it is typically not necessary, you can set these and other JVM properties in the SQLJ command line by using the SQLJ `-J` prefix. Options marked by this prefix are passed to the JVM.

Set the character encoding as in the following example, which specifies shift-JIS Japanese character encoding:

```
-J-Dfile.encoding=SJIS
```

Note: Only a limited number of encodings might be available, depending on platform or operating system considerations.

Set the locale as in the following example (which specifies Japanese locale):

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

The `-J` prefix can be used on the command line or `SQLJ_OPTIONS` environment variable only. It cannot be used in a properties file, because properties files are read after the JVM is invoked.

Note:

- If your `file.encoding`, `user.language`, or any other Java property is to be set routinely to the same value, it is most convenient to specify `-J` settings in the `SQLJ_OPTIONS` environment variable. This way, you do not have to repeatedly specify them on the command line. The syntax is essentially the same as on the command line. For more information, refer to "SQLJ_OPTIONS Environment Variable for Option Settings" on page 8-15.
- Remember that if you do not set the SQLJ `-encoding` option, then setting `file.encoding` will affect encoding for source files as well as error and status messages.
- Be aware that altering the `file.encoding` property might have unforeseen consequences on other aspects of your Java operations. Also, any new setting must be compatible with your operating system.

See Also: "Command-Line Syntax and Operations" on page 8-9 and "Options to Pass to the Java Virtual Machine (-J)" on page 8-48

SQLJ Command-Line Example: Setting Encoding and Locale

Following is a complete SQLJ command line, including JVM `file.encoding` and `user.language` settings:

```
% sqlj -encoding=8859_1 -J-Dfile.encoding=SJIS -J-Duser.language=ja Foo.sqlj
```

This example uses the SQLJ `-encoding` option to specify `8859_1` (Latin-1) for source code representation during SQLJ translation. This encoding is used by the translator in reading the `.sqlj` input file and in generating the `.java` output file. The encoding is then passed to the Java compiler to be used in reading the generated `.java` file. The `-encoding` option, when specified, is always passed to the Java compiler unless the SQLJ `-compiler-encoding-flag` is disabled.

For error and status messages output during translation of `Foo.sqlj`, the SQLJ translator uses the `SJIS` encoding and the `ja` locale.

SQLJ Extended Globalization Support

The Oracle SQLJ implementation includes support for Java types (Unicode character types) derived from existing character and stream types that convey expected usage for globalization. In SQLJ it is not possible to use JDBC statement or result set methods directly that otherwise serve the purpose of globalization support. If you are interested in information about those methods, refer to the *Oracle Database JDBC Developer's Guide and Reference*

If the database natively supports Unicode, then the types described in "Java Types for Globalization Support" are unnecessary. In this case, globalization support will be handled transparently. It is when the database does not natively support Unicode, but has a national language character set that does support Unicode, that you will typically use these types (for columns that use the national language character set).

Java Types for Globalization Support

The Oracle SQLJ implementation provides a number of Java types for globalization support. Table 9–3 notes the correspondence between these globalization support types and general-use JDBC and SQLJ character and stream types. Each globalization support type, except for `NString`, is a subclass of its corresponding JDBC or SQLJ type.

Table 9–3 JDBC and SQLJ Types and Corresponding Globalization Types

JDBC and SQLJ Types	Globalization Support Types
JDBC types:	
`oracle.sql.CHAR`	`oracle.sql.NCHAR`
`java.lang.String`	`oracle.sql.NString`
`oracle.sql.CLOB`	`oracle.sql.NCLOB`
SQLJ types:	
`sqlj.runtime.CharacterStream`	`oracle.sqlj.runtime.NcharCharacterStream`
`sqlj.runtime.AsciiStream` (Deprecated; use `CharacterStream`.)	`oracle.sqlj.runtime.NcharAsciiStream` (Deprecated; use `NcharCharacterStream`.)
`sqlj.runtime.UnicodeStream` (Deprecated; use `CharacterStream`.)	`oracle.sqlj.runtime.NcharUnicodeStream` (Deprecated; use `NcharCharacterStream`.)

In situations where your application must handle national language character strings, either inserting them into or selecting them from national language character set columns, use the globalization support types instead of the corresponding general-use types.

Note:

- All globalization support types add automatic registration of intended usage for `IN` and `OUT` parameters, but are otherwise identical in usage to the corresponding JDBC or SQLJ type (including constructors).
- Use of globalization support types is unnecessary in iterator columns, because the underlying network protocol supports national language characters implicitly for the underlying result sets.

NString Class Usage and Notes

The `oracle.sql.CHAR` class, and therefore its `NCHAR` subclass, provides only constructors that require explicit knowledge of the database character set. Therefore, the `oracle.sql.NString` class, a wrapper for `java.lang.String`, is preferable in most circumstances. The `NString` class provides simpler constructors and ensures that the national language character form of use is registered with the JDBC driver.

Following are the key `NString` methods:

- `NString(String)`: This constructor creates an `NString` instance from an existing `String` instance.
- `String toString()`: This method returns the underlying `String` instance.

- `String getString()`: This method also returns the underlying `String` instance.

The `toString()` method enables you to use the `NString` instance in string concatenation expressions (such as `"a"+b`, where `b` is a string). The `getString()` method, provided in the `CHAR` superclass, is supported as well for uniformity. In addition, the member methods of the `String` class are carried over to the `NString` wrapper class to enable you to write more concise code.

In SQLJ applications, for versions prior to Oracle Database 11*g*, you must use host variables of the `NString` type to bind columns of the `NCHAR` type. For example, consider the following table:

```
CREATE TABLE Tbl1 (
       ColA  CHAR
       NColB NCHAR
)
```

To insert a row in this table through a SQLJ application, use a code similar to the following:

```
...
String v_a = "\uFF5E";
NString v_nb = "\uFF5E";
#sql {INSERT INTO Tbl1 (ColA, NColB) VALUES (:v_a, :v_nb)};
...
```

In Oracle Database 11*g* release 1 (11.1), SQLJ applications can use `String` variables to bind `NCHAR` columns. Therefore, the preceding example can be rewritten as follows:

```
...
String v_a = "\uFF5E";
String v_nb = "\uFF5E";
#sql {INSERT INTO Tbl1 (ColA, NColB) VALUES (:v_a, :v_nb)};
...
```

However, if you want to use `String` host variable to bind to `NCHAR` columns, then you must translate the SQLJ file with the `-ncharconv` SQLJ translator option, as follows:

```
sqlj -ncharconv [-options] app.sqlj
```

In the preceding command, *`options`* can be other SQLJ options and `app.sqlj` is the SQLJ file that contains the code.

When this option is used, the `setFormOfUse` method will be generated for all binds to character columns, that is, `CHAR` or `NCHAR` columns.

> **Note:** When the SQLJ file is compiled with the `-ncharconv` option, the `setFormOfUse` method is used in the generated code for `codegen=oracle`. For `codegen=iso`, this option information is passed to the Oracle SQLJ run time, which internally uses `SetFormOfUse` for bind at run time.

The SQLJ application can use `String` host variable to retrieve data from the server without using the `-ncharconv` option, because the information about the column type is fetched at the client-side and JDBC internally sets the form for the appropriate columns.

Note: There may be a small difference in the performance when using the `-ncharconv` option, depending on the database character set and national charster set and the number of character columns in the table.

See Also: "NCHAR Bind (-ncharconv)" on page 8-47

Globalization Support Examples

The following examples show use of the `NString` class:

- `NString` as `IN` argument

  This example uses an `NString` instance as an input parameter to the database.

```
import oracle.sql.NString;
...
NString nc_name = new NString("Name with strange characters");
#sql { update PEOPLE
       set city = :(new NString("\ufff2")), name = :nc_name
       where num= :n };
...
```

- `NString` as `OUT` argument

  This example uses an `NString` instance as an output parameter from the database.

```
import oracle.sql.NString;
...
NString nstr;
#sql { call foo(:out nstr) };
System.out.println("Result is: "+nstr);
// or, explicitly: System.out.println("Result is: "+nstr.toString());
...
```

- `NString` as `Result Set` column

  This example uses the `NString` type for an iterator column. Such usage is superfluous, given that the underlying network protocol supports national language characters implicitly, but harmless. This example also shows use of one of the `String` methods, `substring()`, that is carried over to `NString`.

```
import oracle.sql.NString;
import oracle.sql.NCLOB;
...
#sql iterator NIter(NString title, NCLOB article);

NIter nit;
#sql nit = { SELECT article, title FROM page_table };
while (nit.next())
{
  System.out.println("<TITLE>"+nit.title()+"</TITLE>");
  ...
  nit.article().substring(0, 1000); ...
}
```

Note: Using the NCHAR type instead of the NString type for the preceding examples requires the following changes:

- Use the appropriate NCHAR constructor. NCHAR constructors mirror CHAR constructors, such as the following:

  ```
  NCHAR(String str, oracle.sql.CharacterSet charset)
  ```

- Although you have the option of using either toString() or getString() to retrieve the underlying String instance from an NString instance, for an NCHAR instance you must use the getString() method. When using the NString type, the toString() method is used automatically for string concatenation, such as in "NString as OUT argument".

Manipulation Outside of SQLJ for Globalization Support

This section discusses ways to manipulate your Oracle Globalization Support configuration outside of SQLJ.

Setting Encoding and Locale at Application Run Time

As with any end user running any Java application, those running your SQLJ application can specify JVM properties, such as file.encoding and user.language directly, as they invoke the JVM to run your application. This determines the encoding and locale used for message output as your application executes.

They can accomplish this as in the following example:

```
% java -Dfile.encoding=SJIS -Duser.language=ja Foo
```

This will use SJIS encoding and Japanese locale.

Using API to Determine Java Properties

In Java code, you can determine values of Java properties by using the java.lang.System.getProperty() method, specifying the appropriate property. For example:

```
public class Settings
{
   public static void main (String[] args)
   {
      System.out.println("Encoding: " + System.getProperty("file.encoding")
                         + ", Language: " + System.getProperty("user.language"));
   }
}
```

You can compile this and run it as a standalone utility.

There is also a getProperties() method that returns the values of all properties, but this will raise a security exception if you try to use it in code that runs in the server.

Using native2ascii for Source File Encoding

If you are using a Sun Microsystems JDK, then there is an alternative to having SQLJ do the character encoding for your source files. You can use the native2ascii utility to convert sources with native encoding to sources in 7-bit ASCII with Unicode escape sequences.

> **Note:** To use SQLJ to translate source created by `native2ascii`, ensure that the JVM that invokes SQLJ has a `file.encoding` setting that supports some superset of 7-bit ASCII. This is not the case with settings for EBCDIC or Unicode encoding.

Run `native2ascii` as follows:

```
% native2ascii <options> <inputfile> <outputfile>
```

Standard input or standard output are used if you omit the input file or output file. Two options are supported:

- `-reverse` (Reverse the conversion. Convert from Latin-1 or Unicode to native encoding)
- `-encoding <encoding>`

For example:

```
% native2ascii -encoding SJIS Foo.sqlj Temp.sqlj
```

For more information, see the following Web site:

http://java.sun.com/j2se/1.3/docs/tooldocs/solaris/native2ascii.html

10

Performance and Debugging

This chapter discusses features, utilities, and tips to enhance performance of your SQLJ application and to debug your SQLJ source code at run time. The following topics are discussed:

- Performance Enhancement Features
- SQLJ Debugging Features
- SQLJ Support for Oracle Performance Monitoring

Performance Enhancement Features

The Oracle SQLJ implementation includes features to enhance performance by making data access more efficient. These include the following:

- Row Prefetching
- Statement Caching
- Update Batching
- Column Definitions
- Parameter Size Definitions

> **See Also:** *Oracle Database JDBC Developer's Guide and Reference*

Your application will likely benefit from the default Oracle-specific code generation. The generated code will be optimized with direct calls to the Oracle Java Database Connectivity (JDBC) driver, eliminating the overhead of intermediate calls to the SQLJ run time, which in turn would call JDBC.

> **See Also:** "Oracle-Specific Code Generation (No Profiles)" on page 3-28

> **Note:** The Oracle SQLJ implementation does *not* support batch fetches, which is the fetching of sets of rows into arrays of values. However, you may be able to use Oracle row prefetching to obtain some of the benefits of batch fetching.

In addition to the preceding SQLJ performance enhancements, you can use optimizer hints in the SQL operations within a SQLJ program, as you can in any Oracle SQL operations.

The Oracle SQL implementation enables you to tune your SQL statements by using "/*+" or "--+" comment notation to pass hints to the Oracle SQL optimizer. The SQLJ translator recognizes and supports these optimizer hints, passing them at run time as part of your SQL statement.

You can also define cost and selectivity information for a SQLJ stored function, as for any other stored function, using the extensibility features for SQL optimization in Oracle Database 11*g*. During SQL execution, the optimizer invokes the cost and selectivity methods for the stored function, evaluates alternate strategies for execution, and chooses an efficient execution plan.

> **See Also:** *Oracle Database SQL Language Reference* for more information

Note that using Oracle performance extensions in your code requires the following:

- Use one of the Oracle JDBC drivers.
- Use the default Oracle-specific code generation, or customize profiles appropriately.

  For ISO standard code generation, the default customizer, `oracle.sqlj.runtime.util.OraCustomizer`, is recommended.
- Use the Oracle SQLJ run time when your application runs.

The Oracle SQLJ run time and an Oracle JDBC driver are required by your application whenever you customize profiles with the Oracle customizer, even if you do not actually use Oracle extensions in your code.

Row Prefetching

Standard JDBC receives the results of a query one row at a time, with each row requiring a separate round trip to the database. Row prefetching enables you to receive the results more efficiently, in groups of multiple rows each.

Use the `setFetchSize()` method of an `ExecutionContext` instance to set the number of rows to be prefetched whenever you execute a `SELECT` statement (for SQLJ statements using the particular `ExecutionContext` instance).

The `getFetchSize()` method of an `ExecutionContext` instance returns the current prefetch size, as an `int` value.

The following is an example of setting the prefetch size to 20 by getting the default execution context instance of the default connection context instance and calling the `setFetchSize()` method:

```
DefaultContext.getDefaultContext().getExecutionContext().setFetchSize(20);
```

It is also possible to set the prefetch size directly on the underlying `OracleConnection` object using the JDBC application programming interface (API), but in SQLJ this is discouraged.

To specify the number of rows to prefetch for queries that use a given connection context instance, use the underlying JDBC connection, cast to an Oracle`Connection` instance. Following is an example that sets the prefetch value to 20 for your default connection:

```
((OracleConnection)DefaultContext.getDefaultContext().getConnection()).setDefaultR
owPrefetch(20);
```

Also, please note that the prefetch size set on the SQLJ connection context overrides the prefetch size set on the underlying JDBC connection.

The prefetch value needs to be setup on each individual connection context. For example, if `ctx` is an instance of a declared connection context class, set its prefetch value as follows:

```
ctx.getExecutionContext().setFetchSize(20);
```

There is no maximum row-prefetch value. The default is 10 in JDBC, and this is inherited by SQLJ. This value is effective in typical circumstances, although you might want to increase it if you receive a large number of rows.

Statement Caching

SQLJ offers a statement caching feature that improves performance by saving executable statements that are used repeatedly, such as in a loop or in a method that is called repeatedly. When a statement is cached before it is reexecuted, the code does not have to be reparsed (either on the server or on the client), the statement object does not have to be recreated, and the parameter size definitions do not have to be recalculated. Without this feature, repeated statements would have to be reparsed on the client, and perhaps in the server as well, depending on whether a statement is still available in the general server-side SQL cache when it is encountered again.

For Oracle-specific code generation, SQLJ statement caching relies on the Oracle JDBC driver, using the JDBC explicit caching mechanism. This is distinct from the JDBC implicit caching mechanism, although there are interdependencies. With Oracle-specific code, statement caching is controlled through connection methods.

> **See Also:** *Oracle Database JDBC Developer's Guide and Reference*

For ISO code generation, SQLJ has its own statement caching mechanism through functionality of the SQLJ run time. With ISO code, statement caching is controlled through the Oracle customizer `stmtcache` option.

> **Note:** For Oracle-specific code generation, explicit caching is the only statement caching mechanism that can be manipulated through SQLJ APIs. For the discussion in this document, it will be referred to as SQLJ/explicit statement caching.

In Oracle Database 11*g* release 1 (11.1), the default statement cache size will be set to 5, provided the JDBC connection is being created by the connection context. If a connection context is created using an already available JDBC connection or data source, then the statement cache size will be set to that of the JDBC connection or the data source.

Connection Context Methods for Statement Caching (Oracle-Specific Code)

If you use Oracle-specific code generation, which is the case with the SQLJ translator default `-codegen=oracle` setting, use connection context methods for statement caching functionality. Note that any statement cache size greater than 0 results in SQLJ/explicit statement caching being enabled. By default, it is enabled with a cache size of 5, that is, five statements.

The following Oracle-specific (nonstandard) static methods have been added to the `sqlj.runtime.ref.DefaultContext` class, and are also included in any connection context classes you declare:

- `static void setDefaultStmtCacheSize(int)`

  This sets the default statement cache size for *all* connection contexts. This becomes the initial statement cache size for any subsequently created instance of *any* connection context class, not just the class upon which you call the method. The method call does not affect connection context instances that already exist.

  Note: `setDefaultStmtCacheSize(int)` affects the statement cache size only for the connections that are created using the SQLJ connection contexts. It does not affect the statement cache size for the connections that are created using JDBC connections.

  Consider the following two code snippets:

  Example 1:

  ```
  ...
  MyContext.setDefaultStmtCacheSize(10);
  OracleConnection conn = DriverManager.getConnection(url, user,
  passwd);
  myctx = new MyContext(conn);
  ```

  Example 2:

  ```
  ...
  MyContext.setDefaultStmtCacheSize(10);
  myctx = new MyContext(url, user, passwd,true);
  ```

  In the preceding two examples, the statement cache size will be set to 10 only in the second example. In the first example, the statement cache size corresponding to this connection will not be affected because the connection is created using the `getConnection` method of the `DriverManager` interface from JDBC specification.

- `static int getDefaultStmtCacheSize()`

  This retrieves the current default statement cache size for connection contexts.

And the following Oracle-specific instance methods have also been added to the `DefaultContext` class and are included in any other connection context classes:

- `void setStmtCacheSize(int) throws java.sql.SQLException`

  This sets the statement cache size for the underlying connection of the particular connection context instance (overrides the default).

  Note: If SQLJ/explicit caching is already disabled, then setting the size to 0 leaves it disabled. If it is already enabled, then setting the size to 0 leaves it enabled, but renders it nonfunctional.

- `int getStmtCacheSize()`

  This verifies whether SQLJ/explicit statement caching is enabled for the underlying connection of the connection context. If so, it returns the current statement cache size. It can also return either of the following integer constants:

  ```
  static int STMT_CACHE_NOT_ENABLED
  static int STMT_CACHE_EXCEPTION
  ```

It is possible for a `getStmtCacheSize()` call to cause a SQL exception. However, for backward compatibility, this method does not throw the exception directly. When an exception occurs, the method returns the constant `STMT_CACHE_EXCEPTION`. In this case, you can call the `getStmtCacheException()` method to find out what exception occurred.

If you call `getStmtCacheSize()` when SQLJ/explicit caching is disabled, then the method returns the constant `STMT_CACHE_NOT_ENABLED`. This is distinguished from a cache size of 0. Technically, it is possible for SQLJ/explicit caching to be enabled (though useless) with a cache size of 0.

- `java.sql.Exception getStmtCacheException()`

  See if there is a statement caching exception. There are two scenarios for using this method:

  – Call it if a `getStmtCacheSize()` call returns `STMT_CACHE_EXCEPTION`.

  – Call it whenever you create a connection context instance with which you want to use statement caching. This is because of automatic manipulation that occurs with respect to statement cache size whenever you create a connection context instance. If you care about statement caching for the connection context instance, call `getStmtCacheException()` after creating the instance, to verify there were no problems.

Enabling and Disabling Statement Caching (Oracle-Specific Code)

With Oracle-specific code, to reiterate what was stated earlier, any nonzero statement cache size results in SQLJ/explicit caching being enabled. Because the default size is 5, statement caching is enabled by default.

You cannot explicitly disable SQLJ/explicit statement caching through SQLJ APIs, although you can effectively disable it (render it nonfunctional) by setting the statement cache size to 0. In this case, the connection context `getStmtCacheSize()` method might return 0, *not* `STMT_CACHE_NOT_ENABLED`.

You *can* explicitly disable SQLJ/explicit statement caching or JDBC implicit caching, through JDBC connection APIs. Because SQLJ/explicit caching and JDBC implicit caching use the same cache size, there might sometimes be reason to do so. The following methods are available through the `OracleConnection` class:

- `void setExplicitCachingEnabled(boolean)`
- `boolean getExplicitCachingEnabled()`
- `void setImplicitCachingEnabled(boolean)`
- `boolean getImplicitCachingEnabled()`

You have access to these methods if you retrieve the `OracleConnection` instance from within a SQLJ connection context instance.

See Also: "SQLJ Connection Context and JDBC Connection Interoperability" on page 7-44

Note: In SQLJ, JDBC implicit caching is disabled by default and remains disabled unless you explicitly enable it through the `setImplicitCachingEnabled()` method.

Key Interactions Between SQLJ/Explicit Caching and JDBC Implicit Caching

With regard to statement caching in Oracle-specific code, this document naturally emphasizes SQLJ/explicit caching rather than JDBC implicit caching. If you do not use JDBC code in your application, SQLJ/explicit caching is the only statement caching that is relevant. However, there are situations where you might want to use both SQLJ and JDBC code in your application, and in these circumstances you might also want to use implicit caching.

SQLJ/explicit caching and JDBC implicit caching are enabled independently of each other. Furthermore, you do not have access to the implicit cache through SQLJ. However, there is a key interaction between the two, in that they share the same cache size. If, for example, the statement cache size is 5, then you can have a maximum of five statements cached for SQLJ/explicit caching and implicit caching combined.

An important point related to this is that if you choose to effectively disable SQLJ/explicit statement caching by setting the cache size to 0, then you have also effectively disabled implicit caching.

Also be aware that if SQLJ/explicit caching is disabled, changing the cache size to a value greater than 0 will enable it, but this does not affect whether implicit caching is enabled.

JDBC Support for Statement Caching (ISO Code)

With ISO standard code generation, specified through the SQLJ translator `-codegen=iso` setting, statement caching is a standard SQLJ feature that does not require any particular JDBC driver. However, using a driver that implements the `sqlj.runtime.profile.ref.ClientDataSupport` interface enables more robust caching. Oracle Database 11*g* JDBC drivers implement this interface, providing the following features:

- A separate cache for each database connection, instead of a single static cache for the entire application
- The ability to share cached statements between multiple instances of a connection context class that share the same underlying connection

When a single cache is used, as is the case with a generic JDBC driver that does not implement `ClientDataSupport`, a statement executed in one connection can cause a cached statement from another connection to be flushed (if the statement cache size, the maximum number of statements that can be cached, is exceeded).

Oracle Customizer Option for Statement Cache Size (ISO Code)

With ISO standard code generation, statement caching is enabled in your application by default with a cache size of 5 (the same default size as with Oracle-specific code) when you use the Oracle customizer, which is typically executed as part of Oracle SQLJ translation.

You can alter the statement cache size as desired, or effectively disable statement caching with a cache size of 0, through the Oracle customizer `stmtcache` option. This is set as `-P-Cstmtcache=`*n*, where *n* is an integer.

> **See Also:** "Oracle Customizer Statement Cache Size Option (stmtcache)" on page A-26

If you use multiple connection context classes and, therefore, have multiple profiles, you can set their statement cache sizes individually by running SQLJ (actually, the customizer) separately for each profile.

At run time, the appropriate SQLJ profile determines the statement cache size for a connection. This would be the profile that corresponds to the first connection context class instantiated for this connection. Its statement cache size setting, if any, is determined according to how you set the Oracle customizer `stmtcache` option when you customized the profile. The run-time statement cache size for a connection is set when the first statement on that connection is executed.

Additional Statement Caching Behavior

When a SQLJ connection context object is instantiated, if the statement cache size on the underlying JDBC connection is smaller than the default size for the connection context class, then the SQLJ run time will attempt to increase the JDBC statement cache size to the connection context default value. This manipulation occurs even with ISO code generation, enabling explicit statement caching in the process, although this is actually of no relevance in the ISO code case.

If, on the other hand, the actual JDBC statement cache size is larger, then the SQLJ run time will not attempt to perform a change in the cache size. The SQLJ run time checks the actual JDBC cache size against the default size set whenever it creates a SQLJ connection context instance.

It is important to note that these methods have the same effect regardless of the context class on which they are issued, because they modify or report the same underlying static field.

As an example, assume the following connection context class declarations:

```
#sql context CtxtA;
#sql context CtxtB;
```

In this case, each of the following three code instructions has the effect that whenever a new SQLJ connection context instance is subsequently created, it will *not* try to enable SQLJ/explicit statement caching:

```
sqlj.runtime.ref.DefaultContext.setDefaultStmtCacheSize(0);

CtxtA.setDefaultStmtCacheSize(0);

CtxtB.setDefaultStmtCacheSize(0);
```

> **Important:** If a SQLJ connection context instance is created on an underlying JDBC pooled connection, then SQLJ will not be able to change the JDBC statement cache size. For Oracle-specific code, you can retrieve the resulting exception through the connection context `getStmtCacheException()` method. In this case, the desired JDBC statement cache size must be set explicitly on the underlying physical connections. For data sources, the cache size is set through vendor-specific data source attributes.

SQLJ/explicit caching and JDBC implicit caching functionality have different semantics and behaviors. As noted earlier, SQLJ statement caching applies only to single statements used repeatedly, such as in a loop or through repeated calls to the same method. Consider the following example:

```
...
#sql { same SQL operaton }; // occurrence #1
...
Java code
```

```
...
#sql { same SQL operaton }; // occurrence #2
...
Java code
...
#sql { same SQL operaton }; // occurrence #3
...
```

Assume the three SQL operations are identical, including white space.

SQLJ caching would consider these three occurrences of the same SQL operation to be three different statements. They will occupy three separate slots in the cache. JDBC implicit caching, however, would recognize these as identical statements, using only a single cache slot for all three. The statement would be reused for occurrence #2 and occurrence #3.

Statement Caching Limitations and Notes

Using a statement cache, even of size 1, will improve the performance of almost any SQLJ application. Be aware of the following, however:

- There is no benefit if each statement is executed only once.
- Try to avoid interleaving statements executed once with statements executed multiple times. The statements being executed only once would needlessly take up space in the statement cache, which becomes an issue when you reach the statement cache size limit. As an alternative, if you use ISO code generation you can use a separate connection context class for statements that are executed only once and disable statement caching for that connection context class.
- Distinct statements with identical SQL operations are treated the same way as any distinct statements. Each is processed and cached separately. As an alternative, put the SQL operation in a method and call the method repeatedly, instead of using distinct statements.
- Be careful in choosing an appropriate statement cache size. If it is too small, then the cache might fill up resulting in statements being flushed before they are reexecuted. If it is too large, then database resources or program resources may be exhausted.

Also be aware of the following general notes regarding statement caching:

- With Oracle-specific code generation, using separate SQLJ connection context instances to have separate statement caching behavior will not work if the connection contexts share the same underlying JDBC connection instance. This is because under Oracle-specific code generation, SQLJ uses the JDBC statement cache.
- For Oracle applications, the statement cache size plus the maximum number of open JDBC statements in your application, both directly and through SQLJ, should be less than the maximum number of cursors available for a session. This is because the maximum number of cursors defines the maximum number of statements that can be open simultaneously.
- Using a statement cache generally does not change the execution semantics of an operation itself, although there are some scenarios where it does. For example, if you have a statement that throws an exception when its resources are released, then using a cache would mean that the exception would not be thrown until the connection is closed or the statement is flushed from the cache, which happens when the cache size is exceeded.

Update Batching

Update batching, referred to as batch updates in the Sun Microsystems JDBC 2.0 specification, allows `UPDATE`, `DELETE`, and `INSERT` statements that are batchable and compatible to be collected into a batch and sent to the database for execution at once, saving round trips to the database. This feature is included in the JDBC and SQLJ specifications and is supported by the Oracle JDBC and SQLJ implementations. Update batching is typically used for an operation that is executed repeatedly within a loop.

In SQLJ, update batching is tied to execution context usage. This feature is enabled or disabled in each execution context, independently of any other execution context, and each execution context instance maintains its own batch.

Note: Be aware of the following for update batching:

- You must use the default Oracle-specific code generation or, for ISO code generation, customize your application with the Oracle customizer.
- It is highly advisable to disable auto-commit mode. This gives you control of what to commit and what to roll back in case of an error during batch execution.

Batchable and Compatible Statements

Two criteria determine whether a statement can be added to an existing batch of statements:

- Is it batchable? You cannot batch some kinds of statements under any circumstances.
- Is it compatible with statements in the existing batch?

For SQLJ, the following kinds of statements are batchable:

- `UPDATE`
- `INSERT`
- `DELETE`

However `UPDATE` and `INSERT` statements with one or more stream host expressions are *not* batchable.

In SQLJ, only multiple instances of the same statement are compatible. This can occur in either of two circumstances:

- A statement is executed repeatedly in a loop.
- A statement is executed in a method, and the method is called repeatedly.

Enabling and Disabling Update Batching

SQLJ performs update batching separately for each execution context instance. Each one can have update batching enabled independently of your other execution context instances, and each maintains its own batch.

To enable or disable update batching for a particular execution context instance, use the `setBatching()` method of that execution context instance. This method takes boolean input, as follows:

```
...
ExecutionContext ec = new ExecutionContext();
```

```
ec.setBatching(true);
...
```

or:

```
...
ExecutionContext ec = new ExecutionContext();
ec.setBatching(false);
...
```

Update batching is disabled by default.

> **Note:** The `setBatching()` method does not affect an existing statement batch. Neither enabling nor disabling update batching causes an existing batch to be executed or canceled.

Use the `isBatching()` method of an execution context instance to determine if update batching is enabled for that execution context:

```
ExecutionContext ec = new ExecutionContext();
...
boolean batchingOn = ec.isBatching();
```

This does not, however, indicate whether a batch is currently pending.

Explicit and Implicit Batch Execution

You can explicitly execute a pending update batch as desired, but it might also be implicitly executed under certain circumstances.

> **Note:** It is important to be aware of what happens when an exception occurs in the middle of a batch execution. Refer to "Error Conditions During Batch Execution" on page 10-16.

Use the `executeBatch()` method of the execution context instance to explicitly execute an update batch. This method returns an `int` array of update counts.

Following is an example of explicitly executing a batch:

```
...
ExecutionContext ec = new ExecutionContext();
ec.setBatching(true);
...
double[] sals = ...;
String[] empnos = ...;
for (int i = 0; i < empnos.length; i++)
{
   #sql [ec] { UPDATE emp SET sal = :(sals[i]) WHERE empno = :(empnos[i]) };
}
int[] updateCounts = ec.executeBatch();
...
```

> **Note:** If you invoke `executeBatch()` when the execution context instance has no pending batch, then the method returns `null`.

When a pending update batch exists, it is implicitly executed in the following circumstances:

- An executable statement is encountered that is not batchable. In this case the existing batch is executed first, then the nonbatchable statement is executed.
- An update statement is encountered that is batchable, but is not compatible with the statements in the existing batch, in other words, is not an instance of the same statement. In this case the batch is executed, then a new batch is created, starting with the incompatible statement.
- A predefined batch limit, that is, a specified number of statements, is reached.

Following is an example. First one batch is created and executed implicitly when an unbatchable statement is encountered, then a new batch is created and executed implicitly when a batchable, but incompatible, statement is encountered:

```
ExecutionContext ec = new ExecutionContext();
ec.setBatching(true);
...
/* Statements in the following loop will be placed in a batch */
double[] sals = ...;
String[] empnos = ...;
for (int i = 0; i < empnos.length; i++)
{
   #sql [ec] { UPDATE emp SET sal = :(sals[i]) WHERE empno = :(empnos[i]) };
}

/* a SELECT is unbatchable so causes the batch to be executed */
double avg;
#sql [ec] { SELECT avg(sal) INTO :avg FROM emp };

/* Statements in the following loop will be placed in a new batch */
double[] comms = ...;
for (int i = 0; i < empnos.length; i++)
{
   #sql [ec] { UPDATE emp SET comm = :(comms[i]) WHERE empno = :(empnos[i]) };
}

/* the following update is incompatible with the second batch, so causes it to be
executed */
int smithdeptno = ...;
#sql [ec] { UPDATE emp SET deptno = :smithdeptno WHERE ename = 'Smith' };
```

To obtain the update count array for a batch executed implicitly, invoke the `getBatchUpdateCounts()` method of the execution context instance. This returns the update counts for the last batch to be executed successfully in this execution context instance. The following code statement could be inserted after the `SELECT` and after the last `UPDATE`:

```
int[] updateCounts = ec.getBatchUpdateCounts();
```

Note: If no update batch has been executed successfully for the execution context instance, then `getBatchUpdateCounts()` returns `null`.

See Also: "Execution Context Update Counts" on page 10-12

Canceling a Batch

To cancel the batch that is pending in an execution context, use the `cancel()` method of the execution context instance. You can, for example, cancel a batch that has been executed, but not yet committed, in the event that an exception occurred during batch execution. Following is an example:

```
...
ExecutionContext ec = new ExecutionContext();
ec.setBatching(true);
...
double[] sals = ...;
String[] empnos = ...;
for (int i = 0; i < empnos.length; i++)
{
   #sql [ec] { UPDATE emp SET sal = :(sals[i]) WHERE empno = :(empnos[i]) };
   if (!check(sals[i], empnos[i])) //assume "check" is a user-supplied function
   {
      ec.cancel();
      throw new SQLException("Process canceled.");
   }
}

try
{
   int[] updateCounts = ec.executeBatch();
} catch ( SQLException exception) { ec.cancel(); }
...
```

When you cancel a batch, the next batchable statement will start a new batch.

Note:

- Calling `cancel()` will also cancel any statement currently executing.
- Canceling a batch does *not* disable update batching.

Execution Context Update Counts

In the Oracle Database 11*g* SQLJ implementation, the array of update counts returned by the `executeBatch()` or `getBatchUpdateCounts()` method of an execution context instance does *not* contain counts of the number of rows updated by the batched statements, but simply values indicating whether each statement was successful. So its functionality differs from that of the single update count returned by the `getUpdateCount()` method of the execution context instance when batching is not enabled. As statements are batched, and after batch execution, the single update count returned by `getUpdateCount()` is also affected.

In a batch-enabled environment, the value available from the `getUpdateCount()` method of the execution context instance is modified after each statement is encountered. It will be updated with one of several `ExecutionContext` class static `int` constant values, as follows:

- `NEW_BATCH_COUNT`: Indicates that a new batch was created for the last statement encountered.
- `ADD_BATCH_COUNT`: Indicates that the last statement encountered was added to an existing batch.

- EXEC_BATCH_COUNT: Indicates that the pending batch was executed, either explicitly or implicitly, after the last statement was encountered.

If you want to refer to these constants, then use the following qualified names:

```
ExecutionContext.NEW_BATCH_COUNT
ExecutionContext.ADD_BATCH_COUNT
ExecutionContext.EXEC_BATCH_COUNT
```

After a batch has been executed, either explicitly or implicitly, the array of values returned by `executeBatch()` or `getBatchUpdateCounts()` indicates only whether the statements executed successfully. There is an array element for each batched statement. In accordance with the JDBC 2.0 specification, a value of `-2` for an array element indicates that the corresponding statement completed successfully, but that the number of rows it affected is unknown.

Checking all the array values after execution of a batch would not be meaningful. As currently implemented, the only useful test of this array would be to verify the number of statements that were in the batch prior to execution, by checking the number of elements in the array after a successful execution (essentially, after a batch execution that does not produce an exception).

Note that the update counts array is not modified as statements are batched, only as the batch is executed.

Setting a Batch Limit

You can specify that each update batch be executed after a predefined number of statements have been batched, before the next statement would be added. Use the `setBatchLimit()` method of the execution context instance, inputting a positive, nonzero integer as follows:

```
...
ExecutionContext ec = new ExecutionContext();
ec.setBatching(true);
ec.setBatchLimit(10);
...
double[] sals = ...;
String[] empnos = ...;
for (int i = 0; i < 20; i++)
{
   #sql [ec] { UPDATE emp1 SET sal = :(sals[i]) WHERE empno = :(empnos[i]) };
}
```

This loop is executed 20 times, with the statements being batched and the batch being executed during the 11th time through the loop, before the 11th statement would be added to the batch. Note that the batch would not be executed a second time in the loop, however. When your application exits the loop, the last ten statements would still be in the batch and would not be executed until another statement is encountered or you execute the batch explicitly.

You can use two special static `int` constants of the `ExecutionContext` class as input to the `setBatchLimit()` method:

- AUTO_BATCH: Enables the SQLJ run time to determine the batch limit.
- UNLIMITED_BATCH (default): Specifies that there is no batch limit.

For example:

```
...
ExecutionContext ec = new ExecutionContext();
```

```
ec.setBatching(true);
ec.setBatchLimit(ExecutionContext.AUTO_BATCH);
...
```

or:

```
ec.setBatchLimit(ExecutionContext.UNLIMITED_BATCH);
...
```

To check the current batch limit, use the `getBatchLimit()` method of the execution context instance.

Batching Incompatible Statements

If you want to batch a statement that is incompatible with statements in an existing batch, without implicitly executing the existing batch, then you will have to use a separate execution context instance. Following is an example:

```
...
ExecutionContext ec1 = new ExecutionContext();
ec1.setBatching(true);
ExecutionContext ec2 = new ExecutionContext();
ec2.setBatching(true);
...
double[] sals = ...;
String[] empnos = ...;
for (int i = 0; i < empnos.length; i++)
{
   #sql [ec1] { UPDATE emp1 SET sal = :(sals[i]) WHERE empno = :(empnos[i]) };
   #sql [ec2] { UPDATE emp2 SET sal = :(sals[i]) WHERE empno = :(empnos[i]) };
}
int[] updateCounts1 = ec1.executeBatch();
int[] updateCounts2 = ec2.executeBatch();
...
```

> **Note:** This example assumes that the two `UPDATE` statements are completely independent of each other. Do not batch interdependent statements in different execution contexts because you cannot completely assure the order in which they will be executed.

An alternative is to use a single execution context and separate loops so that all the `EMP1` updates are batched and executed prior to the `EMP2` updates:

```
...
ExecutionContext ec = new ExecutionContext();
ec.setBatching(true);
...
double[] sals = ...;
String[] empnos = ...;
for (int i = 0; i < empnos.length; i++)
{
   #sql [ec] { UPDATE emp1 SET sal = :(sals[i]) WHERE empno = :(empnos[i]) };
}
for (int i = 0; i < empnos.length; i++)
{
   #sql [ec] { UPDATE emp2 SET sal = :(sals[i]) WHERE empno = :(empnos[i]) };
}
ec.executeBatch();
```

```
...
```

This example executes the first batch implicitly and the second batch explicitly.

Using Implicit Execution Contexts for Update Batching

All the update batching examples so far have created and specified explicit execution context instances. This is not necessary, however, given that every connection context instance has an implicit execution context instance. For example, you can access the implicit execution context instance of the default connection as follows:

```
DefaultContext.getDefaultContext().getExecutionContext().setBatching(true);
...
double[] sals = ...;
String[] empnos = ...;
for (int i = 0; i < empnos.length; i++)
{
   #sql { UPDATE emp SET sal = :(sals[i]) WHERE empno = :(empnos[i]) };
}
// implicitly execute the batch and commit
#sql { COMMIT };
```

Or, you could execute the batch explicitly, as follows:

```
DefaultContext.getDefaultContext().getExecutionContext().executeBatch();
```

General Cautions Regarding Update Batching

If you use update batching, especially if you mix statements using an unbatched execution context instance with statements using a batched execution context instance, then remember the following points:

- If an unbatched statement depends on a batched statement, then be sure the batch is executed prior to the unbatched statement.
- A JDBC `COMMIT` or `ROLLBACK` operation, that is, an auto-commit or any explicit use of the `commit()` or `rollback()` method of a JDBC `Connection` instance, does not execute pending statements in a batch.

  It is important to note, however, that using a SQLJ `COMMIT` or `ROLLBACK` statement, such as follows, *will* execute pending statements in a batch:

  ```
  #sql { COMMIT };
  ```

  or:

  ```
  #sql { ROLLBACK };
  ```

  This is another reason that you should always commit or roll back changes using `#sql` syntax, which cleans up both SQLJ resources and JDBC resources.
- When a batch is implicitly executed as a result of an unbatchable or incompatible statement being encountered, the batch is executed *before* the unbatchable or incompatible statement is executed, but *after* the input parameters of that statement have been evaluated and passed to the statement.
- If you no longer intend to use a particular batch-enabled execution context instance, then explicitly execute or cancel its pending batch to free resources.

Error Conditions During Batch Execution

In the event that a statement causes an exception in the middle of a batch execution, be aware of the following:

- Batched statements following the statement that caused the exception are *not* executed.
- Batched statements that had already been executed prior to the exception are *not* rolled back.
- If the batch where the exception occurred was executed implicitly as the result of another (unbatchable or incompatible) statement being encountered, that statement is *not* executed.

> **Note:** Presumably you have disabled auto-commit mode when using update batching. This gives you commit/rollback control in case of an error during batch execution.

When an exception occurs during batch execution under JDBC 2.0 or later, it is typically an instance of the standard `java.sql.BatchUpdateException` class, a subclass of the `java.sql.SQLException` class. The `BatchUpdateException` class has a `getUpdateCounts()` method that, for batched statements successfully executed before the exception occurred, returns an array of update counts equivalent to what would be returned by the `executeBatch()` or `getBatchUpdateCounts()` method of the `ExecutionContext` class.

Recursive Call-ins and Update Batching

Execution of SQLJ stored procedures, where one calls the other, can result in situations where the two procedures are simultaneously using the same execution context instance. The update-batching flag, set using the `setBatching()` method of the execution context instance, would act in the same way as other execution context attributes. Regardless of which stored procedure sets it, it would affect the next executable statement in either stored procedure.

For this reason, update batching is automatically disabled in the server whenever a recursive call-in occurs. The pending batch is executed, and no batching occurs in the recursively invoked procedure. To avoid this behavior, use explicit execution context instances in batch-enabled stored procedures.

See Also: "Recursive SQLJ Calls in the Server" on page 11-18

Column Definitions

The Oracle SQLJ implementation reflects Oracle JDBC support for column type and size definitions. Depending on the driver implementation, which differs somewhat among the different Oracle JDBC drivers, registering column types and sizes can save a trip to the database for each query. In particular, this is true for the Oracle JDBC Thin driver and use of positional iterators.

Oracle Implementation of Column Definitions

If you enable column definitions, then the Oracle SQLJ implementation takes the following steps to automatically register column types and sizes:

- During customization or during translation when the default Oracle-specific code generation is used, SQLJ connects to a specified database schema to determine types and sizes of columns being retrieved. With ISO standard SQLJ code

generation, the column defaults become part of the SQLJ profile. This can be accomplished during the customization step of source code translation or during separate customization of an existing profile.

- When your application executes, the SQLJ run time will use the column information to register the column types and sizes with the JDBC driver, using a call to the `defineColumnType()` method available in the Oracle JDBC statement classes.

Customizer and Translator Options for Column Definitions

To enable column definitions, set SQLJ options as follows:

- Enable the `optcols` flag. For Oracle-specific code generation, use the SQLJ translator `-optcols` option. For ISO standard code generation, use either the translator option or the Oracle customizer option (`-P-Coptcols` on the SQLJ command line).
- Set the user, password, and URL for a database connection. For Oracle-specific code generation, this is through the SQLJ translator `-user`, `-password`, and `-url` options. For ISO standard code generation, this can be through the translator options or you can separately use the customizer options (`-P-user`, `-P-password`, and `-P-url` on the SQLJ command line). In addition, set the JDBC driver class (`-P-driver` on the SQLJ command line) if you are not using the default `OracleDriver` class.

> **See Also:** "Column Definitions (-optcols)" on page 8-41 and "Connection Options" on page 8-25

For information about the customizer options, refer to the `optcols` section under "Overview of Customizer-Specific Options" on page A-18, and the `user`, `password`, `url`, and `driver` sections under "Overview of Customizer Harness Options" on page A-7.

Parameter Size Definitions

The Oracle JDBC and SQLJ implementations enable you to optimize JDBC resource allocation by defining parameter sizes (sizes of Java host variables) used as any of the following:

- Input or output parameters in stored procedure or function calls
- Return values from stored function calls
- Input or output parameters in SET statements
- Input or output parameters in PL/SQL blocks

Oracle Implementation of Parameter Size Definitions

Oracle implements parameter size definitions through option settings, in combination with hints embedded in source code comments. For ISO standard SQLJ code generation, Oracle customizer options are available. For the default Oracle-specific code generation, equivalent SQLJ translator options are available.

Use options and hints as follows:

- Enable parameter size definitions through the SQLJ translator or Oracle customizer parameter definition flag.
- Specify default sizes for particular data types through the SQLJ translator or Oracle customizer parameter default size option.

- Override data type default sizes for particular parameters by embedding hints in source code comments, following a prescribed format.

For any given host variable, when parameter size definitions are enabled, resources are allocated according to the source code hint if there is one. If there is no source code hint, then the default size for the corresponding data type is used if one was specified. If there is no source code hint or appropriate default size, then maximum resources are allocated according to the JDBC implementation.

When your application executes, the parameter sizes are registered through calls to the `defineParameterType()` and `registerOutParameter()` methods available in the Oracle JDBC statement classes.

> **Note:** If you do not enable the parameter definition flag, then parameter size defaults and source code hints will be ignored and maximum or default resources will be allocated according to the JDBC implementation.

Customizer and Translator Options for Parameter Size Definitions

Use the following SQLJ options for parameter size definitions:

- Use the `optparams` flag to enable parameter size definitions. For Oracle-specific code generation, use the SQLJ translator `-optparams` option. For ISO standard code generation, use either the translator option or the Oracle customizer option, `-P-Coptparams` on the SQLJ command line.
- Use `optparamdefaults` to set default sizes for particular data types. For Oracle-specific code generation, use the SQLJ translator `-optparamdefaults=`*`xxxx`* option. For ISO standard code generation, use either the translator option or the Oracle customizer option, `-P-Coptparamdefaults=`*`xxxx`* on the SQLJ command line.

  See Also:

  - "Parameter Definitions (-optparams)" on page 8-43
  - "Parameter Default Size (-optparamdefaults)" on page 8-44
  - "Overview of Customizer-Specific Options" on page A-18

Source Code Hints for Parameter Size Definitions

Embed source code hints for parameter size definitions within your SQLJ statements in the following format (you can add white space within the comment, as desired):

```
/*(size)*/
```

The size is in bytes. Hints are ignored if the `optparams` flag is disabled.

You can override the default parameter size, without specifying a new size (leaving size allocation to the JDBC implementation), as follows:

```
/*()*/
```

Here is an example:

```
byte[] hash;
String name=Tyrone;
String street=2020 Meryl Street;
String city=Wichita;
String state=Kansas;
```

```
String zipcode=77777;
#sql hash = { /* (5) */ VALUES (ADDR_HASH(:name /* (20) */, :street /* () */,
                                :city, :state, :INOUT zipcode /* (10) */ )) };
```

A hint for a result expression, such as the result expression `hash` in the example, must be the first item appearing inside the brackets of the SQLJ statement, as shown. Hints for input and output host variables must immediately follow the variables, as shown.

The example sets parameter sizes as follows:

- `hash`: 5 bytes
- `name`: 20 bytes
- `street`: override default, but with no setting (leave allocation up to JDBC)
- `city`: none (use appropriate data type default, if any)
- `state`: none (use appropriate data type default, if any)
- `zipcode`: 10 bytes

> **Note:** If any parameter size is altered such that its actual size exceeds its registered size at run time, then a SQL exception will be thrown.

SQLJ Debugging Features

This section summarizes debugging features in the Oracle SQLJ implementation. It covers the following topics:

- SQLJ -linemap Flag for Debugging
- Server-Side debug Option
- Introduction to the AuditorInstaller Specialized Customizer
- Introduction to Developing and Debugging in Oracle10g JDeveloper

SQLJ -linemap Flag for Debugging

The `-linemap` flag instructs SQLJ to map line numbers from a SQLJ source code file to locations in the corresponding `.class` file. This will be the `.class` file created during compilation of the `.java` file generated by the SQLJ translator. As a result of this, when Java run-time errors occur, the line number reported by the Java virtual machine (JVM) is the line number in the SQLJ source code, making it much easier to debug.

If you are using the Sun Microsystems `jdb` debugger, then use the `-jdblinemap` option instead of the `-linemap` option. The options are equivalent, except that `-jdblinemap` does some special processing, necessitated by the fact that `jdb` does not support Java source files with file name extensions other than the `.java` extension.

> **See Also:** "Line-Mapping to SQLJ Source File (-linemap)" on page 8-36 and "Line-Mapping to SQLJ Source File for jdb Debugger (-jdblinemap)" on page 8-37

Note: If you are translating in the server, then class schema objects created during server-side translation automatically reference line numbers that map to the SQLJ source code. This is equivalent to enabling the `-linemap` option when you translate on a client.

Server-Side debug Option

If you are loading SQLJ source into the server and using the server-side embedded translator to translate it, then the server-side `debug` option instructs the server-side compiler to output debugging information when a `.sqlj` or `.java` source file is compiled in the server. This is equivalent to using the `-g` option when running the standard `javac` compiler on a client. This does not aid in debugging your SQLJ code in particular, but aids in debugging your Java code in general.

Refer to "Option Support in the Server Embedded Translator" on page 11-12 for more information about this option and information about how to set options in the server.

For general information about debugging in Oracle JVM, refer to the *Oracle Database Java Developer's Guide*.

Introduction to the AuditorInstaller Specialized Customizer

For ISO code generation, SQLJ provides a special customizer, `AuditorInstaller`. This customizer will insert sets of debugging statements, known as auditors, into profiles specified on the SQLJ command line. These profiles must already exist from previous customization. The debugging statements will execute during SQLJ run time (when someone runs your application), displaying a trace of method calls and values returned.

Use the customizer harness `debug` option, preceded by `-P-` as with any general customization option, to insert the debugging statements.

See Also: "AuditorInstaller Customizer for Debugging" on page A-33

Introduction to Developing and Debugging in Oracle10*g* JDeveloper

The Oracle SQLJ product is fully integrated into the Oracle10*g* JDeveloper visual programming tool.

JDeveloper also includes an integrated debugger that supports SQLJ. SQLJ statements, as with standard Java statements, can be debugged in-line as your application executes. Reported line numbers are according to the line numbers in your SQLJ source code (as opposed to in the generated Java code).

SQLJ Support for Oracle Performance Monitoring

The following sections discuss Oracle SQLJ implementation support for Oracle Dynamic Monitoring Service (DMS):

- Overview of SQLJ DMS Support
- Summary of SQLJ Command-Line Options for DMS
- SQLJ Run Time Commands and Properties File Settings for DMS
- SQLJ DMS Sensors and Metrics
- SQLJ DMS Examples

Note: *Oracle Application Server 10g Performance Guide*

Overview of SQLJ DMS Support

DMS enables users to measure performance statistics for SQLJ programs. SQLJ support for DMS focuses on the overall performance per SQL statement, such as its execution time, but can also provide method-level or class-level performance information, such as with Oracle JDBC support for DMS. You can choose a client-side perspective, such as the overall performance of each `#sql` statement, a server-side perspective, such as server-side tracing of each SQL operation, or both.

Instrumenting a program, which is specified at translation time through SQLJ options, is required in order to enable DMS setup. Specifically, instrumenting is the process of inserting DMS calls into system or application code for measuring its performance.

At run time, any components that were instrumented during translation can be monitored during execution, according to instructions in a SQLJ DMS properties file. During run time, statistics are sent to DMS through DMS APIs. This requires a running DMS system in your environment. You can then access the statistics through DMS tools.

The statistics are intended to help you track and understand SQL statement performance and are reported according to the following hierarchy (from top to bottom):

1. Application: The application, in this context, is defined to consist of the SQLJ and Java components specified in the SQLJ command line for translation. However, only the SQLJ components can be instrumented.
2. Module: A module corresponds to a Java package.
3. Action: An action maps to a Java class defined in a SQLJ program.
4. Statement: A statement is a SQL statement in a SQLJ program.

The following DMS statistics are measured for client-side monitoring:

- Elapsed time for each `#sql` statement, including parsing and execution
- Get-next time, the time to execute each `next()` call
- Get-XXX time, the time to extract a database column through each `get`*`XXX`*`()` call

These statistics require the DMS library to be in your classpath at both translation time and run time, so are not supported on the server, where the DMS library is not available. Server-side SQLJ code cannot be monitored in the way that client-side code can.

The following statistics are measured for server-side SQL monitoring of your SQLJ client program:

- Parsing time
- Execution time
- Fetching time

These statistics are available from the Oracle Database 11*g* trace file, through SQL tracing functionality. This is independent of DMS, but you can enable it through the SQLJ DMS properties file `sqlmonitor.servertracing` setting.

See Also: "SQLJ Run Time Commands and Properties File Settings for DMS" on page 10-23

For a client-side SQLJ program, you can use both DMS statistics and server-side tracing. For example, from DMS you can get the total time required for a `#sql` statement that consists of a query, then from server-side tracing you can find out how much of that time was actually spent executing the SQL query in the server.

Note:

- DMS support currently requires Oracle-specific code generation, which is enabled by default.
- In Oracle Database 11*g*, instrumented code requires Java Development Kit (JDK) 1.5.
- Only SQLJ declarations and statements are instrumented.
- The DMS library is in the file `dms.jar`, in *ORACLE_HOME*`/oc4j/lib` in Oracle Database 11*g*

Summary of SQLJ Command-Line Options for DMS

The Oracle SQLJ implementation provides following translator front-end options to support DMS:

- `-instrument`: Enable instrumentation and designate a name for the application (the collective of the components being translated).

  See Also: "Instrumentation for DMS (-instrument)" on page 8-38

- `-components`: Specify the components (packages and classes) to be instrumented.

  See Also: "Components to Instrument for DMS (-components)" on page 8-39

Typically you would enable instrumentation by specifying a desired application name in the `-instrument` setting, optionally specifying a package as well. Or specify a setting of `true` to use the default application, `defaultApp`. For DMS instrumentation, the term application refers to all the SQLJ and Java components specified for translation in the SQLJ command line.

If instrumentation is enabled, a SQLJ DMS properties file is created according to the `-instrument` setting, starting from the current directory, and also according to any setting of the SQLJ `-d` option. For a setting of `true`, the properties file is named `sqlmonitor.properties` in the current directory.

Note: A setting of `-instrument` is equivalent to `-instrument=true`. A setting of `-instrument=false` (the default) disables instrumentation.

As a simple example, a setting of `-instrument=myapp` will result in creation of the properties file `myapp.properties`. Now consider the following example, for an application name of `stock` and a package name of `com.acme`:

```
% sqlj -instrument=com.acme/stock -d /home Stock.sqlj Trading.sqlj
```

Because of the `-d` option, the `/home/com/acme/stock.properties` file is created.

When instrumentation is enabled through the `-instrument` option, use the `-components` option to specify the subset of translated components to be instrumented for DMS monitoring, typically most or all of them to allow flexibility in what you can monitor during run time. Specify a comma-delimited list of packages (to instrument all classes in each package) or specific classes, or use the default `all` setting to instrument all components being translated.

For example, to instrument the classes `Stock` and `Trading`:

```
% sqlj ... -components=com.acme.Stock,com.acme.Trading
```

At run time, instrumented components are monitored according to what is specified in the SQLJ DMS properties file. Any components that are not instrumented during translation cannot be monitored during run time, regardless of what is specified in the properties file.

SQLJ Run Time Commands and Properties File Settings for DMS

While the SQLJ `-instrument` option specifies whether the SQLJ translator instruments files for monitoring capability, it is the SQLJ DMS properties file that actually determines what is monitored and how, at run time.

This properties file is created by SQLJ during translation, and then you can modify it as desired. Be aware that if you run SQLJ again, however, SQLJ overwrites the properties file. Any changes that you made are lost.

Settings in the SQLJ DMS properties file are as follows:

- `sqlmonitor.components`: This is a comma-delimited list of components (packages or classes) that have been instrumented. This is set automatically by the translator to reflect the setting of the SQLJ `-components` option.
- `sqlmonitor.monitorcomp`: This is a comma-delimited list of components (packages or classes) to be monitored and denotes a subset of the components in the `sqlmonitor.components` setting. The setting for `sqlmonitor.monitorcomp` is initially determined during translation to reflect the `sqlmonitor.components` setting, but you can then adjust it as desired. A setting of `all` means to monitor all components listed in the `sqlmonitor.components` setting.
- `sqlmonitor.dms`: This boolean flag, with a default value of `true`, specifies whether to deliver collected statistics to DMS. This requires a running Oracle Application Server 10*g* instance where you can use DMS tools. Statistics can be accessed through a Web browser or written into a file.

  Note: A setting of `sqlmonitor.dms=false` is not currently supported.

- `sqlmonitor.sysurl`: For server-side tracing, this specifies the database URL.
- `sqlmonitor.sysuser`: For server-side tracing, this specifies the database user. This user must have `sysdba` privileges.
- `sqlmonitor.syspassword`: For server-side tracing, this specifies the password for the `sysuser`.

> **Note:** For `sysurl`, `sysuser`, and `syspassword`, default values are according to the `user`, `password`, and `url` values supplied to SQLJ, either through the SQLJ command line or through the SQLJ properties file.

- `sqlmonitor.servertracing`: Use this to enable server-side tracing, to collect performance statistics in the server, such as for SQL operations. Supported settings are `true` or `false` (the default).
- `sqlmonitor.dumpfile`: If delivering statistics to DMS, then you can use this option to specify a file into which the DMS tool writes the statistics. The default is *`application_name`*`.mtr`, where *`application_name`* is according to the `-instrument` option setting (or is `defaultApp` by default).

SQLJ DMS Sensors and Metrics

Sensors are used by DMS to calculate performance metrics during the execution of instrumented SQLJ programs and delivered to DMS. They are organized as a hierarchy, with each sensor having a path name. Here are typical sensor formats:

```
/SQLJ/application_name/sensor_name
/SQLJ/application_name/module/sensor_name
/SQLJ/application_name/module/class/sensor_name
/SQLJ/application_name/module/class/linenum/sensor_name
```

A sensor is an instance of the `oracle.dms.instrument.Sensor` class, which has methods for calculating and organizing performance statistics. For example, there are methods to instruct the sensor to derive additional metrics and to get the value of one of the metrics.

Be aware that before the end of an instrumented application, there must be a call to the `close()` method of the `oracle.sqlj.runtime.sqlmonitor.SQLMonitor` class, such as in the following example (which also uses the `Oracle` class `close()` method to close the connection context):

```
try
{
   Oracle.close();
   oracle.sqlj.runtime.sqlmonitor.SQLMonitor.close();
}
catch( Throwable e ) { ... }
```

Note the following terms:

- The *`application_name`* is the name of the application according to the SQLJ `-instrument` option, or `defaultApp` by default.
- If a sensor is associated with a package, then the item *`module`* is the package name. The setting `*TopLevel*` is used if the package name is empty.
- If a sensor is associated with a class, then *`class`* is the class name.
- If a sensor is associated with a SQL statement, then *`linenum`* denotes the line number of the SQL statement in the SQLJ program being instrumented. If multiple SQL statements appear in the same line, then their starting column positions are used to distinguish them. For example, a *`linenum`* value of 8.13 indicates that 8 is the line number and 13 is the column number.

The following sensors and associated metrics are typically of particular interest:

- Sensor name: `ContextType`

  `/SQLJ/application_name/module/class/linenum/ContextType`

  Metrics:

  – `value`: A string indicating the connection context type

- Sensor name: `SQLString`

  `/SQLJ/application_name/module/class/linenum/SQLString`

  Metrics:

  – `value`: A string consisting of the SQL statement

    This is the exact string that is passed to JDBC, including any transformations made from the original `#sql` statement.

- Sensor name: `Execute`

  `/SQLJ/application_name/module/class/linenum/Execute`

  Metrics:

  – `time`: The total time, in milliseconds, of all the executions of the JDBC `execute()` method for this statement

    If the statement executes five times, for example, then `time` would be the total time spent in the `execute()` method for the five executions.

  – `completed`: The number of executions completed (such as 5)

  – `minTime`: The shortest time of any single execution

  – `maxTime`: The longest time of any single execution

  – `avg`: The average execution time, which is `time` divided by `completed`

  – `active`: The number of threads executing the statement at the end of program execution, typically 0.

  – `maxActive`: The maximum number of threads that executed the statement during program execution

  To measure the execution time of a JDBC statement, the clock is started immediately before the statement is executed and stopped when a result set is obtained or the statement otherwise finishes executing, or when an exception is caught.

- Sensor name: `ServerExecute`

  `/SQLJ/application_name/module/class/linenum/ServerExecute`

  Metrics:

  – `value`: The total execution time in the server, in milliseconds, for all executions of this SQL statement

  – `count`: The number of executions completed

  – `minValue`: The shortest time of any single execution

  – `maxValue`: The longest time of any single execution

- Sensor name: `ServerFetch`

  `/SQLJ/application_name/module/class/linenum/ServerFetch`

Metrics:

- `value`: The total fetch time in the server, in milliseconds, for all executions of this SQL statement
- `count`: The number of executions completed
- `minValue`: The shortest time of any single execution
- `maxValue`: The longest time of any single execution

- Sensor name: `ServerParse`

  `/SQLJ/`*`application_name`*`/`*`module`*`/`*`class`*`/`*`linenum`*`/ServerParse`

  Metrics:

  - `value`: The total time spent parsing the SQL statement in the server, in milliseconds, for all executions of this SQL statement
  - `count`: The number of executions completed
  - `minValue`: The shortest time of any single execution
  - `maxValue`: The longest time of any single execution

- Sensor name: `Next`

  `/SQLJ/`*`application_name`*`/`*`module`*`/`*`class`*`/`*`linenum`*`/Next`

  Metrics:

  - `time`: The total time, in milliseconds, spent in the `next()` method of the result set iterator for all executions of this SQL statement
  - `completed`: The number of executions completed (such as 5)
  - `minTime`: The shortest time of any single execution
  - `maxTime`: The longest time of any single execution
  - `avg`: The average execution time, which is `time` divided by `count`.
  - `active`: The number of threads executing the statement at the end of program execution, typically 0.
  - `maxActive`: The maximum number of threads that executed the statement during program execution

SQLJ DMS Examples

Following is a sample command line (a single wraparound line) to instrument the SQLJ program `ExprDemo.sqlj`:

```
% sqlj -dir=. -instrument=a.b.c/app -components=all
       -user=scott/tiger -url=jdbc:oracle:oci:@ ExprDemo.sqlj
```

> **Note:** Ensure that `dms.jar` is in your classpath.

This command results in generation of the following files:

- `./a/b/c/ExprDemo.java` (due to the `-dir` option setting and because package `a.b.c` is declared in `ExprDemo.sqlj`)
- `./a/b/c/app.properties` (due to the `-instrument` option setting)

Sample SQLJ DMS Properties File

The following is sample content for `app.properties`. This assumes you edited the file after SQLJ created it, given that some of the settings here are nondefault.

```
sqlmonitor.components=all
sqlmonitor.monitorcomp=all
sqlmonitor.dms=true
sqlmonitor.servertracing=true
sqlmonitor.sysurl=jdbc:oracle:oci:@
sqlmonitor.sysuser=scott
sqlmonitor.syspassword=tiger
sqlmonitor.dumpfile=a/b/c/app.mtr
```

Note: If you run the SQLJ translator again, then `app.properties` is overwritten and you will lose any changes you made.

Sample Statistics

The `sqlmonitor.dms=true` setting specifies that monitoring statistics are to be delivered to DMS. Given the `sqlmonitor.dumpfile` value, the DMS tool writes the statistics to the `./a/b/c/app.mtr` file when you compile and run the program.

To examine statistics for a particular code sample, here is a segment of `ExprDemo.sqlj`:

```
#sql
{
  DECLARE
    n NUMBER;
    s NUMBER;
  BEGIN
    n := 0;
    s := 0;
    WHILE n < 100 LOOP
      n := n + 1;
      s := s + :IN (indx++);
    END LOOP;
    :OUT total := s;
  END;
};
```

And here is a segment of statistics from `app.mtr`, relating to the preceding code example and showing the execution time and server execution times:

```
SQLString.value:      DECLARE         n NUMBER;        s NUMBER;
BEGIN         n :=  0;         s := 0;       WHILE n < 100 LOOP
      n := n + 1;
      s := s +  :1 ;
END LOOP;          :2  := s;       END; statement SQL string
     ServerExecute.maxValue:       20.0 server_execute_time
     ServerExecute.minValue:       20.0 server_execute_time
     ServerExecute.count:  0 ops
     ServerExecute.value:  20.0 server execute time
     ServerFetch.maxValue: 0.0 server_fetch_time
     ServerFetch.minValue: 0.0 server_fetch_time
     ServerFetch.count:    0 ops
     ServerFetch.value:    0.0 server fetch time
     ServerParse.maxValue: 0.0 server_parse_time
```

```
     ServerParse.minValue: 0.0 server_parse_time
     ServerParse.count:    0 ops
     ServerParse.value:    0.0 server parse time
    193.5
     ContextType.value:    class sqlj.runtime.ref.DefaultContext
statement connection context
     Execute.maxActive:    1 threads
     Execute.active:       0 threads
     Execute.avg:  37.0 msecs
     Execute.maxTime:      37 msecs
     Execute.minTime:      37 msecs
     Execute.completed:    1 ops
     Execute.time: 37 msecs
```

These statistics indicate that the total execution time at the JDBC client was 37 milliseconds (in one execution), while the execution time in the server was 20 milliseconds.

Sample Statistics for Iterators

`ExprDemo.sqlj` also defines and executes an iterator type, `Iter`, as follows:

```
#sql public static iterator Iter(String ename);

 ....

Iter iter;
#sql iter = { select ename from emp};
while (iter.next())
{
 System.out.println(iter.ename());
}
```

For iterators, DMS collects the execution time for the `next()` operation. Here is a sample DMS result for the iterator type `Iter`:

```
Iter
 Next.time:    5 msecs
```

This shows that the total time spent on the `next()` operation while iterating through the `Iter` instance was 5 milliseconds.

Sample Statistics for Connection Contexts

The `#sql` statements in `ExprDemo.sqlj` use the default connection context. For the `DefaultContext` instance used throughout the program, DMS returns the following statistics:

```
class_sqlj.runtime.ref.DefaultContext
StmtCacheSize.value:  5 statement cache size
StmtsExecuted.count:  7 ops
StmtsCacheExecuted.count:     7 ops
```

This shows that the context has a statement cache size of five statements. Altogether, seven SQL statements are executed.

11

SQLJ in the Server

SQLJ applications can be stored and run directly in the Oracle Database 11*g* server. You have the option of either translating and compiling them on a client and loading the generated classes and resources into the server or loading SQLJ source code into the server and having it translated and compiled by the embedded translator of the server.

This chapter discusses features and usage of SQLJ in the server, including additional considerations, such as multithreading and recursive SQLJ calls.

The following topics are discussed:

Introduction to Server-Side SQLJ

SQLJ code, as with any Java code, can run in Oracle Database 11*g* in stored procedures, stored functions, or triggers. Data access is through a server-side implementation of the SQLJ run time in combination with the Oracle Java Database Connectivity (JDBC) server-side internal driver. In addition, an embedded SQLJ translator in Oracle Database 11*g* is available to translate SQLJ source files directly in the server.

Considerations for running SQLJ in the server include several server-side coding issues as well as decisions about where to translate your code and how to load it into the server. You must also be aware of how the server determines the names of generated output. You can either translate and compile on a client and load the class and resource files into the server or you can load `.sqlj` source files into the server and have the files automatically translated by the embedded SQLJ translator.

The embedded translator has a different user interface than the client-side translator. Supported options can be specified using a database table, and error output is to a database table. Output files from the translator are transparent to the developer.

Note:

- In Oracle Database 11*g* release 1 (11.1), the server uses a Java Development Kit (JDK) 1.5 Java2 Platform, Standard Edition (J2SE) environment. The server-side SQLJ environment is roughly equivalent to a client-side environment using the `runtime12ee` library, except for SQLJ-specific connection bean support and considering any relevant exceptions noted in "Creating SQLJ Code for Use in the Server" on page 11-2.
- This manual presumes that system configuration issues are outside the duties of most SQLJ developers. Therefore, configuration of the Oracle Database 11*g* Java virtual machine (JVM) is not covered here. For information about setting Java-related configuration parameters, refer to the *Oracle Database Java Developer's Guide*. If you need information about configuring the multithreaded server, dispatcher, or listener, refer to the *Oracle Database Net Services Administrator's Guide*.

Note Regarding Desupport of J2EE in Oracle Database

Since the introduction of Oracle9*i* Application Server Containers for J2EE (OC4J), a new, lighter-weight, easier-to-use, faster, and certified Java2 Platform, Enterprise Edition (J2EE) container, Oracle has desupported the J2EE and Common Object Request Broker Architecture (CORBA) stacks from the database. However, Oracle JVM will still be present and will continue to be enhanced to offer J2SE features, Java stored procedures, JDBC, and SQLJ in the database.

To summarize, Oracle no longer supports the following technologies in the database:

- The J2EE stack, consisting of Enterprise Beans (EJB) container, JavaServer Pages (JSP) container, and Oracle9*i* Servlet Engine (OSE)
- The embedded CORBA framework, based on Visibroker for Java

Customers can no longer deploy servlets, JSP pages, EJBs, and CORBA objects in Oracle Databases. Oracle9*i* Database Release 1 (9.0.1) was the last database release to support the J2EE and CORBA stacks. Oracle encourages customers to migrate to OC4J for J2EE applications that previously ran in the database.

Creating SQLJ Code for Use in the Server

With few exceptions, writing SQLJ code for use within the target Oracle Database 11*g* instance is identical to writing SQLJ code for client-side use. The few differences are due to Oracle JDBC characteristics or general Java characteristics in the server, rather than being specific to SQLJ. There are a few considerations to be aware of, however:

- There is an implicit connection to the server itself.
- There are coding issues, such as lack of auto-commit functionality.
- In the server, the default output device is the current trace file.
- Name resolution functions differently in the server than on a client.
- SQL names must be interpreted and processed differently from Java names.
- There is no JSP, EJB, or CORBA functionality in the server. Because there is no JSP container, you cannot use the SQLJ JSP connection beans in server-side code.

> **Note:** Writing SQLJ code to connect from one server to another through the server-side Thin driver is identical to writing code for an application that uses a client-side JDBC Thin driver. The points in this discussion do not apply.

This section covers the following topics:

Database Connections Within the Server

The concept of connecting to a server is different when your SQLJ code is running within the server itself. There is no explicit database connection. By default, an implicit channel to the database is used for any Java program running in the server. You do not have to initialize this connection, as it is automatically initialized for SQLJ programs. You do not have to register or specify a driver, create a connection instance, specify a default connection context, specify any connection objects for any of your `#sql` statements, or close the connection.

> **Note:** In the server, setting the default connection context to `null`, as follows, will reinstall the default connection context (the implicit connection to the server):
>
> ```
> DefaultContext.setDefaultContext(null);
> ```

Coding Issues Within the Server

There are a few coding issues to consider when your code will run within the target server using the server-side internal driver. Note the following:

- Result sets issued by the internal driver persist across calls, and their finalizers do not release their cursors. Because of this, it is especially important to close all iterators to avoid running out of available cursors, unless you have a particular reason for keeping an iterator open, such as when it is actually used across calls.
- The internal driver does not support auto-commit functionality, so the auto-commit setting is ignored within the server. Use explicit `COMMIT` or `ROLLBACK` statements to implement or cancel your data updates:

```
#sql { COMMIT };
...
#sql { ROLLBACK };
```

> **Note:** If you are using any kind of XA transactions, such as Java Transaction Service (JTS) transactions, you cannot use SQLJ or JDBC `COMMIT/ROLLBACK` statements or methods.

- For ISO standard code generation, if you use SQLJ code that interacts with JDBC code and you use a nondefault connection context instance, then you must eventually close the connection context instance in order to clean up statements cached there, unless you use the same connection context instance for the duration of your session. Following is an example:

  ```
  DefaultContext ctx = new DefaultContext(conn); // conn is JDBC connection
  #sql [ctx] { SQL operation };
  ...
  ctx.close(sqlj.runtime.ConnectionContext.KEEP_CONNECTION);
  ...
  ```

  If you do not close the connection context instance, you are likely to run out of statement handles in your session. Also be aware that simply closing the underlying JDBC connection object does *not* reclaim statement handles, which differs from the behavior when the application executes on a client.

  For the default Oracle-specific code generation, statements are cached in the underlying JDBC statement cache and can be automatically reclaimed.

- With Oracle-specific code generation for code that will run in the server, use an explicit `ExecutionContext` instance. This ensures that your application can fully interoperate with applications translated with ISO standard SQLJ code generation.

  If you use one thread per connection, which translates to one thread per Oracle session, it is sufficient to use one static instance, as in the following example:

  ```
  public static ExecutionContext ec = new ExecutionContext();
  ...
  #sql [ec] { SQL operation };    // use ec for all operations
  ```

  If you use multiple threads per connection, then you must use a separate execution context instance for each method invocation.

See Also: *Oracle Database JDBC Developer's Guide and Reference*

Default Output Device in the Server

The default standard output device in the Oracle Java virtual machine (JVM) is the current trace file. If you want to reroute all standard output from a program executing in the server, for example, output from any `System.out.println()` calls, to a user screen, then you can execute the `SET_OUTPUT()` procedure of the `DBMS_JAVA` package as in the following example. Input the buffer size in bytes (10,000 bytes in this case).

```
sqlplus> execute dbms_java.set_output(10000);
```

Output exceeding the buffer size will be lost.

If you want your code executing in the server to expressly write output to the user screen, then you can also use the PL/SQL `DBMS_OUTPUT.PUT_LINE()` procedure instead of the Java `System.out.println()` method. The `PUT_LINE()` procedure is overloaded, accepting either `VARCHAR2`, `NUMBER`, or `DATE` as input to specify what is printed.

See Also: *Oracle Database PL/SQL Packages and Types Reference*

Name Resolution in the Server

Class loading and name resolution in the server follow a very different paradigm than on a client, because the environments themselves are very different.

Java name resolution in Oracle JVM includes the following:

- Class resolver specs, which are schema lists to search in resolving a class schema object (functionally equivalent to the classpath on a client)
- The resolver, which maintains mappings between class schema objects that reference each other in the server

A class schema object must be resolved before Java objects of the class can be instantiated or methods of the class can be executed. A class schema object is said to be resolved when all of its external references to Java names are bound. In general, all the classes of a Java program should be compiled or loaded before they can be resolved. This is because Java programs are typically written in multiple source files that can reference each other recursively.

When all the class schema objects of a Java program in the server are resolved and none of them have been modified since being resolved, the program is effectively prelinked and ready to run.

Note: The `loadjava` utility resolves references to classes but not to resources. For ISO standard code, which has to be translated on the client, be careful how you load any resources into resource schema objects in the server. If you enabled the SQLJ `-ser2class` flag for your client-side translation, then your SQLJ profiles will be in class files and you will typically not have any resource files. If you did not enable `-ser2class`, then your profiles will be in `.ser` resource files.

See Also: *Oracle Database Java Developer's Guide*

SQL Names Versus Java Names

SQL names, such as names of source, class, and resource schema objects, are not global in the way that Java names are global. The Java Language Specification (JLS) directs that package names use Internet naming conventions to create globally unique names for Java programs. By contrast, a fully qualified SQL name is interpreted only with respect to the current schema and database. For example, the `SCOTT.FIZZ` name in one database does not necessarily denote the same program as `SCOTT.FIZZ` in another database. In fact, `SCOTT.FIZZ` in one database can even call `SCOTT.FIZZ` in another database.

Because of this inherent difference, SQL names must be interpreted and processed differently than Java names. SQL names are relative names and are interpreted from the point of view of the schema where a program is executed. This is central to how the program binds local data stored at that schema. Java names are global names, and the classes that they designate can be loaded at any execution site, with reasonable expectation that those classes will be classes that were used to compile the program.

Translating SQLJ Source on a Client and Loading Components

One approach to deploying SQLJ code in Oracle Database 11*g* is to run the SQLJ translator on a client computer to take care of translation, compilation, and profile

customization, if applicable. Then load the resulting class and resource files, if any, into the server, typically using a Java Archive (JAR) file. In fact, this is the only way to use ISO standard code in the server, because the server-side translator supports only Oracle-specific code generation.

If you are developing your source on a client computer, as is usually the case, and have a SQLJ translator available there, then this approach is advisable. It provides flexibility in running the translator, because option-setting and error-processing are not as convenient in the server.

For ISO standard code, it might also be advisable to use the SQLJ `-ser2class` option during translation when you intend to load an application into the server. This results in SQLJ profiles being converted from `.ser` serialized resource files to `.class` files and simplifies their naming. However, be aware that profiles converted to `.class` files cannot be further customized. To further customize, you would have to rerun the translator and regenerate the profiles.

> **See Also:** "Conversion of .ser File to .class File (-ser2class)" on page 8-53

When you load `.class` files and `.ser` resource files into Oracle Database 11*g*, either directly or using a JAR file, the resulting library units are referred to as Java class schema objects and Java resource schema objects. Your SQLJ profiles, if any, will be in resource schema objects, if you load them as `.ser` files, or in class schema objects if you enabled `-ser2class` during translation and load them as `.class` files.

This section covers the following topics:

- Loading Classes and Resources into the Server
- Naming of Loaded Class and Resource Schema Objects
- Publishing the Application After Loading Class and Resource Files
- Summary: Running a Client Application in the Server

Loading Classes and Resources into the Server

Once you run the translator on the client, use the Oracle `loadjava` client-side utility to load class and resource files into schema objects in the server. Either specify the class and resource files, if any, individually on the `loadjava` command line, or put them into a JAR file and specify the JAR file on the command line. A separate schema object is created for each `.class` or `.ser` file in the JAR file or on the command line.

Consider an example where you do the following:

1. Translate and compile `Foo.sqlj`, which includes an iterator declaration for `MyIter`, using ISO standard code generation.
2. Enable the `-ser2class` option when you translate `Foo.sqlj`.
3. Archive the resulting files, `Foo.class`, `MyIter.class`, `Foo_SJProfileKeys.class`, and `Foo_SJProfile0.class`, into `Foo.jar`.

Then run `loadjava` with the following command line (plus any options you want to specify). This examples uses the default JDBC Oracle Call Interface (OCI) driver:

```
% loadjava -user scott/tiger Foo.jar
```

Alternatively, you can use the original files:

```
% loadjava -user scott/tiger Foo.class MyIter.class Foo_SJProfileKeys.class
Foo_SJProfile0.class
```

or:

```
% loadjava -user scott/tiger Foo*.class MyIter.class
```

You can use the JDBC Thin driver for loading as follows (specifying the `-thin` option and an appropriate URL):

```
% loadjava -thin -user scott/tiger@localhost:1521/myservice Foo.jar
```

> **See Also:** "Code Generation" on page 9-3 and "Java Compilation" on page 9-6

Note:

- When you use the `-codegen=iso` setting during translation, generating profile files and then loading the profiles into the server as `.ser` files, they are first customized if they were not already customized on the client. If they were already customized, then they are loaded as is.
- You can access the `USER_OBJECTS` view in your schema to verify that your classes and resources are loaded properly.

Although the `loadjava` utility is recommended for loading your SQLJ and Java applications into the server, you can also use SQL `CREATE JAVA` commands such as the following:

```
CREATE OR REPLACE <AND RESOLVE> JAVA CLASS <NAMED name>;

CREATE OR REPLACE JAVA RESOURCE <NAMED name>;
```

> **See Also:** *Oracle Database SQL Language Reference* and *Oracle Database Java Developer's Guide*

Naming of Loaded Class and Resource Schema Objects

This section discusses how schema objects for classes and profiles are named when you load classes and profiles into the server. However, remember that profiles are created only for ISO standard code generation.

For ISO standard code generation, if the SQLJ `-ser2class` option was enabled when you translated your application on the client, then profiles were converted to `.class` files and will be loaded into class schema objects in the server. If `-ser2class` was not enabled, then profiles were generated as `.ser` serialized resource files and will be loaded into resource schema objects in the server.

In the following discussion, it is assumed that you use only the default connection context class for any application that will run in the server. Therefore, there will be only one profile.

Full Names and Short Names

There are two forms of schema object names in the server, full names and short names. Full names are fully qualified and are used as the schema object names whenever possible. If any full name is longer than 31 characters, however, or contains characters that are illegal or cannot be converted to characters in the database character set, then

Oracle Database 11*g* converts the full name to a short name to use as the name of the schema object, keeping track of both names and how to convert between them. If the full name is 31 characters or less and has no illegal or inconvertible characters, then the full name is used as the schema object name.

For more information about these and about other file naming considerations, including `DBMS_JAVA` procedures to retrieve a full name from a short name, and vice versa, refer to *Oracle Database Java Developer's Guide*

Full Names of Loaded Classes

Loaded classes will include profile files if you use ISO standard code generation and enable the `-ser2class` flag. The full name of the class schema object produced when you load a `.class` file into the server is determined by the package and class name in the original source code. Any path information you supply on the command line or in the JAR file is irrelevant in determining the name of the schema object. For example, if `Foo.class` consists of the `Foo` class, which was specified in the source code as being in the `x.y` package, then the full name of the resulting class schema object is as follows:

```
x/y/Foo
```

Note that the `.class` extension is dropped.

If `Foo.sqlj` declares an iterator, `MyIter`, then the full name of its class schema object is as follows (unless it is a nested class, in which case it will not have its own schema object):

```
x/y/MyIter
```

Furthermore, if you are using ISO standard code generation:

- The related profile-keys class file, generated by SQLJ when you translate `Foo.sqlj`, is `Foo_SJProfileKeys.class`. Therefore, the full name of its class schema object is:

  ```
  x/y/Foo_SJProfileKeys
  ```

- If the `-ser2class` option was enabled when you translated your application, then the resulting profile is generated in `Foo_SJProfile0.class`. Therefore, the full name of the class schema object is:

  ```
  x/y/Foo_SJProfile0
  ```

Full Names of Loaded Resources

This discussion is relevant only if you are using ISO standard code generation and did not enable the `-ser2class` option when you translated your application, or if you use other Java serialized resource (`.ser`) files in your application.

The naming of resource schema objects is handled differently from class schema objects. Their names are not determined from the contents of the resources. Instead, their full names are identical to the names that appear in a JAR file or on the `loadjava` command line, including path information. Also note that the `.ser` extension is *not* dropped.

It is important to note that because resource names are used to locate the resources at run time. Their names must include the correct path information. In the server, the correct full name of a resource is identical to the relative path and file name that Java would use to look it up on a client.

In the case of a SQLJ profile, this is a subdirectory under the directory specified by the translator `-d` option, according to the package name. If the `-d` option, used to specify the top-level output directory for generated `.class` and `.ser` files, is set to `/mydir` and the application is in the `abc.def` package, then `.class` and `.ser` files generated during translation will be placed in the `/mydir/abc/def` directory.

> **See Also:** "Output Directory for Generated .ser and .class Files (-d)" on page 8-22

At run time, `/mydir` would presumably be in your classpath and Java will look for your application components in the `abc/def` directory underneath it. Therefore, when you load this application into the server, you must run `loadjava` or `jar` from the `-d` directory so that the path you specify on the command line to find the files also indicates the package name, as follows (where `%` is the system prompt):

```
% cd /mydir
% loadjava <...options...> abc/def/*.class abc/def/*.ser
```

Alternatively, to use a JAR file:

```
% cd /mydir
% jar -cvf myjar.jar abc/def/*.class abc/def/*.ser
% loadjava <...options...> myjar.jar
```

If your application is `App` and your profile is `App_SJProfile0.ser`, then either of the preceding examples will correctly result in the following full name of the created resource schema object:

```
abc/def/App_SJProfile0.ser
```

Note that `.ser` is retained.

Note also that if you set `-d` to a directory whose hierarchy has no other contents (which is advisable), you can simply run the JAR utility as follows to recursively get your application components:

```
% cd /mydir
% jar -cvf myjar.jar *
% loadjava <...options...> myjar.jar
```

Publishing the Application After Loading Class and Resource Files

Before using your SQLJ code in the server, you must publish the top-level methods, as is true of any Java code you use in the server. Publishing includes writing call descriptors, mapping data types, and setting parameter modes.

> **See Also:** *Oracle Database Java Developer's Guide*

Summary: Running a Client Application in the Server

This section summarizes the typical steps of running a client application in the server. As an example, it uses a demo application called `NamedIterDemo`.

1. Create a JAR file for your application components. For `NamedIterDemo`, the components include `SalesRec.class` as well as the application class and profile, if any.

   You can create JAR file `niter-server.jar` as follows:

   ```
   % jar cvf niter-server.jar Named*.class Named*.ser SalesRec.class
   ```

```
connect.properties
```

But remember that `.ser` files are only relevant for ISO standard code generation.

2. Load the JAR file into the server.

   Use `loadjava` as follows. This example instructs `loadjava` to use the OCI driver in loading the files. The `-resolve` option results in the class files being resolved.

   ```
   % loadjava -oci -resolve -force -user scott/tiger niter-server.jar
   ```

3. Create a SQL wrapper in the server for your application.

   For example, run a SQL*Plus script that executes the following:

   ```
   set echo on
   set serveroutput on
   set termout on
   set flush on

   execute dbms_java.set_output(10000);

   create or replace procedure SQLJ_NAMED_ITER_DEMO as language java
   name 'NamedIterDemo.main (java.lang.String[])';
   /
   ```

   The `DBMS_JAVA.SET_OUTPUT()` routine reroutes default output to your screen, instead of to a trace file. The input parameter is the buffer size in bytes.

4. Execute the wrapper.

   For example:

   ```
   sqlplus> call SQLJ_NAMED_ITER_DEMO();
   ```

Loading SQLJ Source and Translating in the Server

Another approach to developing SQLJ code for the server is loading the source code into the server and translating it directly in the server. This uses the embedded SQLJ translator in Oracle JVM. This discussion still assumes you created the source on a client computer.

> **Note:** The server-side SQLJ translator does not support ISO standard code generation. If you want to use such code in the server, you must translate on a client and load the individual class files and resources into the server.

As a general rule, loading SQLJ source into the server is identical to loading Java source into the server, with translation taking place implicitly when a compilation option is set, such as the `loadjava -resolve` option. When you load `.sqlj` source files into Oracle Database 11*g*, either directly or using a JAR file, the resulting library units containing the source code are referred to as Java source schema objects. A separate schema object is created for each source file.

When translation and compilation take place, the resulting library units for the generated classes are referred to as Java class schema objects, just as they are when loaded directly into the server from `.class` files created on a client. A separate schema object is created for each class. Resource schema objects are used for properties files that you load into the server.

This section covers the following topics:

- Loading SQLJ Source Code into the Server
- Option Support in the Server Embedded Translator
- Naming of Loaded Source and Generated Class and Resource Schema Objects
- Error Output from the Server Embedded Translator
- Publishing the Application After Loading Source Files

> **See Also:** *Oracle Database SQL Language Reference* and *Oracle Database Java Developer's Guide*

Loading SQLJ Source Code into the Server

Use the Oracle `loadjava` client-side utility on a `.sqlj` file, instead of on a `.class` file, to load source into the server. If you enable the `loadjava -resolve` option in loading a `.sqlj` file, then the server-side embedded translator is run to perform the translation and compilation of your application as it is loaded. Otherwise, the source is loaded into a source schema object without any translation. However, in this case, the source *is* implicitly translated and compiled the first time an attempt is made to use a class defined in the source. Such implicit translation might seem surprising at first, because there is nothing comparable in client-side SQLJ.

For example, run `loadjava` as follows from the system prompt:

```
% loadjava -user scott/tiger -resolve Foo.sqlj
```

Alternatively, you can use the JDBC Thin driver to load:

```
% loadjava -thin -user scott/tiger@localhost:1521/myservice -resolve Foo.sqlj
```

Either of these will result in appropriate class schema objects being created in addition to the source schema object.

Before running `loadjava`, however, you must set SQLJ options appropriately. Note that encoding can be set on the `loadjava` command line, instead of through the server-side SQLJ `encoding` option, as follows:

```
% loadjava -user scott/tiger -resolve -encoding SJIS Foo.sqlj
```

The `loadjava` script, which runs the actual utility, is in the `bin` subdirectory under your *`ORACLE_HOME`* directory. This directory should already be in your path once Oracle has been installed.

Note:

- In processing a JAR file, `loadjava` first processes `.sqlj`, `.java`, and `.class` files. It then makes a second pass and processes everything else as Java resource files.
- You cannot load a `.sqlj` file along with `.class` files that were generated from processing of the same `.sqlj` file. This would create an obvious conflict, because the server would be trying to load the same classes that it would also be trying to generate.
- You can put multiple `.sqlj` files into a JAR file and specify the JAR file to `loadjava`.
- You can access the `USER_OBJECTS` view in your schema to verify that your classes are loaded properly.

Although the `loadjava` utility is recommended for loading your SQLJ and Java applications into the server, you can also use SQL `CREATE JAVA` commands such as the following:

```
CREATE OR REPLACE <AND COMPILE> JAVA SOURCE <NAMED srcname> <AS loadname>;
```

If you specify `AND COMPILE` for a `.sqlj` file, then the source is translated and compiled at that time, creating class schema objects as appropriate in addition to the source schema object. Otherwise, it is not translated and compiled. In this case, only the source schema object is created. In this latter case, however, the source *is* implicitly translated and compiled the first time an attempt is made to use a class contained in the source.

Note: When you first load a source file, some checking of the source code is performed, such as determining what classes are defined. If any errors are detected at this time, the load fails.

Option Support in the Server Embedded Translator

The following options are available in the server-side SQLJ translator:

- `encoding`
- `online`
- `debug`

This section discusses these options, after leading off with some discussion of fixed settings in server-side SQLJ. There is also discussion of the `loadjava` utility and its `-resolve` option.

Fixed Settings in the Server-Side SQLJ Translator

The following settings, supported by SQLJ translator options on a client, are fixed in the server-side translator:

- Both online semantics-checking and offline parsing are enabled in the server by default, equivalent to the default `-parse=both` setting on a client. You can override this to disable online semantics-checking through the `online` option, but cannot disable offline parsing.

> **See Also:** "Online Semantics-Checking Versus Offline Parsing" on page 8-57

- Oracle-specific code generation is used in the server, equivalent to the default `-codegen=oracle` setting on a client. This is a fixed setting.

  > **See Also:** "Oracle-Specific Code Generation (No Profiles)" on page 3-28

- Class schema objects created during server-side translation reference line numbers that map to the SQLJ source code. This is equivalent to enabling the `-linemap` option when you translate on a client.

  > **See Also:** "Line-Mapping to SQLJ Source File (-linemap)" on page 8-36

The encoding Option

This option determines any encoding used to interpret your source code when it is loaded into the server. The `encoding` option is used at the time the source is loaded, regardless of whether it is also compiled. Alternatively, when using `loadjava` to load your SQLJ application into the server, you can specify encoding on the `loadjava` command line. Any `loadjava` command-line setting for encoding overrides this `encoding` option.

> **Note:** If no encoding is specified, either through this option or through `loadjava`, then encoding is performed according to the `file.encoding` setting of the client from which you run `loadjava`.

> **See Also:** "Encoding for Input and Output Source Files (-encoding)" on page 8-21

The online Option

A `true` setting for the `online` option (the default value) enables online semantics-checking. Semantics-checking is performed relative to the schema in which the source is loaded. You do not specify an exemplar schema, as you do for online-checking on a client. If the `online` option is set to `false`, offline checking is performed.

In either case, the default checker is `oracle.sqlj.checker.OracleChecker`, which will choose an appropriate checker according to your JDBC driver version and Oracle version.

> **See Also:** "Semantics-Checkers and the OracleChecker Front End (default checker)" on page 8-56

The `online` option is used at the time the source is translated and compiled. If you load it with the `loadjava -resolve` option enabled, then this will occur immediately. Otherwise it will occur the first time an attempt is made to use a class defined in the source, resulting in implicit translation and compilation.

> **Note:** The `online` option is used differently in the server than on a client. In the server, the `online` option is only a flag that enables online checking using a default checker. On a client, the `-online` option specifies which checker to use, but it is the `-user` option that enables online checking.

The debug Option

Setting this option to `true` instructs the server-side Java compiler to output debugging information when a `.sqlj` or `.java` source file is compiled in the server. This is equivalent to using the `-g` option when running the standard `javac` compiler on a client.

Source is compiled during loading if you use the `loadjava -resolve` option, right after SQLJ translation in the case of a `.sqlj` file. If you do not use the `-resolve` option, then implicit translation and compilation occurs the first time an attempt is made to use a class defined in the source.

Setting SQLJ Options in the Server

There is no command line and there are no properties files when running the SQLJ translator in the server. Information about translator and compiler options is held in each schema in a table named `JAVA$OPTIONS`. Manipulate options in this table through the following functions and procedures of the `DBMS_JAVA` package:

- `DBMS_JAVA.GET_COMPILER_OPTION()`
- `DBMS_JAVA.SET_COMPILER_OPTION()`
- `DBMS_JAVA.RESET_COMPILER_OPTION()`

Use `set_compiler_option()` to specify separate option settings for individual packages or sources. It takes the following as input, with each parameter enclosed by single-quotes:

- Package name, using dotted names, or source name

  Specify this as a full name, not a short name. If you specify a package name, then the option setting applies to all sources in that package and subpackages, except where you override the setting for a particular subpackage or source.
- Option name
- Option setting

Execute the `DBMS_JAVA` routines using SQL*Plus as follows:

```
sqlplus> execute dbms_java.set_compiler_option('x.y', 'online', 'true');
sqlplus> execute dbms_java.set_compiler_option('x.y.Create', 'online', 'false');
```

These two commands enable online checking for all sources in the `x.y` package, then override that for the `Create` source by disabling online checking for that particular source.

Similarly, set encoding for the `x.y` package to `SJIS` as follows:

```
sqlplus> execute dbms_java.set_compiler_option('x.y', 'encoding', 'SJIS');
```

Server-Side Option Notes

Be aware of the following:

- The `set_compiler_option()` parameter for package and source names uses dotted names, such as `abc.def` as a package name, even though schema object names use slash syntax, such as `abc/def` as a package name.
- When you specify a package name, be aware that the option will apply to any included packages as well. A setting of `a.b.MyPackage` sets the option for any source schema objects whose names are of the following form:

  ```
  a/b/MyPackage/subpackage/...
  ```

- Specifying `''` (empty set of single-quotes) as a package name makes the option apply to the root and all subpackages, effectively making it apply to all packages in your schema.

Naming of Loaded Source and Generated Class and Resource Schema Objects

When you use the server-side SQLJ translator, such as when you use `loadjava` on a `.sqlj` file with the `-resolve` option enabled, the output generated by the server-side translator is essentially identical to what would be generated on a client. This output consists of a compiled class for each class you defined in the source and a compiled class for each iterator and connection context class.

As a result, the following schema objects will be produced when you load a `.sqlj` file into the server with `loadjava` and have it translated and compiled:

- A source schema object for the original source code
- A class schema object for each class you defined in the source
- A class schema object for each iterator or connection context class you declared in the source

  But presumably you will not need to declare connection context classes in code that will run in the server, unless it is to specify type maps for user-defined types.

The full names of these schema objects are determined as described in the following subsections. Use the `loadjava -verbose` option for a report of schema objects produced and what they are named.

Full Name of Source

When you load a source file into the server, regardless of whether it is translated and compiled, a source schema object is produced. The full name of this schema object is determined by the package and class names in the source code. Any path information you supply to `loadjava` on the command line is irrelevant to the determination of the name of the schema object.

For example, if `Foo.sqlj` defines the `Foo` class in the `x.y` package and defines or declares no other classes, then the full name of the resulting source schema object is:

```
x/y/Foo
```

Note that the `.sqlj` extension is dropped.

If you define additional classes or declare iterator or connection context classes, then the source schema object is named according to the first public class definition or declaration encountered, or, if there are no public classes, the first class definition. In the server, there can be more than one public class definition in a single source.

For example, if `Foo.sqlj` is still in the `x.y` package, defines public class `Bar` first and then class `Foo`, and has no public iterator or connection context class declarations

preceding the definition of `Bar`, then the full name of the resulting source schema object is:

```
x/y/Bar
```

However, if the declaration of public iterator class `MyIter` precedes the `Bar` and `Foo` class definitions, then the full name of the resulting source schema object is:

```
x/y/MyIter
```

Full Names of Generated Classes

Class schema objects are generated for each class you defined in the source, each iterator you declared, and the profile-keys class. The naming of the class schema objects is based on the class names and the package name from the source code.

This discussion continues the example in "Full Name of Source". Presume your source code specifies the `x.y` package, defines public class `Bar` then class `Foo`, then declares public iterator class `MyIter`. The full names of the class schema objects for the classes you define and declare are as follows:

```
x/y/Bar
x/y/Foo
x/y/MyIter
```

Note that `.class` is not appended.

> **Note:** It is recommended that the source name always match the first public class defined or, if there are no public classes, the first class defined. This will avoid possible differences between client-side and server-side behavior.

The name of the original source file, as well as any path information you specify when loading the source into the server, is irrelevant in determining the names of the generated classes. If you define inner classes or anonymous classes in your code, then they are named according to the conventions of the standard `javac` compiler.

Error Output from the Server Embedded Translator

SQLJ error processing in the server is similar to general Java error processing in the server. SQLJ errors are directed into the `USER_ERRORS` table of the user schema. You can `SELECT` from the `TEXT` column of this table to get the text of a given error message.

However, if you use `loadjava` to load your SQLJ source, then `loadjava` also captures and writes the error messages from the server-side translator.

Informational messages and suppressible warnings are withheld by the server-side translator in a way that is equivalent to the operation of the client-side translator with a `-warn=noportable,noverbose` setting, which is the default.

> **See Also:** "Translator Warnings (-warn)" on page 8-33

Publishing the Application After Loading Source Files

Before using your SQLJ code in the server, you must publish the top-level methods, as is true of any Java code you use in the server. Publishing includes writing call

descriptors, mapping data types, and setting parameter modes. For information, refer to *Oracle Database Java Developer's Guide*.

Dropping Java Schema Objects

To complement the `loadjava` utility, Oracle provides the `dropjava` utility to remove Java source, class, and resource schema objects. It is recommended that any schema object loaded into the server using `loadjava` be removed using `dropjava` only.

The `dropjava` utility transforms command-line file names and JAR file contents to schema object names, then removes the schema objects. You can enter `.sqlj`, `.java`, `.class`, `.ser`, and `.jar` files on the command line in any order.

You should always remove Java schema objects in the same way that you first loaded them. If you load a `.sqlj` source file and translate it in the server, then run `dropjava` on the same source file. If you translate on a client and load classes and resources directly, then run `dropjava` on the same classes and resources.

For example, if you run `loadjava` on `Foo.sqlj`, then execute `dropjava` on the same file name, as follows:

```
% dropjava -user scott/tiger Foo.sqlj
```

If you translate your program on the client and load it using JAR file containing the generated components, then use the same JAR file name to remove the program:

```
% dropjava -user scott/tiger Foo.jar
```

If you translate your program on the client and load the generated components using the `loadjava` command line, then remove them using the `dropjava` command line, as follows (assume `-codegen=oracle` and no iterator classes):

```
% dropjava -user scott/tiger Foo*.class
```

See Also: *Oracle Database Java Developer's Guide*

Additional Server-Side Considerations

This section discusses Java multithreading in the server and recursive SQLJ calls in the server. It covers the following topics:

- Java Multithreading in the Server
- Recursive SQLJ Calls in the Server
- Verifying that Code is Running in the Server

Java Multithreading in the Server

Programs that use Java multithreading can execute in Oracle Database 11*g* without modification. However, while client-side programs use multithreading to improve throughput for users, there are no such benefits when Java-multithreaded code runs in the server. If you are considering porting a multithreaded application into the server, be aware of the following important differences in the functionality of multithreading in Oracle JVM, as opposed to in client-side JVMs:

- Threads in the server run sequentially, not simultaneously.
- In the server, threads within a call die at the end of the call.

- Threads in the server are not preemptively scheduled. If one thread goes into an infinite loop, then no other threads can run.

Do not confuse Java multithreading in Oracle Database 11*g* with general Oracle server multithreading. The latter refers to simultaneous Oracle sessions, not Java multithreading. In the server, scalability and throughput are gained by having many individual users, each with his own session, executing simultaneously. The scheduling of Java execution for maximum throughput, such as for each call within a session, is performed by Oracle Database 11*g*, and not by Java.

See Also: "Multithreading in SQLJ" on page 7-31

Recursive SQLJ Calls in the Server

SQLJ generally does not allow multiple SQLJ statements to use the same execution context instance simultaneously. Specifically, a statement trying to use an execution context instance that is already in use will be blocked until the first statement completes.

However, this functionality would be less desirable in the server than on a client. This is because different stored procedures or functions, which all typically use the default execution context instance, can inadvertently try to use this same execution context instance simultaneously in recursive situations. For example, one stored procedure might use a SQLJ statement to call another stored procedure that uses SQLJ statements. When these stored procedures are first created, there is probably no way of knowing when such situations might arise, so it is doubtful that particular execution context instances are specified for any of the SQLJ statements.

To address this situation, SQLJ *does* allow multiple SQLJ statements to use the same execution context instance simultaneously if this results from recursive calls.

Consider an example of a recursive situation to see what happens to status information in the execution context instance. Presume that all statements use the default connection context instance and its default execution context instance. If stored procedure `proc1` has a SQLJ statement that calls stored procedure `proc2`, which also has SQLJ statements, then the statements in `proc2` will each be using the execution context instance while the procedure call in `proc1` is also using it.

Each SQLJ statement in `proc2` results in status information for that statement being written to the execution context instance, with the opportunity to retrieve that information after completion of each statement, as desired. The status information from the statement in `proc1` that calls `proc2` is written to the execution context instance only after `proc2` has finished executing, program flow has returned to `proc1`, and the operation in `proc1` that called `proc2` has completed.

To avoid confusion about execution context status information in recursive situations, execution context methods are carefully defined to update status information about a SQL operation only after the operation has completed.

Note:

- To avoid confusion, use distinct execution context instances as appropriate whenever you plan to use execution context status or control methods in code that will run in the server.
- Be aware that if the preceding example does not use distinct execution context instances and `proc2` has any method calls to the execution context instance to change control parameters, then this will affect operations subsequently executed in `proc1`.
- Update batching is not supported across recursive calls. By default, only the top-level procedure will perform batching, if enabled. This limitation can be avoided by using explicit execution context instances.

See Also: "Execution Context Methods" on page 7-26

Verifying that Code is Running in the Server

A convenient way to verify that your code is actually running in the server is to use the static `getProperty()` method of the `java.lang.System` class to retrieve the `oracle.server.version` Java property. If this property contains a version number, then you are running in the server. If it is `null`, then you are not. Here is an example:

```
...
if (System.getProperty("oracle.server.version") != null
{
   // (running in server)
}
...
```

Note: Do not use the `getProperties()` method, as this causes a security exception in the server.

A

Customization and Specialized Customizers

Profiles and profile customization are introduced in "SQLJ Profiles" on page 3-33. This appendix presents more technical detail and discusses customizer options and how to use customizers other than the default Oracle customizer.

There is also discussion of Oracle specialized customizers, particularly the `SQLCheckerCustomizer` for semantics-checking profiles, and the `AuditorInstaller` for installing *auditors* for debugging.

The following topics are covered:

> **Note:** If you use the default Oracle-specific code generation (`-codegen=oracle`), the discussion in this appendix does not pertain to your application.

More About Profiles

SQLJ profiles contain information about your embedded SQL operations, with a separate profile being created for each connection context class that your application uses. Profiles are created during the SQLJ translator code generation phase and customized during the customization phase. Customization enables your application to use vendor-specific database features. Separating these vendor-specific operations into your profiles enables the rest of your generated code to remain generic.

Each profile contains a series of entries for the SQLJ statements that use the relevant connection context class, where each entry corresponds to one SQL operation in your application.

Profiles exist as serialized objects stored in resource files packaged with your application. Because of this, profiles can be loaded, read, and modified (added to or recustomized) at any time. When profiles are customized, information is only added, never removed. Multiple customizations can be made without losing preceding customizations, so that your application maintains the capability to run in multiple environments. This is known as **binary portability**.

For profiles to have binary portability, SQLJ industry-standard requirements have been met in the Oracle SQLJ implementation.

Creation of a Profile During Code Generation

During code generation, the translator creates each profile as follows:

1. It creates a profile object as an instance of the `sqlj.runtime.profile.Profile` class.
2. It inserts information about your embedded SQL operations into the profile object, for SQLJ statements that use the relevant connection context class.
3. It serializes the profile object into a Java resource file, referred to as a *profile file*, with a `.ser` file name extension.

> **Note:** The Oracle SQLJ implementation provides an option to have the translator automatically convert these `.ser` files to `.class` files. The `.ser` files are not supported by some browsers, and can be cumbersome when loading translated applications into the server. However, this prevents any further customization of the profile. For information, see "Conversion of .ser File to .class File (-ser2class)" on page 8-53.

As discussed in "Code Generation" on page 9-3, profile file names for application `Foo` are of the form:

```
Foo_SJProfilen.ser
```

SQLJ generates `Foo_SJProfile0.ser`, `Foo_SJProfile1.ser`, and so on, as needed, depending on how many connection context classes you use in your code. Or, if the `-ser2class` option is enabled, then SQLJ generates `Foo_SJProfile0.class`, `Foo_SJProfile1.class`, and so on.

Each profile has a `getConnectedProfile()` method that is called during SQLJ runtime. This method returns something equivalent to a JDBC `Connection` object, but with added functionality. This is further discussed in "Functionality of a Customized Profile at Run Time" on page A-6.

> **Note:** Referring to a "profile object" indicates that the profile is in its original nonserialized state. Referring to a "profile file" indicates that the profile is in its serialized state in a `.ser` file.

Sample Profile Entry

Following is a sample SQLJ executable statement with the profile entry that would result. For simplicity, the profile entry is presented as plain text with irrelevant portions omitted.

Note that in the profile entry, the host variable is replaced by JDBC syntax (the question mark).

SQLJ Executable Statement

Presume the following declaration:

```
#sql iterator Iter (double sal, String ename);
```

And presume the following executable statements:

```
String empname = 'Smith';
Iter it;
...
#sql it = { SELECT ename, sal FROM emp WHERE ename = :empname };
```

Corresponding SQLJ Profile Entry

```
====================================================================
...
#sql { SELECT ename, sal FROM emp WHERE ename = ? };
...
PREPARED_STATEMENT executed through EXECUTE_QUERY
role is QUERY
descriptor is null
contains one parameter
1. mode: IN, java type: java.lang.String (java.lang.String),
   sql type: VARCHAR, name: ename, ...
result set type is NAMED_RESULT
result set name is Iter
contains 2 result columns
1. mode: OUT, java type: double (double),
   sql type: DOUBLE, name: sal, ...
2. mode: OUT, java type: java.lang.String (java.lang.String),
   sql type: VARCHAR, name: ename, ...
====================================================================
```

Note: This profile entry is presented here as text for convenience only; profiles are not actually in text format. They can be printed as text, however, using the SQLJ `-P-print` option, as discussed in "Overview of Customizer Harness Options" on page A-7.

More About Profile Customization

When using ISO SQLJ code, running the `sqlj` script on a SQLJ source file includes an automatic customization process, where each profile created during the code generation phase is customized for use with your particular database. The default customizer is the Oracle customizer, `oracle.sqlj.runtime.OraCustomizer`, which optimizes your profiles to use type extensions and performance enhancements specific to Oracle10*i*.

You can also run the `sqlj` script to customize profiles created previously. On the SQLJ command line, you can specify `.ser` files individually, JAR files containing `.ser` files, or both.

Note:

- Whenever you use the default Oracle customizer during translation, your application will require the Oracle SQLJ run time and an Oracle JDBC driver when it runs, even if you do not use Oracle extensions in your code.
- If an application has no customizations, or none suitable for the connection, then the generic SQLJ run time is used.
- You can run SQLJ to process `.sqlj` and `.java` files (for translation, compilation, and customization) or to process `.ser` and `.jar` files (for customization only), but not both categories at once.

Overview of the Customizer Harness and Customizers

Regardless of whether you use the Oracle customizer or an alternative customizer, SQLJ uses a front-end customization utility known as the *customizer harness* in accomplishing your customizations.

When you run SQLJ, you can specify customization options for the customizer harness (for general customization settings that apply to any customizer you use) and for your customizer (for settings used by the particular customizer). In either case, you can specify these options either on the command line or in a properties file. This is discussed in "Customization Options and Choosing a Customizer" on page A-6.

A customizer is required to be a JavaBeans component adhering to the standard JavaBeans API to expose its properties, and must implement the `sqlj.runtime.profile.util.ProfileCustomizer` interface, which specifies a `customize()` method. For each profile to be customized, the customizer harness calls the `customize()` method of the customizer object.

The Oracle customizer meets the preceding requirements and is defined in the `oracle.sqlj.runtime.OraCustomizer` class.

Steps in the Customization Process

The SQLJ customization process during translation consists of the following steps, as applicable, either during the customization stage of an end-to-end SQLJ run, or when you run SQLJ to customize existing profiles only:

1. SQLJ instantiates and invokes the customizer harness and passes it any general customization options you specified.
2. The customizer harness instantiates the customizer you are using and passes it any customizer-specific options you specified.
3. When you run SQLJ for customization only, specifying one or more JAR files on the command line, the customizer harness discovers and extracts the profile files within these JAR files.
4. The customizer harness deserializes each profile file into a profile object (`.ser` files automatically created during an end-to-end SQLJ run, `.ser` files specified on the command line for customization only, or `.ser` files extracted from JAR files specified on the command line for customization only).
5. If the customizer you use requires a database connection, the customizer harness establishes that connection.

6. For each profile, the harness calls the `customize()` method of the customizer object instantiated in step 2 (customizers used with SQLJ must have a `customize()` method).
7. For each profile, the `customize()` method typically creates and registers a profile customization within the profile. This depends on the intended functionality of the customizer, however. Some might have a specialized purpose that does not require a customization to be created and registered in this way.
8. The customizer harness reserializes each profile and puts it back into a `.ser` file.
9. When you run SQLJ for customization only, specifying one or more JAR files on the command line, the customizer harness recreates the JAR contents, inserting each customized `.ser` file to replace the original corresponding uncustomized `.ser` file.

Note:

- If an error occurs during customization of a profile, the original `.ser` file is not replaced.
- If an error occurs during customization of any profile in a JAR file, the original JAR file is not replaced.
- SQLJ can run only one customizer at a time. If you want to accomplish multiple customizations on a single profile, you must run SQLJ multiple times. For the additional customizations, enter the profile name directly on the SQLJ command line.

Creation and Registration of a Profile Customization

When the harness calls the `customize()` method to customize a profile, it passes in the profile object, a SQLJ connection context object (if you are using a customizer that requires a connection), and an error log object (which is used in logging error messages during the customization).

The same error log object is used for all customizations throughout a single running of SQLJ, but its use is transparent. The customizer harness reads messages written to the error log object and reports them in real-time to the standard output device (whatever SQLJ uses, typically your screen).

Recall that each profile has a set of entries, where each entry corresponds to a SQL operation. (These would be the SQL operations in your application that use instances of the connection context class associated with this profile.)

A `customize()` method implements special processing on these entries. It could be as simple as checking each entry to verify its syntax, or it could be more complicated, such as creating new entries that are equivalent to the original entries but are modified to use features of your particular database.

Note:

- Any `customize()` processing of profile entries does not alter the original entries.
- Customizing your profiles for use in a particular environment does not prevent your application from running in a different environment. You can customize a profile multiple times for use in multiple environments, and these customizations will not interfere with each other.

Customization Error and Status Messages

The customizer harness outputs error and status messages in much the same way as the SQLJ translator, outputting them to the same output device. None of the warnings regarding customization are suppressible, however.

Error messages reported by the customizer harness fall into four categories:

- Unrecognized or illegal option
- Connection instantiation error
- Profile instantiation error
- Customizer instantiation error

Status messages reported by the customizer harness during customization enable you to determine whether a profile was successfully customized. They fall into three categories:

- Profile modification status
- JAR file modification status
- Name of backup file created (if the customizer harness `backup` option is enabled)

Additional customizer-specific errors and warnings might be reported by the `customize()` method of the particular customizer.

During customization, the profile customizer writes messages to its error log, and the customizer harness reads the log contents in real-time and outputs these messages to the SQLJ output device, along with any other harness output. You never have to access error log contents directly.

Functionality of a Customized Profile at Run Time

A customized profile is a static member of the connection context class with which it is associated. For each SQLJ statement in your application, the SQLJ run time determines the connection context class and instance associated with that statement, then uses the customized profile of the connection context class, together with the underlying JDBC connection of the particular connection context instance, to create a *connected profile*. This connected profile is the vehicle that the SQLJ run time uses in applying vendor-specific features to the execution of your SQLJ application.

Customization Options and Choosing a Customizer

This section discusses options for profile customization, which fall into three categories:

- Options you specify to the customizer harness, which apply to whatever customizer you use.

  This includes general options, connection options, and options that invoke specialized customizers.
- Customizer-specific options you specify to your customizer through the customizer harness.
- SQLJ options, which determine basic aspects of customization, such as whether to customize at all and which customizer to use.

All categories of options are specified through the SQLJ command line or properties files.

The following topics are included in this section:

To choose a customizer other than the default Oracle customizer, you can use either the customizer harness `customizer` option (discussed in "Overview of Customizer Harness Options" on page A-7) or the SQLJ `-default-customizer` option (discussed in "SQLJ Translator Options for Profile Customization" on page A-29).

Overview of Customizer Harness Options

The customizer harness provided with the Oracle SQLJ implementation offers a number of options that are not specific to a particular customizer. The harness uses these options in its front-end coordination of the customization process.

Syntax for Customizer Harness Options

Customizer harness option settings on the SQLJ command line have the following syntax:

```
-P-option=value
```

Alternatively, in a SQLJ properties file:

```
profile.option=value
```

Enable boolean options (flags) either with:

```
-P-option
```

or:

```
-P-option=true
```

Boolean options are disabled by default, but you can explicitly disable them with:

```
-P-option=false
```

This option syntax is also discussed in "Options to Pass to the Profile Customizer (-P)" on page 8-50 and "Properties File Syntax" on page 8-13.

Options Supported by the Customizer Harness

The customizer harness supports the following general options:

- `backup`: Save a backup copy of the profile before customizing it.
- `context`: Limit customizations to profiles associated with the listed connection context classes.
- `customizer`: Specify the customizer to use.
- `digests`: Specify digests for JAR file manifests (relevant only if specifying JAR files to customize).
- `help`: Display customizer options (specified only in SQLJ command line).
- `verbose`: Display status messages during customization.

The customizer harness supports the following options for customizer database connections. Currently, these are used by the Oracle customizer if you enable its `optcols` option for column definitions (for performance optimization). In addition, they are used by the `SQLCheckerCustomizer` if you use this specialized customizer to perform online semantics-checking on profiles.

- `user`: Specify the user name for the connection used in this customization.
- `password`: Specify the password for the connection used in this customization.
- `url`: Specify the URL for the connection used in this customization.
- `driver`: Specify the JDBC driver for the connection used in this customization.

For information about the Oracle customizer `optcols` flag, see "Oracle Customizer Column Definition Option (optcols)" on page A-20. For information about the `SQLCheckerCustomizer`, see "SQLCheckerCustomizer for Profile Semantics-Checking" on page A-30.

The following commands function as customizer harness options, but are implemented through specialized customizers provided with the Oracle SQLJ implementation.

- `debug`: Insert debugging information into the specified profiles, to be output at run time. This is a shortcut to invoke the Oracle SQLJ `AuditorInstaller`, which is described in "AuditorInstaller Customizer for Debugging" on page A-33.
- `print`: Output the contents of the specified profiles, in text format.
- `verify`: Perform semantics-checking on a profile that was produced during a previous execution of the SQLJ translator (equivalent to semantics-checking performed on source code during translation). This is a shortcut to invoke the Oracle SQLJ `SQLCheckerCustomizer`, which is described in "SQLCheckerCustomizer for Profile Semantics-Checking" on page A-30.

General Customizer Harness Options

This section describes general options supported by the customizer harness.

Profile Backup Option (backup)

Use the `backup` flag to instruct the harness to save a backup copy of each `.jar` file and standalone `.ser` file before replacing the original. (Separate backups of `.ser` files that are within `.jar` files are not necessary.)

Backup file names are given the extension `.bak`*n*, where *n* indicates digits used as necessary where there are similarly named files. For each backup file created, an informational message is issued.

If an error occurs during customization of a standalone `.ser` file, then the original `.ser` file is not replaced and no backup is created. Similarly, if an error occurs during customization of any `.ser` file within a JAR file, then the original JAR file is not replaced and no backup is created.

The command-line syntax for this option is:

```
-P-backup<=true|false>
```

Command-line example is:

```
-P-backup
```

Properties file syntax is:

```
profile.backup<=true|false>
```

Default value is:

```
false
```

Customization Connection Context Option (context)

Use the `context` option to limit customizations to profiles that correspond to the specified connection context classes. Fully qualify the class names and use a comma-delimited list to specify multiple classes. For example:

```
-P-context=sqlj.runtime.ref.DefaultContext,foo.bar.MyCtxtClass
```

There must be *no* space on either side of the comma.

If this option is not specified, then all profiles are customized, regardless of their associated connection context classes.

Command-line syntax is:

```
-P-context=ctx_class1<,ctx_class2,...>
```

Command-line example is:

```
-P-context=foo.bar.MyCtxtClass
```

Properties file syntax is:

```
profile.context=ctx_class1<,ctx_class2,...>
```

Properties file example is:

```
profile.context=foo.bar.MyCtxtClass
```

Customizer Option (customizer)

Use the `customizer` option to specify which customizer to use. Fully qualify the class name, such as in the following example:

```
-P-customizer=oracle.sqlj.runtime.util.OraCustomizer
```

If you do not set this option, then SQLJ will use the customizer specified in the SQLJ `-default-customizer` option. Unless set otherwise, this is the following:

```
oracle.sqlj.runtime.util.OraCustomizer
```

Command-line syntax is:

```
-P-customizer=customizer_class
```

Command-line example is:

```
-P-customizer=a.b.c.MyCustomizer
```

Properties file syntax is:

```
profile.customizer=customizer_class
```

Properties file example is:

```
profile.customizer=a.b.c.MyCustomizer
```

Default value is:

```
None
```

Customization JAR File Digests Option (digests)

When a JAR file is produced, the JAR utility can optionally include one or more *digests* for each entry, based on one or more specified algorithms, so that the integrity of the JAR file entries can later be verified. Digests are similar conceptually to checksums, for readers familiar with those.

If you are customizing profiles in a JAR file and want the JAR utility to add new digests (or update existing digests) when the JAR file is updated, use the `digests` option to specify a comma-delimited list of one or more algorithms. These are the algorithms that the JAR utility will use in creating the digests for each entry. The JAR utility produces one digest for each algorithm for each JAR file entry in the JAR manifest file. Specify algorithms as follows:

```
-P-digests=SHA,MD5
```

There must be no space on either side of the comma.

In this example, there will be two digests for each entry in the JAR manifest file: an `SHA` digest and an `MD5` digest.

> **Note:** Visit the Sun site more information about JAR manifest file.

Command-line syntax is:

```
-P-digests=algo1<,algo2,...>
```

Command-line example is:

```
-P-digests=SHA,MD5
```

Properties file syntax is:

```
profile.digests=algo1<,algo2,...>
```

Properties file example is:

```
profile.digests=SHA,MD5
```

Default value is:

```
SHA,MD5
```

Customization Help Option (help)

Use the `help` option to display the option lists of the customizer harness and the default customizer or a specified customizer. For the harness and Oracle customizer, this includes a brief description and the current setting of each option.

Display the option lists for the harness and default customizer as follows (where the default customizer is the Oracle customizer or whatever you have specified in the SQLJ `-default-customizer` option):

```
-P-help
```

Use the `help` option in conjunction with the `customizer` option to display the option list of a particular customizer, as follows:

```
-P-help -P-customizer=sqlj.runtime.profile.util.AuditorInstaller
```

Note:

- You can use the `-P-help` option on the SQLJ command line only, not in a SQLJ properties file.
- No customizations are performed if the `-P-help` flag is enabled, even if you specify profiles to customize on the command line.

Command-line syntax is:

```
-P-help <-P-customizer=customizer_class>
```

Command-line example is:

```
-P-help
```

Properties file syntax is:

```
NA
```

Properties file example is:

```
NA
```

Default value is:

```
None
```

Customization Verbose Option (verbose)

Use the `verbose` flag to instruct the harness to display status messages during customizations. These messages are written to the standard output device, wherever SQLJ writes its other messages.

Command-line syntax is:

```
-P-verbose<=true|false>
```

Command-line example is:

```
-P-verbose
```

Properties file syntax is:

```
profile.verbose<=true|false>
```

Properties file example is:

```
profile.verbose
```

Default value is:

```
false
```

Customizer Harness Options for Connections

This section describes connection options supported by the customizer harness. These are used as follows:

- The Oracle customizer uses database connections only for column definitions. If you do not enable the Oracle customizer `optcols` option, then there is no need to set the customizer harness `user`, `password`, `url`, and `driver` options.
- The `SQLCheckerCustomizer`, a specialized customizer that performs semantics-checking on profiles, uses the customizer harness `user`, `password`, `url`, and `driver` settings for online checking.

  Use `-P-verify` on the SQLJ command line to invoke this customizer.

> **Note:** Do not confuse the customizer harness `user`, `password`, `url`, and `driver` options with the translator options of the same names, which are for semantics-checking during the translation step. However, the translator settings are passed to the customizer for convenience, in case customization is to use the same connection as translation. Override these initial settings through the customizer harness options if you wish.

Customization User Option (user)

Set the `user` option to specify a database schema if your customizer uses database connections.

In addition to specifying the schema, you can optionally specify the password, URL, or both in your `user` option setting. The password is preceded by a forward-slash (/), and the URL is preceded by an "at" sign (@), as in the following examples:

```
-P-user=scott/tiger
```

```
-P-user=scott@jdbc:oracle:oci:@
-P-user=scott/tiger@jdbc:oracle:oci:@
```

> **Note:** When you use column definitions (`optcols` option), the user setting for the SQLJ translator is forwarded to the profile customizer as well, but you can use the customizer `user` option to override the translator setting.

Command-line syntax is:

```
-P-user=username</password><@url>
```

Command-line examples is:

```
-P-user=scott
-P-user=scott/tiger
-P-user=scott/tiger@jdbc:oracle:oci:@
```

Properties file syntax is:

```
profile.user=username</password><@url>
```

Properties file examples is:

```
profile.user=scott
profile.user=scott/tiger
profile.user=scott/tiger@jdbc:oracle:oci:@
```

Default value is:

```
null
```

Customization Password Option (password)

Use the `password` option if your customizer uses database connections.

The password can also be set with the `user` option, as described in "Customization User Option (user)" on page A-12.

> **Note:** When you use column definitions (`optcols` option), the password setting for the SQLJ translator is forwarded to the profile customizer as well, but you can use the customizer `password` option to override the translator setting.

Command-line syntax is:

```
-P-password=password
```

Command-line example is:

```
-P-password=tiger
```

Properties file syntax is:

```
profile.password=password
```

Properties file example is:

```
profile.password=tiger
```

Default value is:

```
null
```

Customization URL Option (url)

Use the `url` option if your customizer uses database connections.

The URL can also be set with the `user` option, as described in "Customization User Option (user)" on page A-12.

> **Note:** When you use column definitions (`optcols` option), the URL setting for the SQLJ translator is forwarded to the profile customizer as well, but you can use the customizer `url` option to override the translator setting.

Command-line syntax is:

```
-P-url=url
```

Command-line example is:

```
-P-url=jdbc:oracle:oci:@
```

Properties file syntax is:

```
profile.url=url
```

Properties file example is:

```
profile.url=jdbc:oracle:oci:@
```

Default value is:

```
jdbc:oracle:oci:@
```

Customization JDBC Driver Option (driver)

Use the `driver` option to register a comma-delimited list of JDBC driver classes if your customizer uses database connections. For example:

```
-P-driver=sun.jdbc.odbc.JdbcOdbcDriver,oracle.jdbc.OracleDriver
```

There must be no space on either side of the comma.

Command-line syntax is:

```
-P-driver=dvr_class1<,dvr_class2,...>
```

Command-line example is:

```
-P-driver=sun.jdbc.odbc.JdbcOdbcDriver
```

Properties file syntax is:

```
profile.driver=dvr_class1<,dvr_class2,...>
```

Properties file example is:

```
profile.driver=sun.jdbc.odbc.JdbcOdbcDriver
```

Default value is:

```
oracle.jdbc.OracleDriver
```

Customizer Harness Options that Invoke Specialized Customizers

The customizer harness supports the following options that invoke specialized customizers:

- `debug`: This invokes the `AuditorInstaller` customizer, used in debugging.
- `print`: This invokes a customizer that prints a text version of a profile.
- `verify`: This invokes the `SQLCheckerCustomizer` customizer, which performs semantics-checking on a profile.

Important: Because each of these options invokes a customizer, and only one customizer can run in a single execution of SQLJ, you cannot perform any other customization when you use any of these options.

You also cannot use more than one of `print`, `debug`, or `verify` simultaneously.

Specialized Customizer: Profile Debug Option (debug)

The `debug` option runs a specialized customizer, called the `AuditorInstaller`, that inserts debugging statements into profiles. Use this option in conjunction with a SQLJ command line file list to insert debugging statements into the specified profiles. These profiles must already be customized from a previous SQLJ run.

For detailed information about this customizer, including additional options that it supports, see "AuditorInstaller Customizer for Debugging" on page A-33.

The debugging statements will execute during SQLJ run time (when someone runs your application), displaying a trace of method calls and values returned.

Following are examples of how to specify the `debug` option:

```
% sqlj -P-debug Foo_SJProfile0.ser Bar_SJProfile0.ser

% sqlj -P-debug *.ser
```

Command-line syntax is:

```
sqlj -P-debug profile_list
```

Command-line example is:

```
sqlj -P-debug Foo_SJProfile*.ser
```

Properties file syntax is:

```
profile.debug
```

(Also specify profiles in the SQLJ file list.)

Properties file example is:

```
profile.debug
```

Default value is:

```
NA
```

Specialized Customizer: Profile Print Option (print)

The `print` option runs a specialized customizer that prints profiles in text format. Use this option in conjunction with a SQLJ command line file list to output the contents of one or more specified profiles. The output goes to the standard SQLJ output device, typically the user screen. For example:

```
% sqlj -P-print Foo_SJProfile0.ser Bar_SJProfile0.ser
```

Use the following command, if you want to see all the customizer options:

```
% sqlj -P-print *.ser
```

The output of the preceding command is like the following:

```
printing contents of profile Sample_SJProfile0
created 1154609279331 (8/3/06 5:47 AM)
associated context is Mycontext1
profile loader is sqlj.runtime.profile.DefaultLoader@12a3793
contains one customization
OracleCustomization Options  :
Version is :2300
Cstmtcache :5
Ccompat    :false
Cforce     :false
Coptcols   :false
Coptparams :false
Coptparamdefaults:null
CshowSQL   :false
Csummary   :false
CuserSQL   :true

#sql { SELECT a    FROM test WHERE name= ?  }
setFixedchar is enabled
Ncharconv  is disabled

#sql { commit }
setFixedchar is disabled
Ncharconv  is disabled
……
```

Command-line syntax is:

```
sqlj -P-print profile_list
```

Command-line example is:

```
sqlj -P-print Foo_SJProfile*.ser
```

Properties file syntax is:

```
profile.print
```

(Also specify profiles in SQLJ file list.)

Properties file example is:

```
profile.print
```

Default value is:

NA

Specialized Customizer: Profile Semantics-Checking Option (verify)

The `verify` option runs a specialized customizer, called the `SQLCheckerCustomizer`, that performs semantics-checking on a profile. This is equivalent to the semantics-checking that is performed on source code during translation. The profile will have been created during a previous execution of the SQLJ translator.

This option is useful for checking semantics against the run time database, after deployment, and after the source code may no longer be available.

For detailed information about this customizer, including additional options that it supports, see "SQLCheckerCustomizer for Profile Semantics-Checking" on page A-30.

> **Note:** For online semantics-checking of the profile, you must also use the customizer harness `user`, `password`, and `url` options.

Following are examples of how to specify the `verify` option. Both of these examples use the `SQLCheckerCustomizer` default semantics-checker, which employs online checking through the specified database connection. (The first is a single wraparound command.)

```
% sqlj -P-verify -P-user=scott -P-password=tiger -P-url=jdbc:oracle:oci:@
Foo_SJProfile0.ser Bar_SJProfile0.ser

% sqlj -P-verify -P-user=scott -P-password=tiger -P-url=jdbc:oracle:oci:@ *.ser
```

Command-line syntax is:

```
sqlj -P-verify <conn params> profile_list
```

Command-line example is:

```
sqlj -P-verify <conn params> Foo_SJProfile*.ser
```

Properties file syntax is:

```
profile.verify
```

(You must also specify profiles, and typically customizer harness connection options, in the SQLJ command line.)

Properties file example is:

```
profile.verify
```

Default value is:

```
NA
```

Overview of Customizer-Specific Options

You can set customizer-specific options, such as options for the Oracle customizer, on the SQLJ command line or in a SQLJ properties file. The syntax is similar to that for setting customizer harness options.

Set a customizer option on the SQLJ command line by preceding it with:

```
-P-C
```

Alternatively, you can set it in a SQLJ properties file by preceding it with:

```
profile.C
```

This option syntax is also discussed in "Options to Pass to the Profile Customizer (-P)" on page 8-50 and "Properties File Syntax" on page 8-13.

The remainder of this section discusses features of the Oracle customizer, which supports several options. Most of these options are boolean and are enabled as follows:

```
-P-Coption
```

or:

```
-P-Coption=true
```

Boolean options are disabled by default, but you can explicitly disable them with:

```
-P-Coption=false
```

Numeric or string options are set similarly:

```
-P-Coption=value
```

Oracle Customizer Options

This section describes options that are specific to the Oracle customizer, beginning with an overview of the options supported.

Options Supported by the Oracle Customizer

The Oracle customizer implements the following options:

- `compat`: Display version-compatibility information.
- `force`: Instruct the customizer to customize even if a valid customization already exists.
- `optcols`: Enable iterator column type and size definitions to optimize performance.
- `optparams`: Enable parameter size definitions to optimize JDBC resource allocation (used in conjunction with `optparamdefaults`).
- `optparamdefaults`: Set parameter size defaults for particular data types (used in conjunction with `optparams`).
- `fixedchar`: Enable `CHAR` comparisons with blank padding for `WHERE` clauses.
- `showSQL`: Display SQL statement transformations.
- `stmtcache`: Set the statement cache size (the number of statements that can be cached for each connection during run time) for performance optimization, or set it to zero to disable statement caching.

- `summary`: Display a summary of Oracle features used in your application.

Any output displayed by these options is written to the standard output device, wherever SQLJ writes its other messages.

Oracle Customizer Version Compatibility Option (compat)

Use the `compat` flag to instruct the Oracle customizer to display information about compatibility of your application with different versions of Oracle Database and Oracle JDBC drivers. This can be accomplished either during a full SQLJ translation run or on profiles previously created.

For example, to see compatibility output when translating and customizing the application `MyApp`:

```
% sqlj <...SQLJ options...> -P-Ccompat MyApp.sqlj
```

In this example, the `MyApp` profiles will be created, customized, and checked for compatibility in a single running of SQLJ.

To see compatibility output for `MyApp` profiles previously created:

```
% sqlj <...SQLJ options...> -P-Ccompat MyApp_SJProfile*.ser
```

In this example, the `MyApp` profiles were created (and possibly customized) in a previous running of SQLJ and will be customized (if needed) and checked for compatibility in the above running of SQLJ.

Following are two output samples from a -P-C`compat` setting when using the default Oracle customizer. The first example indicates that the application can be used with all Oracle JDBC driver versions:

```
MyApp_SJProfile0.ser: Info: compatible with all Oracle JDBC drivers
```

This second example indicates that the application can be used only with the JDBC implementation from an Oracle 8.1.x or later release:

```
MyApp_SJProfile0.ser: Info: compatible with Oracle 8.1 or later JDBC driver
```

> **Note:** If customization does not take place because a valid previous customization is detected, the `compat` option reports compatibility regardless.

Command-line syntax is:

```
-P-Ccompat<=true|false>
```

Command-line example is:

```
-P-Ccompat
```

Properties file syntax is:

```
profile.Ccompat<=true|false>
```

Properties file example is:

```
profile.Ccompat
```

Default value is:

```
false
```

Oracle Customizer Force Option (force)

Use the `force` flag to instruct the Oracle customizer to force the customization of a given profile (specified on the command line) even if a valid customization already exists in that profile. For example:

```
% sqlj -P-Cforce MyApp_SJProfile*.ser
```

This will customize all the `MyApp` profiles, regardless of whether they have already been customized. Otherwise, by default, the Oracle customizer will not reinstall over a previously existing customization unless the previous one had been installed with an older version of the customizer.

Command-line syntax is:

```
-P-Cforce<=true|false>
```

Command-line example is:

```
-P-Cforce
```

Properties file syntax is:

```
profile.Cforce<=true|false>
```

Properties file example is:

```
profile.Cforce
```

Default value is:

```
false
```

Oracle Customizer Column Definition Option (optcols)

Use the `optcols` flag to instruct the Oracle customizer to determine types and sizes of iterator or result set columns and add this information to the profile. This enables the SQLJ run time to automatically register the columns with the Oracle JDBC driver when your application runs, saving round trips to Oracle depending on the particular driver implementation. Specifically, this is effective for the Thin driver and positional iterators.

For an overview of column definitions, see "Column Definitions" on page 10-16.

An error will be generated if you enable the Oracle customizer `optcols` option without setting the user name, password, and URL for a database connection. You can accomplish this through the translator `-user`, `-password`, and `-url` options, which are forwarded to the customizer during ISO standard code generation, or directly through the customizer `user`, `password`, and `url` options.

The customizer does not have to connect to the same schema or even the same database that your application will connect to at run time, but the relevant columns will have to be in the same order and of identical types and sizes to avoid run time errors.

For information about the customizer harness connection options, see the `user`, `password`, `url`, and `driver` sections under "Overview of Customizer Harness Options" on page A-7.

Note: You can use the SQLJ translator `-optcols` option instead. This sets the customizer option automatically. (And for Oracle-specific code generation, which uses no profiles, you *must* use the translator option instead.) See "Column Definitions (-optcols)" on page 8-41.

That section also has some additional conceptual information.

You can enable or disable the customizer `optcols` flag on the SQLJ command line or in a properties file.

Enable it on the command line as follows:

```
-P-Coptcols
```

or:

```
-P-Coptcols=true
```

This flag is disabled by default, but you can also disable it explicitly. Disable it on the command line as follows:

```
-P-Coptcols=false
```

Column definitions require the customizer to make a database connection to examine columns of tables being queried, so the customizer harness `user`, `password`, and `url` options must be set appropriately (as well as the customizer harness `driver` option if you are not using the default `OracleDriver` class). For example:

```
% sqlj <...SQLJ options...> -P-user=scott/tiger@jdbc:oracle:oci:@ -P-Coptcols MyApp.sqlj
```

Note that as with the SQLJ translator, you can optionally set the password and URL in the `user` option instead of in the `password` and `url` options.

Alternatively, you can insert column definitions into a previously existing profile. In this case you must also use the Oracle customizer `force` option to force a recustomization:

```
% sqlj -P-user=scott/tiger@jdbc:oracle:oci:@ -P-Cforce -P-Coptcols MyApp_SJProfile*.ser
```

You also can insert column definitions into previously existing profiles in a JAR file:

```
% sqlj -P-user=scott/tiger@jdbc:oracle:oci:@ -P-Cforce -P-Coptcols MyAppProfiles.jar
```

When you run the Oracle customizer with its `optcols` flag enabled, either during translation and creation of a new profile or during customization of an existing profile, you can also enable the customizer harness `verbose` flag. This will instruct the Oracle customizer to display information about what iterators and result sets are being processed and what their column type and size definitions are. For example:

```
% sqlj -P-user=scott/tiger@jdbc:oracle:oci:@ -P-verbose -P-Cforce -P-Coptcols MyApp_SJProfile*.ser
```

For general information about the `verbose` flag, see that section under "Overview of Customizer Harness Options" on page A-7.

You can execute the Oracle customizer with its `summary` flag enabled on an existing profile to determine if column definitions have been added to that profile:

```
% sqlj -P-Csummary MyApp_SJProfile*.ser
```

For general information about the `summary` flag, see that section under "Overview of Customizer-Specific Options" on page A-18.

Command-line syntax is:

```
-P-Coptcols<=true|false>
```

Command-line example is:

```
-P-Coptcols
```

Properties file syntax is:

```
profile.Coptcols<=true|false>
```

Properties file example is:

```
profile.Coptcols
```

Default value is:

```
false
```

Oracle Customizer Parameter Definition Option (optparams)

Use the `optparams` flag to enable parameter size definitions. If this flag is enabled, SQLJ will register your input and output parameters (host variables) to optimize JDBC resource allocations according to sizes you specify.

For an overview of parameter size definitions and a discussion of source code hints, see "Parameter Size Definitions" on page 10-17.

> **Note:** You can use the SQLJ translator `-optparams` option instead. This sets the customizer option automatically. (And for Oracle-specific code generation, which uses no profiles, you *must* use the translator option instead.) See "Parameter Definitions (-optparams)" on page 8-43.
>
> That section also has some additional conceptual information.

You can enable or disable the `optparams` flag on the command line or in a SQLJ properties file.

Enable it on the command line as follows:

```
-P-Coptparams
```

or:

```
-P-Coptparams=true
```

This flag is disabled by default, but you can also disable it explicitly. Disable it on the command line as follows:

```
-P-Coptparams=false
```

> **Note:** Unlike the `optcols` option, the `optparams` option does not require a database connection by the customizer, because you are providing the size specifications yourself.

Following is a command-line example (omitting a setting for the `optparamdefaults` option, which is discussed in the next section):

```
% sqlj <...SQLJ options...> -P-Coptparams -P-Coptparamdefaults=defaults_string MyApp.sqlj
```

Alternatively, to enable parameter size definitions for a previously existing profile:

```
% sqlj -P-Coptparams -P-Coptparamdefaults=
defaults_string
MyApp_SJProfile*.ser
```

You can also use previously existing profiles in a JAR file:

```
% sqlj -P-Coptparams -P-Coptparamdefaults=
defaults_string
MyAppProfiles.jar
```

Command-line syntax is:

```
-P-Coptparams<=true|false>
```

Command-line example is:

```
-P-Coptparams
```

Properties file syntax is:

```
profile.Coptparams<=true|false>
```

Properties file example is:

```
profile.Coptparams
```

Default value is:

```
false
```

Oracle Customizer Parameter Default Size Option (optparamdefaults)

If you enable the `optparams` option to set parameter sizes, use the `optparamdefaults` option as desired to set default sizes for specified data types. If `optparams` is *not* enabled, then any `optparamdefaults` setting is ignored.

For an overview of parameter size definitions and a discussion of source code hints, see "Parameter Size Definitions" on page 10-17.

> **Note:** You can use the SQLJ translator `-optparamdefaults` option instead. This sets the customizer option automatically. (And for Oracle-specific code generation, which uses no profiles, you *must* use the translator option instead.) See "Parameter Default Size (-optparamdefaults)" on page 8-44.
>
> That section also has important additional conceptual and syntax information. Functionality of the two options is equivalent.

You can set the `optparamdefaults` flag on the command line or in a SQLJ properties file.

Set it on the command line as follows:

```
-P-Coptparamdefaults=datatype1(size1),datatype2(size2),...
```

Following is a command-line example, including the `optparams` setting as well:

```
% sqlj <..SQLJ options..> -P-Coptparams -P-Coptparamdefaults=CHAR_TYPE(50),RAW_TYPE(500),CHAR(10) MyApp.sqlj
```

The syntax is explained in "Parameter Default Size (-optparamdefaults)" on page 8-44.

Alternatively, you can specify parameter size defaults for a previously existing profile, in which case you must also use the Oracle customizer `force` option to force a recustomization:

```
% sqlj -P-Cforce -P-Coptparams -P-Coptparamdefaults=CHAR_TYPE(50),RAW_TYPE(500),CHAR(10) MyApp_SJProfile*.ser
```

You also can specify parameter size defaults for previously existing profiles in a JAR file:

```
% sqlj -P-Cforce -P-Coptparams -P-Coptparamdefaults=CHAR_TYPE(50),RAW_TYPE(500),CHAR(10) MyAppProfiles.jar
```

Note: If at run time the actual size exceeds the registered size of any parameter, run time errors will occur.

Command-line syntax is:

```
-P-Coptparamdefaults=defaults_string
```

Command-line example is:

```
-P-Coptparamdefaults=VAR%(50),LONG%(500),RAW_TYPE()
```

Properties file syntax is:

```
profile.Coptparamdefaults=defaults_string
```

Properties file example is:

```
profile.Coptparamdefaults=VAR%(50),LONG%(500),RAW_TYPE()
```

Default value is:

```
null
```

Oracle Customizer CHAR Comparisons with Blank Padding (fixedchar)

Set this flag to `true` to account for blank padding in `CHAR` database columns when binding character strings for `WHERE` clause comparisons. This way, for example, "mystring" would compare positively against "mystring ".

Here is an example of Oracle customizer `fixedchar` usage:

```
% sqlj -P-Cfixedchar MyProgram.sqlj AnotherProg.java ...
```

Note:

- You can use the SQLJ translator `-fixedchar` option instead. This sets the customizer option automatically. (And for Oracle-specific code generation, which uses no profiles, you *must* use the translator option instead.) See "CHAR Comparisons with Blank Padding (-fixedchar)" on page 8-46. That section also has some additional conceptual information.
- If you also enable the Oracle customizer `summary` flag, the number of usages of the Oracle `setFixedCHAR()` API (used behind the scenes for `fixedchar` functionality) will be displayed. See "Oracle Customizer Summary Option (summary)" on page A-27 for an example.

Command-line syntax is:

```
-P-Cfixedchar<=true|false>
```

Command-line example is:

```
-P-Cfixedchar
```

Properties file syntax is:

```
profile.Cfixedchar<=true|false>
```

Properties file example is:

```
profile.Cfixedchar
```

Default value is:

```
false
```

Oracle Customizer Show-SQL Option (showSQL)

Use the `showSQL` flag to display any SQL statement transformations performed by the Oracle customizer. Such transformations are necessary in cases where SQLJ supports syntax that Oracle10*i* does not.

To show SQL transformations when translating and customizing the application `MyApp`:

```
% sqlj <...SQLJ options...> -P-CshowSQL MyApp.sqlj
```

In this example, the `MyApp` profiles will be created and customized and their SQL transformations displayed in a single running of SQLJ.

To show SQL transformations when customizing `MyApp` profiles previously created:

```
% sqlj <...SQLJ options...> -P-CshowSQL MyApp_SJProfile*.ser
```

In this example, the `MyApp` profiles were created (and possibly customized) in a previous running of SQLJ and will be customized (if needed) and have their SQL transformations displayed in the above running of SQLJ.

The `showSQL` output might include an entry such as this:

```
MyApp.sqlj:14: Info: <<<NEW SQL>>> #sql {BEGIN  ? := VALUES(tkjsSET_f1); END};
```

```
in file MyApp, line 14, we had:

#sql {set :v1= VALUES(tkjsSET_f1) };
```

SQLJ supports the `SET` statement, but Oracle10*i* does not. During customization, the Oracle customizer replaces the `SET` statement with an equivalent PL/SQL block.

> **Note:** If customization does not take place because a valid previous customization is detected, the `showSQL` option shows SQL transformations regardless.

Command-line syntax is:

```
-P-CshowSQL<=true|false>
```

Command-line example is:

```
-P-CshowSQL
```

Properties file syntax is:

```
profile.CshowSQL<=true|false>
```

Properties file example is:

```
profile.CshowSQL
```

Default value is:

```
false
```

Oracle Customizer Statement Cache Size Option (stmtcache)

Use the Oracle customizer `stmtcache` option to set the statement cache size—the number of statements that can be cached for each database connection as your application runs—or to disable statement caching.

The default statement cache size is 5. For an overview of statement caching, see "Statement Caching" on page 10-3.

> **Important:** With the default Oracle-specific code generation (`-codegen=oracle`), SQLJ does not produce profiles and skips the customization step. In this case, use connection context methods to control SQLJ statement caching. See "Connection Context Methods for Statement Caching (Oracle-Specific Code)" on page 10-3.

You can set the statement cache size on the command line or in a properties file.

To use the command line to set the statement cache size to 15 (for example) for the application `MyApp`:

```
% sqlj <...SQLJ options...> -P-Cstmtcache=15 MyApp.sqlj
```

To disable statement caching, set the cache size to 0:

```
% sqlj <...SQLJ options...> -P-Cstmtcache=0 MyApp.sqlj
```

You also can alter the statement cache size in an existing profile without retranslating the application, but you must also use the Oracle customizer `force` option to force a recustomization, as follows:

```
% sqlj -P-Cforce -P-Cstmtcache=15 MyApp_SJProfile0.ser
```

If you have multiple profiles, you can set their statement cache sizes individually by running SQLJ separately for each profile, after you have translated your application:

```
% sqlj -P-Cforce -P-Cstmtcache=10 MyApp_SJProfile0.ser
% sqlj -P-Cforce -P-Cstmtcache=15 MyApp_SJProfile1.ser
% sqlj -P-Cforce -P-Cstmtcache=0 MyApp_SJProfile2.ser
```

Of course, you must determine which profile corresponds to each of your connection context classes. This is determined as follows: profile 0 will correspond to the connection context class used for the first executable statement in your application; profile 1 will correspond to the connection context class used for the first executable statement that does not use the first connection context class, and so on. You can verify the correlation by using the customizer harness `print` option to examine each profile.

Command-line syntax is:

```
-P-Cstmtcache=value
```

Command-line example is:

```
-P-Cstmtcache=10
```

Properties file syntax is:

```
profile.Cstmtcache=value
```

Properties file example is:

```
profile.Cstmtcache=10
```

Default value is:

```
5
```

Oracle Customizer Summary Option (summary)

Use the `summary` flag to instruct the Oracle customizer to display a summary of Oracle features used in an application being translated, or in specified profile files. This is useful in identifying features that would prevent portability to other platforms and can be accomplished either during a full SQLJ translation run or on profiles previously created.

To see summary output when translating and customizing the application `MyApp`:

```
% sqlj <...SQLJ options...> -P-Csummary MyApp.sqlj
```

In this example, the `MyApp` profiles will be created, customized, and summarized in a single running of SQLJ.

To see summary output for `MyApp` profiles previously created:

```
% sqlj <...SQLJ options...> -P-Csummary MyApp_SJProfile*.ser
```

In this example, the `MyApp` profiles were created (and possibly customized) in a previous running of SQLJ and will be customized (if needed) and summarized in the above running of SQLJ.

Following are two samples resulting from a `-P-Csummary` setting when using the default Oracle customizer. The first example indicates no Oracle features are used:

```
MyApp_SJProfile0.ser: Info: Oracle features used:
MyApp_SJProfile0.ser: Info: * none
```

This second example indicates that Oracle features *are* used—several Oracle extended data types from the `oracle.sql` package—and lists them:

```
MyApp_SJProfile0.ser: Info: Oracle features used:
MyApp_SJProfile0.ser: Info: * oracle.sql.NUMBER: 2
MyApp_SJProfile0.ser: Info: * oracle.sql.DATE: 2
MyApp_SJProfile0.ser: Info: * oracle.sql.CHAR: 2
MyApp_SJProfile0.ser: Info: * oracle.sql.RAW: 2
```

The following example prints out the number of usages of the Oracle `setFixedCHAR()` API (enabled through the Oracle customizer `fixedchar` option, to account for blank padding when binding a string into a `WHERE` clause for comparison against `CHAR` data):

```
% sqlj -P-Cfixedchar -P-Csummary -P-Cforce *.ser
FC_SJProfile0.ser: Info: re-installing Oracle customization
FC_SJProfile0.ser: Info: Oracle features used:
FC_SJProfile0.ser: Info: * setFixedCHAR(): 4
```

> **Note:** If customization does not take place because a valid previous customization is detected, the `summary` option produces a summary regardless.

Command-line syntax is:

```
-P-Csummary<=true|false>
```

Command-line example is:

```
-P-Csummary
```

Properties file syntax is:

```
profile.Csummary<=true|false>
```

Properties file example is:

```
profile.Csummary
```

Default value is:

```
false
```

Options for Other Customizers

The Oracle SQLJ implementation provides additional, specialized customizers described later in this chapter. These customizers also have command-line options:

- `SQLCheckerCustomizer` (for profile semantics-checking): See "SQLCheckerCustomizer for Profile Semantics-Checking" on page A-30 for general information, and "SQLCheckerCustomizer Options" on page A-32 for information about its options.

- `AuditorInstaller` (for debugging): See "AuditorInstaller Customizer for Debugging" on page A-33 for general information, and "AuditorInstaller Options" on page A-36 for information about its options.

SQLJ Translator Options for Profile Customization

The following SQLJ translator options relate to profile customization and are described elsewhere in this manual:

- `-default-customizer`: Specify the default profile customizer to use if none is specified in the customizer harness `-customizer` option.

  See "Default Profile Customizer (-default-customizer)" on page 8-67.

- `-profile`: Specify whether to customize during this running of SQLJ.

  See "Profile Customization Flag (-profile)" on page 8-52.

JAR Files for Profiles

As discussed previously, you can specify a JAR file on the SQLJ command line in order to customize any profiles that the JAR file contains.

Note:

- Remember that you can specify `.sqlj` or `.java` files or both on the SQLJ command line for standard SQLJ processing, or you can specify `.ser` or `.jar` files or both on the command line for customization only, but not both categories.
- It is permissible for the `.jar` file to contain files that are not profiles. Any file whose manifest entry indicates that the file is not a profile will be ignored during customization.
- The `.jar` file is used as the class-loading context for each profile it contains. If a profile contains a reference to a class contained within the `.jar` file, then that class is loaded from the `.jar` file. If a profile contains a reference to a class not in the `.jar` file, then the system class loader will find and load the class according to your classpath, as usual.

JAR File Requirements

There are requirements for the manifest entry of each profile.

Create a plain text file with two lines for each profile that will be included in the JAR file. One line starts with "Name:", followed by the path or package and name. The other line is the following:

```
SQLJProfile: TRUE
```

The two lines must be consecutive (no blank line in between), and there must be a blank line preceding line-pairs for additional profiles.

Use the JAR utility `-m` option to input this file.

For example, presume your `MyApp` application (in the directory `foo/bar`) has three profiles, and you will be creating a JAR file that will include these profiles. Complete the following steps:

1. Create a text file with the following eight lines (including the blank lines used as separators):

```
Name: foo/bar/MyApp_SJProfile0.ser
SQLJProfile: TRUE

Name: foo/bar/MyApp_SJProfile1.ser
SQLJProfile: TRUE

Name: foo/bar/MyApp_SJProfile2.ser
SQLJProfile: TRUE
```

   Presume you call this file `MyAppJarEntries.txt`.

2. When you run `jar` to create the JAR file, use the `-m` option to input your text file as follows (presume you want to call the JAR file `myjarfile.jar`):

```
% jar -cvfm myjarfile.jar MyAppJarEntries.txt foo/bar/MyApp_SJProfile*.ser foo/bar/*.class
```

As the JAR utility constructs the manifest during creation of the JAR file, it reads your text file and inserts the `SQLJProfile: TRUE` line into the manifest entry of each profile. It accomplishes this by matching the names in the manifest with the names you specify in your text file.

JAR File Logistics

When you specify a JAR file on the SQLJ command line, each profile in the JAR file is deserialized and customized.

A JAR file is successfully customized only if all the profiles it contains are successfully customized. After a successful customization, each profile has been reserialized into a `.ser` file, the JAR file has been modified to replace the original `.ser` files with the customized `.ser` files, and the JAR file manifest has been updated to indicate the new entries.

If any error is encountered in the customization of any profile in a JAR file, then the JAR file customization has failed, and the original JAR file is left completely unchanged.

> **Note:** If you use signature files for authentication, the signature files that appeared in the original JAR file will appear unchanged in the updated JAR file. You are responsible for resigning the new JAR file if the profiles require signing.

SQLCheckerCustomizer for Profile Semantics-Checking

Oracle provides a special customizer, `SQLCheckerCustomizer`, that will perform semantics-checking on a profile that was produced during previous execution of the translator. This semantics-checking is similar to what is usually performed during translation of the source code.

This is particularly valuable when the database to be used at run time is not available for semantics-checking during translation. In these circumstances, you can use `SQLCheckerCustomizer` after deployment, against the run time database, typically in a scenario where the source code is no longer available.

You can specify the checker to use. If you accept the default `OracleChecker` front end, `SQLCheckerCustomizer` will perform online semantics-checking using an appropriate online checker.

Note: For online semantics-checking of the profile, you must also specify connection parameters using the customizer harness connection options.

Invoking SQLCheckerCustomizer with the Customizer Harness verify Option

Following are examples of how to specify the Oracle customizer harness `verify` option to run `SQLCheckerCustomizer` in its default mode. Because it defaults to an online checker, you typically must provide connection parameters through the customizer harness `user`, `password`, and `url` options. (The first example is a single wraparound command line.)

```
% sqlj -P-verify -P-user=scott -P-password=tiger -P-url=jdbc:oracle:oci:@
Foo_SJProfile0.ser Bar_SJProfile0.ser

% sqlj -P-verify -P-user=scott -P-password=tiger -P-url=jdbc:oracle:oci:@ *.ser
```

The `verify` option results in the customizer harness instantiating and invoking the following class:

```
sqlj.runtime.profile.util.SQLCheckerCustomizer
```

This class coordinates semantics-checking of the SQL operations in the profile. You can specify a semantics-checker or accept the default `OracleChecker` semantics-checker front end.

The `-P-verify` option is equivalent to the following:

```
-P-customizer=sqlj.runtime.profile.util.SQLCheckerCustomizer
```

This overrides the customizer specified in the SQLJ `-default-customizer` option.

Note:

- As with any Oracle customizer, help output and an option list will be provided if you specify `-P-verify` together with `-P-help` on the SQLJ command line.
- It is important to realize that because the `verify` option invokes a customizer, and only one customizer can run in any single running of SQLJ, you cannot do any other customization when you use this option.
- You also cannot use more than one of `-P-print`, `-P-debug`, and `-P-verify` simultaneously, because each of these invokes a specialized customizer.

Command-line syntax is:

```
sqlj -P-verify <conn params> profile_list
```

Command-line example is:

```
sqlj -P-verify <conn params> Foo_SJProfile*.ser
```

Properties file syntax is:

```
profile.verify
```

(You must also specify profiles, and typically customizer harness connection options, in the SQLJ command line.)

Properties file example is:

```
profile.verify
```

Default value is:

```
NA
```

SQLCheckerCustomizer Options

Like any customizer, `SQLCheckerCustomizer` has its own options, which can be set using the `-P-C` prefix on the SQLJ command line or the `profile.C` prefix in a SQLJ properties file.

`SQLCheckerCustomizer` supports the following options:

- `checker`: Specify the semantics-checker to use. The default is the `OracleChecker` front end, as for checking during SQLJ translation.
- `warn`: Specify the categories of warnings and messages to display during semantics-checking of the profile. This is equivalent to the SQLJ `-warn` flag for warning categories during translation-time semantics-checking, supports the same settings, and uses the same defaults. See "Translator Warnings (-warn)" on page 8-33.

SQLCheckerCustomizer Semantics-Checker Option (checker)

The `checker` option enables you to specify the semantics-checker to use in checking the SQL operations in a profile.

This defaults to the Oracle semantics-checker front end, `oracle.sqlj.checker.OracleChecker`, which for `SQLCheckerCustomizer` chooses an appropriate online checker for your environment. For more information about `OracleChecker`, see "Semantics-Checkers and the OracleChecker Front End (default checker)" on page 8-56.

Following is a full command-line example, showing how to use the `SQLCheckerCustomizer checker` option, in conjunction with the customizer harness `verify` option and connection options.

```
% sqlj -P-verify -P-user=scott -P-password=tiger -P-url=jdbc:oracle:oci:@
-P-Cchecker=abc.def.MyChecker *.ser
```

(This is a single wraparound command line.)

Command-line syntax is:

```
-P-Cchecker=checker_class
```

Command-line example is:

```
-P-Cchecker=a.b.c.MyChecker
```

Properties file syntax is:

```
profile.Cchecker=checker_class
```

Properties file example is:

```
profile.Cchecker=a.b.c.MyChecker
```

Default value is:

```
oracle.sqlj.checker.OracleChecker
```

SQLCheckerCustomizer Warnings Option (warn)

The `warn` option is equivalent to the SQLJ translator `-warn` option, enabling you to choose the categories of warnings and messages to be displayed as semantics-checking is performed on a profile.

For a complete description of the functionality and possible settings of these options, see "Translator Warnings (-warn)" on page 8-33.

This defaults to the `all,noverbose,noportable` settings, resulting in all warning categories except `verbose` and `portable` being enabled. You will receive any warnings regarding inheritance hierarchy requirements, data precision, conversion loss for nullable data, and strict matching for named iterators. These are the same defaults as for warnings during SQLJ translation.

Following is a full command-line example showing how to use the `SQLCheckerCustomizer warn` option, in conjunction with the customizer harness `verify` option and connection options. This would result in only portability warnings being displayed.

```
% sqlj -P-verify -P-user=scott -P-password=tiger -P-url=jdbc:oracle:oci:@
-P-Cwarn=none,portable *.ser
```

(This is a single wraparound command line.)

Command-line syntax is:

```
-P-Cwarn=comma-delimited_list_of_flags
```

Command-line example is:

```
-P-Cwarn=none,verbose
```

Properties file syntax is:

```
profile.Cwarn=comma-delimited_list_of_flags
```

Properties file example is:

```
profile.Cwarn=none,verbose
```

Default value is:

```
all,noverbose,noportable
```

AuditorInstaller Customizer for Debugging

For ISO code generation, SQLJ provides a special customizer, `AuditorInstaller`. This customizer will insert sets of debugging statements, known as *auditors*, into profiles specified on the SQLJ command line. These profiles must already exist from previous customization.

The debugging statements will execute during SQLJ run time (when someone runs your application), displaying a trace of method calls and values returned.

Use the customizer harness `debug` option, preceded by `-P-` as with any general customization option, to insert the debugging statements. (Syntax for this option is discussed in "Invoking AuditorInstaller with the Customizer Harness debug Option" on page A-34.)

Overview of Auditors and Code Layers

When an application is customized, the Oracle customizer implements profiles in *layers* of code (typically less than five) for different levels of run time functionality. The deepest layer uses straight Oracle JDBC calls and implements any of your SQLJ statements that can be executed through JDBC functionality. Each higher layer is a specialized layer for some category of SQLJ functionality that is not supported by JDBC and so must be handled specially by the SQLJ run time. For example, a layer for iterator conversion statements (`CAST`) is used to convert JDBC result sets to SQLJ iterators. Another layer is used for assignment statements (`SET`).

At run time, each SQLJ executable statement is first passed to the shallowest layer and then passed, layer-by-layer, until it reaches the layer that can process it (usually the deepest layer, which executes all JDBC calls).

You can install debugging statements at only one layer during a single execution of `AuditorInstaller`. Each set of debugging statements installed at a particular layer of code is referred to as an individual *auditor*. During run time, an auditor is activated whenever a call is passed to the layer at which the auditor is installed.

Any one of the specialized code layers above the JDBC layer is usually of no particular interest during debugging, so it is typical to install an auditor at either the deepest layer or the shallowest layer. If you install an auditor at the shallowest layer, its run time debugging output will be a trace of method calls resulting from all your SQLJ executable statements. If you install an auditor at the deepest layer, its run time output will be a trace of method calls from all your SQLJ executable statements that result in JDBC calls.

Use multiple executions of `AuditorInstaller` to install auditors at different levels. You might want to do that to install auditors at both the deepest layer and the shallowest layer, for example.

See "AuditorInstaller Depth Option (depth)" on page A-36 for information about how to specify the layer at which to install an auditor.

Invoking AuditorInstaller with the Customizer Harness debug Option

Following are examples of how to specify the Oracle customizer harness `debug` option to run `AuditorInstaller` in its default mode:

```
% sqlj -P-debug Foo_SJProfile0.ser Bar_SJProfile0.ser

% sqlj -P-debug *.ser

% sqlj -P-debug myappjar.jar
```

The `debug` option results in the customizer harness instantiating and invoking the following class:

```
% sqlj.runtime.profile.util.AuditorInstaller
```

This class performs the work of inserting the debugging statements.

The `-P-debug` option is equivalent to the following:

```
-P-customizer=sqlj.runtime.profile.util.AuditorInstaller
```

This overrides the customizer specified in the SQLJ `-default-customizer` option.

Be aware of the following:

- To run an application with auditors installed, the SQLJ file `translator.jar` must be in your classpath. (Usually, running a already translated SQLJ application requires only a runtime library.)
- As with any Oracle customizer, help output and an option list will be provided if you specify `-P-debug` together with `-P-help` on the SQLJ command line.
- It is important to realize that because the `debug` option invokes a customizer, and only one customizer can run in any single running of SQLJ, you cannot perform any other customization when you use this option.
- You also cannot use more than one of `-P-print`, `-P-debug`, and `-P-verify` simultaneously, because each of these invokes a specialized customizer.

Command-line syntax is:

```
sqlj -P-debug profile_list
```

Command-line example is:

```
sqlj -P-debug Foo_SJProfile*.ser
```

Properties file syntax is:

```
profile.debug
```

(You must also specify profiles in the file list.)

Properties file example is:

```
profile.debug
```

Default value is:

```
NA
```

AuditorInstaller Run Time Output

During run time, debugging statements placed by `AuditorInstaller` result in a trace of methods called and values returned. This happens for all profile layers that had debugging statements installed. There is no means of selective debug output at run time.

`AuditorInstaller` output relates to profiles only; there is currently no mapping to lines in your original `.sqlj` source file.

Following is a sample portion of `AuditorInstaller` run-time output. This is what the output might look like for a SQLJ `SELECT INTO` statement:

```
oracle.sqlj.runtime.OraProfile@1 . getProfileData (  )
oracle.sqlj.runtime.OraProfile@1 . getProfileData returned
sqlj.runtime.profile.ref.ProfileDataImpl@2
oracle.sqlj.runtime.OraProfile@1 . getStatement ( 0 )
oracle.sqlj.runtime.OraProfile@1 . getStatement returned
oracle.sqlj.runtime.OraRTStatement@3
oracle.sqlj.runtime.OraRTStatement@3 . setMaxRows ( 1000 )
oracle.sqlj.runtime.OraRTStatement@3 . setMaxRows returned
oracle.sqlj.runtime.OraRTStatement@3 . setMaxFieldSize ( 3000 )
```

```
oracle.sqlj.runtime.OraRTStatement@3 . setMaxFieldSize returned
oracle.sqlj.runtime.OraRTStatement@3 . setQueryTimeout ( 1000 )
oracle.sqlj.runtime.OraRTStatement@3 . setQueryTimeout returned
oracle.sqlj.runtime.OraRTStatement@3 . setBigDecimal ( 1 , 5 )
oracle.sqlj.runtime.OraRTStatement@3 . setBigDecimal returned
oracle.sqlj.runtime.OraRTStatement@3 . setBoolean ( 2 , false )
oracle.sqlj.runtime.OraRTStatement@3 . setBoolean returned
oracle.sqlj.runtime.OraRTStatement@3 . executeRTQuery (  )
oracle.sqlj.runtime.OraRTStatement@3 . executeRTQuery returned
oracle.sqlj.runtime.OraRTResultSet@6
oracle.sqlj.runtime.OraRTStatement@3 . getWarnings (  )
oracle.sqlj.runtime.OraRTStatement@3 . getWarnings returned null
oracle.sqlj.runtime.OraRTStatement@3 . executeComplete (  )
oracle.sqlj.runtime.OraRTStatement@3 . executeComplete returned
oracle.sqlj.runtime.OraRTResultSet@6 . next (  )
oracle.sqlj.runtime.OraRTResultSet@6 . next returned true
oracle.sqlj.runtime.OraRTResultSet@6 . getBigDecimal ( 1 )
oracle.sqlj.runtime.OraRTResultSet@6 . getBigDecimal returned 5
oracle.sqlj.runtime.OraRTResultSet@6 . getDate ( 7 )
oracle.sqlj.runtime.OraRTResultSet@6 . getDate returned 1998-03-28
```

There are two lines for each method call. The first shows the call and input parameters; the second shows the return value.

> **Note:** The classes you see in the `oracle.sqlj.runtime` package are SQLJ run time classes with equivalent functionality to similarly named JDBC classes. For example, `OraRTResultSet` is the SQLJ run time implementation of the JDBC `ResultSet` interface, containing equivalent attributes and methods.

AuditorInstaller Options

As with any customizer, `AuditorInstaller` has its own options that can be set using the `-P-C` prefix on the SQLJ command line (or `profile.C` in a SQLJ properties file).

`AuditorInstaller` supports the following options:

- `depth`: Specify how deeply you want to go into the layers of run time functionality in your profiles.
- `log`: Specify the target file for run time output of the debugging statements of the installed auditor.
- `prefix`: Specify a prefix for each line of run time output that will result from this installation of debugging statements.
- `showReturns`: Enable the installed auditor to include return arguments in its run time call tracing.
- `showThreads`: Enable the installed auditor to include thread names in its run time call tracing (relevant only for multithreaded applications).
- `uninstall`: Remove the debugging statements placed into the profiles during the most recent previous invocation of `AuditorInstaller` on those profiles.

AuditorInstaller Depth Option (depth)

As discussed in "Overview of Auditors and Code Layers" on page A-34, `AuditorInstaller` can install a set of debugging statements, known as an auditor,

at only a single layer of code during any one execution. The `AuditorInstaller depth` option enables you to specify which layer. Use multiple executions of `AuditorInstaller` to install auditors at different levels.

Layers are numbered in integers. The shallowest depth is layer 0; a maximum depth of 2 or 3 is typical. The only depth settings typically used are 0 for the shallowest layer or -1 for the deepest layer. In fact, it is difficult to install an auditor at any other particular layer, because the layer numbers used for the various kinds of SQLJ executable statements are not publicized.

The `depth` option is sometimes used in conjunction with the `prefix` option. By running `AuditorInstaller` more than once, with different prefixes for different layers, you can see at runtime what information is coming from which layers.

If you do not set the `depth` option, or the specification exceeds the number of layers in a given profile, then an auditor will be installed at the deepest layer.

Command-line syntax is:

```
-P-Cdepth=n
```

Command-line example is:

```
-P-Cdepth=0
```

Properties file syntax is:

```
profile.Cdepth=n
```

Properties file example is:

```
profile.Cdepth=0
```

Default value is:

```
-1 (deepest layer)
```

AuditorInstaller Log File Option (log)

Use the `log` option to specify an output file for runtime output that will result from the auditor that you are currently installing. Otherwise, standard output will be used, so that debug output will go to wherever SQLJ messages go.

When auditors write messages to an output file, they append; they do not overwrite. Therefore, you can specify the same log file for multiple auditors without conflict. In fact, it is typical in this way to have debug information from all layers of your application go to the same log file.

Command-line syntax is:

```
-P-Clog=log_file
```

Command-line example is:

```
-P-Clog=foo/bar/mylog.txt
```

Properties file syntax is:

```
profile.Clog=log_file
```

Properties file example is:

```
profile.Clog=foo/bar/mylog.txt
```

Default value is:

```
Empty (for standard output)
```

AuditorInstaller Prefix Option (prefix)

Use the `prefix` option to specify a prefix for each line of runtime output resulting from the debugging statements installed during this invocation of `AuditorInstaller`.

This option is often used in conjunction with the `depth` option. By running `AuditorInstaller` multiple times with different prefixes for different layers, you can easily see at runtime what information is coming from which layers.

Command-line syntax is:

```
-P-Cprefix="string"
```

Command-line example is:

```
-P-Cprefix="layer 2: "
```

Properties file syntax is:

```
profile.Cprefix="string"
```

Properties file example is:

```
profile.Cprefix="layer 2: "
```

Default value is:

```
Empty
```

AuditorInstaller Return Arguments Option (showReturns)

Use the `showReturns` option to enable or disable the display of return arguments as part of the runtime call tracing. This is enabled by default.

The following few lines show sample output with `showReturns` enabled (default):

```
oracle.sqlj.runtime.OraRTStatement@3 . executeComplete (  )
oracle.sqlj.runtime.OraRTStatement@3 . executeComplete returned
oracle.sqlj.runtime.OraRTResultSet@6 . next (  )
oracle.sqlj.runtime.OraRTResultSet@6 . next returned true
oracle.sqlj.runtime.OraRTResultSet@6 . getBigDecimal ( 1 )
oracle.sqlj.runtime.OraRTResultSet@6 . getBigDecimal returned 5
oracle.sqlj.runtime.OraRTResultSet@6 . getDate ( 7 )
oracle.sqlj.runtime.OraRTResultSet@6 . getDate returned 1998-03-28
```

With `showReturns` disabled, the output would appear as follows:

```
oracle.sqlj.runtime.OraRTStatement@3 . executeComplete (  )
oracle.sqlj.runtime.OraRTResultSet@6 . next (  )
oracle.sqlj.runtime.OraRTResultSet@6 . getBigDecimal ( 1 )
oracle.sqlj.runtime.OraRTResultSet@6 . getDate ( 7 )
```

Instead of both a call line and a return line for each method call, there is only a call line.

Command-line syntax is:

```
-P-CshowReturns<=true|false>
```

Command-line example is:

```
-P-CshowReturns=false
```

Properties file syntax is:

```
profile.CshowReturns<=true|false>
```

Properties file example is:

```
profile.CshowReturns=false
```

Default value is:

```
true
```

AuditorInstaller Thread Names Option (showThreads)

Use the `showThreads` option to enable or disable the display of thread names as part of the runtime call tracing (relevant only for multithreaded applications). This is disabled by default.

When this option is enabled, thread names prefix the method names in the trace output.

Command-line syntax is:

```
-P-CshowThreads<=true|false>
```

Command-line example is:

```
-P-CshowThreads
```

Properties file syntax is:

```
profile.CshowThreads<=true|false>
```

Properties file example is:

```
profile.CshowThreads
```

Default value is:

```
false
```

AuditorInstaller Uninstall Option (uninstall)

Use the `uninstall` option to remove debugging statements placed during previous invocations of `AuditorInstaller`. Each time you use the `uninstall` option, it will remove the auditor most recently installed.

To remove all auditors from a profile, run `AuditorInstaller` repeatedly until you get a message indicating that the profile was unchanged.

Command-line syntax is:

```
-P-Cuninstall
```

Command-line example is:

```
-P-Cuninstall
```

Properties file syntax is:

```
profile.Cuninstall
```

Properties file example is:

```
profile.Cuninstall
```

Default value is:

```
Disabled
```

Full Command-Line Examples

Following are some full SQLJ command-line examples showing the specification of `AuditorInstaller` options.

Insert a set of debugging statements, or auditor, into the deepest layer (which is the default layer), with runtime output to standard output:

```
% sqlj -P-debug MyApp_SJProfile*.ser
```

Insert an auditor into the deepest layer, with runtime output to `log.txt`:

```
% sqlj -P-debug -P-Clog=foo/bar/log.txt MyApp_SJProfile*.ser
```

Insert an auditor into the deepest layer, with runtime output to standard output, showing thread names but not return arguments:

```
% sqlj -P-debug -P-CshowThreads=true -P-CshowReturns=false MyApp_SJProfile*.ser
```

Insert an auditor into layer 0 (the shallowest layer). Send runtime output to `log.txt`; prefix each line of runtime output with "`Layer 0: `" (the following command is a single wraparound line):

```
% sqlj -P-debug -P-Clog=foo/bar/log.txt -P-Cdepth=0 -P-Cprefix="Layer 0: "
MyApp_SJProfile*.ser
```

Uninstall an auditor (this uninstalls the auditor most recently installed; do it repeatedly to uninstall all auditors):

```
% sqlj -P-debug -P-Cuninstall MyApp_SJProfile*.ser
```

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W