Oracle8*i*

SQLJ Developer's Guide and Reference

Release 8.1.5

February, 1999

Part No. A64684-01

SQLJ Developer's Guide and Reference, Release 8.1.5

Part No. A64684-01

Primary Author: Brian Wright

Contributing Authors: Jack Melnick, Tim Smith, Bill Courington, Tom Pfaeffle

Contributors: Pierre Dufour, Julie Basu, Ekkehard Rohwedder, Brian Becker, Risto Lankinen, Alan Thiesen, Jerry Schwarz, Cheuk Chau, Vishu Krishnamurthy, Rafiul Ahad, Tom Portfolio, Ellen Barnes, Susan Kraft, Angie Long

Contents

Send Us Your Comments

SQLJ Developer's Guide and Reference, Release 8.1.5

Part No. A64684-01

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this publication. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most about this manual?

If you find any errors or have any other suggestions for improvement, please indicate the chapter, section, and page number (if available). You can send comments to us in the following ways:

- Electronic mail — jpgcomnt@us.oracle.com
- FAX - 650-506-7225. Attn: Java Products Group, Information Development Manager
- Postal service:
  Oracle Corporation
  Information Development Manager
  500 Oracle Parkway, Mailstop 4op978
  Redwood Shores, CA 94065
  USA

If you would like a reply, please give your name, address, and telephone number below.

If you have problems with the software, please contact your local Oracle World Wide Support Center.

Preface

This preface introduces you to the *Oracle8i SQLJ Developer's Guide and Reference*, discussing the intended audience, structure, and conventions for this document. A list of related Oracle documents is also provided.

Intended 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
- Oracle PL/SQL
- JDBC
- Oracle databases

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

See "Related Documents" on page xx below for the names of Oracle documents that discuss SQL and JDBC.

Document Structure

The two major aspects of using SQLJ are:

- creating your SQLJ source code
- running the SQLJ translator

Chapters 3 through 7 provide information you must have to create your code, with chapters 3 and 4 covering the most important aspects.

Chapter 8 provides information you must have to run the translator.

In all, this document consists of twelve chapters and two appendixes, as follows:

Chapter 1	"Overview"—Introduces SQLJ concepts, components, and processes. Discusses possible alternative deployment or development scenarios.
Chapter 2	"Getting Started"—Guides you through the steps of testing and verifying the installation of the Oracle database, Oracle JDBC drivers, and Oracle SQLJ.
Chapter 3	"Basic Language Features"—Discusses SQLJ programming features you must have for basic applications. Focuses largely on standard SQLJ constructs, as opposed to Oracle extended functionality.
Chapter 4	"Key Programming Considerations"—Discusses key issues to consider as you write your source code, such as connections, null-handling, and exception-handling.
Chapter 5	"Type Support"—Lists Java types that Oracle SQLJ supports, discusses use of stream types, and discusses Oracle type extensions in the database and the Java types that correspond to them.
Chapter 6	"Objects and Collections"—Discusses Oracle SQLJ support of user-defined object and collection types, including use of the Oracle JPublisher utility to generate corresponding Java types.
Chapter 7	"Advanced Language Features"—Discusses additional SQLJ programming features you may need for more advanced applications.
Chapter 8	"Translator Command Line and Options"—Documents command-line syntax, properties files, and options for the Oracle SQLJ translator.

Chapter 9	"Translator and Runtime Functionality"—Discusses the functionality of translator operations, translator and runtime error messages, and NLS support.
Chapter 10	"Profiles and Customization"—Describes SQLJ profiles (used in implementing SQL operations); documents options you can specify during translation regarding the customization of your profiles for particular environments.
Chapter 11	"SQLJ in the Server"—Discusses how to create and load SQLJ applications to run in the server, typically as stored procedures or functions. This includes optional use of the server-side embedded translator.
Chapter 12	"Sample Applications"—Contains source code for a range of fully functional SQLJ sample applications. Oracle provides these applications in the `demo` directory on the product CD.
Appendix A	"Performance and Debugging"—Briefly discusses performance tuning and refers to other useful documentation; discusses debugging scenarios, focusing on the `AuditorInstaller` utility that is provided with Oracle SQLJ.
Appendix B	"SQLJ Error Messages"—Lists Oracle SQLJ translator and runtime error messages, their causes, and what actions you should take in response.

Document Conventions

This document uses UNIX syntax for file paths (for example: `/myroot/myfile.html`). If you are using some other kind of operating system, then substitute the appropriate syntax.

This document uses `[Oracle Home]` to indicate your Oracle home directory.

In addition, this document uses the following conventions:

Convention	Meaning
italicized regular text	Italicized regular text is used either for emphasis or to indicate a special term that is being defined or will be defined shortly.

Convention	Meaning
`. . .`	Horizontal ellipsis points in sample code indicate the omission of a statement or statements or part of a statement. This is done when you would normally expect additional statements or code to appear, but such statements or code would not be related to the example.
`code text`	Code text in the midst of regular text indicates class names, object names, method names, variable names, Java types, Oracle datatypes, file names, and directory names.
`italicized_code_text`	Italicized code text in a program statement indicates something that must be provided by the user.
<*`italicized_code_text`*>	Angle brackets enclosing italicized code text in a program statement indicates something that can *optionally* be provided by the user.

In this document, it was not feasible to use more standard conventions, such as square brackets [] to enclose optional items to be provided, because of the particulars of SQLJ coding syntax.

For example, in the following statement the square brackets and curly brackets are part of SQLJ coding syntax, but the angle brackets indicate that `conntxt_exp`, `execctxt_exp`, and `results_exp` are optional entries. You must provide a SQL operation, however.

```
#sql <[<conntxt_exp><,><execctxt_exp>]> <results_exp> = { SQL operation };
```

And in the following SQLJ command line option (`-user`), the angle brackets indicate that `conn_context_class` and the password (with preceding slash) are optional entries. You must provide a username, however.

```
-user<@conn_context_class>=username</password>
```

Related Documents

For information about Java-related aspects of Oracle8*i*, see the following Oracle documents from the Java Products documentation group:

- *Oracle8i Java Developer's Guide* (overview and configuration information)
- *Oracle8i JDBC Developer's Guide and Reference*

- *Oracle8i JPublisher User's Guide* (relevant to support for objects and collections)
- *Oracle8i Java Stored Procedures Developer's Guide*
- *Oracle8i Enterprise JavaBeans and CORBA Developer's Guide*

The *Java Developer's Guide* and *JPublisher User's Guide* are not included on the release 8.1.5 documentation CD.

Also, the following Oracle documents from the Server Technology group may be useful and are sometimes referenced in this document:

- *Oracle8i SQL Reference*
- *Oracle8i Application Developer's Guide—Fundamentals*
- *Oracle8i Supplied Packages Reference*
- *Net8 Administrator's Guide*
- *PL/SQL User's Guide and Reference*

For documentation of SQLJ standard features and syntax, refer to the ANSI SQL / OLB standard X3.135.10-1998. You can access this at the following Web site:

`http://www.sqlj.org/`

1 Overview

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

Introduction to SQLJ

SQLJ allows applications programmers to embed static SQL operations in Java code in a way that is compatible with the Java design philosophy. A SQLJ program is a Java program containing embedded static SQL statements that comply with the ANSI-standard SQLJ Language Reference syntax; *static* SQL operations are predefined. The operations themselves do not change in real-time as a user runs the application, although the data values transmitted can change dynamically. Typical applications contain much more static SQL than dynamic SQL; *dynamic* SQL operations are not predefined. The operations themselves can change in real-time and require direct use of JDBC statements. Note, however, that you can use SQLJ statements and JDBC statements in the same program.

SQLJ consists of both a translator and a runtime component and is smoothly integrated into your development environment. The translator is run by the developer, with translation, compilation, and customization taking place transparently. Translation replaces embedded SQL with calls to the SQLJ runtime, which implements the SQL operations. In standard SQLJ this is typically, but not necessarily, done through calls to a JDBC driver. In the case of an Oracle database, you would typically use an Oracle JDBC driver. When the end user runs the SQLJ application, the runtime is invoked to handle the SQL operations in real-time.

The Oracle SQLJ translator is conceptually similar to other Oracle precompilers and allows the developer to check SQL syntax, compare SQL operations to what is available in the schema, and check the compatibility of Java types with corresponding database types. In this way, errors can be caught by the developer instead of by a user at runtime. The translator performs the following checks:

- Checks the syntax of the embedded SQL.
- Checks data types to ensure that the data exchanged between Java and SQL have compatible types and proper type conversions.
- Checks SQL constructs against the database schema to ensure consistency.

The SQLJ methodology of embedding SQL operations directly in Java code is much more convenient and concise than the JDBC methodology. In this way, SQLJ reduces development and maintenance costs in Java programs that require database connectivity. When dynamic SQL is required, however, SQLJ supports interoperability with JDBC such that you can intermix SQLJ code and JDBC code in the same source file.

For Oracle developers, it is also noteworthy that Oracle SQLJ can interoperate with PL/SQL, allowing anonymous PL/SQL blocks within SQLJ statements, and providing a syntax for calling PL/SQL stored procedures and functions.

Overview of SQLJ Components

This section introduces the main SQLJ components and the concept of SQLJ *profiles*.

SQLJ Translator and SQLJ Runtime

Oracle SQLJ consists of two major components:

- Oracle SQLJ **translator**—This component is a preprocessor or precompiler that developers run after creating SQLJ source code.

  The translator, written in pure Java, supports a programming syntax that allows you to embed SQL operations inside SQLJ executable statements. SQLJ executable statements, as well as 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 translator produces a `.java` file and one or more SQLJ *profiles*, which contain information about your SQL operations. SQLJ then automatically invokes a Java compiler to produce `.class` files from the `.java` file.

- Oracle SQLJ **runtime**—This component is invoked automatically each time an end-user runs a SQLJ application.

  The SQLJ runtime, also written in pure Java, implements the desired actions of your SQL operations, accessing the database using a JDBC driver. The generic SQLJ standard does not require that a SQLJ runtime use a JDBC driver to access the database, but the Oracle SQLJ runtime does require a JDBC driver, and in fact requires an Oracle JDBC driver if your application uses Oracle-specific features.

  For more information about the runtime, see "SQLJ Runtime" on page 9-16.

In addition to the translator and runtime, there is a component known as the **customizer**. SQLJ automatically invokes a customizer to tailor your SQLJ profiles for a particular database implementation and any vendor-specific features and datatypes. By default, SQLJ uses the Oracle customizer so that your application can use Oracle-specific features.

SQLJ Profiles

SQLJ *profiles* are serialized Java objects (or optionally classes) generated by the SQLJ translator which contain details about the embedded SQL operations in your SQLJ source code. The translator creates these profiles, then either serializes them and puts them into binary resource files, or puts them into `.class` files (according to your translator option settings).

About Profiles

SQLJ profiles are used in implementing the embedded SQL operations in your SQLJ executable statements. Profiles contain information about your 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 executing this instruction.

SQLJ generates a profile for each connection context class in your application, where typically each connection context class corresponds to a type of database schema you connect to. (There is one default connection context class, and you can declare additional classes.) The 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 allows database vendors to add value in two ways:

- Vendors can support their own specific datatypes 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.

For example, you must customize your profile in order to use Oracle objects in your SQLJ application.

Notes:

- By default, SQLJ profile file names end in the `.ser` extension, but this does not mean that all `.ser` files are profiles. Any serialized object uses that 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 of Profiles

SQLJ-generated profile files feature *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 employed vendor-specific database types or features. This is true of generated `.class` files as well.

Overview of Oracle Extensions to the SQLJ Standard

The Oracle SQLJ translator accepts a broader range of SQL syntax than the ANSI SQLJ Standard specifies.

The standard addresses only the SQL92 dialect of SQL, but allows extension beyond that. Oracle SQLJ supports Oracle's SQL dialect, which is a superset of SQL92. Therefore, if you must create SQLJ programs that work with other DBMS vendors, you should avoid using SQL syntax and SQL types that are not in the standard and therefore might not be supported in other environments.

Oracle SQLJ supports the following Java types as extensions to the SQLJ standard. Do not use these or other types if you may want to use your code in other environments. To ensure that your application is portable, use the Oracle SQLJ `-warn=portable` flag. (See "Translator Warnings (-warn)" on page 8-42.)

- instances of `oracle.sql.*` classes as wrappers for SQL data (see "Oracle Type Extensions" on page 5-22)
- custom Java classes (classes that implement the `CustomDatum` interface), typically produced by the Oracle JPublisher utility, to correspond to SQL objects, object references, and collections (see "About Custom Java Classes and the CustomDatum Interface" on page 6-6)
- stream instances (`AsciiStream`, `BinaryStream`, `UnicodeStream`) used as output parameters (see "Support for Streams" on page 5-8)
- iterator and result set instances as input or output parameters anywhere (the standard specifies them only in result expressions or cast statements; see "Using Iterators and Result Sets as Host Variables" on page 3-48 and "Using Iterators and Result Sets as Stored Function Returns" on page 3-60)

For information about the `oracle.sql` package, see "Oracle Type Extensions" on page 5-22.

For general information about Oracle SQLJ extensions, see Chapter 5, "Type Support", and Chapter 6, "Objects and Collections".

Basic Translation Steps and Runtime Processing

This section introduces the following:

- basic steps of the Oracle SQLJ translator in translating SQLJ source code
- processing by the Oracle SQLJ runtime when a user runs your application

This is followed by a summary of translator input and output.

For more detailed information about the translation steps, see "Internal Translator Operations" on page 9-2.

SQLJ source code contains a mixture of standard Java source together with SQLJ class declarations and SQLJ executable statements containing embedded SQL operations.

SQLJ source files have the `.sqlj` file name extension. If the source file declares a public class (maximum of one), 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 must still be a legal Java identifier, and it is recommended that the file name match one of the defined classes.

Translation Steps

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

```
sqlj Foo.sqlj
```

What this command actually runs is a front-end script or utility (depending on the platform) that reads the command line, invokes a Java VM, and passes arguments to the VM. The Java VM invokes the SQLJ translator and acts as a front end.

This document refers to running the front-end as "running SQLJ" and to its command line as the "SQLJ command line". For information about command-line syntax, see "Command-Line Syntax and Operations" on page 8-10.

From this point the following sequence of events occurs, presuming each step completes without fatal error.

1. The Java VM invokes the SQLJ translator.
2. The translator parses the source code in the `.sqlj` file, checking for proper SQLJ syntax and looking for type mismatches between your declared SQL datatypes and corresponding Java host variables. (Host variables are local Java

variables that are used as input or output parameters in your SQL operations. "Java Host Expressions, Context Expressions, and Result Expressions" on page 3-15 describes them.)

3. The translator invokes the semantics-checker, which checks the semantics of embedded SQL statements.

   The developer can use online or offline checking, according to SQLJ option settings. If online checking is performed, then SQLJ will connect to the database to verify that the database supports all of the database tables, stored procedures, and SQL syntax that the application uses, and that the host variable types in the SQLJ application are compatible with datatypes of corresponding database columns.

4. The translator processes your SQLJ source code, converts SQL operations to SQLJ runtime calls, and generates Java output code and one or more SQLJ profiles. A separate profile is generated for each connection context class in your source code, where a different connection context is typically used for each type of database schema you will connect to.

   Generated Java code is put into a `.java` output file, which contains the following:

   - any class definitions and Java code from your `.sqlj` source file
   - class definitions created as a result of your SQLJ declarations (see "Overview of SQLJ Declarations" on page 3-2)
   - a class definition for a specialized class (known as the *profile-keys* class) that SQLJ generates and uses in conjunction with your profiles
   - calls to the SQLJ runtime to implement the actions of your embedded SQL operations

     (The SQLJ runtime, in turn, uses the JDBC driver to access the database. See "SQLJ Runtime" on page 9-16 for more information.)

   Generated profiles contain information about all of the embedded SQL statements in your SQLJ source code, such as actions to take, datatypes being manipulated, and tables being accessed. When your application is run, the SQLJ runtime accesses the profiles to retrieve your SQL operations and pass 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.

5. The Java VM 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 you defined, a `.class` file for each of your SQLJ declarations, and a `.class` file for the profile-keys class.
7. The Java VM invokes the Oracle SQLJ customizer or other specified customizer.
8. The customizer customizes the profiles generated in step 4.

Notes:

- SQLJ generates profiles and the profile-keys class only if your source code includes SQLJ executable statements.
- This is a very generic example. It is also possible to specify pre-existing `.java` files on the command line to be compiled (and to be available for type resolution as well), or to specify pre-existing profiles to be customized, or to specify `.jar` files containing profiles to be customized. See "Translator Command Line and Properties Files" on page 8-2 for more information.

Runtime Processing

When a user runs the application, the SQLJ runtime 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 mapping between SQLJ executable statements in the application and SQL operations in the profile.
2. The SQLJ-generated application code calls the SQLJ runtime, which reads the SQL operations from the profile.
3. The SQLJ runtime calls the JDBC driver and passes the SQL operations to the driver.
4. The SQLJ runtime passes any input parameters to the JDBC driver.
5. The JDBC driver executes the SQL operations.

6. If any data is to be returned, the database sends it to the JDBC driver, which sends it to the SQLJ runtime for use by your application.

Note: Passing input parameters (step 4) 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 to or output from SQL operations.

Summary of Translator Input and Output

This section summarizes what the SQLJ translator takes as input, what it produces as output, and where it puts its output.

Note: This discussion mentions iterator class and connection context class declarations. Iterators are similar to JDBC result sets; connection contexts are used for database connections. For more information about these class declarations, see "Overview of SQLJ Declarations" on page 3-2.

Input

In its most basic operation, the SQLJ translator takes one or more `.sqlj` source files as input in its command line. The name of your main `.sqlj` file is based on the public class it defines, if it defines one, or else on the first class it defines if there are no public class definitions. Each public class you define must be in its own `.sqlj` file.

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

```
MyClass.sqlj
```

This will also be the file name if there are no public class definitions but `MyClass` is the first class defined.

When you run SQLJ, you can also specify numerous SQLJ options in the command line or properties files.

For more information about SQLJ input, including additional types of files you can specify in the command line, see "Translator Command Line and Properties Files" on page 8-2.

Output

The translation step produces a Java source file for each `.sqlj` file in your application, and at least one application *profile* (presuming your source code uses SQLJ executable statements).

SQLJ generates source files and profiles as follows:

- Java source files will be `.java` files with the same base names as your `.sqlj` files.

  For example, `MyClass.sqlj` defines class `MyClass` and the translator produces `MyClass.java`.

- The application profile files contain information about the SQL operations of your SQLJ application. There will be one profile for each kind of database schema you connect to in your application. The profiles will have names with the same base name as your main `.sqlj` file, plus the following extensions:

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

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

  ```
  MyClass_SJProfile0.ser
  ```

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

> **Note:** There is a translator option, `-ser2class`, that instructs the translator to generate profiles as `.class` files instead of `.ser` files. Other than the file extension, the naming is the same.

The compilation step compiles the Java source file into multiple class files. There are at least two class files: one for each class you define in your `.sqlj` source file (minimum of one) and one for a class that the translator generates and uses with the profiles to implement your SQL operations (presuming your source code uses SQLJ executable statements). Additional `.class` files are produced if you declared any SQLJ iterators or connection contexts (see "Overview of SQLJ Declarations" on page 3-2). The `.class` files are named as follows:

- The class file for each class you define consists of the name of the class with the `.class` extension.

  For example, `MyClass.sqlj` defines `MyClass`, the translator produces `MyClass.java`, and the compiler produces `MyClass.class`

- The class that the translator generates is named according to the base name of your 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 produce:

  ```
  MyClass_SJProfileKeys.class
  ```

- The translator names iterator classes and connection context classes according to how you declare them. For example, if you declare an iterator `MyIter`, the resulting class file will be `MyIter.class`.

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

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

Output File Locations

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

By default, SQLJ places generated `.class` and `.ser` files in the same directory as the generated `.java` files. You can specify a different `.class` and `.ser` file location, however, 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 either the `-d` or `-dir` option, you must specify a directory that already exists. For more information about these options, see "Options for Output Files and Directories" on page 8-26.

Alternative Deployment Scenarios

The discussion in this book is mostly from the perspective of writing for client-side SQLJ applications, but 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)
- against an Oracle Lite database

Running SQLJ in Applets

The SQLJ runtime is pure Java, so you can use SQLJ source code in applets as well as applications. There are, however, a few considerations:

- You must package all of the SQLJ runtime packages with your applet:

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

  As well as the following if you used Oracle customization:

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

  These are all included in the file `runtime.zip`, which comes with your Oracle SQLJ installation.

  The total size of the downloaded SQLJ packages will be approximately 150K-250K (uncompressed) or 40K-70K (compressed), depending on whether your applet requires Oracle customization and/or internationalized messages.

- You must specify a pure Java JDBC driver, such as the Oracle JDBC Thin driver, for your database connection.
- 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 used for profiles. However, Oracle SQLJ provides the `-ser2class` option to convert `.ser` files to `.class` files. See "Conversion of .ser File to .class File (-ser2class)" on page 8-54 for more information.

- Oracle SQLJ requires the runtime environment of JDK 1.1.x or higher. You cannot employ browsers using JDK 1.0.x, such as Netscape Navigator 3.0 and Microsoft Internet Explorer 3.0, without a plug-in or some other means of using JRE 1.1.x instead of the browser's default JRE.

  Sun Microsystems offers such a plug-in. For information, refer to the following Web site:

  `http://www.javasoft.com/products/plugin`

- 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 address space. (For information about connections, see "Connection Considerations" on page 4-9.)

For additional information, please see the SQLJ `README` file and examine the sample applet (not included in this document) in the SQLJ `demo` directory. These will explain any additional steps to take.

For applet issues that apply more generally to the Oracle JDBC drivers, see the *Oracle8i JDBC Developer's Guide and Reference*. This includes discussion of firewalls and security issues as well.

Introduction to SQLJ in the Server

In addition to its use in client applications, SQLJ code can run in the Oracle8*i* Server in stored procedures, stored functions, triggers, Enterprise JavaBeans, or CORBA objects. Server-side access is through the Oracle JDBC server-side driver. There is also an embedded SQLJ translator in the Oracle8*i* Server so that SQLJ source files for server-side use can optionally be translated directly in the server.

The two main areas to consider, which are discussed in detail in Chapter 11, "SQLJ in the Server", are:

- creating SQLJ code for server-side use

  In creating a SQLJ application for use against the Oracle8*i* Server, there is very little difference between coding for server-side use as opposed to client-side use. What issues do exist are due to general JDBC characteristics as opposed to SQLJ-specific characteristics. The main differences involve connections:

  - You only have one connection.
  - The connection must be 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 does not support auto-commit.

- 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, you can then load the class and resource files into the server from your client machine, either pushing them from the client using the Oracle `loadjava` utility or pulling them in from the server using SQL commands. (It is convenient to have them all in a single `.jar` file first.)

  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 you can specify that translation and compilation be 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 if requested.

  See "Loading SQLJ Source and Translating in the Server" on page 11-13 for information about using the embedded server-side translator.

Using SQLJ with an Oracle Lite Database

You can use SQLJ on top of an Oracle Lite database. This section provides a brief overview of this functionality. For more information, refer to the *Oracle Lite Java Developer's Guide*.

Overview of Oracle Lite and Java Support

Oracle Lite is a lightweight database that offers flexibility and versatility that larger databases cannot. It requires only 350K to 750K of memory for full functionality, natively synchronizes with the Palm Computing platform, and can run on Windows NT (3.51 or higher), Windows 95, and Windows 98. It offers an embedded environment that requires no background or server processes.

Oracle Lite is compatible with Oracle8*i*, previous versions of Oracle8, and Oracle7. It provides comprehensive support for Java, including JDBC, SQLJ, and Java stored procedures. There are two alternatives for access to the Oracle Lite database from Java programs:

- native JDBC driver

  This is intended for Java applications that use the relational data model, allowing them direct communication with the object-relational database engine.

  Use the relational data model if your program has to access data that is already in SQL format, must run on top of other relational database systems, or uses very complex queries.

- Java Access Classes (JAC)

  This is intended for Java applications that use either the Java object model or the Oracle Lite object model, allowing them to access persistent information stored in the Oracle Lite database without having to map between the object model and the relational model. Use of JAC also requires a persistent Java proxy class to model the Oracle Lite schema. This can be generated by Oracle Lite tools.

  Use the object model if you want your program to have a smaller footprint and run faster and you do not require the full capability of the SQL language.

There is interoperability between Oracle Lite JDBC and JAC, with JAC supporting all types that JDBC supports, and JDBC supporting JAC types that meet certain requirements.

Requirements to Run Java on Oracle Lite

Note the following requirements if you intend to run a Java program on top of an Oracle Lite database:

- Windows NT 3.51 or higher, Windows 95, or Windows 98
- Oracle Lite 3.0 or higher
- JDK 1.1 or higher
- Java Runtime Environment (JRE) that supports Java Native Interface (JNI)

  The JREs supplied with JDK 1.1, Oracle JDeveloper, and Symantec Visual Cafe support JNI.

Support for Oracle Extensions

The JDBC driver implemented with Oracle Lite versions 3.6 and prior supports standard SQL92 types only, so Oracle-specific functionality cannot be used on top of these versions. Therefore, you cannot use Oracle type extensions such as `BLOB`, `CLOB`, `BFILE`, `ROWID`, and user-defined object and collection types.

Beginning with version 4.0, however, Oracle Lite will include an Oracle-specific JDBC driver and Oracle-specific SQLJ runtime classes (including the Oracle semantics-checkers and customizer), allowing use of Oracle-specific features and type extensions.

Alternative Development Scenarios

The discussion in this book assumes that you are coding manually in a UNIX environment for English-language deployment. However, you can use SQLJ on other platforms and with IDEs. There is also NLS support for deployment to other languages. This section introduces these topics:

- NLS support
- SQLJ in IDEs
- Windows considerations

SQLJ NLS Support

Oracle SQLJ support for native languages and character encodings is based on Java's built-in NLS capabilities.

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

For information, see "NLS Support in the Translator and Runtime" on page 9-20.

SQLJ in JDeveloper and Other IDEs

Oracle SQLJ includes a programmatic API so that it can be embedded in integrated development environments (IDEs) such as Oracle JDeveloper. The IDE takes on a role that is similar to that of the `sqlj` script that is used as a front-end in Solaris, invoking the translator, semantics-checker, compiler, and customizer.

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

The JDeveloper Suite includes Oracle JDeveloper, Oracle Application Server, Oracle8*i* Enterprise Edition, and Oracle Procedure Builder.

The Oracle JDBC OCI and Thin drivers are included with JDeveloper, as well as drivers to access an Oracle Lite database.

JDeveloper's compilation functionality includes an integrated Oracle SQLJ translator so that your SQLJ application is translated automatically as it is compiled.

Information about `JDeveloper` is available at the following URL:

`http://technet.us.oracle.com`

Windows Considerations

Note the following if you are using a Windows platform instead of Solaris:

- This manual uses Solaris/UNIX syntax. Use platform-specific file names and directory separators (such as "\" on Windows) that are appropriate for your platform, because your Java VM expects file names and paths in the platform-specific format. This is true even if you are using a shell (such as `ksh` on NT) that permits a different file name syntax.
- For Solaris, Oracle SQLJ provides a front-end script, `sqlj`, that you use to invoke the SQLJ translator. On Windows, Oracle SQLJ instead provides an executable file, `sqlj.exe`. Using a script is not feasible on Windows platforms 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.
- Be aware of any limitations for the maximum size of your command line and environment variables.
- On Windows, the SQLJ translation process may sometimes suspend during compilation. If you encounter this problem, use the translator `-passes` option, which is discussed in "Run SQLJ in Two Passes (-passes)" on page 8-67.

Refer to the Windows platform `README` file for additional information.

2 Getting Started

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

Note that if you are using an Oracle database and Oracle JDBC driver, then you should also verify your JDBC installation according to the *Oracle8i JDBC Developer's Guide and Reference*.

The following topics are discussed in this chapter:

Assumptions and Requirements

This section discusses basic assumptions about your environment and requirements of your system so that you can run Oracle SQLJ.

Assumptions About Your Environment

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

- You have a standard Java environment that is operational on your system. This would typically be using the Sun Microsystems JDK, but other kinds of Java are permissible.

  To translate and run Oracle SQLJ applications on a Sun JDK you must use a JDK 1.1.x version. SQLJ does not support JDK 1.0.2, including applets running in browsers that use that JDK, unless special preparations have been made. (Applets are not discussed here. Refer to "Running SQLJ in Applets" on page 1-14.)

  Make sure you can run Java (typically `java`) and your Java compiler (typically `javac`).

- You can already run JDBC applications in your environment.

  If you are using an Oracle database and Oracle JDBC driver, then you should complete the steps in Chapter 2, "Getting Started", of the *Oracle8i JDBC Developer's Guide and Reference*. This chapter also includes introductory information about the Oracle JDBC drivers and how to decide which driver is appropriate for your situation.

Notes:

- Oracle JDBC drivers do not yet support JDK 1.2; however, Oracle SQLJ supports JDK 1.2 in case you are using a non-Oracle JDBC driver that supports it as well.
- If you are using a non-Oracle JDBC driver, then you must modify `connect.properties`, as discussed in "Set Up the Runtime Connection" on page 2-6, and the sample applications, as discussed in "Driver Selection and Registration for Runtime" on page 4-7, so that your driver is registered before the call to the `Oracle.connect()` method.

Requirements for Using Oracle SQLJ

The following are required in order to use Oracle SQLJ:

- JDBC driver implementing the standard `java.sql` JDBC interfaces from Sun Microsystems

  Oracle SQLJ works with any standards-compliant JDBC driver.
- database system that is accessible using your JDBC driver
- class files for the SQLJ translator, SQLJ profile customizer, and SQLJ runtime

  These class files are all available in the file:

  `[Oracle Home]/sqlj/lib/translator.zip`

Note: If you have a system where you will only be running SQLJ applications that have already been translated, compiled, and customized, you will need only the runtime classes, not the translator or profile customizer classes. The runtime classes are in the file:

`[Oracle Home]/sqlj/lib/runtime.zip`

This is a subset of the `translator.zip` file.

JServer Configuration

This document presumes that system configuration issues are outside the duties of most SQLJ developers. Therefore, configuration of the Oracle8*i* JServer (formerly referred to as the Java Option) is not covered here. For information about setting Java-related configuration parameters (such as `JAVA_POOL_SIZE`), see the JServer `README` file.

If you need information about configuring the multi-threaded server, dispatcher, or listener (which may be particularly relevant if you are coding Enterprise JavaBeans or CORBA objects), see the *Net8 Administrator's Guide*.

Checking the Installation and Configuration

Once you have verified that the above assumptions and requirements are satisfied, you must check your Oracle SQLJ installation.

Check for Installed Directories and Files

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

Directories for Oracle JDBC

If you are using one of the Oracle JDBC drivers, refer to the *Oracle8i JDBC Developer's Guide and Reference* for information about JDBC files that should be installed on your system.

Directories for Oracle SQLJ

Installing Oracle Java products will include, among other things, a `sqlj` directory under your `[Oracle Home]` directory, containing the following subdirectories:

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

Also, directly under `[Oracle Home]` is the following directory containing utilities for all Java product areas:

- `bin`

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

Set the PATH and CLASSPATH

Make sure your `PATH` and `CLASSPATH` environment variables have the necessary settings for Oracle SQLJ (and Oracle JDBC if applicable).

PATH and CLASSPATH for Oracle JDBC

If you are using one of the Oracle JDBC drivers, refer to the *Oracle8i JDBC Developer's Guide and Reference* for information about required settings of the `PATH` and `CLASSPATH` environment variables for Oracle JDBC.

PATH and CLASSPATH for Oracle SQLJ

Set your PATH and CLASSPATH variables as follows.

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

```
[Oracle Home]/bin
```

Use backward-slashes for Windows. Replace `[Oracle Home]` with your actual Oracle Home directory.

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

```
[Oracle Home]/sqlj/lib/translator.zip
```

Use backward-slashes for Windows. Replace `[Oracle Home]` with your actual Oracle Home directory.

Verify Installation of sqljutl Package

Note: This step is applicable only if you are using an Oracle database, and is not critical until you are at a point where you are using SQLJ stored procedures or functions.

The package `sqljutl` is required for online checking of stored procedures and functions in an Oracle database. It should have been installed automatically under the SYS schema during installation of your database, unless you are using a previously existing database. To check the installation of `sqljutl`, issue the following SQL command (from `SQL*Plus`, for example):

```
describe package sqljutl
```

This should result in a brief description of the package. If you get a message indicating the package cannot be found, then you must install it manually. To do so, use `SQL*Plus` to run the `sqljutl.sql` script, which is located as follows:

```
[Oracle Home]/sqlj/lib/sqljutl.sql
```

Consult your installation instructions if necessary.

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.

These demo applications are provided with your SQLJ installation in the `demo` directory:

```
[Oracle Home]/sqlj/demo
```

You must edit some of the source files as necessary and translate and/or compile them as appropriate (as explained in the following subsections).

The demo applications provided with the Oracle SQLJ installation refer to tables on a database account with user name `scott` and password `tiger`. Most Oracle 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 (".") be in your `CLASSPATH`, as described earlier.

Set Up the Runtime Connection

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

```
# 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:ORCL
#sqlj.url=jdbc:oracle:oci8:@
#sqlj.url=jdbc:oracle:oci7:@
```

```
#
# User name and password here
sqlj.user=scott
sqlj.password=tiger
```

(User `scott` and password `tiger` are used for the demo applications.)

If Using an Oracle JDBC Driver

If you are using a JDBC OCI driver (OCI 8 or OCI 7), then uncomment the `oci8` URL line or the `oci7` URL line, as appropriate, in `connect.properties`.

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

If Using a non-Oracle JDBC Driver

If you are using a non-Oracle JDBC driver, then add a line to `connect.properties` to set the appropriate URL, as follows:

```
sqlj.url=your_URL_here
```

Use the same URL that was specified when your JDBC driver was set up.

You must also register the driver explicitly in your code (this is handled automatically in the demo and test programs if you use an Oracle JDBC driver). See "Driver Selection and Registration for Runtime" on page 4-7.

Create a Table to Verify the Database

The following tests assume a table called SALES. If you compile and run `TestInstallCreateTable` it will create the table for you if the database and your JDBC driver are working and your connection is set up properly in `connect.properties`.

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

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, you can instead create the SALES table using the following command in a database command-line processor (such as SQL*Plus):

```
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, use the TestInstallJDBC demo.

Verify that your connection is set up properly in connect.properties as described above, then compile and run TestInstallJDBC:

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

The program should print:

```
Hello, JDBC!
```

Verify the SQLJ Translator and Runtime

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

```
sqlj  TestInstallSQLJ.sqlj
```

After a brief wait you should get your system prompt back with no error output. Note that this command also compiles the application and customizes it to use an Oracle database.

On Solaris, the `sqlj` script is in `[Oracle Home]/bin` which should already be in your `PATH` as described above. (On Windows, use the `sqlj.exe` executable in the `bin` directory.) The SQLJ `translator.zip` file has the class files for the SQLJ translator and runtime, is located in `[Oracle Home]/sqlj/lib`, and should already be in your `CLASSPATH` as described above.

Now run the application:

```
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 (called `sqlj.properties` by default).

While still in the `demo` directory, edit the file `sqlj.properties` and update, comment, or uncomment the `sqlj.password`, `sqlj.url`, and `sqlj.driver` lines as appropriate to reflect your database connection information, as you did in `connect.properties`. For some assistance, see 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 OCI8 driver (the username will be discussed next):

```
sqlj.url=jdbc:oracle:oci8:@
sqlj.driver=oracle.jdbc.driver.OracleDriver
sqlj.password=tiger
```

Online semantics-checking is enabled as soon as you specify a username for the translation-time database connection. You can specify the username 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. This is explained in "Connection Options" on page 8-31.)

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

```
Hello, SQLJ Checker!
```

3 Basic Language Features

This chapter discusses basic SQLJ language features and constructs that you use in coding your application.

SQLJ statements always begin with a `#sql` token and can be broken into two main categories: 1) declarations, used for creating Java classes for iterators (similar to JDBC result sets) or connection contexts (which can be used to establish database connections to different kinds of schemas); and 2) executable statements, used to execute embedded SQL operations.

For more advanced topics, see Chapter 7, "Advanced Language Features".

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. Different connection context classes are typically used for connections to different types of database schemas. That is to say, instances of a particular connection context class are typically used to connect to schemas that have the same SQL objects (tables, views, stored procedures, and so on). SQLJ implements each database connection as an instance of a connection context class.

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

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

These are described in "Declaration IMPLEMENTS Clause" on page 3-5 and in "Declaration WITH Clause" on page 3-6.

Rules for SQLJ Declarations

SQLJ declarations are allowed in your SQLJ source code in the top-level scope, a class scope, or a nested-class scope but not inside method blocks. 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; // ILLEGAL (method block)
   }
}
```

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`, etc. Type declarations are separated by commas.

There are two categories of iterators—*named iterators* and *positional iterators*. For named iterators, you specify column names and types; for positional iterators, you 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 datatypes (`CHAR` and `NUMBER`).

Note: As with standard Java, any public class should be declared in a separate source file (this is a requirement if you are using the standard `javac` compiler provided with the Sun JDK). The base name of the file should be the same as the class name. Given the above example, declare the iterator class in a file `EmpIter.sqlj`.

Declaring `EmpIter` as a positional iterator instead of a named iterator would be done as follows:

```
#sql public iterator EmpIter (String, double);
```

For more information about iterators, see "Multi-Row Query Results—SQLJ Iterators" on page 3-35.

Connection Context Declarations

A connection context declaration creates a connection context class, whose instances are typically used for database connections to a particular type of database schema.

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. The following is an 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 a schema of a given type, where a schema type has a particular set of objects, such as tables, views, and stored procedures.

Note: As with standard Java, any public class should be declared in a separate source file (this is a requirement if you are using the standard `javac` compiler provided with the Sun JDK). The base name of the file should be the same as the class name. Given the above example, declare the connection context class in a file `MyContext.sqlj`.

Specified connection contexts are an advanced topic and are not necessary for basic SQLJ applications that connect to only one type of schema. In more 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 "Connection Considerations" on page 4-9 for an overview of connections and connection contexts.

For information about creating additional connection contexts, see "Connection Contexts" on page 7-2.

Declaration IMPLEMENTS Clause

In declaring 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;
```

There is potential usefulness for the `implements` clause in either an iterator declaration or a connection context declaration, but as a general comment it is more likely to be useful in iterator declarations. For information, see "Use of the IMPLEMENTS Clause in Iterator Declarations" on page 7-22 and "Use of the IMPLEMENTS Clause in Connection Context Declarations" on page 7-10.

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 empname, int empnum);
```

The declared class, `MyIter`, will implement the interface `mypackage.MyIterIntfc`.

This next example declares a connection context class that implements an interface named `MyConnCtxtIntfc` (presume it also is in package `mypackage`).

```
#sql public context MyContext implements mypackage.MyConnCtxtIntfc;
```

Declaration WITH Clause

In declaring any iterator class or connection context class, you can specify and initialize one or more constants to be included in the definition of the generated class. The constants 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 known as 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:

```
#sql <modifiers> context context_classname
     with (var1=value1,..., varN=valueN);
```

Note: Unlike the `implements` clause, there is no Java clause that corresponds to the SQLJ `with` clause.

Note: There is a predefined set of standard SQLJ constants that can be defined in a `with` clause. Attempts to define constants other than the standard constants (as in the example below) is legal with an Oracle8*i* database, but may 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, see "Translator Warnings (-warn)" on page 8-42.)

The standard constants mostly involve cursor states and can take only particular values. These constants are not meaningful to an Oracle8*i* database, but because they are supported by the SQLJ standard, they are listed here with their possible values:

- `sensitivity` (`SENSITIVE`/`ASENSITIVE`/`INSENSITIVE`)
- `holdability` (`true`/`false`)
- `returnability` (`true`/`false`)
- `updateColumns` (a `String` literal containing a comma-separated list of column names)

An iterator declaration with a `with` clause that specifies `updateColumns` must also have an `implements` clause that specifies `sqlj.runtime.ForUpdate`.

The following example uses a `with` clause in declaring a named iterator.

```
#sql public iterator MyIter with (TYPECODE=OracleTypes.NUMBER)
     (String empname, int empnum);
```

The declared class, `MyIter`, will define an attribute `TYPECODE` that will be `public static final` of type `int` and initialized to the value of the typecode for the `NUMBER` datatype, as defined in the Oracle JDBC class `oracle.jdbc.driver.OracleTypes`.

Here is another example:

```
#sql public iterator MyAsensitiveIter with (sensitivity=ASENSITIVE)
     (String empname, int empnum);
```

This declaration sets the cursor `sensitivity` to `ASENSITIVE` for a named iterator class (but note that `sensitivity` is not supported in the Oracle8*i* database).

The following example uses both an `implements` clause and a `with` clause.

```
#sql public iterator MyIter implements mypackage.MyIterIntfc
     with (holdability=true) (String empname, int empnum);
```

Note that the `implements` clause must precede the `with` clause.

This declaration implements the interface `mypackage.MyIterIntfc` and enables cursor `holdability` for a named iterator class (but note that `holdability` is not currently supported in the Oracle8*i* database).

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 that is supported by your JDBC driver (such as DML, DDL, and transaction control).

Rules for SQLJ Executable Statements

A SQLJ executable statement must follow these rules:

- It is permitted in Java code wherever Java block statements are permitted (in other words, it is permitted inside method definitions and static initialization blocks).
- Its embedded SQL must be enclosed in curly braces: `{...}`.

Notes:

- 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 (which are described in "Java Host Expressions, Context Expressions, and Result Expressions" on page 3-15).
- During examination of SQL operations, only DML operations (such as `SELECT`, `UPDATE`, `INSERT`, and `DELETE`) can be parsed and checked for syntax and semantics by the SQLJ translator using a database connection. DDL operations (such as `CREATE...`, or `ALTER...`), transaction-control operations (such as `COMMIT` and `ROLLBACK`), or any other kinds of SQL operations cannot.

SQLJ Clauses

A SQLJ *clause* is the executable part of a statement (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` below):

```
#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 example, is known as a *statement clause*. A clause that does have a result expression, such as in the second 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.

A SQL operation in a SQLJ statement can use standard SQL syntax only, or can use a clause with syntax that is specific to SQLJ (see Table 3-1 and Table 3-2 below).

For reference, Table 3-1 lists supported SQLJ statement clauses and Table 3-2 lists supported SQLJ assignment clauses. Details of how to use the various kinds of clauses are discussed elsewhere, as indicated. The last entry in Table 3-1 is a general category for statement clauses that use standard SQL syntax, as opposed to SQLJ-specific syntax.

Table 3–1 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 3-32
FETCH clause	fetch data from a positional iterator	"Using Positional Iterators" on page 3-45
COMMIT clause	commit changes to the database	"Using Manual COMMIT and ROLLBACK" on page 4-26
ROLLBACK clause	cancel changes to the database	"Using Manual COMMIT and ROLLBACK" on page 4-26
set transaction clause	use advanced transaction control—access mode and isolation level	"Advanced Transaction Control" on page 7-24
procedure clause	call a stored procedure	"Calling Stored Procedures" on page 3-57

Table 3–1 SQLJ Statement Clauses (Cont.)

Category	Functionality	More Information
assignment clause	assign values to Java host expressions	"Assignment Statements (SET)" on page 3-55
SQL clause	use standard SQL syntax and functionality: `SELECT`, `UPDATE`, `INSERT`, `DELETE`	*Oracle8i SQL Reference*

Table 3–2 SQLJ Assignment Clauses

Category	Functionality	More Information
query clause	select data into a SQLJ iterator	"Multi-Row Query Results—SQLJ Iterators" on page 3-35
function clause	call a stored function	"Calling Stored Functions" on page 3-58
iterator conversion clause	convert a JDBC result set to a SQLJ iterator	"Converting from Result Sets to Named or Positional Iterators" on page 7-31

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, use the following syntax:

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

"Connection Considerations" on page 4-9 discusses connection context instances.

If you have defined one or more *execution context* instances and want to specify one of them for use with an executable statement, use the following syntax (similar to that for connection context instances):

```
#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. This is an advanced topic; for example, you can use execution context instances in multithreading situations where multiple operations are occurring on the same connection. See "Execution Contexts" on page 7-13 for information.

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, the connection context instance must come first.

Executable Statement Examples

Examples of elementary SQLJ executable statements appear below. More complicated statements are discussed later in this chapter.

Elementary INSERT

The following example demonstrates a basic `INSERT`. The statement clause does not require any syntax that is specific to SQLJ.

Assume this table has been created:

```
CREATE TABLE EMP (
   EMPNAME CHAR(30),
   SALARY NUMBER );
```

Use the following SQLJ executable statement (simply using standard SQL syntax) to insert Joe as a new employee into the `EMP` table, specifying his name and salary.

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

Elementary INSERT with Connection Context or Execution Context Instances

The following examples use `ctx` as a connection context instance (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 (empname, salary) VALUES ('Joe', 43000) };

#sql [execctx] { INSERT INTO emp (empname, salary) VALUES ('Joe', 43000) };

#sql [ctx, execctx] { INSERT INTO emp (empname, salary) 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 Oracle SQL. The statement also uses Java host expressions, marked by colons (:), to define the values. (Host expressions are used to pass data between your Java code and SQL instructions. They are discussed in "Java Host Expressions, Context Expressions, and Result Expressions" on page 3-15.)

```
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) };
}
```

Notes:

- The `throws SQLException` is required. For information about exception-handling, see "Exception-Handling Basics" on page 4-20.
- 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 goes through a loop that inserts new employees in the `emp` table, creating employee numbers 2001-2100 (this example presumes that data other than the employee number will be filled in later).

Simple PL/SQL blocks can also be coded in a single line:

```
#sql { <DECLARE ...> BEGIN ... END; };
```

Note: Remember that using PL/SQL in your SQLJ code would prevent portability to other platforms, because PL/SQL is Oracle-specific.

Java Host Expressions, Context Expressions, and Result Expressions

There are three categories of Java expressions used in SQLJ code: *host expressions*, *context expressions*, and *result expressions*. Host expressions are the most frequently used and merit the most discussion.

SQLJ uses Java host expressions to pass arguments between your Java code and your SQL operations. This is how you pass information between Java and SQL. Host expressions are interspersed within the embedded SQL operations in 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.

(Result expressions and the specification of connection context instances and execution context instances were first introduced in "Overview of SQLJ Executable Statements" on page 3-9.)

Overview of Host Expressions

Any valid Java expression can be used as a host expression. In the simplest case, which is typical, the expression consists of just a single Java variable. Other kinds of host expressions include: arithmetic expressions, Java method calls with return values, Java class field values, array elements, conditional expressions (`a ? b : c`), logical expressions, or bitwise expressions.

Java identifiers used as host variables or in host expressions can represent any of the following:

- local variables
- declared parameters
- class fields (such as `myclass.myfield`)
- 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 and from a SQL datatype.

Host expressions can be input, output, or input-output.

See "Supported Types for Host Expressions" on page 5-2 for information about data conversion between Java and SQL during input and output operations.

Basic Host Expression Syntax

A host expression is preceded by a colon. If the desired mode of the host expression (input, output, or input-output) is not the default, then the colon must be followed (before the host expression itself) by `IN`, `OUT`, or `INOUT`, as appropriate. 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`. (When using the default, you can still include the mode specifier if desired.)

Any `OUT` or `INOUT` host expression must be assignable (an *l-value*, meaning something that can logically appear on the left side of an equals sign).

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. Therefore, to avoid any possible ambiguity, any host expression that is not a simple host variable (in other words, that is more complex than a non-dotted 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 white space (space, tab, or newline) 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 the above example, because the parenthesis is a suitable separator, but it 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())
  ```

Notes:

- 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—SQL comments after the colon and before the mode specifier, or after the colon and before the host expression if there is no mode specifier, or after the mode specifier and before the host expression (these are all in the SQL namespace), or Java comments within the host expression (inside the parentheses—this is the Java namespace).
- The `IN`, `OUT`, and `INOUT` syntax used for host variables and expressions is not case sensitive; these tokens can be uppercase, lowercase, or mixed.
- In Oracle SQLJ, the same host variable can be used multiple times in a statement. This is vendor-specific, however, and will generate warnings if you set the translator `-warning=portable` flag.
- Do not confuse the `IN`, `OUT`, and `INOUT` syntax of SQLJ host variables and expressions with similar `IN`, `OUT`, and `IN OUT` syntax used to specify the mode of parameters passed to PL/SQL stored functions and procedures.

Examples of Host Expression

The following examples will help clarify the preceding syntax discussion. (Some of these examples use `SELECT INTO` statements, which are described in "Single-Row Query Results—SELECT INTO Statements" on page 3-32.)

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 also shown in the example.

   ```
   String name = "SMITH";
   double salary = 25000.0;
   ...
   #sql { UPDATE emp SET sal = :salary WHERE ename = :name };
   ```

   `IN` is the default, but you can state it explicitly as well:

   ```
   #sql { UPDATE emp SET sal = :IN salary WHERE ename = :IN name };
   ```

   As you can see, ":" can immediately precede the variable when not using the `IN` token, but ":IN" must be followed by white space before the host variable.

2. This example uses an output host variable in a `SELECT INTO` statement, where you want to find out the name of employee number 28959.

   ```
   String empname;
   ...
   #sql { SELECT ename INTO :empname FROM emp WHERE enum = 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 enum = 28959 };
   ```

   This looks in the `enum` 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.

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 };
```

Or, to use the `IN` token:

```
#sql { UPDATE creditacct SET minPayment = :IN (balance * minPmtRatio)
       WHERE acctnum = 537845 };
```

4. This example displays 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 output from `getNewSal()` to update the `sal` column in the `emp` table for the employee (in the `ename` column) who is specified by the Java variable `ename`. Java code initializing the host variables is also shown.

```
String ename = "SMITH";
double raise = 0.1;
...
#sql {UPDATE emp SET sal = :(getNewSal(raise, ename)) WHERE ename = :ename};
```

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 that is 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 that is 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 (such as `myclass.myfield`)
- array elements

Result expressions and context expressions appear lexically in the SQLJ space, unlike host expressions, which appear lexically in the SQL space (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 Runtime

This section discusses the evaluation of Java host expressions, connection context expressions, execution context expressions, and result expressions when your application executes.

Here is a simplified representation of a SQLJ executable statement that uses all of these types of expressions:

```
#sql [connctxt_exp, execctxt_exp] result_exp = { SQL with host expression };
```

Java expressions can be used as any of the following:

- connection context expression (optional; evaluated to specify the connection context instance to be used)
- execution context expression (optional; evaluated to specify the execution context instance to be used)
- result expression (when appropriate; to receive results from a stored function, for example)
- host expression

The evaluation of Java expressions *does* have side effects in a Java program because they are evaluated by Java, not by the SQL engine. Furthermore, the order of evaluation of these expressions can be critical if any of the expressions have side effects.

The following is a summary of the overall order of evaluation, execution, and assignment of Java expressions for each statement that executes during runtime.

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.

"Examples of Evaluation of Java Expressions at Runtime" on page 3-22, has a series of examples that clarifies this sequence and discusses a number of special considerations.

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 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 results 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 runtime all host expressions are treated as distinct values, whether or not they share the same name or reference the same object. The execution of each statement is treated as 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, and the expression itself is not re-evaluated, nor any side-effects repeated, when the output assignment is made.

In discussing the evaluation order of host expressions, several points must be highlighted, as discussed in the remainder of this section.

Examples of Evaluation of Java Expressions at Runtime

This section discusses some of the subtleties of how Java expressions are evaluated when your application executes, and provides examples. (Some of these examples use `SELECT INTO` statements, which are described in "Single-Row Query Results—SELECT INTO Statements" on page 3-32; some use assignment statements, which are described in "Assignment Statements (SET)" on page 3-55; and some use stored procedure and function calls, which are described in "Stored Procedure and Function Calls" on page 3-57.)

Prefix Operators Act Before Evaluation; Postfix Operators Act After Evaluation

When a Java expression contains a Java postfix increment or decrement operator, the incrementation or decrementation occurs *after* the expression has been evaluated. Similarly, when a Java expression contains a Java prefix increment or decrement operator, the incrementation or decrementation occurs *before* the expression is evaluated.

This is equivalent to how these operators are handled in standard Java code.

Consider the following examples.

Example 1: postfix operator

```
int indx = 1;
...
#sql { ... :OUT (array[indx]) ... :IN (indx++) ... };
```

This is evaluated as follows:

```
#sql { ... :OUT (array[1]) ... :IN (1) ... };
```

The variable `indx` is incremented to 2 and will have that value the next time it is encountered, but not until after `:IN (indx++)` has been evaluated.

Example 2: postfix operators

```
int indx = 1;
...
#sql { ... :OUT (array[indx++]) ... :IN (indx++) ... };
```

This 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.

Example 3: prefix and postfix operators

```
int indx = 1;
...
#sql { ... :OUT (array[++indx]) ... :IN (indx++) ... };
```

This 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.

Example 4: postfix operator

```
int grade = 0;
int count1 = 0;
...
#sql { SELECT count INTO :count1 FROM staff
       WHERE grade = :(grade++) OR grade = :grade };
```

This is evaluated as follows:

```
#sql { SELECT count INTO :count1 FROM staff
       WHERE grade = 0 OR grade = 1 };
```

The variable `grade` is incremented to 1 after `:(grade++)` is evaluated and has that value when `:grade` is evaluated.

Example 5: postfix operators

```
int count = 1;
int[] x = new int[10];
int[] y = new int[10];
int[] z = new int[10];
...
#sql { SET :(z[count++]) = :(x[count++]) + :(y[count++]) };
```

This is evaluated as follows:

```
#sql { SET :(z[1]) = :(x[2]) + :(y[3]) };
```

The variable `count` 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 but before the third expression is evaluated; it is incremented to 4 after the third expression is evaluated and will have that value the next time it is encountered.

Example 6: postfix operator

```
int[] arr = {3, 4, 5};
int i = 0;
...
#sql { BEGIN
         :OUT (arr[i++]) := :(arr[i]);
       END; };
```

This is evaluated as follows:

```
#sql { BEGIN
         :OUT (a[0]) := :(a[1]);
       END; };
```

The variable `i` is incremented to 1 after the first expression is evaluated but before the second expression is evaluated, therefore output will be assigned to `arr[0]`. Specifically, `arr[0]` will be assigned the value of `arr[1]`, which is 4. After execution of this statement, array `arr` will have the values `{4, 4, 5}`.

IN vs. INOUT vs. OUT Makes No Difference in Evaluation Order

Host expressions are evaluated from left to right. Whether an expression is `IN`, `INOUT`, or `OUT` makes no difference in when it is evaluated; all that matter is its position in the left-to-right order.

Example 7: `IN` vs. `INOUT` vs. `OUT`

```
int[5] arry;
int n = 0;
...
#sql { SET :OUT arry[n] = :(++n) };
```

This 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. `:OUT arry[n]` is evaluated first because it is the left-most expression. Then `n` is incremented prior to evaluation of `++n`, since 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.

Example 8: evaluation of expressions in a PL/SQL block

```
int x=3;
int z=5;
...
#sql { BEGIN :OUT x := 10; :OUT z := :x; END; };
System.out.println("x=" + x + ", z=" + z);
```

This 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.

Example 9: evaluation of expressions in a PL/SQL block (with postfix)

Consider an additional 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 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 (the initial value of `y`). The second statement, `:OUT z := :x`, is evaluated before the first statement is executed, so `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.

Example 10: statements in one block vs. separate SQLJ executable statements

This 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 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, so `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 PL/SQL block example above, `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 and Only Once

Each host expression is evaluated once and only once, regardless of program flow and logic.

Example 11: 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 variable `count` will have the value 0 when it is passed to SQL (since it is operated on by a postfix operator, as opposed to a prefix operator), then will 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.

Example 12: evaluation of host expressions in conditional blocks

This example demonstrates how each expression is always evaluated, regardless of program flow. As the block is executed, only one branch of the `IF...THEN...ELSE` construct can be executed. Before the block is executed, however, 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. So `x++` is evaluated, then `x` is incremented, then each `x` is evaluated in the `(x * x)` expression. The `IF...THEN...ELSE` block is, therefore, 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 there is one, 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.

Note: Some of these examples use stored procedure and function calls, whose syntax is explained in "Stored Procedure and Function Calls" on page 3-57.

Example 13: 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.

Example 14: 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, 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.

Example 15: results assignment taking precedence over host expression assignment

This 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 an input `invar`, an output `outvar`, 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 to 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. Proceeding left to right, however, 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`. So `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 and an `IN` host expression inside a single statement or PL/SQL block. The behavior in such cases is well-defined in Oracle SQLJ, but this practice is not covered in the SQLJ specification and so code written in this manner will not be portable. Such code will generate a warning from the Oracle 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 case-insensitive.

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 from the database, SQLJ allows you to assign selected items directly to Java host expressions inside SQL syntax. This is done using the `SELECT INTO` statement. The syntax is as follows:

```
#sql { SELECT expression1,..., expressionN  INTO :host_exp1,..., :host_expN
       FROM datasource <optional clauses> };
```

Where:

- *`expression1`* through *`expressionN`* are expressions specifying what is to be selected from the database. These can be any expressions that are 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 (see the examples below).

- *`host_exp1`* through *`host_expN`* are target host expressions, such as variables or array indexes. This list of host expressions is referred to as the *INTO-list*.
- *`datasource`* is the name of the database table, view, or snapshot from which you are selecting the data.
- *`optional clauses`* are any additional clauses that 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 runtime.

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.

Notes:

- Permissible syntax for *`expression1`* through *`expressionN`*, the *`datasource`*, and the optional clauses is the same as for any SQL `SELECT` statement. For information about what is permissible in Oracle SQL, see the *Oracle8i SQL Reference*.
- There can be any number of SELECT-list and INTO-list items, as long as they match (one INTO-list item per SELECT-list item, with compatible types).

Examples of SELECT INTO Statements

The examples below assume the following table definition:

```
CREATE TABLE EMP (
   EMP_NUM NUMBER,
   EMP_NAME CHAR(30),
   HIRE_DATE DATE );
```

The first example is a `SELECT INTO` statement with a single host expression in the INTO-list:

```
String name;
#sql { SELECT emp_name INTO :name FROM emp WHERE emp_num=28959 };
```

The second example is a `SELECT INTO` statement with multiple host expressions in the INTO-list:

```
String name;
Date hiredate;
#sql { SELECT emp_name, hire_date INTO :name, :hiredate FROM emp
       WHERE emp_num=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):

```
...
#sql { SELECT :name1 INTO :name2 FROM emptable WHERE emp_num=28959 };
...
```

More realistically, you may want to perform an operation or concatenation on the data that is selected, as in the following examples (assume Java variables were previously declared and assigned, as necessary):

```
...
#sql { SELECT salary + :raise INTO :newsal FROM emptable WHERE emp_num=28959 };
...
```

```
...
#sql { SELECT :(firstname + " ") || emp_last_name INTO :name FROM emptable
       WHERE emp_num=28959 };
...
```

In the second example, `firstname` is concatenated to " " using a Java host expression and Java string concatenation (the **+** operator). This result is then passed to the SQL engine which uses SQL string concatenation (the **||** operator) to append the last name.

Multi-Row Query Results—SQLJ Iterators

A large number of SQL operations are multi-row queries. Processing multi-row query-results in SQLJ requires a SQLJ *iterator*, which is a strongly typed version of a JDBC result set and is associated with the underlying database cursor. SQLJ iterators are used first and foremost to take query results from a `SELECT` statement.

Additionally, Oracle SQLJ offers extensions that allow 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`.

For information about use as stored function returns, see "Using Iterators and Result Sets as Stored Function Returns" on page 3-60, after stored procedures and stored functions have been discussed. The other uses listed above are documented later in this section.

For information about advanced iterator topics, see "Iterator Class Implementation and Advanced Functionality" on page 7-21. This section discusses how iterator classes are implemented and what advanced functionality is available, such as interoperability with JDBC result sets.

Iterator Concepts

Before using an 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 types (and optionally the names) of the columns of data in the iterator.

A SQLJ iterator object is an instantiation 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 an instantiation of the generic java.sql.ResultSet class and can, in principle, contain any number of columns of any type.

When you declare an iterator, you specify either just the datatypes of the selected columns, or both the datatypes and the names of the selected columns:

- Specifying the names and datatypes defines a *named iterator* class.
- Specifying just the datatypes defines a *positional iterator* class.

The datatypes (and names, if applicable) that you declare determine how query results will be stored in iterator objects you instantiate from that class. SQL data that are retrieved into an iterator object are converted to the Java types that are specified in the iterator declaration.

When you query to populate a named iterator object, the names and datatypes of the SELECT-fields must match the names and types of the iterator columns (case-insensitive). The order of the SELECT-fields is irrelevant—all that matters is that each SELECT-field name matches an iterator column name. In the simplest case, the database column names directly match the iterator column names. For example, data from an EMP column in a database table can be selected and put into an iterator emp column. Alternatively, you can use an alias to map a database column name to an iterator column name. Furthermore, 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.

An an example, assume the following table:

```
CREATE TABLE EMPSAL (
   ID NUMBER(5),
   NAME VARCHAR(30),
   OLDSAL NUMBER(10),
   RAISE NUMBER(10) );
```

Given this table, you can declare and use a named iterator as follows.

Declaration:

```
#sql iterator SalNamedIter (int id, String empname, float newsalary);
```

Executable code:

```
class MyClass {
   void func() throws SQLException {
      ...
      SalNamedIter niter = null;
      #sql niter = { SELECT name AS empname, id, oldsal + raise
                     AS newsalary FROM empsal };
      ...
   }
}
```

The `id` columns match directly, an alias is used to map the database `name` column to the iterator `empname` column, and the `newsalary` target for the `oldsal + raise` operation matches the `newsalary` column of the iterator. The order of items in the `SELECT` statement does not matter.

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 datatypes 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 above, you can declare and use a positional iterator as follows.

Declaration:

```
#sql iterator SalPosIter (int, String, float);
```

Executable code:

```
class MyClass {
   void func() throws SQLException {
      ...
      SalPosIter piter = null;
      #sql piter = { SELECT id, name, oldsal + raise FROM empsal };
      ...
   }
}
```

Note that the data items are in the same order in the table, iterator, and `SELECT` statement.

Notes:

- `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 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 type of each iterator column match a database table column. (If the number of columns in the table is greater, however, a warning will be generated unless the translator `-warn=nostrict` flag is set. For information about this flag, see "Translator Warnings (-warn)" on page 8-42.)
- 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 (it can be passed and returned as a method parameter, for example) and can be declared using Java class modifiers, such as public or private.
- SQLJ supports interoperability and conversion between SQLJ iterators and JDBC result sets. For information, see "SQLJ Iterator and JDBC Result Set Interoperability" on page 7-31.
- The contents of an iterator, like a result set, is determined only by the state of the database at the time of execution of the SELECT statement that populated it. Subsequent updates, insertions, deletions, commits, or rollbacks have no affect on the iterator or its contents. This is further discussed in "Effect of Commit and Rollback on Iterators and Result Sets" on page 4-27.

General Steps in Using an Iterator

Five general steps are involved in using either kind of SQLJ 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, as explained below).
5. When you finish processing the results of the query, close the iterator to release its resources.

Named Iterators vs. Positional Iterators

There are advantages and appropriate situations for each of the two kinds of SQLJ iterators.

Named iterators allow greater flexibility. Because data selection into a named iterator matches SELECT-fields 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 don't match, the SQLJ translator will generate an error when it checks your 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 row of the iterator upon completion.)

Positional iterators do, however, 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. You also must select data into all columns of the iterator, and it is possible to have data written into the wrong iterator column if the type of that column happens to match the datatype 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 `emp()` method to retrieve data from an `emp` iterator column). With positional iterators, you must fetch data directly into Java host expressions with your `FETCH INTO` statement and the host expressions must be in the correct order.

Note:

- 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. (For information about this flag, see "Translator Warnings (-warn)" on page 8-42.)
- 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 is because you are fetching data from the iterator, not the database.

Using Named Iterators

When you declare a named iterator class, you declare the name as well as the datatype of each column of the iterator.

When you select data into a named iterator, the SELECT-fields must match the iterator columns in two ways:

- The name of each SELECT-field (either a table column name or an alias) must match an iterator column name (case-insensitive, so `emp` would match `EMP`).
- The type of each iterator column must be compatible with the datatype of the corresponding SELECT-field, according to standard JDBC type mappings.

The order in which attributes are declared in your 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 name is identical to the column name. (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—`next()`, `close()`, `getResultSet()`, `isClosed()`.
- Column names cannot have the same characters as each other, such as `ENAME` and `Ename`. Name-matching is case-insensitive, so this would present an ambiguous situation. Similarly, if you want to use a named iterator to select columns from a table that has two column names with the same characters (again, `ENAME` and `Ename`), then you must use a different name for one of them in your iterator and use an alias in your `SELECT` statement to match this name.

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 );
```

Where *`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 (convertible from) 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. These are discussed in "Declaration IMPLEMENTS Clause" on page 3-5 and "Declaration WITH Clause" on page 3-6.

For example, you might declare the following named iterator for use with company projects:

```
#sql public iterator ProjIter (String projname, int id, Date deadline);
```

This will result in an iterator class with columns of data that are accessible using the following provided accessor methods: `projname()`, `id()`, and `deadline()`.

Instantiating and Populating Named Iterators

Declare a named iterator variable and then instantiate and populate it using a `SELECT` statement. For example, declare an instance of the above `ProjIter` named iterator:

```
ProjIter projs;
```

Now presume you want to populate this iterator with data from a database table defined as follows:

```
CREATE TABLE PROJECTS (
   ID NUMBER(4),
   NAME VARCHAR(30),
   START_DATE DATE,
   DURATION NUMBER(3) );
```

There are columns in this table whose names and datatypes match the `id` and `name` columns of the iterator, but you must use an alias (and perform an operation as appropriate) to populate the `deadline` column of the iterator. Here is an example:

```
#sql projs = { SELECT start_date + duration AS deadline, name, 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 iterator column `deadline`. It also uses a `WHERE` clause so that only future deadlines are processed (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 database table `Emp` with a column `comm` and a function `maximum()` that takes a `comm` entry and an integer as input and returns the maximum of the two. (For example, you could input a 0 to avoid negative numbers in your iterator.) Additionally, suppose you declare an iterator class with a corresponding `maxComm` column.

Now consider the following `SELECT` syntax:

```
SELECT maximum(comm, 0) FROM Emp
```

This is valid `SELECT` syntax, but you cannot use it to populate a named iterator because "maximum(comm, 0)" is not an iterator column name (and cannot possibly be, because that is not a valid Java identifier). You can, however, work around this problem by using an alias, as follows:

```
#sql emps = { SELECT maximum(comm, 0) AS maxComm FROM Emp };
```

Generally, you must use an alias in your query for any SELECT-field whose name is not a legal Java identifier or does not match a column name in your 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 (unmatched columns are ignored), but 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 that are generated by SQLJ, typically inside a `while` loop.

Whenever `next()` is called:

- If there is another row to retrieve from the iterator, `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 used under "Instantiating and Populating Named Iterators" on page 3-42).

Note: Each iterator has a `close()` method, which you must always call once 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 projs = null;
```

```
// Instantiate and populate iterator; order of SELECT doesn't matter
#sql projs = { SELECT start_date + duration as deadline, projname, id
               FROM projects WHERE start_date + duration >= sysdate };

// Process the results
while (projs.next()) {
   System.out.println("Project name is " + projs.projname());
   System.out.println("Project ID is " + projs.id());
   System.out.println("Project deadline is " + projs.deadline());
}

// Close the iterator
projs.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.

Remember, however, that accessor method names are created with the case exactly as in your declaration of the iterator class. The following will generate compilation errors.

Declaration:

```
#sql iterator Cursor1 (String NAME);
```

Executable code:

```
...
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()`, not `name()`. You would have to use `c1.NAME()` in the `System.out.println` statement.

For a complete sample of using a named iterator, see "Named Iterator—NamedIterDemo.sqlj" on page 12-5.

Using Positional Iterators

When you declare a positional iterator class, you only declare the datatype of each column; you do not define column names. The Java types into which the columns of the SQL query results are selected must be compatible with the datatypes of the SQL data. The names of the database columns or SELECT-fields 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 (as detailed under "Accessing Positional Iterators" on page 3-46).

Declaring Positional Iterator Classes

Use the following syntax to declare a positional iterator class:

```
#sql <modifiers> iterator classname <implements clause> <with clause>
     ( position-list );
```

Where *`modifiers`* is an optional sequence of legal Java class modifiers, and the *`position-list`* is a list of the 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. These are discussed in "Declaration IMPLEMENTS Clause" on page 3-5 and "Declaration WITH Clause" on page 3-6.

For example, consider a database table defined as follows:

```
#sql public iterator EmpIter (int, String, float);
```

This defines Java class `EmpIter` with unnamed integer, string, and float columns.

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 your SELECT-fields are in the proper order.

Declare a variable of the `EmpIter` positional iterator class:

```
EmpIter emps = null;
```

Now presume that you want to populate this iterator with data from a database table defined as follows:

```
CREATE TABLE EMPLOYEES (
   EMPNUM NUMBER(5),
   EMPSAL NUMBER(8,2),
   EMPNAME VARCHAR2(30) );
```

These three datatypes are compatible with the types of the `EmpIter` positional iterator columns, but be careful about how you select the data because the order is different. The following will work—instantiating and populating the iterator—as the SELECT-fields are in the same order as the iterator columns:

```
// Populate iterator emps
#sql emps = { SELECT empnum, empname, empsal FROM employees };
```

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 that is provided with all positional iterator classes to determine when you have reached the end of the data.

Notes:

- The `endFetch()` method initially returns `true` before any rows have been fetched, then returns `false` once a row has been successfully fetched, then returns `true` again after the last row has been fetched. 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, though, because the `FETCH` does not throw a SQL exception when it reaches the end of the data. 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, which 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 used under "Instantiating and Populating Positional Iterators" on page 3-45).

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 (int, String, float);
...
```

Populate and then access an instance of this iterator class as follows:

```
// Declare and initialize host variables
int id=0;
String name=null;
float salary=0.0f;
```

```
// Declare an iterator instance
EmpIter emps;

#sql emps = { SELECT empnum, empname, empsal FROM employees };

while (true) {
   #sql { FETCH :emps INTO :id, :name, :salary };
   if (emps.endFetch()) break;   // This test must be AFTER fetch,
                                 // but before results are processed.
   System.out.println("Name is " + name);
   System.out.println("Employee number is " + id);
   System.out.println("Salary is " + salary);
   ...
}

emps.close();
...
```

The `id`, `name`, and `salary` variables are Java host variables whose types must match the types of the iterator columns.

Explicitly using the `next()` method is not necessary for a positional iterator, because `FETCH` 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 only assign the variables if there was a row to be fetched.)

For a complete sample of using a positional iterator, see "Positional Iterator—PosIterDemo.sqlj" on page 12-9.

Using Iterators and Result Sets as Host Variables

SQLJ supports SQLJ iterators and JDBC result sets as host variables, as illustrated in the examples below.

Notes:

- Additionally, SQLJ supports iterators and result sets as return variables for stored functions. This is discussed in "Using Iterators and Result Sets as Stored Function Returns" on page 3-60.
- The Oracle JDBC drivers do not currently support result sets as input host variables. There is a `setCursor()` method in the `OraclePreparedStatement` class but it raises an exception at runtime if called.

As you will see from the following examples, using iterators and result sets is fundamentally the same, with differences in declarations and in accessor methods to retrieve the data.

Example: Use of Iterator as OUT Host Variable This example uses a named iterator as an output host variable.

Declaration:

```
#sql iterator EmpIter (String ename, int empno);
```

Executable code:

```
...
EmpIter iter;
...
#sql { BEGIN
          OPEN :OUT iter FOR SELECT ename, empno FROM emp;
       END; };

while (iter.next())
{
   String name = iter.ename();
   int empno = iter.empno();
}
iter.close();
...
```

This example opens 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, then loops through the iterator to retrieve data into local variables.

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 name = rs.getString(1);
   int empno = rs.getInt(2);
}
rs.close();
...
```

This example opens 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, then loops through the result set 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. For information about `SELECT INTO` statements and syntax, see "Single-Row Query Results—SELECT INTO Statements" on page 3-32.)

Declaration:

```
#sql iterator DNameIter (String dname);
```

Executable code:

```
...
DNameIter dnameIter;
String ename;
...

#sql { SELECT ename, cursor (SELECT dname FROM dept WHERE deptno = emp.deptno)
       INTO :ename, :dnameIter FROM emp WHERE empno = 7788 };
```

```
System.out.println(ename);
while (dnameIter.next())
{
   System.out.println(dnameIter.dname());
}
dnameIter.close();
...
```

This example uses nested `SELECT` statements to accomplish the following:

- Selects the name of employee number 7788 from the `emp` table, selecting it into the output host variable `ename`.
- Selects data from the `dept` table into a cursor wherever the department number matches one of employee 7788's department numbers, selecting that cursor into the output host variable `dnameIter`, which is a named iterator.
- Prints the employee's name.
- Loops through the named iterator, printing all of his or her department numbers.

Using Iterators and Result Sets as Iterator Columns

Oracle SQLJ includes extensions that allow iterator declarations to specify columns of type `ResultSet` or columns of other iterator types that were declared within the current scope. In other words, iterators and result sets can exist within iterators in Oracle SQLJ. 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, but the first example uses named iterators within a named iterator, the second example uses result sets within a named iterator, and the third example uses named iterators within a positional iterator.

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).

Declarations:

```
#sql iterator DNameIter (String dname);
#sql iterator NameDeptIter2 (String ename, DNameIter depts);
```

Executable code:

```
...
NameDeptIter2 iter;
...
#sql iter = { SELECT ename, cursor
             (SELECT dname FROM dept WHERE deptno = emp.deptno)
              AS depts FROM emp };

while (iter.next())
{
   System.out.println(iter.ename());
   DNameIter deptsIter = iter.depts();
   while (deptsIter.next())
   {
      System.out.println(deptsIter.dname());
   }
   deptsIter.close();
}
iter.close();
...
```

This example uses a nested iterator (iterators in a column within another iterator) to print all the departments of each employee in the `emp` table, as follows:

- Selects each `ename` (employee name) from the `emp` table.
- Does a nested `SELECT` into a cursor to get all the departments from the `dept` table for each employee (each department that the employee belongs to).
- Puts this name and department data into the outer iterator (`iter`), which has a name column and an iterator column. The cursor with the department information for any given employee goes into the iterator column of that employee's row of the outer iterator.
- Goes through a nested loop that, for each employee, prints their name and then loops through the inner iterator to print their department names. (Each employee can be in multiple departments in this example.)

Example: Result Set Column in a Named Iterator This example uses a column of type `ResultSet` in a named iterator. This example is essentially equivalent to the previous example, except it uses result sets inside an iterator instead of nested iterators.

Declaration:

```
#sql iterator NameDeptIter1 (String ename, ResultSet depts);
```

Executable code:

```
...
NameDeptIter1 iter;
...
#sql iter = { SELECT ename, cursor
             (SELECT dname FROM dept WHERE deptno = emp.deptno)
              AS depts FROM emp };

while (iter.next())
{
   System.out.println(iter.ename());
   ResultSet deptsRs = iter.depts();
   while (deptsRs.next())
   {
      String deptName = deptsRs.getString(1);
      System.out.println(deptName);
   }
   deptsRs.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 there was when the outer iterator was a named iterator in the earlier example.

Declarations:

```
#sql iterator DNameIter (String dname);
#sql iterator NameDeptIter3 (String, DNameIter);
```

Executable code:

```
...
NameDeptIter3 iter;
...
#sql iter = { SELECT ename, cursor
             (SELECT dname FROM dept WHERE deptno = emp.deptno) FROM emp };

while (true)
{
   String ename = null;
   DNameIter deptsIter = null;
   #sql { FETCH :iter INTO :ename, :deptsIter };
   if (iter.endFetch()) break;
   System.out.println(ename);
   while (deptsIter.next())
   {
      System.out.println(deptsIter.dname());
   }
   deptsIter.close();
}
iter.close();
...
```

Assignment Statements (SET)

SQLJ allows 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:

```
#sql { BEGIN :OUT host_exp := expression; END; };
```

Here is a simple example of an assignment statement:

```
#sql { SET :x = foo1() + foo2() };
```

This 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 (750 + 250). (The `substr()` calls take the first three characters of the strings, or '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 normal function call syntax as explained in "Stored Procedure and Function Calls" on page 3-57. You also do not need an assignment statement to manipulate output from Java functions, because you can accomplish that in a normal Java statement. So you can presume that `foo1()` and `foo2()` above 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, as described immediately below. These procedures and functions could be written in Java, PL/SQL (in an Oracle database), 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 can optionally take input, output, or input-output parameters as well.

A stored procedure does not have a return value but can optionally 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, then call the Java wrappers as you would any other Java methods. JPublisher is discussed in "JPublisher and the Creation of Custom Java Classes" on page 6-18. For additional information, see the *Oracle8i JPublisher User's Guide*.

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, as shown below. The word "CALL" is followed by a space and then the procedure call. 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, as explained in "Calling Stored Functions" on page 3-58).

```
#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.

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 table `projects`, looks at the `start_date` and `duration` columns, calculates `start_date + duration` in each row, then takes the maximum `start_date + duration` total and selects it into `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` (optional/default), `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, 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, as shown below. This syntax consists of the word "VALUES" followed by the function call. In standard SQLJ, the function call must be enclosed in a set of outer parentheses, as shown. In Oracle SQLJ, the outer parentheses are optional. When using the outer parentheses, it does not matter if there is a space between the `VALUES` token and the begin-parenthesis. (A `VALUES` token can also be used in `INSERT INTO` *`table`* `VALUES` syntax that is supported by Oracle SQL, but these situations are unrelated semantically.)

```
#sql result = { VALUES(FUNC(<PARAM_LIST>)) };
```

Where *`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.

Referring back to the example in "Calling Stored Procedures" on page 3-57, consider defining the stored procedure as a stored function instead, as follows:

```
CREATE OR REPLACE FUNCTION get_max_deadline() RETURN DATE IS
   DECLARE
      DATE deadline;
   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 that is compatible with the return type of the function.

In Oracle SQLJ, the following syntax (outer parentheses omitted) is also allowed:

```
#sql maxDeadline = { VALUES get_max_deadline };
```

For stored function calls, as with stored procedures, you must use the host expression tokens `IN` (optional—default), `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.

Notes: 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.

If you want your application to be compatible with Oracle7, then do not include empty parentheses for the parameter list if the function takes no parameters. For example:

```
#sql maxDeadline = { VALUES(get_max_deadline) };
```

not:

```
#sql maxDeadline = { VALUES(get_max_deadline()) };
```

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, provided that 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. (`REF CURSOR` types are described in "Support for Oracle REF CURSOR Types" on page 5-33.)

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;
```

Use this function as follows:

```
...
#sql iterator EmpIter (String ename, int empno);
...
EmpIter iter;
...
#sql iter = { VALUES(sqlj_refcursor.job_listing('SALES')) };

while (iter.next())
{
   String name = iter.ename();
   int empno = iter.empno();
}
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.

4

Key Programming Considerations

This chapter discusses key issues you must consider before developing and running your SQLJ application, concluding with a summary and sample applications. The following topics are discussed:

Naming Requirements and Restrictions

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

Java Namespace—Local Variable and Class Naming Restrictions

The *Java namespace* applies to all of your standard Java statements and declarations, including the naming of Java classes and local variables. All standard Java naming restrictions apply, and you should avoid 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 3-31.

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 executable SQL statement with a statement block, where the executable SQL 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_
```

(There are two underscores at the beginning and one at the end.)

The following declarations are examples of those which 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 if `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`, SQLJ generates a profile-keys class called `Foo_SJProfileKeys`. Do not declare a class name that conflicts with that.

- 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 that is 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 the user-defined type `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 that is outside the curly braces.

Note: Restrictions that are particular to the naming of iterator columns are discussed in "Using Named Iterators" on page 3-40.

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`.

Note, however, 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 that is inside the curly braces. Normal SQL naming restrictions apply here.

Note, however, that host expressions follow rules of the Java namespace, not the SQL namespace. This applies to the name of a host variable, or 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 one of the defined classes.

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 are according to the Java Language Specification and are not SQLJ-specific. They do not apply in the Oracle8*i* server, but are still recommended.

Selection of the JDBC Driver

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

This discussion begins with a brief description of the Oracle JDBC drivers, but SQLJ supports any standard JDBC driver.

About Oracle JDBC Drivers

If you are connecting to an Oracle database and using an Oracle JDBC driver, your choices are the following:

- Thin driver
- OCI 8 driver
- OCI 7 driver
- Server-side driver

The Thin driver is a pure Java alternative that you must use for applets but can also use for applications.

The OCI 8 and OCI 7 drivers are written partly in native C code and communicate with the database through the OCI (Oracle Call Interface) layer. These drivers require an Oracle client installation and therefore cannot be used for applets.

The server-side driver, formally known as the KPRB (Kernel PRogram Bundled calls) driver, is the JDBC driver in the Oracle8*i* server. This is used if your SQLJ code will run as a stored procedure or function, trigger, Enterprise JavaBean, or CORBA object in the database.

All four drivers are supported by the `oracle.jdbc.driver.OracleDriver` class.

For information about these drivers and about which will be most appropriate for your particular situation, see the *Oracle8i JDBC Developer's Guide and Reference*.

Remember that your choices may differ between translation time and runtime. For example, you may want to use the Oracle JDBC OCI 8 driver at translation time for semantics-checking but the Oracle JDBC Thin driver at runtime.

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 Thin, OCI 8, or server-side for an Oracle database) as part of the connection URL you specify in the SQLJ `-url` option.

For information about these options, see "Connection Options" on page 8-31.

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

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

Driver Selection and Registration for Runtime

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

If you are using an Oracle JDBC driver and call the standard `Oracle.connect()` method to create a default connection, then SQLJ handles this automatically—`Oracle.connect()` registers the `oracle.jdbc.driver.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.driver.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, username, and password. This is described in "Single Connection or Multiple Connections Using DefaultContext" on page 4-9. This section also further discusses the `Oracle.connect()` method.

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, will you want to use connection context classes other than the default `sqlj.runtime.ref.DefaultContext` class?
- Will you need different connection specifications for translation and runtime, 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 thereby use the default connection (an instance of `DefaultContext` that you previously set as the default).

Note: If you will be connecting to different database schema types, then you will typically want to declare and use your own connection context classes. This is discussed in "Connection Contexts" on page 7-2.

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 to the same type of database schema. (It is permissible, however, to use different `DefaultContext` instances with different types of schemas.)

Single Connection

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

You can use the `connect()` method of the `oracle.sqlj.runtime.Oracle` class to accomplish this. This method has several signatures, including ones that allow you to specify username, 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");
```

(Where `MyClass` is the name of your class. There is an example of `connect.properties` in `[Oracle Home]/sqlj/demo`, and also in "Set Up the Runtime Connection" on page 2-6.)

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

Alternatively, you can specify username, password, and URL directly:

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

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

Notes:

- `Oracle.connect()` will not set your default connection if one had already been set. In that case, it returns `null`. If you do want to override your default connection, use the static `setDefaultContext()` method of the `DefaultContext` class, as described in the next section.
- 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; this is done in some of the SQLJ demo applications.

In the second example, the connection will use the JDBC Thin driver to connect user `scott` (password `tiger`) to a database on the machine `localhost` through port 1521, where `orcl` is the SID of the database to connect to on that machine.

Once you have completed these steps you do not need to specify the connection for any of the SQLJ executable statements in your application.

Multiple Connections

For multiple connections, you can create and use additional instances of the `DefaultContext` class, while optionally still using the default connection created under "Single Connections" above.

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 created above, 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:orcl2", "bill", "lion");
```

(Or `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 };
```

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 allows you to switch your default back and forth, for example.

The following statements establish multiple connections to the same schema (in case you want to use multiple 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:orcl1", "scott", "tiger");
DefaultContext ctx2 = Oracle.getConnection (
   "jdbc:oracle:thin:@localhost1:1521:orcl1", "scott", "tiger");
```

This creates two connection context instances that would use the same schema, connecting to `scott/tiger` on database SID `orcl1` on the machine `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:orcl1", "scott", "tiger");
DefaultContext ctx2 = Oracle.getConnection (
   "jdbc:oracle:thin:@localhost2:1521:orcl2", "bill", "lion");
```

This creates two connection context instances that both use the Oracle JDBC Thin driver but use different schemas. The `ctx1` object connects to `scott/tiger` on database SID `orcl1` on the machine `localhost1`, while the `ctx2` object connects to `bill/lion` on database SID `orcl2` on the machine `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:** Remember to include the square brackets around the connection context instance name; they are part of the syntax.

or:

- 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 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.

Multiple Connections Using Declared Connection Context Classes

For multiple connections to different types of database schemas, you typically use connection context declarations to define your own connection context classes. Having a separate connection context class for each type of schema you use allows SQLJ to do more rigorous semantics-checking of your code.

See "Connection Contexts" on page 7-2 for more information.

More About the Oracle Class

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

The static `connect()` method instantiates a `DefaultContext` object and implicitly installs this instance as your default connection. You do not need to assign or use the `DefaultContext` instance that is returned by `connect()`. If you had already established a default connection, then `connect()` returns `null`.

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

Both methods register the Oracle JDBC driver manager automatically if the `oracle.jdbc.driver.OracleDriver` class is found in your `CLASSPATH`.

Each method has signatures that take the following parameters as input:

- URL (`String`), username (`String`), password (`String`)
- URL (`String`), username (`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 username and password

- URL (`String`), auto-commit flag (`boolean`)
- `java.lang.Class` object for class used to load properties file, name of properties file (`String`)
- `java.lang.Class` object, name of properties file (`String`), auto-commit flag (`boolean`)
- `java.lang.Class` object, name of properties file (`String`), username (`String`), password (`String`)
- `java.lang.Class` object, name of properties file (`String`), username (`String`), password (`String`), auto-commit flag (`boolean`)
- JDBC `Connection` object
- 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.

Note: The auto-commit flag is used to specify whether or not operations are automatically committed. 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.

The auto-commit flag is discussed in "Basic Transaction Control" on page 4-24.

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. (This interface is described in "Implementation and Functionality of Connection Context Classes" on page 7-8.)

The `DefaultContext` class has the same class definition that would have been generated by the SQLJ translator from the declaration:

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

The `DefaultContext` class has three methods of note:

- `getConnection()`—Gets the underlying JDBC `Connection` object. This is useful if you must use JDBC in your application for 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()`—This is a `static` method that sets the default connection your application uses; 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 that you define using `Oracle.connect()`).
- `getDefaultContext()`—This is a `static` method that returns the `DefaultContext` instance currently defined as the default connection for your application (through earlier use of the `setDefaultContext()` class method).

Note: On a client, `getDefaultContext()` returns `null` if `setDefaultContext()` was not previously called. In the server, it returns the default connection (the connection to the server itself).

It is typical to instantiate `DefaultContext` using the `Oracle.connect()` or `Oracle.getConnection()` method. If you want to create an instance directly, however, there are five constructors for `DefaultContext`, which take the following parameters as input:

- URL (`String`), username (`String`), password (`String`), auto-commit (`boolean`)
- URL (`String`), `java.util.Properties` object, auto-commit (`boolean`)
- URL (`String` fully specifying connection and including username and password), auto-commit setting (`boolean`)
- JDBC `Connection` object only
- SQLJ connection context object only

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

Notes:

- To use any of the first three constructors above, you must first register your JDBC driver. This happens automatically if you are using an Oracle JDBC driver and call `Oracle.connect()`. Otherwise, see "Driver Selection and Registration for Runtime" on page 4-7.
- Any connection context class that you declare will have the same constructor signatures as the `DefaultContext` class.
- 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 or not SQL operations are automatically committed. For more information, see "Basic Transaction Control" on page 4-24.

Connection for Translation and Runtime

You can use different connections for translation and runtime. For example, you might want a different connection for translation than for runtime because you are not developing in the same kind of environment that your application will be running in. Or you may need different connections to specify different drivers, perhaps because you want to use the OCI 8 driver for semantics-checking but must use the Thin driver for runtime because you are developing an applet.

Notes:

- Oracle SQLJ supports specified connections for profile customization as well (this could be done through the `-P` command-line prefix), but the Oracle customizer does not use a connection.
- Before specifying your connection, consider which JDBC driver to use and register the appropriate driver class (for Oracle JDBC drivers this is `oracle.jdbc.driver.OracleDriver`; this is registered automatically if you use `Oracle.connect()` to create your default connection). See "Selection of the JDBC Driver" on page 4-6 for more information.

Connection at Translation Time

Use SQLJ translator option settings, either on the command line or in a properties file, to specify a connection for translation. Use the SQLJ `-url`, `-user`, and `-password` options.

The `-url` setting includes which JDBC driver to use.

For information about these options, see "Connection Options" on page 8-31.

You will usually, but not necessarily, use the same URL, username, and password as you do in your source code for the runtime connection.

Connections at Runtime

Specify a connection or connections for runtime in your SQLJ source code or in a properties file (such as `connect.properties`) to be read by `Oracle.connect()`. The connection URL includes which JDBC driver to use. See "Single Connection or Multiple Connections Using DefaultContext" on page 4-9.

Null-Handling

Java primitive types (such as `int`, `double`, or `float`) cannot legally have null values, which you must consider in choosing your result expression and host expression types.

Wrapper Classes for Null-Handling

When receiving data from the database, SQL nulls are converted to Java null values. Therefore, do not use Java primitive types for output variables in situations where a SQL null may be received.

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 values 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`

Notes:

- `SQLNullException` is a subclass of the standard `java.sql.SQLException` class. See "Using SQLException Subclasses" on page 4-23.
- Because Java objects can have null values, there is no need in SQLJ for indicator variables such as those used in other host languages (C, C++, and COBOL for example).

Examples of Null-Handling

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

Example: Null Input Host Variable In the following example, a `Float` object is used to pass a null value to the database. You cannot use the Java primitive type `float` to accomplish this.

Presume the following declarations:

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

Example:

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

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

Presume the following declarations:

```
#sql iterator EmployeeIter (String name, Double commission);
EmployeeIter ei;
```

Example:

```
#sql ei = { SELECT name, commission FROM emps WHERE salary >= 50000 };

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

Exception-Handling Basics

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

SQLJ/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 `java.sql.SQLException` class. You can either import it as follows, or fully qualify its name whenever you need it:

```
import java.sql.SQLException;        // or optionally java.sql.*
```

Example: Exception Handling This example demonstrates the kind of basic exception-handling that is required of SQLJ applications, with a `main` method with a `try/catch` block, and another method which is called from `main` and throws exceptions back to `main` when they are encountered.

```
/* 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.SQLException ;
import oracle.sqlj.runtime.Oracle;

// iterator for the select

#sql iterator MyIter (String ITEM_NAME);

class TestInstallSQLJ
{
  //Main method
  public static void main (String args[])
  {
    try {
      /* if you're using a non-Oracle JDBC Driver, add a call here to
         DriverManager.registerDriver() to register your Driver
      */
      // 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();
      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 runtime, exceptions may come from any of the following:

- SQLJ runtime
- JDBC driver
- RDBMS

Printing Error Text

The example in the previous section showed how to catch SQL exceptions and output the error messages, which is repeated again here.

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

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

For error messages from the SQLJ runtime, see "Runtime Messages" on page B-42. You can also retrieve SQL state information, as described below.

For error messages from the Oracle JDBC driver, see the *Oracle8i JDBC Developer's Guide and Reference*.

For error messages from the Oracle RDBMS, which will be preceded by the prefix `ORA-xxxxx`, where `xxxxx` is a five-digit error code, see the *Oracle8i Error Messages* reference.

Retrieving SQL States and Error Codes

The `java.sql.SQLException` class and subclasses include the methods `getSQLState()`, `getErrorCode()`, and `getMessage()`. Depending on where the exception came from and how error conditions are implemented there, these methods may provide additional information as follows.

For exceptions from the Oracle SQLJ runtime:

- `getSQLState()` returns a five-digit string containing the SQL state.
- `getErrorCode()` returns 0 (no meaningful information).
- `getMessage()` returns an error message (with no prefix).

For exceptions from the Oracle JDBC drivers:

- `getSQLState()` returns null (no meaningful information).
- `getErrorCode()` returns 0 (no meaningful information).
- `getMessage()` returns an error message (with no prefix).

For exceptions from the RDBMS:

- `getSQLState()` returns an empty string (no meaningful information).

- `getErrorCode()` returns the Oracle error code, which is the `xxxxx` portion of the `ORA-xxxxx` prefix. (For example, this would return `942` for the message prefixed by `ORA-00942`.)
- `getMessage()` returns an error message, including the `ORA-xxxxx` prefix.

For example, you can use the following error processing. This prints the error message as in the preceding example, but 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);
}
...
```

SQL states and error messages for SQLJ runtime errors are documented in "Runtime Messages" on page B-42.

Using SQLException Subclasses

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

SQLJ provides one such subclass, the `sqlj.runtime.NullException` class, which you can catch in situations where a null value might be returned into a Java primitive variable. (Java primitives cannot handle nulls.)

When you use a `SQLException` subclass, catch the subclass exception first, before catching a `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 would not drop through for any special processing you want to use for the subclass exception.

Basic Transaction Control

This section discusses how to manage your changes to the database.

For information about SQLJ support for more advanced transaction control functions—access mode and isolation level—see "Advanced Transaction Control" on page 7-24.

About Transactions

A *transaction* is a sequence of SQL statements 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 changes to the database, either automatically or manually)
- rollback (canceling changes to the database)

A transaction ends with a commit or rollback.

(Note that in Oracle SQLJ, all DDL commands such as `CREATE...` and `ALTER...` include an implicit commit.)

Automatic Commits vs. Manual Commits

In using SQLJ or JDBC, you can either have your changes automatically committed to the database or commit them manually. In either case, each commit starts a new transaction. You can specify automatic commits by enabling the auto-commit flag, 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. You have no option to roll back changes, for example. You also cannot save a batch of updates to be committed all at once—operations are committed as soon as they are executed. This results in slower overall execution because every individual operation incurs the full overhead of a transaction, instead of this overhead being spread over several operations.

There are also some SQLJ or JDBC features that are incompatible with automatic commits. 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. There are signatures of these methods, however, that allow 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:

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

Or you can specify a `true` setting:

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

For the complete list of signatures for `Oracle.connect()` and `Oracle.getConnection()`, see "More About the Oracle Class" on page 4-13.

If you use a constructor to create a connection context instance, either of `DefaultContext` or of a declared connection context class, 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:orcl", "scott", "tiger", false);
```

For the complete list of signatures for `DefaultContext` constructors, see "More About the DefaultContext Class" on page 4-14.

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, as described in "Modifying Auto-Commit in an Existing Connection" on page 4-26.)

Note: Auto-commit functionality is not supported by the server-side JDBC 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 desired. 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 you have declared).

You can accomplish these two steps at once, as follows. In these examples, `ctx` is a SQLJ connection context instance:

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

or:

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

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 changes to the database.

To commit any changes (such as updates, inserts, or deletes) that have been executed since the last commit, use the SQLJ `commit` statement, as follows:

```
#sql { commit };
```

To rollback (cancel) any changes that have been executed since the last commit, use the SQLJ `rollback` statement, as follows:

```
#sql { rollback };
```

Do not use the `commit` or `rollback` commands when auto-commit is enabled. This will result in unspecified behavior (or perhaps SQL exceptions).

Notes:

- All DDL statements in Oracle SQL include an implicit `commit`. There is no special SQLJ functionality in this regard; such statements follow standard Oracle SQL rules.
- If auto-commit is off and you close a connection context instance from a client application, then any changes since your last `commit` will be rolled back (unless you close the connection context instance with `KEEP_CONNECTION`, which is explained in "Implementation and Functionality of Connection Context Classes" on page 7-8).

Effect of Commit and Rollback on Iterators and Result Sets

Commits (either automatic or manual) and rollbacks do not affect open result sets and iterators. The result sets and iterators will still be open, and 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.

This also applies to `UPDATE`, `INSERT`, and `DELETE` statements which 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`, then `COMMIT`. A 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`, then `ROLLBACK`. A result set or iterator populated by the `SELECT` will still contain the updated data, regardless of the subsequent `ROLLBACK`.

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, make sure to 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 of the `java.sql` package, however. Key classes there are `java.sql.SQLException` and any classes that you refer to explicitly (for example, `java.sql.Date`, `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 runtime classes directly in your code, import the following packages:

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

If your code does not use any SQLJ runtime classes directly, however, it will be sufficient to have them in your `CLASSPATH` as described in "Set the PATH and CLASSPATH" on page 2-4.

(Key runtime classes include `AsciiStream`, `BinaryStream`, and `ResultSetIterator` 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. If you are using a non-Oracle JDBC driver, you must add code to register it (as mentioned in the code comments below).

This uses a signature of connect() that takes the URL, username, and password from the file connect.properties. An example of this file is in the directory [Oracle Home]/sqlj/demo and also in "Set Up the Runtime Connection" on page 2-6.

```
public class SimpleExample {

  public SimpleExample() throws SQLException {
    /* If you are using a non-Oracle JDBC driver, add a call here to
       DriverManager.registerDriver() to register your driver.  */
    // Set default connection (as defined in connect.properties).
    Oracle.connect(getClass(), "connect.properties");
  }
```

(The main() method is defined below.)

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);
   }
}
```

(The runExample() method is defined below.)

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.

```
void runExample() throws SQLException {

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

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

     // Use SELECT INTO statement to execute query and retrieve values.
      #sql { SELECT Ename, Sal INTO :ename, :salary FROM Emp
             WHERE Empno = 7499 };

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

This declares `salary` and `ename` 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 `ename` are printed out.

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 {
    /* If you are using a non-Oracle JDBC driver, add a call here to
       DriverManager.registerDriver() to register your driver.  */
    // 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);
    }
  }

  void runExample() throws SQLException {

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

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

     // Use SELECT INTO statement to execute query and retrieve values.
        #sql { SELECT Ename, Sal INTO :ename, :salary FROM Emp
              WHERE Empno = 7499 };

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

Set Up Named Iterator

The next example will build 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, Integer is required.)
      String ename,
      String job,
      Integer mgr,
      Date hiredate,
      Float sal,
      Float comm,
      int deptno);
```

Later, when needed, 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 when you are done.

```
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.

Aside 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 {
      /* If you are using a non-Oracle JDBC driver, add a call here to
         DriverManager.registerDriver() to register your driver.  */
      // 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 );
    }
  }

  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() ;
  }
}
```

5 Type Support

This chapter documents data types supported by Oracle SQLJ, listing supported SQL types and the Java types that correspond to them, including information about backwards compatibility to Oracle8 and Oracle7. 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.

For information about Oracle SQLJ support for user-defined types—SQL objects, object references, and collections—see Chapter 6, "Objects and Collections".

This chapter includes the following topics:

Supported Types for Host Expressions

This section summarizes the types that are supported by Oracle SQLJ, including information about backwards compatibility for the 8.0.5 and 7.3.4 Oracle JDBC drivers.

Supported Types for Oracle8*i*

Table 5-1 lists the Java types that you can use in host expressions when employing the 8.1.5 Oracle JDBC drivers. This table also documents the correlation between Java types, SQL types whose *typecodes* are defined in the `oracle.jdbc.driver.OracleTypes` class, and datatypes in the Oracle database. (The `OracleTypes` class simply defines a typecode, which is an integer constant, for each Oracle datatype. For standard SQL types, the `OracleTypes` entry is the same as the entry in the standard `java.sql.Types` type definitions class.)

SQL data that is output to a Java variable is converted to the corresponding Java type. A Java variable that is input to SQL is converted to the corresponding Oracle datatype.

Where objects, object references, and arrays are referred to as "JPub-generated", this refers to the Oracle JPublisher utility that can be used in defining Java classes to correspond to Oracle8*i* objects, object references, and arrays. The JPublisher utility is discussed in "JPublisher and the Creation of Custom Java Classes" on page 6-18, and documented in further detail in the *Oracle8i JPublisher User's Guide.*

Table 5–1 Type Mappings for Supported Host Expression Types

Java Type	OracleTypes Definition	Oracle Datatype
STANDARD JDBC TYPES		
boolean	BIT	NUMBER
byte	TINYINT	NUMBER
short	SMALLINT	NUMBER
int	INTEGER	NUMBER
long	BIGINT	NUMBER
float	REAL	NUMBER
double	FLOAT, DOUBLE	NUMBER

Table 5–1 Type Mappings for Supported Host Expression Types (Cont.)

Java Type	OracleTypes Definition	Oracle Datatype
java.lang.String	CHAR	CHAR
java.lang.String	VARCHAR	VARCHAR2
java.lang.String	LONGVARCHAR	LONG
byte[]	BINARY	RAW
byte[]	VARBINARY	RAW
byte[]	LONGVARBINARY	LONGRAW
java.sql.Date	DATE	DATE
java.sql.Time	TIME	DATE
java.sql.Timestamp	TIMESTAMP	DATE
java.math.BigDecimal	NUMERIC	NUMBER
java.math.BigDecimal	DECIMAL	NUMBER
JAVA WRAPPER CLASSES		
java.lang.Boolean	BIT	NUMBER
java.lang.Byte	TINYINT	NUMBER
java.lang.Short	SMALLINT	NUMBER
java.lang.Integer	INTEGER	NUMBER
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.AsciiStream	LONGVARCHAR	LONG
sqlj.runtime.UnicodeStream	LONGVARCHAR	LONG
ORACLE EXTENSIONS		
oracle.sql.NUMBER	NUMBER	NUMBER
oracle.sql.CHAR	CHAR	CHAR
oracle.sql.RAW	RAW	RAW

Table 5–1 Type Mappings for Supported Host Expression Types (Cont.)

Java Type	OracleTypes Definition	Oracle Datatype
oracle.sql.DATE	DATE	DATE
oracle.sql.ROWID	ROWID	ROWID
oracle.sql.BLOB	BLOB	BLOB
oracle.sql.CLOB	CLOB	CLOB
oracle.sql.BFILE	BFILE	BFILE
oracle.sql.STRUCT	STRUCT	STRUCT
oracle.sql.REF	REF	REF
oracle.sql.ARRAY	ARRAY	ARRAY
JPub-generated objects	STRUCT	STRUCT
JPub-generated object references	REF	REF
JPub-generated arrays	ARRAY	ARRAY
client-customized types (customization of any `oracle.sql` types, including objects, references, and collections)	depends on what is being customized	depends on what is being customized
QUERY RESULT OBJECTS		
java.sql.ResultSet	CURSOR	CURSOR
SQLJ iterator objects	CURSOR	CURSOR

The following points relate to type support for basic SQLJ features:

- Remember that all numeric types in an Oracle database are stored as `NUMBER`. While you can specify additional precision when you declare a `NUMBER` during table creation (you can declare the total number of places and the number of places to the right of the decimal point), 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` would preserve full information.)
- The Java wrapper classes (such as `Integer` and `Float`) are useful in cases where null values may be returned by the SQL statement. Primitive types (such as `int` and `float`) cannot contain null values. See "Null-Handling" on page 4-18 for more information.

- For information about SQLJ support for result set and iterator host variables, see "Using Iterators and Result Sets as Host Variables" on page 3-48.
- The SQLJ stream classes are required in using streams as host variables. For information, see "Support for Streams" on page 5-8.

And the following points relate to Oracle extensions, which are covered in "Oracle Type Extensions" on page 5-22 and in Chapter 6, "Objects and Collections":

- The `oracle.sql` classes are wrappers for SQL data for each of the Oracle datatypes. The `ARRAY`, `STRUCT`, `REF`, `BLOB`, and `CLOB` classes correspond to standard JDBC 2.0 interfaces. For background information about these classes and Oracle extensions, see the *Oracle8i JDBC Developer's Guide and Reference*.
- Custom objects and references refer to Oracle8*i* object support, which allows strongly typed custom objects in the database. The Oracle JPublisher utility creates Java classes to correspond to your custom objects and can also create Java classes corresponding to custom collections (variable arrays or nested tables).

Wrapping PL/SQL BOOLEAN, RECORD, and TABLE Types

Oracle JDBC drivers do not support calling arguments or return values of the PL/SQL types `TABLE` (now known as *indexed-by tables*), `RECORD`, or `BOOLEAN`.

As a workaround, you can create wrapper procedures that handle the data as types supported by JDBC. For example, to wrap a stored procedure that uses PL/SQL booleans, 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 tables, you can break the data into components or perhaps use Oracle collection types.

Here 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;
```

Backwards Compatibility for Oracle 8.0.5 and 7.3.4

Some of the Oracle type extensions supported by the Oracle8*i* JDBC drivers are either not supported or supported differently by the Oracle 8.0.5 and 7.3.4 JDBC drivers. Following are the key points:

- The 8.0.5 and 7.3.4 drivers have no `oracle.sql` package, meaning there are no wrapper types such as `oracle.sql.NUMBER` and `oracle.sql.CHAR` that you can use to wrap raw SQL data.
- The 8.0.5 and 7.3.4 drivers do not support Oracle object and collection types.
- The 8.0.5 and 7.3.4 drivers support the Oracle `ROWID` datatype with the `OracleRowid` class in the `oracle.jdbc.driver` package.
- The 8.0.5 drivers support the Oracle `BLOB`, `CLOB`, and `BFILE` datatypes with the `OracleBlob`, `OracleClob`, and `OracleBfile` classes in the `oracle.jdbc.driver` package. These classes do not include LOB and BFILE manipulation methods such as those discussed in "Support for BLOB, CLOB, and BFILE" on page 5-23. You must instead use the PL/SQL `DBMS_LOB` package, which is discussed in the same section.
- The 7.3.4 drivers do not support `BLOB`, `CLOB`, and `BFILE`.

Table 5-2 summarizes these differences.

Table 5–2 Type Support Differences for 8.0.5 and 7.3.4 JDBC Drivers

Java Type	OracleTypes Definition	Oracle Datatype
ORACLE EXTENSIONS		
oracle.sql.NUMBER	not supported	n/a
oracle.sql.CHAR	not supported	n/a
oracle.sql.RAW	not supported	n/a
oracle.sql.DATE	not supported	n/a

Table 5–2 Type Support Differences for 8.0.5 and 7.3.4 JDBC Drivers (Cont.)

Java Type	OracleTypes Definition	Oracle Datatype
oracle.jdbc.driver.OracleRowid	ROWID	ROWID
oracle.jdbc.driver.OracleBlob	BLOB in 8.0.5 not supported in 7.3.4	BLOB in 8.0.5 n/a in 7.3.4
oracle.jdbc.driver.OracleClob	CLOB in 8.0.5 not supported in 7.3.4	CLOB in 8.0.5 n/a in 7.3.4
oracle.jdbc.driver.OracleBfile	BFILE in 8.0.5 not supported in 7.3.4	BFILE in 8.0.5 n/a in 7.3.4
oracle.sql.STRUCT	not supported	n/a
oracle.sql.REF	not supported	n/a
oracle.sql.ARRAY	not supported	n/a
JPub-generated objects	not supported	n/a
JPub-generated object references	not supported	n/a
JPub-generated arrays	not supported	n/a
client-customized types (customization of any `oracle.sql` types, including objects, references, and collections)	not supported	n/a

Support for Streams

Standard SQLJ provides three specialized classes, listed below, for convenient handling of long data in streams. 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 handling and transferring large data items in manageable chunks.

- `BinaryStream`
- `AsciiStream`
- `UnicodeStream`

These classes are in the `sqlj.runtime` package.

This section discusses general use of these classes, Oracle SQLJ extended functionality, and stream class methods.

General Use of SQLJ Streams

With respect to an Oracle8*i* database, Table 5-1 in "Supported Types for Host Expressions" on page 5-2 lists the datatypes you would typically process using these stream classes. To summarize: `AsciiStream` and `UnicodeStream` are typically used for datatype `LONG` (`java.sql.Types.LONGVARCHAR`) but might also be used for datatype `VARCHAR2` (`Types.VARCHAR`); `BinaryStream` is typically used for datatype `LONG RAW` (`Types.LONGVARBINARY`) but might also be used for datatype `RAW` (`Types.BINARY` or `Types.VARBINARY`).

Of course, any use of streams is at your discretion. As Table 5-1 documents, `LONG` and `VARCHAR2` data can also be manifested in Java strings, while `RAW` and `LONGRAW` data can also be manifested in Java byte arrays. Furthermore, if your database supports large object types such as `BLOB` (binary large object) and `CLOB` (character large object), you may find these to be preferable to using types such as `LONG` and `LONG RAW` (although streams may still be used in extracting data from large objects). Oracle8*i* supports large object types—see "Support for BLOB, CLOB, and BFILE" on page 5-23.

All three SQLJ stream classes are subclasses of the standard Java input stream class, `java.io.InputStream`, and act as wrappers to provide the functionality required by SQLJ. This functionality is to communicate to SQLJ the type and length of data in the underlying stream so that it can be handled and formatted properly.

You can use the SQLJ stream types for host variables to send data to the database or iterator columns to receive data from the database.

Note: In using any method that takes an `InputStream` object as input, you can use an object of any of the SQLJ stream classes instead.

Using SQLJ Streams to Send Data to the Database

Standard SQLJ allows 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. This will be further discussed below.

You can use a SQLJ stream to send data to the database as follows:

1. Determine the length of your data.
2. Create a standard Java input stream—an instance of`java.io.InputStream` or some subclass—as you normally would.
3. Create an instance of the appropriate SQLJ stream class (depending on the type of data), passing the input stream and length (as an `int`) 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 (this is not required but is recommended).

This section now goes into more detail regarding two typical examples of sending a SQLJ stream to the database:

- Using an operating system file to update a `LONG` or `LONG RAW` column. (This can be either a binary file to update a `LONG RAW` column or an ASCII or Unicode file to update a `LONG` column.)
- Using a byte array to update a `LONG RAW` column.

Updating LONG or LONG RAW from a File

In updating a database column (presumably a `LONG` or `LONG RAW` 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.

Here are the steps in 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 make sure 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. Create a `java.io.FileInputStream` object from your `File` object. You can pass the `File` object to the `FileInputStream` constructor.
4. Create an appropriate SQLJ stream object. This would be a `BinaryStream` object for a binary file, an `AsciiStream` object for an ASCII file, or a `UnicodeStream` object for a Unicode file. Pass the `FileInputStream` object and data length (as an `int`) to the SQLJ stream class constructor.

   The SQLJ stream constructor signatures are all identical, as follows:

   ```
   BinaryStream (InputStream in, int length)
   AsciiStream (InputStream in, int length)
   UnicodeStream (InputStream in, int length)
   ```

   An instance of `java.io.InputStream` or of any subclass, such as `FileInputStream`, can be input to these constructors.
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 you want to insert the file contents into a `LONG` column called `asciidata` in a database table named `filetable`.

Imports:

```
import java.io.*;
import sqlj.runtime.*;
```

Executable code:

```
File myfile = new File ("/private/mydir/myfile.html");
int length = (int)myfile.length();     // Must cast long output to int.
FileInputStream fileinstream = new FileInputStream(myfile);
AsciiStream asciistream = new AsciiStream(fileinstream, length);
#sql { INSERT INTO filetable (asciidata) VALUES (:asciistream) };
asciistream.close();
...
```

Updating LONG RAW from a Byte Array

You must determine the length of the data before updating the database from a byte array. (Presumably you would be updating a `LONG RAW` column.) This is more trivial for arrays than for files, though, because all Java arrays have functionality to return the length.

Here 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 `ByteArrayInputStream`.
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 called `bindata` in a database table named `bintable`.

Imports:

```
import java.io.*;
import sqlj.runtime.*;
```

Executable code:

```
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—you can also update the database directly from a byte array.

Retrieving Data into Streams—Precautions

You can also use the SQLJ stream classes to retrieve data from the database, 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 an Oracle8*i* database and 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 as 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 way streams must be processed 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.

To summarize the precautions you must take:

- When using a positional iterator, you can only have 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 similar 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: Oracle8*i* and the Oracle JDBC drivers do not support use of streams in `SELECT INTO` statements.

Using SQLJ Streams to Retrieve Data from the Database

To retrieve data from a database column as a stream, standard SQLJ allows 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. This discussion takes into account the precautions documented in "Retrieving Data into Streams—Precautions" on page 5-12.

These are general steps. For more information, see "Processing SQLJ Streams" on page 5-15 and "Examples of Retrieving and Processing Stream Data" on page 5-16.

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. Query the database to populate the iterator you declared in step 2.
5. Process the iterator as usual (see "Using Positional Iterators" on page 3-45). 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.
6. In the iterator processing loop, after each iterator row is accessed, immediately read and process the local input stream, storing or outputting the stream data as desired.
7. Close the local input stream each time through the iterator processing loop (this is not required but is recommended).
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 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 do not have to be SQLJ stream types; they can be standard `java.io.InputStream` if desired. (They do not have to be SQLJ stream types because the issue of correctly formatting the data from the database was already taken care of as a result of the iterator columns being of appropriate SQLJ stream types.)

4. Query the database to populate the iterator you declared in step 2.
5. Process the iterator as usual (see "Using Named Iterators" on page 3-40). 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.
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 outputting the stream data as desired.
7. Close the local input stream each time through the iterator processing loop (this is not required but is recommended).
8. Close the iterator.

Note: When you populate a SQLJ stream object with data from an iterator or the database, the length attribute of the stream will not be meaningful. This attribute is only meaningful when you set it explicitly, either using the `setLength()` method that each SQLJ stream class provides, or specifying the length to the constructor (as discussed in "Using SQLJ Streams to Send Data to the Database" on page 5-9).

Processing SQLJ Streams

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` type. In either case, standard input stream methods are supported.

If the local stream variable is a SQLJ stream type—`BinaryStream`, `AsciiStream`, or `UnicodeStream`—you have the option of either reading data directly from the SQLJ stream object, or retrieving the underlying `java.io.InputStream` object and reading data from that. This is just a matter of preference—the former approach is simpler; the latter approach involves more direct and efficient data access.

The following important methods of the `InputStream` class—the `skip()` method, `close()` method, and three forms of the `read()` method—are supported by the SQLJ stream classes 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, the value -1 is returned. This method blocks until one of the following: 1) input data is available; 2) the end of the stream is detected; or 3) 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 until input is available.
- `int read (byte b[], int off, int len)`—Reads up to `len` (length) 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. In some circumstances, however, 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.

In addition, SQLJ stream classes support the following important method:

- `InputStream getInputStream()`—Returns the underlying input stream being wrapped, as a `java.io.InputStream` object.

Examples of Retrieving and Processing Stream Data

This section provides examples of various scenarios of retrieving stream data from the database, as follows:

- using a `SELECT` statement to select data from a `LONG` column and populate a SQLJ `AsciiStream` column in a named iterator
- using a `SELECT` statement to select data from a `LONG RAW` column and populate a SQLJ `BinaryStream` column in a positional iterator

Example: Selecting LONG Data into AsciiStream Column of Named Iterator This example selects data from a `LONG` database column, populating a SQLJ `AsciiStream` column in a named iterator.

Presume 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 ASCII.

Imports and declarations:

```
import sqlj.runtime.*;
import java.io.*;

#sql iterator MyNamedIter (String filename, AsciiStream filecontents);
```

Executable code:

```
MyNamedIter namediter = null;
String fname;
AsciiStream ascstream;
#sql namediter = { SELECT filename, filecontents FROM filetable };
while (namediter.next()) {
   fname = namediter.filename();
   ascstream = namediter.filecontents();
   System.out.println("Contents for file " + fname + ":");
   printStream(ascstream);
   ascstream.close();
}
namediter.close();
...
public void printStream(InputStream in) throws IOException
{
   int asciichar;
   while ((asciichar = in.read()) != -1) {
      System.out.print((char)asciichar);
   }
}
```

Remember that you can pass a SQLJ stream to any method that takes a standard `java.io.InputStream` as an input parameter.

Example: Selecting LONG RAW Data into BinaryStream Column of Positional Iterator This example selects data from a `LONG RAW` database column, populating a SQLJ `BinaryStream` column in a positional iterator.

As explained in "Retrieving Data into Streams—Precautions" on page 5-12, there can be only one stream column in a positional iterator and it must be the last column.

Presume 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.

Imports and declarations:

```
import sqlj.runtime.*;

#sql iterator MyPosIter (int, BinaryStream);
```

Executable code:

```
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 use of the `BinaryStream`, `AsciiStream`, and `UnicodeStream` classes in the package `sqlj.runtime` for retrieval of stream data into iterator columns.

In addition, the Oracle SQLJ implementation allows the following uses of SQLJ stream types if you are using an Oracle database, Oracle JDBC driver, and the Oracle customizer:

- 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 types `AsciiStream`, `BinaryStream` and `UnicodeStream` as the assignment type for a stored procedure or stored function `OUT` or `INOUT` parameter.

Presume the following table definition:

```
CREATE TABLE streamexample (name VARCHAR2 (256), data LONG);
INSERT INTO streamexample (data, name)
   VALUES
   ('0000000000111111111112222222222333333333344444444445555555555',
   '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.

Imports:

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

Executable code:

```
AsciiStream 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 types `AsciiStream`, `BinaryStream` and `UnicodeStream` as the assignment type for a stored function return result.

Presume the same `streamexample` table definition as in the preceding stored procedure example.

Also presume 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.

Imports:

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

Executable code:

```
AsciiStream 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.

Stream Class Methods

The SQLJ stream classes in the `sqlj.runtime` package—`BinaryStream`, `AsciiStream`, and `UnicodeStream`—are all subclasses of the `sqlj.runtime.StreamWrapper` class.

The `StreamWrapper` class provides the following methods that are inherited by the SQLJ stream classes:

- `InputStream getInputStream()`—As discussed in "Processing SQLJ Streams" on page 5-15, you can optionally use this method to get the underlying `java.io.InputStream` object of any SQLJ stream object. This is not required, however, as 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.

  Bear in mind that 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 only meaningful 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.

Note: The `sqlj.runtime.StreamWrapper` class is a subclass of `java.io.FilterInputStream`, which is a subclass of `java.io.InputStream`.

Oracle Type Extensions

Oracle SQLJ supports the following Oracle-specific datatypes:

- LOBs (`BLOB`, `CLOB`, `BFILE`)
- `ROWID`
- `REF CURSOR` types
- other Oracle8*i* datatypes (such as `NUMBER` and `RAW`)

These datatypes are supported by classes in the `oracle.sql` package, discussed below, which requires you to use one of the Oracle JDBC drivers and to customize your profiles appropriately (the default Oracle customizer, `oracle.sqlj.runtime.util.OraCustomizer`, is recommended).

Additionally, Oracle SQLJ offers extended support for the following standard JDBC type:

- `BigDecimal`

User-defined database objects (both weakly and strongly typed), object references, and collections (variable arrays and nested tables) are also supported. These are discussed in Chapter 6, "Objects and Collections".

Note: You must use one of the Oracle JDBC drivers to use Oracle type extensions.

Package oracle.sql

SQLJ users as well as JDBC users should be aware of the `oracle.sql` package, which includes classes to support all of the Oracle8*i* datatypes (for example, `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.*` datatype class is a subclass of the `oracle.sql.Datum` class.

To use these classes, you must use an Oracle JDBC driver and customize your SQLJ profiles appropriately. The default Oracle customizer, `oracle.sqlj.runtime.util.OraCustomizer`, is recommended. This is used automatically when you run the `sqlj` script unless you specify otherwise.

You also must import the package, as follows (unless you use the fully qualified class names in your code):

```
import oracle.sql.*;
```

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.

For information about translator options relating to semantics-checking, see "Connection Options" on page 8-31 and "Semantics-Checking Options" on page 8-55.

For more information about the `oracle.sql` classes, see the *Oracle8i JDBC Developer's Guide and Reference*.

Support for BLOB, CLOB, and BFILE

Oracle JDBC and SQLJ support three LOB (large object) datatypes: `BLOB` (binary LOB), `CLOB` (single-character LOB), and `BFILE` (read-only binary files stored outside the database). These datatypes are supported by the following classes:

- `oracle.sql.BLOB`
- `oracle.sql.CLOB`
- `oracle.sql.BFILE`

See the *Oracle8i JDBC Developer's Guide and Reference* for more information about LOBs and files and use of supported stream APIs.

The `oracle.sql.BLOB`, `oracle.sql.CLOB`, and `oracle.sql.BFILE` classes can be used in Oracle-specific SQLJ applications in the following ways:

- as `IN`, `OUT`, or `INOUT` host variables in executable SQLJ statements (including use in INTO-lists)
- as return values from stored function calls
- as column types in iterator declarations (both named and positional)

You can manipulate LOBs by using methods defined in the `BLOB` and `CLOB` classes (recommended) or by using the procedures and functions defined in the PL/SQL package DBMS_LOB. 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 (recommended) or by using the file-handling routines of DBMS_LOB.

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, CLOBs contain character data; therefore, Java strings are used to hold chunks of data. BLOBs contain binary data; therefore, Java byte arrays are used to hold chunks of data.

Note: DBMS_LOB is a database package, requiring a round trip to the server.

Methods in the BLOB, CLOB, and BFILE classes may also result in a round trip to the server.

The following examples contrast use of the oracle.sql methods with use of the DBMS_LOB package. For each example using oracle.sql methods, the example that follows it is functionally identical but uses DBMS_LOB instead.

Example: 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: 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 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.

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: 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 type `String`.

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: 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 type `byte[]`.

LOB Stored Function Results

Host variables of type `BLOB`, `CLOB`, and `BFILE` can be assigned to the result of a stored function call.

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 function defined above.

```
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 Host Variables and SELECT INTO Targets

Host variables of type `BLOB`, `CLOB`, and `BFILE` can appear in the INTO-list of a `SELECT INTO` executable statement.

Assume the following table definition:

```
CREATE TABLE basic_lob_table(x varchar2(30), b blob, c clob);
INSERT INTO basic_lob_table
   VALUES('one', '010101010101010101010101010101', 'onetwothreefour');
INSERT INTO basic_lob_table
   VALUES('two', '020202020202020202020202020202', 'twothreefourfivesix');
```

The following example uses a BLOB and a CLOB as host variables that receive data from the table defined above, 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 the `basic_lob_table`. It then inserts a third row into the table using the BLOB and CLOB selected in the previous operation.

LOB Iterator Declarations

The types `BLOB`, `CLOB`, and `BFILE` 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.

Here are sample declarations that will be repeated and used below.

```
#sql iterator NamedLOBIter(CLOB c);
#sql iterator PositionedLOBIter(BLOB);
```

LOB Host Variables and Named Iterator Results

The following example employs the table `basic_lob_table` and the method `readFromLongest()` defined in previous examples, and uses a CLOB in a named iterator.

Declaration:

```
#sql iterator NamedLOBIter(CLOB c);
```

Executable code:

```
...
NamedLOBIter iter;
#sql iter = { SELECT c FROM basic_lob_table };
iter.next();
CLOB c1 = iter.c();
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 the `basic_lob_table`, then prints the larger of the two using the `readFromLongest()` method.

LOB Host Variables and Positional Iterator FETCH INTO Targets

Host variables of type `BLOB`, `CLOB`, and `BFILE` 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 identical type.

The following example employs table `basic_lob_table` and method `writeToBlob()` defined in previous examples.

Declaration:

```
#sql iterator PositionedLOBIter(BLOB);
```

Executable code:

```
...
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 type `ROWID` stores the unique address for each row in a database table. The class `oracle.sql.ROWID` wraps ROWID information and is used to bind and define variables of type `ROWID`.

Variables of type `oracle.sql.ROWID` can be employed in SQLJ applications connecting to an Oracle database in the following ways:

- as `IN`, `OUT` or `INOUT` host variables in SQLJ executable statements (including use in INTO-lists)
- as a return value from a stored function call
- as column types in iterator declarations (both named and positional)

Note: Oracle does not currently support positioned `UPDATE` or positioned `DELETE` by way of a `WHERE CURRENT OF` clause, as specified by the SQLJ specification. Instead, Oracle recommends the use of ROWIDs to simulate this functionality.

ROWID in Iterator Declarations

You can use the type `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 employ `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.

Declaration:

```
#sql iterator NamedRowidIter (String ename, ROWID rowid);
```

Executable code:

```
...
NamedRowidIter iter;
ROWID rowid;
#sql iter = { SELECT ename, rowid FROM emp };
while (iter.next())
{
   if (iter.ename().equals("TURNER"))
   {
       rowid = iter.rowid();
       #sql { UPDATE emp SET sal = sal + 500 WHERE rowid = :rowid };
   }
}
iter.close();
...
```

The preceding example increases the salary of the employee named Turner by $500 according to the ROWID. Note that this is the recommended way to encode `WHERE CURRENT OF` semantics.

ROWID Stored Function Results

Presume the following function exists in the database.

```
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('TURNER')) };
#sql { UPDATE emp SET sal = sal + 500 WHERE rowid = :rowid };
```

This example increases the salary of the employee named Turner by $500 according to the ROWID.

ROWID SELECT INTO Targets

Host variables of type `ROWID` can appear in the INTO-list of a `SELECT INTO` statement.

```
ROWID rowid;
#sql { SELECT rowid INTO :rowid FROM emp WHERE ename='TURNER' };
#sql { UPDATE emp SET sal = sal + 500 WHERE rowid = :rowid };
```

This example increases the salary of the employee named Turner by $500 according to the ROWID.

ROWID Host Variables and Positional Iterator FETCH INTO Targets

Host variables of type `ROWID` can appear in the INTO-list of a `FETCH INTO` statement if the corresponding column attribute in the iterator is of identical type.

Declaration:

```
#sql iterator PositionedRowidIter (String, ROWID);
```

Executable code:

```
...
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("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.

Support for Oracle REF CURSOR Types

Oracle PL/SQL and Oracle SQLJ support the use of cursor variables that represent database cursors.

About 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, 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 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.

For more information about cursor variables, see the *PL/SQL User's Guide and Reference*.

REF CURSOR Types in SQLJ

In Oracle SQLJ, a `REF CURSOR` type can be mapped to iterator columns or host variables of any iterator class type or of type `java.sql.ResultSet`, but host variables can be `OUT` only. Support for `REF CURSOR` types can be summarized as follows:

- as result expressions for the results from stored functions
- 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 Oracle SQL `CURSOR` operator for a nested `SELECT` within an outer `SELECT` statement. This is how you can write a `REF CURSOR` to an iterator column or `ResultSet` column in an iterator, or write a `REF CURSOR` to an iterator host variable or `ResultSet` host variable in an INTO-list.

"Using Iterators and Result Sets as Host Variables" on page 3-48 has examples showing the use of implicit `REF CURSOR` variables, including an example of the `CURSOR` operator.

Notes:

- The Oracle typecode for REF CURSOR types is OracleTypes.CURSOR.
- There is no oracle.sql class for REF CURSOR. Use either an iterator class or java.sql.ResultSet.

REF CURSOR Example

The following sample method shows a REF CURSOR type being retrieved from an anonymous block. This is part of a full sample application that is in "REF CURSOR—RefCursDemo.sqlj" on page 12-52.

```
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 Oracle8*i* Datatypes

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 oracle.sql.NUMBER, oracle.sql.CHAR, or oracle.sql.RAW.

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 need to convert the data to standard Java types, however, to use it with standard Java programs or perhaps 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 Table 5-1 in "Supported Types for Host Expressions" on page 5-2 for more information about JDBC default mappings.)

In Oracle SQLJ, however, you can map to non-default types as long as the datatype is convertible from numeric and you are using Oracle8*i*, an Oracle JDBC driver, and the Oracle customizer. The datatypes CHAR, VARCHAR2, LONG, and NUMBER are convertible. For example, you can retrieve data from a CHAR column into a `BigDecimal` variable. To avoid errors, however, you must be careful that the character data consists only of numbers.

Note: To use `BigDecimal`, import `java.math` or specify `BigDecimal` by its fully qualified name.

---

6 Objects and Collections

This chapter discusses how Oracle SQLJ supports user-defined SQL types, namely objects (and related object references) and collections (variable arrays and nested tables). This includes discussion of the Oracle JPublisher utility, which you can use to generate Java classes corresponding to user-defined SQL types.

The following topics are discussed:

Introduction

Oracle8*i* and Oracle SQLJ support user-defined SQL *object* types (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.

Oracle SQLJ 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 the interface `oracle.sql.CustomDatum`. This paradigm is supported by the Oracle JPublisher utility, which you can use to automatically generate such custom Java classes. Weakly typed representations use the class `oracle.sql.STRUCT` (for objects), `oracle.sql.REF` (for references), or `oracle.sql.ARRAY` (for collections).

To use Oracle-specific object, reference, and collection types, you must use one of the Oracle JDBC drivers and you must customize your profiles appropriately (the default Oracle customizer, `oracle.sqlj.runtime.util.OraCustomizer`, is recommended). This is performed automatically when you run SQLJ, unless you specify otherwise.

You also must import the `oracle.sql` package, as follows (unless you use the fully qualified class names in your code):

```
import oracle.sql.*;
```

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.

For information about translator options relating to semantics-checking, see "Connection Options" on page 8-31 and "Semantics-Checking Options" on page 8-55.

Notes:

- This chapter primarily discusses the use of custom Java classes with user-defined types; however, they can be used for other Oracle SQL types as well. A custom Java class can be employed to perform any kind of desired processing or conversion in the course of transferring data between the database and Java.
- Custom Java classes for objects, references, and collections are referred to as *custom object classes*, *custom reference classes*, and *custom collection classes*, respectively.
- User-defined SQL object types and user-defined SQL collection types are referred to as *user-defined types* (UDTs).

About Oracle Objects and Collections

This section provides some background conceptual information about Oracle8*i* objects and collections.

For additional conceptual and reference information about Oracle objects, references, and collections, refer to the *Oracle8i SQL Reference* and the *Oracle8i Application Developer's Guide—Fundamentals*.

For information about how to declare objects and collections, see "User-Defined Types in the Database" on page 6-13.

Oracle Object Fundamentals

Oracle objects (SQL objects) are composite data structures that group related data items, such as various 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` (type `CHAR`), `address` (type `CHAR`), `phonenumber` (type `CHAR`), and `employeenumber` (type `NUMBER`).

Oracle objects can also have methods—stored procedures that are associated with the object type. These methods can be either static methods or instance methods that can be implemented either in PL/SQL or in Java. Their signatures can include any number of input, output, or input-output parameters. All of this depends on how they are initially defined.

Oracle Collection Fundamentals

There are two categories of Oracle collections (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, while 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, while a variable of any nested table type can be referred to as a nested table.

A VARRAY, like any array, is an ordered set of data elements with each element having an index and all elements being of the same datatype. The *size* of a VARRAY refers to the maximum number of elements. Oracle VARRAYs are of variable size (hence the name), 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, but not in SQLJ. A nested table, like any table, is not created with any particular number of rows. This is determined dynamically.

Notes:

- 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 does not, however, support any nesting of collection types. The elements of a VARRAY or rows of a nested table cannot be of another VARRAY or nested table type, nor can these elements be of a user-defined object type that has VARRAY or nested table attributes.
- In Oracle SQLJ, collection types can be retrieved only as a whole.

Object and Collection Datatypes

User-specified object and collection definitions in Oracle8*i* function as SQL datatype definitions. These datatypes can then be used like any other datatype in defining table columns, SQL object attributes, and stored procedure or function output parameters or return parameters. Also, once you have defined an object type, the related object reference type can be used like any other SQL reference type.

Once you have defined `EMPLOYEE` as an Oracle object, as described in "Oracle Object Fundamentals" on page 6-4, it becomes an Oracle datatype and 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 datatypes, and you can have table columns of either type. Each row of a `MYVARR` column consists of an array of up to ten numbers; each row of an `NTBL` column consists of ten characters.

About Custom Java Classes and the CustomDatum Interface

The key functionality of custom Java classes is to provide a way to convert data between the database and your Java application and making the data accessible, particularly in supporting objects and collections or if you want to do custom data conversions.

It is advisable to provide custom Java classes for all user-defined types (objects and collections) 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 classes `oracle.sql.STRUCT`, `REF`, and `ARRAY`.

Custom Java classes are first-class types that you can use to read from and write to user-defined SQL types transparently.

CustomDatum and CustomDatumFactory Specifications

Oracle provides the interface `oracle.sql.CustomDatum` and the related interface `oracle.sql.CustomDatumFactory` as vehicles to use in mapping Oracle object types, reference types, and collection types to custom Java classes and in converting data between the database and your application. A custom Java class must implement `CustomDatum` to be used in SQLJ iterators and host expressions, and either the custom Java class or some other class must implement `CustomDatumFactory` to create instances of the custom Java class.

Data is passed to or from the database 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 codified database format; the `oracle.sql.Datum` object is just a wrapper. (For information about classes in the `oracle.sql` package that support Oracle type extensions, see the *Oracle8i JDBC Developer's Guide and Reference*.)

The `CustomDatum` interface specifies a `toDatum()` method for data conversion from Java format to database format. This method takes as input your `OracleConnection` object (which is required by the Oracle JDBC drivers) and converts data to the appropriate `oracle.sql.*` representation. The `OracleConnection` object is necessary so that the JDBC driver can perform appropriate type checking and type conversions at runtime. Here is the `CustomDatum` and `toDatum()` specification:

```
interface oracle.sql.CustomDatum
{
   oracle.sql.Datum toDatum(OracleConnection c);
}
```

The `CustomDatumFactory` interface specifies a `create()` method that constructs instances of your custom Java class, converting from database format to Java format. This method takes as input a `Datum` object containing data from the database and an integer indicating the SQL type of the underlying data, such as `OracleTypes.RAW`. It returns an object of your custom Java class, which implements the `CustomDatum` interface. This object receives its data from the `Datum` object that was input. Here is the `CustomDatumFactory` and `create()` specification:

```
interface oracle.sql.CustomDatumFactory
{
   oracle.sql.CustomDatum create(oracle.sql.Datum d, int sqlType);
}
```

To complete the relationship between the `CustomDatum` and `CustomDatumFactory` interfaces, there is a requirement for a static `getFactory()` method that you must implement in any custom Java class that implements the `CustomDatum` interface. This method returns an object that implements the `CustomDatumFactory` interface, and that therefore can be used to create instances of your custom Java class. This returned object may 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.

Custom Java classes generated by JPublisher automatically implement the `CustomDatum` and `CustomDatumFactory` interfaces and the `getFactory()` method.

Note: JPublisher implements `CustomDatum` and its `toDatum()` method and `CustomDatumFactory` and its `create()` method in a single custom Java class; however, the reason `toDatum()` and `create()` are in different interfaces is to allow the option of implementing them in separate classes. You can have one custom Java class that implements `CustomDatum`, its `toDatum()` method, and the `getFactory()` method, and have a separate factory class that implements `CustomDatumFactory` and its `create()` method. For purposes of discussion here, though, the presumption is that both interfaces are implemented in a single class.

For more information about the `CustomDatum` and `CustomDatumFactory` interfaces, the `oracle.sql` classes, and the `OracleTypes` class, see the *Oracle8i JDBC Developer's Guide and Reference*.

Custom Java Class Support for Object Methods

Methods of Oracle objects can be implemented as wrappers in custom Java classes. 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 that is 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 (as an attribute of your custom Java class, for example).

You can write each wrapper method as an instance method of the custom Java class, regardless of whether the server method that the wrapper method invokes is an instance method or a static method. Custom Java classes generated by JPublisher use this technique. JPublisher also provides a public no-argument constructor for each custom Java class it generates, so that an instance can be conveniently created for the purpose of calling a wrapper method that invokes a static server method. This is also a recommended technique if you are writing your own custom Java classes.

There are also issues regarding output and input-output parameters in methods of Oracle objects. In the database, if a stored procedure (Oracle object method) modifies the internal state of one of its arguments, the actual argument that was 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 is 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, 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—each output or input-output parameter is passed in a one-element array.

See the *Oracle8i JPublisher User's Guide* for more information.

Custom Java Class Requirements

All custom Java classes must satisfy certain requirements to be recognized by the SQLJ translator as valid host variable types. These requirements are primarily the same for any kind of custom Java class but vary slightly depending on whether the class is a custom object class, custom reference class, custom collection class, or some other kind of custom Java class.

These requirements are as follows:

- The class implements the interface `oracle.sql.CustomDatum`.
- The class implements a method `getFactory()` that returns an `oracle.sql.CustomDatumFactory`, as follows:

  ```
  public static oracle.sql.CustomDatumFactory getFactory();
  ```

- The class has a constant `_SQL_TYPECODE` initialized to the `oracle.jdbc.driver.OracleTypes` typecode of the `Datum` subclass that `toDatum()` returns.

  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 typecode may be appropriate. For example, for using a custom Java class to serialize and deserialize Java objects into or out of `RAW` fields in the database, a `_SQL_TYPECODE` of `OracleTypes.RAW` is used. See "Serializing Java Objects through Custom Java Classes" on page 6-56.

  (The `OracleTypes` class simply defines a typecode, which is an integer constant, for each Oracle datatype. For standard SQL types, the `OracleTypes` entry is the same as the entry in the standard `java.sql.Types` type definitions class.)

- For custom Java classes with `_SQL_TYPECODE` of `STRUCT`, `REF`, or `ARRAY` (in other words, for custom Java classes that represent objects, object references, or collections), the class has a constant indicating the relevant user-defined type name.

  Custom object classes and custom collection classes must have a constant (string) `_SQL_NAME` initialized to the SQL name you declared for the user-defined type, as follows:

  ```
  public static final String _SQL_NAME = UDT name;
  ```

Custom object class example (for a user-defined PERSON object):

```
public static final String _SQL_NAME = "PERSON";
```

Or (to specify the schema, if that is appropriate):

```
public static final String _SQL_NAME = "SCOTT.PERSON";
```

Custom collection class example (for a collection of PERSON objects, which you have declared as PERSON_ARRAY):

```
public static final String _SQL_NAME = "PERSON_ARRAY";
```

Custom reference classes must have a constant (string) _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;
```

Custom reference class example (for PERSON references):

```
public static final String _SQL_BASETYPE = "PERSON";
```

For other uses, a constant for the UDT name is not applicable.

Notes: The 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.

Compiling Custom Java Classes

You can include the .java files for your custom Java classes on the SQLJ command line together with your .sqlj file. For example, if ObjectDemo.sqlj uses Oracle object types ADDRESS and PERSON, and you have run JPublisher or otherwise produced custom Java classes for these objects, then you can run SQLJ as follows:

```
% sqlj ObjectDemo.sqlj Address.java AddressRef.java Person.java PersonRef.java
```

Otherwise, you can use your Java compiler to compile them directly. If you do this, it must be done prior to translating the .sqlj file.

(Running SQLJ is discussed in Chapter 8, "Translator Command Line and Options".)

Note: Because custom Java classes rely on Oracle-specific features, SQLJ will report numerous portability warnings unless you use the translator option setting `-warn=noportable` (the default). For information about this flag, see "Translator Warnings (-warn)" on page 8-42.

Reading and Writing Custom Data

This section describes how data from custom Java class instances is read from the database and written to the database.

How Custom Data is Read from the Database

In reading data from the database, the conversion of codified bytes of an Oracle object and its attributes (or collection and its elements) into an instance of the corresponding custom Java class takes place in three steps:

1. The JDBC driver reads the codified bytes from the database and creates an instance of the appropriate `oracle.sql` class (typically `STRUCT` or `ARRAY`) to contain the bytes.
2. The SQLJ runtime calls the static `getFactory()` method of your custom Java class to obtain a `CustomDatumFactory` object, which is typically a static instance of your custom Java class.
3. The JDBC driver calls the `create()` method of the `CustomDatumFactory` object obtained in step 2. This creates and populates an instance of your custom class, using the data from the `oracle.sql.*` instance created in step 1.

How Custom Data is Written to the Database

In writing an instance of a custom object class or custom collection class to the database, the `toDatum()` method is called, returning an instance of `oracle.sql.STRUCT` (or `oracle.sql.ARRAY`) that is then written to the database.

Additional Uses for Custom Java Classes

Most discussion of custom Java classes involves their use as one of the following:

- wrappers for SQL objects—custom object classes, for use with `oracle.sql.STRUCT` instances from the database
- wrappers for SQL references—custom reference classes, for use with `oracle.sql.REF` instances from the database
- wrappers for SQL collections—custom collection classes, for use with `oracle.sql.ARRAY` instances from the database

It may be useful to provide custom Java classes to wrap other `oracle.sql.*` types and perhaps implement customized conversions or functionality as well. The following are some examples:

- to perform encryption and decryption or validation of data
- to perform logging of values that have been read or are being written
- to parse character columns (such as character fields containing URL information) into smaller components
- to map character strings into numeric constants
- to map data into more desirable Java formats (such as mapping a DATE field to `java.util.Date` format)
- to customize data representation (for example, data in a table column is in feet but you want it represented in meters after it is selected)
- to serialize and deserialize Java objects—into or out of RAW fields, for example

This last use is further discussed in "Serializing Java Objects through Custom Java Classes" on page 6-56.

In "General Use of CustomDatum—BetterDate.java" on page 12-47, there is a sample class `BetterDate` that can be used instead of `java.sql.Date` to represent dates.

User-Defined Types in the Database

This section contains examples of creating and using user-defined object types and collection types in Oracle8*i*. A full SQL script for all the user-defined types employed in the object and collection sample applications is in "Definition of Object and Collection Types" on page 12-20.

For more information about any of the SQL commands used here, refer to the *Oracle8i SQL Reference*.

Creating Object Types

Oracle 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 datatypes.

The rest of this section provides an example of creating user-defined object types in Oracle8*i*.

The following items are created using the SQL script below:

- 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

```
/*** Using UDTs in SQLJ ***/
SET ECHO ON;
/**
/*** Clean up in preparation ***/
DROP TABLE EMPLOYEES
/
DROP TABLE PERSONS
/
DROP TYPE PERSON FORCE
```

```
/
DROP TYPE ADDRESS FORCE
/
/*** 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')
/
COMMIT
/
QUIT
```

Note: Use of a table alias, such as `p` above, is a recommended general practice in Oracle SQL, 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. See the *Oracle8i SQL Reference* for more information about table aliases.

Creating Collection Types

There are two categories of collections you can define: variable-length arrays (VARRAYs) and nested tables.

Oracle SQL commands to create VARRAY types are of the following form:

```
CREATE TYPE typename IS VARRAY(n) OF datatype;
```

Where *typename* is the desired name of your VARRAY type, *n* is the desired maximum number of elements in the array, and *datatype* is the datatype of the array elements. You must specify the maximum number of elements in the array. For example:

```
CREATE TYPE myvarr IS VARRAY(10) OF INTEGER;
```

Oracle SQL commands to create nested table types are of the following form:

```
CREATE TYPE typename AS TABLE OF datatype;
```

Where `typename` is the desired name of your nested table type and `datatype` is the datatype of the table elements (this can be a user-defined type as well as a standard datatype). A nested table is limited to one column, although that one column type can be a complex object with multiple attributes. The nested table, like any database table, can have any number of rows. For example:

```
CREATE TYPE person_array AS TABLE OF person;
```

This 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 Oracle8*i*.

The following items are created and populated using the SQL script below:

- 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)`
- an `employees` table, which includes a `PHONE_ARRAY` column

```
Rem This is a SQL*Plus script used to create schema to demonstrate collection
Rem manipulation in SQLJ

DROP TABLE projects
/
DROP TABLE employee
/
DROP TYPE MODULETBL_T
/
DROP TYPE MODULE_T
/
DROP TYPE PARTICIPANT_T
/
DROP TYPE PHONE_ARRAY
/
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 TABLE  employees
( empnumber            INTEGER PRIMARY KEY,
  person_data     REF  person,
  manager         REF  person,
  office_addr          address,
  salary               NUMBER,
  phone_nums           phone_array
)
/
commit;
exit;
```

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 Oracle JPublisher to automatically generate custom Java classes and using those classes directly without modification
- using JPublisher to automatically generate custom Java classes and subclassing them to create custom Java classes with added functionality
- manually coding custom Java classes without using JPublisher, provided that 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, using JPublisher is advisable. If you need special functionality, you can subclass a class that JPublisher generates or use it as a field in a class that you create.

This manual provides only a minimal level of information and detail regarding the JPublisher utility. See the *Oracle8i JPublisher User's Guide* for more information.

What JPublisher Produces

When you run JPublisher for a user-defined object type, it automatically creates the following:

- a custom object class to act as a type definition to correspond to your Oracle object type

  This class includes getter and setter methods for each attribute. The method names are of the form `getFoo()` and `setFoo()` for attribute `foo`.

  Also, you can optionally instruct JPublisher to generate wrapper methods in your class that invoke the associated Oracle object methods executing in the server. This is specified by the `-methods=true` option, described later in this section.

Note: For release 8.1.5, generation of wrapper methods by JPublisher is a Beta feature.

- 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.

- 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, it 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 `CustomDatum` interface, the `CustomDatumFactory` interface, and the `getFactory()` method.

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 method wrappers. These key points can be summarized as follows:

- Specify user-defined types to map to Java; optionally specify custom object class names and custom collection class names (this involves the JPublisher `-sql`, `-user`, and `-case` options).
- Optionally specify attribute type mappings (this involves the JPublisher `-mapping` option).

- Optionally instruct JPublisher to create wrapper methods, in particular for Oracle object methods (this involves the JPublisher `-methods` option).

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 the same as the user-defined type names as you type them. Because SQL names in the database are case-insensitive, you can capitalize them to ensure that your class names are capitalized per 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 lower-level classes, such as for `home_address` objects that are attributes of `employee` objects, are determined by the JPublisher `-case` option. If you do not set the `-case` option, 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 source file `HomeAddress.java`.

Notes:

- Remember that Java class names are case-sensitive, while Oracle object and collection names (and SQL names in general) are not.
- For backwards 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 (you can specify multiple actions in a single option setting). Use the `-user` option to specify the database schema.

```
-sql=udt1<:mapclass1><,udt2<:mapclass2>>,...,<udtN<:mapclassN>>
```

Following is an example:

```
% jpub -sql=Myobj,mycoll:MyCollClass -user=scott/tiger
```

(There can be no space before or after the comma.)

For the Oracle object `myobj`, this command will name it as you typed it, creating source `Myobj.java` to define class `Myobj`. For the Oracle collection `mycoll`, this command will create source `MyCollClass.java` to define class `MyCollClass`.

You can optionally specify schema names as well—for example, 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. For example, if JPublisher produces Java source `Myobj.java` to define custom object class `Myobj`, then it will also produce Java source `MyobjRef.java` to define custom reference class `MyobjRef`.

Note: When specifying the schema, such as `scott` in the above 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 in the Database" on page 6-13, you can run JPublisher as follows:

```
%jpub -user=scott/tiger -sql=Address,Person,Phone_array,Participant_t,
Module_t,Moduletbl_t
```

Or, to explicitly specify the custom object class and custom collection class names:

```
%jpub -user=scott/tiger -sql=Address,Person,phone_array:PhoneArray,
participant_t:ParticipantT,module_t:ModuleT,moduletbl_t:ModuletblT
```

The second example will produce Java source files `Address.java`, `AddressRef.java`, `Person.java`, `PersonRef.java`, `PhoneArray.java`, `ParticipantT.java`, `ParticipantTRef.java`, `ModuleT.java`, `ModuleTRef.java`, and `ModuletblT.java`. Examples of some of these source files are provided in "JPublisher Custom Java Class Examples" on page 6-29.

So that it knows how to populate the custom Java classes, JPublisher connects to the specified schema (here `scott/tiger`) to determine attributes of your specified object types or elements of your specified collection types.

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, 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; illegal characters are discarded.
- `-case=upper`—Lowercase letters are converted to uppercase. Underscores and dollar signs are retained; illegal characters are discarded.
- `-case=lower`—Uppercase letters are converted to lowercase. Underscores and dollar signs are retained; illegal characters are discarded.

Notes: If you run JPublisher without specifying the user-defined types to map to Java, it will process all user-defined types in your schema. Generated class names, for both your top-level custom classes and any lower-level classes for object attributes or collection elements, will be based on the setting of the `-case` option.

Specify Attribute and Element Type Mappings

Oracle datatypes of attributes or elements in user-defined SQL types can be mapped to Java types in one of the following three ways, as specified by the JPublisher `-mapping` option. This determines what kinds of conversions are performed in transferring data between the database and your application.

- JDBC mapping (default)—Standard Java types such as `int`, `float`, or `java.lang.BigDecimal` are used. This does not protect against null data—assigning null data to primitive types such as `int` and `float` will cause an exception.

  See Table 5-1 in "Supported Types for Host Expressions" on page 5-2 to see how Oracle datatypes map to Java types.

- Object JDBC mapping—This is like JDBC mapping except that wrapper classes of the `java.lang` package are used instead of Java primitive types. For example, `java.lang.Integer` and `java.lang.Float` are used instead of `int` and `float`. This allows null values.
- Oracle mapping—The `oracle.sql` class that corresponds to the Oracle datatype is used. For example, a `NUMBER` attribute maps to `oracle.sql.NUMBER`. Data items in `oracle.sql` objects are maintained in their raw formats.

Specify attribute and element type mapping with one of the following command-line settings of the JPublisher `-mapping` option:

```
-mapping=jdbc
-mapping=objectjdbc
-mapping=oracle
```

Generate Method Wrappers

In creating custom object classes to map Oracle objects to Java, you can optionally include Java wrappers for Oracle object methods. You can use the JPublisher `-methods=true` option setting to accomplish this.

You can specify `-methods=false` to ensure that wrappers are not generated. For release 8.1.5, this is the default.

Note: For release 8.1.5, generation of wrapper methods by JPublisher is a Beta feature.

Wrapper methods generated by JPublisher are always instance methods, even when the original object methods are static. See "Custom Java Class Support for Object Methods" on page 6-8 for more information.

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 named in the same fashion as custom Java class names (as described in "Specify User-Defined Types to Map to Java" on page 6-20), 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 method names, but this requires use of a JPublisher input file and is discussed in "Creating Custom Java Classes and Specifying Member Names" on page 6-27.

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 `_`*`nn`* to function names, where *`nn`* is a number. This is to ensure that no two methods in the generated custom Java class have the same name and signature. Consider, for example, SQL functions with the following signatures:

```
F(INTEGER, INTEGER)
F(FLOAT, FLOAT)
```

Without precaution, these would both result in the following name and signature in Java:

```
F(oracle.sql.NUMBER, oracle.sql.NUMBER)
```

(Because both integers and floating point numbers map to the `NUMBER` class.)

Instead, JPublisher might call one `F_1` and the other `F_2`. (The `_`*`nn`* is unique for each. In simple cases it will likely be `_1`, `_2`, and so on, but it may sometimes be arbitrary other than being unique for each).

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—see the *Oracle8i JPublisher User's Guide* for information.

Generate Custom Java Classes and Map Alternative 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, subclass 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 custom Java class `JAddress` for the purpose of subclassing it to produce a class `MyAddress`. 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`.

Another alternative is to manually create a class that does not subclass the generated class, but instead uses it as a field.

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 above example, use this setting:

```
-sql=ADDRESS:JAddress:MyAddress
```

This generates class `JAddress` in source file `JAddress.java`, but does the following:

- Maps the object type `ADDRESS` 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, presumably you will retrieve it into an instance of `MyAddress`.
- Creates a `MyAddressRef` class in `MyAddressRef.java`, instead of creating a `JAddressRef` class.

You must manually produce a `MyAddress` class in source file `MyAddress.java`. This class can implement your required functionality either by subclassing `JAddress` or by using `JAddress` as a class field.

For further discussion about subclassing JPublisher-generated classes or using them as fields (continuing the preceding example), see "Extending or Wrapping Classes Generated by JPublisher" on page 6-33.

Using 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 "`AS`" or "`GENERATE...AS`" syntax is equivalent to 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`*.

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's how you typed it—JPublisher creates source `Myobj.java` (and `MyobjRef.java`).
- User-defined type `MYCOLL` is mapped to `MyCollClass`. JPublisher creates source `MyCollClass.java`.
- User-defined type `EMPLOYEE` is mapped to class `MyEmployee`. JPublisher creates source `Employee.java` and `MyEmployeeRef.java`. If you retrieve an object from the database that has an `EMPLOYEE` attribute, 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 an instance of MyEmployee. You must manually create source MyEmployee.java to define class MyEmployee. MyEmployee would either subclass Employee or wrap it by using one or more Employee objects as attributes.

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". As on the command line, but unlike in an input file, you can specify multiple mappings in a single jpub.sql setting.

Properties File Example In the following example, JPublisher will pick up the -user option from the command line and go to properties file jpub.properties 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.mapping=oracle
```

This produces the same results as the input-file example above, except that the oracle attribute-mapping setting is used.

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. This cannot be specified on the JPublisher command line, however. It can only be accomplished 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 Oracle object type `EMPLOYEE` 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 members 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; `getHomeAddress()`, for example, follows naming conventions, while `gethomeAddress()` does not.

JPublisher Implementation of Wrapper Methods

Note the following points about how JPublisher-generated method wrappers are generated:

- JPublisher-generated method wrappers are implemented in SQLJ; therefore, whenever `-methods=true`, the custom object class will be defined in a `.sqlj` file instead of a `.java` file. Run SQLJ to translate the `.sqlj` file.
- 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.

- Any output or input-output parameter is passed as the element of a one-element array. (This is to work around logistical issues with output and input-output parameters, as discussed in "Custom Java Class Support for Object Methods" on page 6-8.) 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.
- The `this` member of an instance of a JPublisher-generated custom Java class is treated differently, however. It can be used as either an input or an input-output parameter, depending on whether the server method actually modifies it. If `this` is input-output, then it is modified in place. This is as opposed to the way output and input-output parameters are generally handled in custom Java classes, where copies must be made if they are not passed as array elements.

Note: For release 8.1.5, generation of wrapper methods by JPublisher is a Beta feature.

JPublisher Custom Java Class Examples

This section provides examples of JPublisher Java source code output for the following user-defined types (created in "User-Defined Types in the Database" on page 6-13):

- 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`)

Example: Custom Object Class Source Code for Address.java 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 6-13, assume the Oracle object `ADDRESS` has only the `street` and `zip_code` attributes.

```
package bar;

import java.sql.SQLException;
import oracle.jdbc.driver.OracleConnection;
import oracle.jdbc.driver.OracleTypes;
import oracle.sql.CustomDatum;
import oracle.sql.CustomDatumFactory;
import oracle.sql.Datum;
import oracle.sql.STRUCT;
import oracle.jpub.MutableStruct;

public class Address implements CustomDatum, CustomDatumFactory
{
  public static final String _SQL_NAME = "SCOTT.ADDRESS";
  public static final int _SQL_TYPECODE = OracleTypes.STRUCT;

  public static CustomDatumFactory getFactory()
  { ... }

  /* constructor */
  public Address()
  { ... }

  /* CustomDatum interface */
  public Datum toDatum(OracleConnection c) throws SQLException
  { ... }

  /* CustomDatumFactory interface */
  public CustomDatum create(Datum d, int sqlType) throws SQLException
  { ... }

  /* accessor methods */
  public String getStreet() throws SQLException
  { ... }

  public void setStreet(String street) throws SQLException
  { ... }


  public String getZipCode() throws SQLException
  { ... }

  public void setZipCode(String zip_code) throws SQLException
  { ... }

}
```

Example: Custom Reference Class Source Code for 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 oracle.jdbc.driver.OracleConnection;
import oracle.jdbc.driver.OracleTypes;
import oracle.sql.CustomDatum;
import oracle.sql.CustomDatumFactory;
import oracle.sql.Datum;
import oracle.sql.REF;
import oracle.sql.STRUCT;

public class AddressRef implements CustomDatum, CustomDatumFactory
{
  public static final String _SQL_BASETYPE = "SCOTT.ADDRESS";
  public static final int _SQL_TYPECODE = OracleTypes.REF;

  public static CustomDatumFactory getFactory()
  { ... }

  /* constructor */
  public AddressRef()
  { ... }

  /* CustomDatum interface */
  public Datum toDatum(OracleConnection c) throws SQLException
  { ... }

  /* CustomDatumFactory interface */
  public CustomDatum create(Datum d, int sqlType) throws SQLException
  { ... }

  public Address getValue() throws SQLException
  { ... }

  public void setValue(Address c) throws SQLException
  { ... }
}
```

Example: Custom Collection Class Source Code for 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 oracle.jdbc.driver.OracleConnection;
import oracle.jdbc.driver.OracleTypes;
import oracle.sql.CustomDatum;
import oracle.sql.CustomDatumFactory;
import oracle.sql.Datum;
import oracle.sql.ARRAY;
import oracle.sql.ArrayDescriptor;
import oracle.jpub.runtime.MutableArray;

public class ModuletblT implements CustomDatum, CustomDatumFactory
{
  public static final String _SQL_NAME = "SCOTT.MODULETBL_T";
  public static final int _SQL_TYPECODE = OracleTypes.ARRAY;

  public static CustomDatumFactory getFactory()
  { ... }

  /* constructors */
  public ModuletblT()
  { ... }

  public ModuletblT(ModuleT[] a)
  { ... }

  /* CustomDatum interface */
  public Datum toDatum(OracleConnection c) throws SQLException
  { ... }

  /* CustomDatumFactory interface */
  public CustomDatum 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 or Wrapping 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 in either of the following ways:

- You can subclass the JPublisher-generated class.

or:

- You can write a new class that wraps the JPublisher-generated class. You can use the generated class type for a field in your custom class, delegating functionality to this field.

For example, suppose you want JPublisher to generate the class `JAddress` from the SQL object type `ADDRESS`. You also want to write a class `MyAddress` to represent `ADDRESS` objects and implement special functionality. The `MyAddress` class that you write can either extend `JAddress` or can have a `JAddress` field.

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, you would have to save a copy and then manually merge your changes back in.

JPublisher Functionality for Extending or Wrapping Generated Classes

As discussed in "Generate Custom Java Classes and Map Alternative Classes" on page 6-25, the JPublisher syntax to generate `JAddress` but map to `MyAddress` is as follows:

```
-sql=ADDRESS:JAddress:MyAddress
```

Or, in an input file:

```
SQL ADDRESS GENERATE JAddress AS MyAddress
```

As a result of this, JPublisher will generate the `REF` class `MyAddressRef` (in `MyAddressRef.java`) rather than `JAddressRef`.

Also, JPublisher alters the code it generates to implement the following functionality:

- The `MyAddress` class is used instead of the `JAddress` class to represent attributes whose database type is `ADDRESS`.
- The `MyAddress` class is used instead of the `JAddress` class to represent method arguments and function results whose type is `ADDRESS`.
- The `MyAddress` factory is used instead of the `JAddress` factory to construct Java objects whose database type is `ADDRESS`.

You would presumably use `MyAddress` similarly in any additional code that you write.

At runtime the Oracle JDBC driver will map any occurrences of `ADDRESS` data in the database to `MyAddress` instances instead of `JAddress` instances.

Requirements of Extended or Wrapper Classes

The class that you create (for example, `MyAddress.java`) must have the following features:

- It 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.
- It must implement the `CustomDatum` interface. If you are subclassing the class generated by JPublisher, then you can inherit the `toDatum()` method from the

generated class. If you are using the generated class as a field, then you can call its `toDatum()` method.

- It must define the fields `_SQL_NAME` and `_SQL_TYPECODE`. If you are subclassing the JPublisher-generated class, then you can inherit these. If you are wrapping the generated class then you must add this code yourself. You can copy the field definitions from the generated class.
- It or another class must implement the `CustomDatumFactory` interface. For example, you can have a class `MyAddress` that implements `CustomDatum` and a class `MyAddressFactory` that implements `CustomDatumFactory`.
- It must have a `getFactory()` method that returns an instance of your map class (such as a `MyAddress` object).

Example of Class Generated by JPublisher

Continuing the example in the preceding sections, here is sample code for the JPublisher-generated class (`JAddress`). Implementation details have been omitted.

```
import java.sql.SQLException;
import oracle.jdbc.driver.OracleConnection;
import oracle.jdbc.driver.OracleTypes;
import oracle.sql.CustomDatum;
import oracle.sql.CustomDatumFactory;
import oracle.sql.Datum;
import oracle.sql.STRUCT;
import oracle.jpub.runtime.MutableStruct;

public class JAddress implements CustomDatum, CustomDatumFactory
{
  public static final String _SQL_NAME = "SCOTT.ADDRESS";
  public static final int _SQL_TYPECODE = OracleTypes.STRUCT;

  public static CustomDatumFactory getFactory()
  { ... }

  /* constructor */
  public JAddress()
  { ... }

  /* CustomDatum interface */
  public Datum toDatum(OracleConnection c) throws SQLException
  { ... }
  /* CustomDatumFactory interface */
  public CustomDatum create(Datum d, int sqlType) throws SQLException
  { ... }

  /* shallow copy method: give object same attributes as argument */
  void shallowCopy(JAddress d) throws SQLException
  { ... }

  /* accessor methods */
  public String getStreet() throws SQLException
  { ... }

  public void setStreet(String street) throws SQLException
  { ... }


  public String getCity() throws SQLException
  { ... }

  public void setCity(String city) throws SQLException
  { ... }

  public String getState() throws SQLException
  { ... }

  public void setState(String state) throws SQLException
  { ... }

  public java.math.BigDecimal getZip() throws SQLException
  { ... }

  public void setZip(java.math.BigDecimal zip) throws SQLException
  { ... }

}
```

Example of Reference Class Generated by JPublisher

Continuing the example in the preceding sections, here is sample code for the JPublisher-generated reference class (`MyAddressRef`, as opposed to `JAddressRef`, because `MyAddress` is the class that `ADDRESS` objects map to). Implementation details have been omitted.

```
import java.sql.SQLException;
import oracle.jdbc.driver.OracleConnection;
import oracle.jdbc.driver.OracleTypes;
import oracle.sql.CustomDatum;
import oracle.sql.CustomDatumFactory;
import oracle.sql.Datum;
import oracle.sql.REF;
import oracle.sql.STRUCT;

public class MyAddressRef implements CustomDatum, CustomDatumFactory
{
  public static final String _SQL_BASETYPE = "SCOTT.ADDRESS";
  public static final int _SQL_TYPECODE = OracleTypes.REF;

  public static CustomDatumFactory getFactory()
  { ... }

  /* constructor */
  public MyAddressRef()
  { ... }

  /* CustomDatum interface */
  public Datum toDatum(OracleConnection c) throws SQLException
  { ... }

  /* CustomDatumFactory interface */
  public CustomDatum create(Datum d, int sqlType) throws SQLException
  { ... }

  public MyAddress getValue() throws SQLException
  { ... }

  public void setValue(MyAddress c) throws SQLException
  { ... }
}
```

Example of Extending JPublisher-Generated Class

Continuing the example in the preceding sections, here is sample code for a `MyAddress` class that subclasses the JPublisher-generated `JAddress` class. This code is somewhat generic, showing such things as what is inherited from `JAddress`. Implementation detail has been omitted.

```
import java.sql.SQLException;
import oracle.sql.CustomDatum;
import oracle.sql.CustomDatumFactory;
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 CustomDatumFactory getFactory()
  {
    return _myAddressFactory;
  }

  /* constructor */
  public MyAddress()
  { super(); }

  /* CustomDatum interface */
  /* toDatum() inherited from JAddress */

  /* CustomDatumFactory interface */
  public CustomDatum create(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.
  */
}
```

Example of Wrapping JPublisher-Generated Class

This is another example of producing a `MyAddress` class to enhance the functionality of the generated `JAddress` class, but using a `JAddress` field (`data`) instead of subclassing `JAddress`.

```
import java.sql.SQLException;
import oracle.jdbc.driver.OracleConnection;
import oracle.jdbc.driver.OracleTypes;
import oracle.sql.CustomDatum;
import oracle.sql.CustomDatumFactory;
import oracle.sql.Datum;
import oracle.sql.STRUCT;
import oracle.jpub.runtime.MutableStruct;

public class MyAddress implements CustomDatum, CustomDatumFactory {
  /* the container for the wrapped object */
  private JAddress data;

  // use these from JAddress
  public static final String _SQL_NAME = "SCOTT.ADDRESS";
  public static final int _SQL_TYPECODE = OracleTypes.STRUCT;

  static final MyAddress _MyAddressFactory = new MyAddress();
  public static CustomDatumFactory getFactory()
  {
    return _MyAddressFactory;
  }

  /* constructor */
  public MyAddress()
  {
    data = new JAddress();
  }

  /* CustomDatum interface */
  public Datum toDatum(OracleConnection c) throws SQLException
  {
    return data.toDatum(c);
  }
  /* CustomDatumFactory interface */
  public CustomDatum create(Datum d, int sqlType) throws SQLException
  {
    if (d == null) return null;
    MyAddress o = new MyAddress();
    o.data = (JAddress) JAddress.getFactory().create(d, sqlType);
    return o;
  }

  /* accessor methods.  These simply delegate to the wrapped object */
  public String getV1() throws SQLException
  { return data.getV1(); }

  public void setV1(String v1) throws SQLException
  { data.setV1(v1); }

  public String getV2() throws SQLException
  { return data.getV2(); }

  public void setV2(String v2) throws SQLException
  { data.setV2(v2); }

  public String getV3() throws SQLException
  { return data.getV3(); }

  public void setV3(String v3) throws SQLException
  { data.setV3(v3); }

  public String getV4() throws SQLException
  { return data.getV4(); }

  public void setV4(String v4) throws SQLException
  { data.setV4(v4); }

  /* add methods here for any additional desired MyAddress functionality */
}
```

Strongly Typed Objects and References in SQLJ Executable Statements

Oracle SQLJ is flexible in how it allows 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 attribute datatypes in the database.

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 attribute datatypes in the database.

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 first two examples, "Selecting Objects and Object References into Iterator Columns" and "Updating an Object", operate at the object level. The third example, "Inserting an Object Created from Individual Object Attributes" operates at the scalar-attribute level. The fourth example, "Updating an Object Reference", operates through a reference.

Refer back to the Oracle object types `ADDRESS` and `PERSON` in "Creating Object Types" on page 6-13.

For a complete sample application that includes most of the code in the following examples, see "Oracle Objects—ObjectDemo.sqlj" on page 12-26.

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 Oracle object type `ADDRESS`:

```
CREATE TYPE ADDRESS AS OBJECT
(  street VARCHAR(40),
   zip NUMBER );
```

And the following definition of the table `emps`, which includes an `ADDRESS` column and an `ADDRESS` reference column:

```
CREATE TABLE emps
(  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 Oracle object type `ADDRESS`, you can use `Address` and `AddressRef` in a named iterator as follows:

Declaration:

```
#sql iterator EmpIter (String name, Address home, AddressRef loc);
```

Executable code:

```
EmpIter ecur;
#sql ecur = { SELECT name, home, loc FROM emps };
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 method call `ecur.home()` extracts an `Address` object from the `home` column of the iterator and assigns it to the local variable `homeAddr` (for efficiency). The attributes of that object can then be accessed using standard Java dot syntax such as `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 6-13.

Updating an Object

This example declares and sets an input host variable of Java type `Address` 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 type `Address` and printed for verification.

```
...
// Updating an object

static void updateObject()
{

   Address addr;
   Address new_addr;
   int empno = 1001;

   try {
      #sql {
         SELECT office_addr
         INTO :addr
         FROM employees
         WHERE empnumber = :empno };
      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 = :empno };
         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 = :empno };

            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. For the source code of this method, see "Oracle Objects—ObjectDemo.sqlj" on page 12-26.

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 (`int` and `String`) 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 empno = 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 = :empno };

      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 = :empno } ;

      System.out.println("Current manager of "+empno+":");
      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. See the *Oracle8i SQL Reference* for more information about table aliases, REF, and DEREF.

Strongly Typed Collections in SQLJ Executable Statements

As is the case with strongly typed objects and references, Oracle SQLJ supports various 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.

Oracle SQLJ, and Oracle SQL in general, support various syntax that allows nested tables to be accessed and manipulated either apart from their outer tables or together with their outer tables. In this section, manipulation of a nested table by itself will be referred to as *detail-level* manipulation, while 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 back to the Oracle collection type `MODULETBL_T` and related tables and object types that were defined in "Creating Collection Types" on page 6-15.

For complete nested table sample applications, including one that incorporates the sample code below, see "Oracle Nested Tables—NestedDemo1.sqlj and NestedDemo2.sqlj" on page 12-35.

Following the nested table discussion are some brief VARRAY examples. There are also complete VARRAY sample applications, including one that incorporate this code, in "Oracle VARRAYs—VarrayDemo1.sqlj and VarrayDemo2.sqlj" on page 12-43.

Note: In Oracle SQLJ, both VARRAY types and nested table types can only be retrieved in their entirety. This is as opposed to Oracle SQL, where nested tables can be selectively queried.

Accessing Nested Tables—TABLE syntax and CURSOR syntax

Oracle SQLJ supports the use of nested iterators to access the data in nested tables. This involves use of the `CURSOR` keyword, used 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 6-52.

Oracle SQLJ 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 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. See the *Oracle8i SQL Reference* for more information about table aliases.

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 type MODULETBL_T, 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 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.)

Presume the following declaration:

```
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` for the nested table, and `p` for the `participants` table. See the *Oracle8i SQL Reference* for more information about table aliases.

```
/* 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. See the *Oracle8i SQL Reference* for more information about table aliases.

```
...

//  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 below:

#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 below 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 host expression. Presume the following SQL definitions:

```
CREATE TYPE PHONE_ARRAY IS VARRAY (10) OF varchar2(30)
/

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 custom collection class `PhoneArray` to map from the `PHONE_ARRAY VARRAY` type in the database.

The following method selects a row from this table, placing the data into a host variable of type `PhoneArray`.

```
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);
}
```

Serializing Java Objects through Custom Java Classes

In "Additional Uses for Custom Java Classes" on page 6-12, there are examples of situations where you may 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 in the database, with a custom Java class that maps to type `oracle.sql.RAW`.

This section provides an example of such an application, creating a class `SerializableDatum` that implements the `CustomDatum` interface and follows the general form of custom Java classes, as described in "About Custom Java Classes and the CustomDatum Interface" on page 6-6.

The example starts with a step-by-step approach to the development of `SerializableDatum`, followed by the complete sample code.

SerializableDatum (Step by Step Example)

1. Begin with a skeleton of the class.

Note: This application uses classes from the packages `java.io`, `java.sql`, `oracle.sql`, and `oracle.jdbc.driver`. The import statements are not shown here.

```
public class SerializableDatum implements CustomDatum
{
   // <Client methods for constructing and accessing the Java object>

   public Datum toDatum(OracleConnection c) throws SQLException
   {
      // <Implementation of toDatum()>
   }

   public static CustomDatumFactory getFactory()
   {
      return FACTORY;
   }

   private static final CustomDatumFactory FACTORY =
         // <Implementation of a CustomDatumFactory for SerializableDatum>

   // <Construction of SerializableDatum from oracle.sql.RAW>

   public static final int _SQL_TYPECODE = OracleTypes.RAW;
}
```

`SerializableDatum` does not implement the `CustomDatumFactory` interface, but its `getFactory()` method returns a static member that implements this interface.

The `_SQL_TYPECODE` is set to `OracleTypes.RAW` because this is the datatype being read from and written to the database. The SQLJ translator needs this typecode information in performing online type-checking to verify compatibility between the user-defined Java type and the SQL type in the database.

2. Define client methods which 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 instance of `oracle.sql.RAW`.

```
// 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 a `CustomDatumFactory`. In this case, it is implemented as an anonymous class.

```
// Implementation of a CustomDatumFactory for SerializableDatum

new CustomDatumFactory()
{
   public CustomDatum 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

Following is an example of using 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

Here is an example of using `SerializableDatum` as a named iterator column.

Declaration:

```
#sql iterator PersonIter (SerializableDatum info, String name);
```

Executable code:

```
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)

This section shows you the entire `SerializableDatum` class previously developed in step-by-step fashion.

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

public class SerializableDatum implements CustomDatum
{
// 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(OracleConnection 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 CustomDatumFactory getFactory()
   {
      return FACTORY;
   }

// Implementation of a CustomDatumFactory for SerializableDatum

   private static final CustomDatumFactory FACTORY =

      new CustomDatumFactory()
      {
         public CustomDatum 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 general use is not recommended and there are some specific restrictions, but in some circumstances they may be useful. For example, you may have generic code that can use "any `STRUCT`" or "any `REF`" (although if this uses dynamic SQL it would require coding in JDBC instead of SQLJ).

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 `oracle.sql` classes instead of the strongly typed custom Java classes that implement the `CustomDatum` interface.

The following `oracle.sql` classes can be used for iterator columns or host expressions in Oracle SQLJ:

- `oracle.sql.STRUCT` for objects
- `oracle.sql.REF` for object references
- `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 classes 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.*` type (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 do 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` instance), reference (`REF` instance), or collection (`ARRAY` instance) *cannot* be used in host expressions in the following circumstances:

- `IN` parameter if null
- `OUT` or `INOUT` parameter in stored procedure or function call
- `OUT` parameter in 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 in order to materialize an instance of a user-defined type in Java.

7 Advanced Language Features

This chapter discusses advanced SQLJ language features used in coding your application. For more basic topics, see Chapter 3, "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

Connection Contexts

SQLJ supports *connection contexts* for connecting to different types of database schemas from the same application.

Connection Context Concepts

When connecting to different schema types you will typically want to declare one or more connection context classes, as discussed in "Overview of SQLJ Declarations" on page 3-2. Each connection context class can be used for a particular type of schema, meaning that all the connections you define using a particular connection context class will use the same set of SQL objects (such as tables, views, and stored procedures). Note, however, that a connection context declaration does not define a type of schema that the connection context class is used for, and it is permissible to use the same connection context class for different schema types.

An example of a schema type is the set of tables and stored procedures used by the Human Resources department. Perhaps they use tables `EMPLOYEES` and `DEPARTMENTS` and stored procedures `CHANGE_DEPT` and `UPDATE_HEALTH_PLAN`. Another schema type might be the set of tables and procedures used by the Payroll department, perhaps consisting of the table `EMPS` (another table of employees, but different than the one used by HR) and stored procedures `GIVE_RAISE` and `CHANGE_WITHHOLDING`.

The advantage in tailoring connection context classes to database schemas is in the degree of online semantics-checking that this allows. To avoid semantics errors when doing online checking, all of the SQL objects used in SQLJ statements that use a given connection context class must match SQL objects found in the *exemplar schema* you provide for online checking of that connection context class. (The exemplar schema is the database connection you provide using the SQLJ translator `-user`, `-password`, and `-url` options. See "Connection Options" on page 8-31 for information about these options.)

If you have SQLJ statements that relate to a variety of schemas but use a single connection context class, then the exemplar schema you provide for this connection context class must be very general, containing all of the tables, views, and stored procedures that are used in any of the schemas. Alternatively, if all of the SQLJ statements using a given connection context class use a set of SQL objects belonging to a single schema type, then you can provide a more meaningful exemplar schema which allows more accurate semantics-checking.

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, you specify a particular schema (username, password, and URL) and a particular session and transaction in which SQL operations will execute. You typically accomplish this by specifying a username, 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, as discussed in "Specifying Connection Context Instances and Execution Context Instances" on page 3-11.

The following example shows basic declaration and use of a connection context class, `MyContext`, to connect to two different schemas (for typical usage, we will assume these schemas are of the same schema type):

Declaration:

```
#sql context MyContext;
```

Executable code:

```
MyContext mctx1 = new MyContext
     ("jdbc:oracle:thin@localhost:1521:ORCL", "scott", "tiger", false);
MyContext mctx2 =  new MyContext
     ("jdbc:oracle:thin@localhost:1521:ORCL", "brian", "mypasswd", false);
```

Note that connection context class constructors specify a boolean auto-commit parameter (this is further discussed in "Declaring and Using a Connection Context Class" on page 7-4).

Also note that you can connect to the identical schema with different connection context instances. During runtime, however, one connection does not see changes to the database made from the other connection until the changes are committed. In the example above, both `mctx1` and `mctx2` could connect to `scott/tiger` if desired.

Note: When some SQLJ documentation uses the term *connection context*, the term refers to a connection context *class* rather than a particular connection context *instance*. Remember that a connection context instance defines a particular connection, specifying a database URL, username, and password. A connection context class can be used for any number of such instances, typically to the same schema type. For clarity, this document specifies whether it is discussing a connection context class or a connection context instance.

When to Declare Connection Contexts

The following are examples of situations where you will typically declare one or more connection context classes:

- if you are connecting to different kinds of databases from the same application (such as Oracle8 and Sybase)
- if you are connecting to the same kind of database (such as Oracle8) but are using multiple types of schemas from the same application (in other words, you are using different sets of SQL objects such as tables, views, and stored procedures)

The program in "Example of Multiple Connection Contexts" on page 7-7 demonstrates the use of multiple contexts. It uses the default context to access a table of employees and a user-defined context to access employee department information. By using distinct contexts, it is possible for you to store the employee and department information in different schemas or even physically different databases.

For an overview of single connections vs. multiple connections, see "Connection Considerations" on page 4-9.

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 given any of the following (as with the `DefaultContext` class):

- URL (`String`), username (`String`), password (`String`), auto-commit (`boolean`)
- URL (`String`), `java.util.Properties` object, auto-commit (`boolean`)
- URL (`String` fully specifying connection and including username and password), auto-commit setting (`boolean`)
- JDBC `Connection` object only
- SQLJ connection context instance only

Notes:

- 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, see "Basic Transaction Control" on page 4-24.

Declaring the 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 `ConnectionContext`. 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. "Declaration IMPLEMENTS Clause" on page 3-5 and "Declaration WITH Clause" on page 3-6 discuss these.

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);
}
```

Instantiating a Connection Object

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:orcl", "scott", "tiger", true);
```

This is accomplished in the same way as instantiating the `DefaultContext` class using one of its constructors, as discussed in "More About the DefaultContext Class" on page 4-14.

Note: You will typically have to register your JDBC driver prior to constructing a connection context instance. See "Driver Selection and Registration for Runtime" on page 4-7.

Specifying a Connection 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 object is `myOrderConn` from the previous example:

```
#sql [myOrderConn] { UPDATE TAB2 SET COL1 = :w WHERE :v < COL2 };
```

This is the same way you might specify instances of `DefaultContext` if you require multiple connections but to the same type of database schema (as discussed in "Connection Considerations" on page 4-9).

Note: Your default connection must be an instance of the `DefaultContext` class, not of a declared connection context class. This means that any executable statement that uses an instance of a declared connection context class must explicitly specify the appropriate connection instance, as shown above.

Example of Multiple Connection Contexts

The following is an example of a SQLJ application using multiple connection contexts. It implicitly uses an object of the `DefaultContext` class for one type of schema and uses an instance of the declared connection context class `DeptContext` for another type of schema.

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
  {
    /* if you're using a non-Oracle JDBC Driver, add a call here to
       DriverManager.registerDriver() to register your Driver
    */

    // 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: Subclassing connection context classes is not permitted in the current SQLJ specification and is not supported by Oracle SQLJ.

ConnectionContext Interface

Each connection context class implements the `sqlj.runtime.ConnectionContext` interface.

Basic methods specified by the 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. Additionally, it closes the underlying JDBC connection if you pass in `CLOSE_CONNECTION`, a static boolean constant of the `ConnectionContext` interface. It does not close the underlying JDBC connection if you pass in the boolean `KEEP_CONNECTION`, which is also a static constant of `ConnectionContext`.

  If you do not pass in a parameter, the default is `CLOSE_CONNECTION`. If you do not explicitly close a connection context instance, the finalizer will close it with a `KEEP_CONNECTION` setting.

  Specify one of these constants as follows (assume a connection context instance `ctx`):

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

  For further discussion, see "Closing Shared Connections" on page 7-30.

- `getConnection()`—Returns the underlying JDBC `Connection` object for this connection context instance.
- `getExecutionContext()`—Returns the default `ExecutionContext` instance for this connection context instance. For more information, see "Execution Contexts" on page 7-13.

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:

- `getDefaultContext()`—This is a static method that returns the default connection context instance for a given connection context class.

- `setDefaultContext(`*`Your_Ctx_Class conn_ctx_instance`*`)`—This is a static method that defines the default context instance for a given connection context 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 allow you, for example, to specify a connection context instance for a SQLJ executable statement as follows.

Declaration:

```
#sql context MyContext;
```

Executable code:

```
...
MyContext myctx1 = new MyContext(url, user, password, auto-commit);
...
MyContext.setDefaultContext(myctx1);
...
#sql [MyContext.getDefaultContext()] { SQL operations };
...
```

Use of the IMPLEMENTS Clause in Connection Context Declarations

There might be situations where it is useful to implement an interface in your connection context declarations. For general information and syntax, see "Declaration IMPLEMENTS Clause" on page 3-5.

You might, for example, want to define an interface that exposes just a subset of the functionality of a connection context class. More specifically, you might want the capability of a class that has `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 type `HasConnection` instead of to a variable that has the type of your declared connection context class.

The declaration will be as follows (presume `HasConnection` is in package `mypackage`):

```
#sql public context MyContext implements mypackage.HasConnection;
```

Then you can 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:orcl", "scott", "tiger", true);
```

Semantics-Checking of Your Connection Context Usage

A significant feature of SQLJ is the strong typing of connection context classes, with each class typically used with a particular type of schema. This allows SQLJ semantics-checking to verify during translation that you are using your connection context instances correctly in your code. The type of schema specifies such properties as names and privileges associated with tables and views, the datatypes of their rows, and names and definitions of stored procedures.

To use online semantics-checking during translation, provide an example of the type of schema for any particular connection context class. You accomplish this by setting the SQLJ `-user`, `-password`, and `-url` options. These schema examples are sometimes referred to as *exemplar schemas*. For information about these SQLJ options, see "Connection Options" on page 8-31.

During semantics-checking, the translator connects to the specified exemplar schema for a particular connection context class and performs the following:

- 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)
- verifies that items in the SQL operations match the set of items existing in the exemplar schema

It is the responsibility of the application developer to pick an exemplar schema that represents the runtime schema in appropriate ways. For example, it must have tables, views, stored functions, and stored procedures with identical names and types, and with privileges set appropriately.

If no appropriate exemplar schema is available during translation for one of your connection context classes, or if it is inconvenient to connect to a schema of that type during translation, then you do not have to specify SQLJ options (`-user`, `-password`, `-url`) for that particular connection context class. In that case, SQLJ statements specifying connection objects of that connection context class are only semantically checked to the extent possible.

Note: Remember that the exemplar schema you specify in your translator option settings does not specify the actual schema that SQLJ allows in your code at runtime. The exemplar schema only furnishes the translator with an example of the type of schema that you use for a particular connection context class.

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 execution control, execution status, and execution cancellation, which function in the following ways:

- Execution control operations of a given execution context instance modify the semantics of subsequent SQL operations executed using that instance.
- Execution status operations of a given execution context instance describe the results of the most recent SQL operation that completed using that instance.
- Execution cancellation operations of a given execution context instance terminate the SQL operation that is currently executing using that instance.

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* refers to a *class* that you have declared, the term *execution context* refers to an *instance* of the `ExecutionContext` class. This document specifies *connection context class*, *connection context instance*, and *execution context instance* to avoid confusion.

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 employ the same connection context instance.
- You want to retain different sets of SQL status information from multiple SQL operations that employ the same connection context instance.

  As you execute successive SQL operations that employ the same execution context instance, the status information from each operation overwrites the status information from the previous operation.

Although execution context instances may appear to be associated with connection context instances (given that each connection context instance has its own default execution context instance, and that you can specify a connection context instance and an execution context instance together for a particular SQLJ statement), they actually operate independently. You can employ different execution context instances in statements that employ the same connection context instance, and vice versa.

For example, you would use multiple execution context instances with a single connection context instance if you were using multithreading (with a separate execution context instance for each thread). And you would use multiple connection context instances with a single execution context instance if you wanted the same set of SQL control parameters to apply to all of the connection context instances. (See "ExecutionContext Methods" on page 7-16 for information about SQL control settings.)

To employ 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. This is described in "Creating and Specifying Execution Context Instances" on page 7-14.

Creating and Specifying Execution Context Instances

To employ an execution context instance other than the default with a given connection context instance, you must construct another execution context instance. This is convenient, as there are no input parameters for the `ExectionContext` constructor:

```
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 declare and instantiate a connection context class `MyConnCtxClass` and create an instance `myConnCtx`, 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 };
```

Notes:

- 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 your default connection and its default execution context instance.

Execution Context Synchronization

`ExecutionContext` methods (discussed in "ExecutionContext Methods" on page 7-16) are all `synchronized` methods. Therefore, generally speaking, anytime a SQLJ statement tries to use an execution context instance that is 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, as discussed in "Multithreading in SQLJ" on page 7-19.

The exception to the preceding discussion is for recursion, which is typically encountered only in the server. Multiple SQLJ statements 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, and is further discussed in "Recursive SQLJ Calls in the Server" on page 11-32.

ExecutionContext Methods

This section lists the methods of the `ExecutionContext` class, categorized as status methods, control methods, and cancellation methods.

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:

- `getWarnings()`—Returns a `java.sql.SQLWarning` object containing the warnings reported by the most recent SQL operation that completed using this execution context instance. The `SQLWarning` object is initially created for the first warning reported, and any subsequent warnings are chained to the same object. The `SQLWarning` object ultimately represents all warnings generated during the execution of the SQL operation and the subsequent outputting of parameters to the output host expressions.

- `getUpdateCount()`—Returns an `int` 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 DML statement. The constant `QUERY_COUNT` is returned if the last SQL operation produced an iterator or result set. The constant `EXCEPTION_COUNT` is returned if the last SQL operation terminated before completing execution, or if no operation has yet been attempted using this execution context instance.

- `getNextResultSet()`—Under some circumstances a SQL procedure call can return multiple result sets (these are known as *side-channel result sets* and are not supported by Oracle8*i*). In such a case, this method returns the next result set, in a JDBC `ResultSet` object. Further calls move to and return subsequent result sets. When there are no further side-channel result sets, `null` is returned. This method implicitly closes each open result set that it had previously obtained.

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).

- `getMaxFieldSize()`—Returns an `int` specifying the maximum amount of data (in bytes) that would be returned from a SQL operation subsequently using this execution context instance. (This can be modified using the `setMaxFieldSize()` method.) This applies only to columns of type `BINARY`, `VARBINARY`, `LONGVARBINARY`, `CHAR`, `VARCHAR`, or `LONGVARCHAR`.

  By default this parameter is set to 0, meaning there is no size limit.
- `setMaxFieldSize()`—Takes an `int` as input to modify the field-size maximum.
- `getMaxRows()`—Returns an `int` specifying the maximum number of rows that can be contained by any SQLJ iterator or JDBC result set created using this execution context instance. (This can be modified using the `setMaxRows()` method.) If the limit is exceeded, 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()`—Takes an `int` as input to modify the row maximum.
- `getQueryTimeout()`—Returns an `int` specifying the timeout limit, in seconds, for any SQL operation that uses this execution context instance. (This can be modified using the `setQueryTimeout()` method.) If a SQL operation exceeds this limit, a SQL exception is thrown.

  By default this parameter is set to 0, meaning there is no query timeout limit.
- `setQueryTimeout()`—Takes an `int` as input to modify the query timeout limit.

Cancellation Method

The following method can be used to cancel SQL operations in a multithreading environment.

- `cancel()`—This method is used by one thread to cancel a SQL operation currently being executed by 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.

Example: Use of 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, 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, perform the following:

- Instantiate a unique execution context instance for use with each thread.
- Specify execution contexts with your `#sql` statements so that each thread uses its own execution context (see "Specifying an Execution Context" above).

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.

See "Multithreading in SQLJ" on page 7-19 for more information about multithreading.

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.

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.

For information about how to specify connection context instances and execution context instances for your SQLJ statements, see "Specifying Connection Context Instances and Execution Context Instances" on page 3-11.

If you are using one of the Oracle JDBC drivers, 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 the user. Note, however, that database access is sequential—only one thread is accessing the database at any given time. (Synchronization refers to the control flow of the various stages of the SQL operations executing through your threads. Each statement, for example, may bind input parameters, then execute, then bind output parameters. With some JDBC drivers, special care must be taken not to intermingle these stages of different operations.)

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, the results of a SQL operation performed by one thread would be visible in the other thread. If both threads were executing SQL operations, 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.

For a complete multithreading sample application, see "Multithreading—MultiThreadDemo.sqlj" on page 12-55.

Iterator Class Implementation and Advanced Functionality

This section discusses how iterator classes are implemented and what additional functionality they offer beyond the essential methods discussed in "Using Named Iterators" on page 3-40 and "Using Positional Iterators" on page 3-45.

Note: Subclassing iterator classes is not permitted in the current SQLJ specification and is not supported by Oracle SQLJ.

Implementation and Functionality of Iterator Classes

Any named iterator class you declare will be generated by the SQLJ translator to implement the interface `sqlj.runtime.NamedIterator`. Classes implementing the `NamedIterator` interface have functionality that maps iterator columns to database columns by name, as opposed to by position.

Any positional iterator class you declare will be generated by the SQLJ translator to implement the interface `sqlj.runtime.PositionedIterator`. Classes implementing the `PositionedIterator` interface have functionality that maps iterator columns to database columns by position, as opposed to by name.

Both the `NamedIterator` interface and the `PositionedIterator` interface, and therefore all generated SQLJ iterator classes as well, implement or extend the interface `sqlj.runtime.ResultSetIterator`.

The `ResultSetIterator` interface specifies the following methods for all SQLJ iterators (both named and positional):

- `close()`—Closes the iterator.
- `getResultSet()`—Extracts the underlying JDBC result set from the iterator.
- `isClosed()`—Determines if the iterator has been closed.
- `next()`—Moves to the next row of the iterator.

The `PositionedIterator` interface adds the following method specification for positional iterators:

- `endFetch()`—Determines if you have reached the last row of a positional iterator.

As discussed in "Using Named Iterators" on page 3-40, 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.

As discussed in "Using Positional Iterators" on page 3-45, 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 and 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.

Use the `close()` method to close any iterator once you are done with it.

The `getResultSet()` method is central to SQLJ-JDBC interoperability and is discussed in "SQLJ Iterator and JDBC Result Set Interoperability" on page 7-31.

Note: You can use a `ResultSetIterator` object directly as a weakly typed iterator if you are only interested in converting it to a JDBC result set and you do not need named or positional iterator functionality. For information, see "Using and Converting Weakly Typed Iterators (ResultSetIterator)" on page 7-34.

Use of the IMPLEMENTS Clause in Iterator Declarations

There may be situations where it will be useful to implement an interface in your iterator declaration. For general information and syntax, see "Declaration IMPLEMENTS Clause" on page 3-5.

You may, for example, have an iterator class where you want to restrict access to one or more columns. As discussed in "Using Named Iterators" on page 3-40, a named iterator class generated by SQLJ will have 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 objects of the interface type to the user instead of exposing objects of the iterator class type.

Presume you are creating a named iterator of employee data, with columns `empname`, `empnum`, and `empsalary`. Accomplish this as follows:

```
#sql iterator EmpIter (String empname, int empnum, float empsalary);
```

This generates a class `EmpIter` with accessor methods `empname()`, `empnum()`, and `empsalary()`.

Presume, though, that you want to prevent access to the `empsalary` column. You can create an interface `EmpIterIntfc` that has `empname()` and `empnum()` methods but no `empsalary()` method, then you can use the following iterator declaration instead of the declaration above (presume `EmpIterIntfc` is in package `mypackage`):

```
#sql iterator EmpIter implements mypackage.EmpIterIntfc
     (String empname, int empnum, float empsalary);
```

Then if you code your application so that users can access data only through `EmpIterIntfc` objects, they will not have access to the `empsalary` column.

Advanced Transaction Control

SQLJ supports the SQL `SET TRANSACTION` statement to specify the access mode and isolation level of any given transaction. Supported settings for access mode are `READ ONLY` and `READ WRITE`. Supported settings for isolation level are `SERIALIZABLE`, `READ COMMITTED`, `READ UNCOMMITTED`, and `REPEATABLE READ`. Oracle SQL, however, does not support `READ UNCOMMITTED` or `REPEATABLE READ`.

`READ WRITE` is the default access mode in both standard SQL and Oracle SQL.

`READ COMMITTED` is the default isolation level in Oracle SQL; `SERIALIZABLE` is the default in standard SQL.

Access modes and isolation levels are briefly described below. For more information, see the *Oracle8i SQL Reference*. You might also consult any guide to standard SQL for additional conceptual information.

For an overview of transactions and information about SQLJ support for more basic transaction control functions, such as `COMMIT` and `ROLLBACK`, see "Basic Transaction Control" on page 4-24.

SET TRANSACTION Syntax

In SQLJ, the `SET TRANSACTION` statement has the following syntax:

```
#sql { SET TRANSACTION <access_mode>, <ISOLATION LEVEL isolation_level> };
```

If you use `SET TRANSACTION` it must be the first statement in your transaction (in other words, the first statement since your connection to the database or your most recent `COMMIT` or `ROLLBACK`), preceding any DML statements.

In a `SET TRANSACTION` statement, you can optionally specify the isolation level first, or specify only the access mode or only the isolation level. Following are some examples:

```
#sql { SET TRANSACTION READ ONLY };

#sql { SET TRANSACTION ISOLATION LEVEL SERIALIZABLE };

#sql { SET TRANSACTION READ WRITE, ISOLATION LEVEL SERIALIZABLE };

#sql { SET TRANSACTION ISOLATION LEVEL READ COMMITTED, READ ONLY };
```

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 have the following functionality:

- READ WRITE (default)—In a READ WRITE transaction, the user is allowed to update the database. SELECT, INSERT, UPDATE, and DELETE are all legal.
- READ ONLY—In a READ ONLY transaction, the user is 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 (not supported by Oracle8*i*)—Dirty reads, non-repeatable reads, and phantom reads are all allowed.
- READ COMMITTED—*Dirty reads* are prevented; *non-repeatable 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. (See below for definitions of the italicized terms.)
- REPEATABLE READ (not supported by Oracle8*i*)—Dirty reads and non-repeatable reads are prevented; phantom reads are allowed.
- SERIALIZABLE—Dirty reads, non-repeatable reads, and phantom reads are all prevented. Any DML statements in the transaction cannot update any resource that may 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 *non-repeatable 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 an Oracle database—use a "greater" setting (one further to the left) to ensure having at least the level of isolation that you want.

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 `java.sql.Connection` instance of your connection context instance.

Note that this is not recommended. SQLJ code using these JDBC methods is not portable.

Following are the `Connection` class methods for access mode and isolation level settings:

- `public 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`
- `public abstract void setTransactionIsolation(int)`—Sets the transaction isolation level, taking as input one of the preceding constant values.
- `public abstract boolean isReadOnly()`—Returns `true` if the transaction is `READ ONLY`; returns `false` if the transaction is `READ WRITE`.
- `public 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

As described in "Introduction to SQLJ" on page 1-2, you can use SQLJ statements for static SQL operations but must use JDBC statements for dynamic SQL operations. There may be situations where your application will require both static and dynamic SQL operations. SQLJ allows you to use SQLJ statements and JDBC statements concurrently and provides interoperability between SQLJ constructs 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

For general information about JDBC functionality, see the *Oracle8i JDBC Developer's Guide and Reference*.

SQLJ Connection Context and JDBC Connection Interoperability

SQLJ allows 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 physical database connection. See "About Shared Connections" on page 7-30.

Converting from Connection Contexts to JDBC Connections

If you want to perform a dynamic SQL operation through a database connection you have established in SQLJ (for example, an operation where the name of the table to select from is not determined until runtime), 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 `java.sql.Connection` instance and a `getConnection()` method that returns that `Connection` instance. Use the `Connection` instance to create JDBC statement objects if you want to use any dynamic SQL operations.

Following is an example of how to use the `getConnection()` method.

Imports:

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

Executable code:

```
DefaultContext ctx = new DefaultContext
         ("jdbc:oracle:thin:@localhost:1521:orcl", "scott", "tiger", true);
...
(static operations through SQLJ ctx connection context instance)
...
Connection conn = ctx.getConnection();
...
(dynamic 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 "Connection Considerations" on page 4-9 for an introduction to connection context instances and default connections. See "More About the Oracle Class" on page 4-13 for information about the `Oracle.connect()` method.)

Following is an example.

Imports:

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

Executable code:

```
...
Connection conn = Oracle.connect(
     "jdbc:oracle:thin:@localhost:1521:orcl", "scott", "tiger").getConnection();
...
(dynamic 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. The dynamic operations are performed using JDBC `java.sql.Connection`, `java.sql.PreparedStatement`, and `java.sql.ResultSet` objects. (For information about such basic features of JDBC programming, see the *Oracle8i JDBC Developer's Guide and Reference*.)

```
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();
}
```

Converting from JDBC Connections to Connection Contexts

If you initiate a connection as a JDBC `java.sql.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 database connection. As a result, the following is true:

- When you get a JDBC `java.sql.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 this can be modified using 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 database session that is being altered.

Closing Shared Connections

Whether you get a JDBC `java.sql.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 only need to close the connection context instance. By default, calling the `close()` method of a connection context instance closes the associated JDBC `Connection` instance and the underlying database connection, thereby freeing all resources associated with the connection.

Note, however, that closing the JDBC `Connection` instance will *not* close the associated SQLJ connection context instance. The underlying database connection would be closed, but the resources of the connection context instance would not be freed until garbage collection.

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 `KEEP_CONNECTION` to the `close()` method, as follows (presume you have been using 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 `ConnectionContext` interface.

If you do not explicitly close a connection context instance, then it will be closed by the finalizer during garbage collection with `KEEP_CONNECTION`, meaning the resources of the JDBC `Connection` instance would not be freed until released explicitly or by garbage collection.

SQLJ Iterator and JDBC Result Set Interoperability

SQLJ allows 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 may find yourself manipulating JDBC result sets. For example, another package may be implemented in JDBC and may provide access to data only through result sets, or may require `ResultSetMetaData` information because it is a routine written generically for any kind of result set. Or your SQLJ application may 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. Presume `myEmpQuery()` is a static Java function in a class called `RSClass`, with a predefined query that returns a JDBC result set object.

Imports and declarations:

```
import java.sql.*;
...
#sql public iterator MyIterator (String empname, float empsal);
...
```

Executable code:

```
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, the SQLJ translator will generate a warning unless you use the `-warn=nostrict` option setting.)

- The result set being cast must implement the interface `java.sql.ResultSet`. (The class `oracle.jdbc.driver.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.

For a complete example of how SQLJ and JDBC can interoperate in the same program, see "Interoperability with JDBC—JDBCInteropDemo.sqlj" on page 12-57.

Converting from Named or Positional Iterators to JDBC Result Sets

You may also encounter situations where you want to define a query using SQLJ but ultimately need a result set. (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.

Imports and declarations:

```
import java.sql.*;

#sql public iterator MyIterator (String empname, float empsal);
...
```

Executable code:

```
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 may have a situation similar to what is discussed in "Converting from Named or Positional Iterators to JDBC Result Sets" on page 7-33, but where you do not at any time require the strongly typed functionality of the iterator. All you may 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 interface `sqlj.runtime.ResultSetIterator` to receive query data, and avoid having to declare a named or positional iterator class.

In using `ResultSetIterator` 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.

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.

As discussed in "Iterator Class Implementation and Advanced Functionality" on page 7-21, the `ResultSetIterator` interface underlies all named and positional iterator classes and specifies the `getResultSet()` and `close()` methods.

Following is an example of how to use and convert a weakly typed iterator.

Imports:

```
import sqlj.runtime.*;
import java.sql.*;
```

```
...
```

Executable code:

```
ResultSetIterator rsiter;
...
#sql rsiter = { SELECT * FROM table };
ResultSet rs = rsiter.getResultSet();
...
(process result set)
...
rsiter.close();
...
```

The following rules apply when converting a `ResultSetIterator` object to a JDBC result set and processing the data:

- There is no data-access functionality in a `ResultSetIterator` object. You must convert it to a result set to access the query data.
- When you finish, close the `ResultSetIterator` object, not the result set. Closing the `ResultSetIterator` will also close the result set, but closing the result set will not close the `ResultSetIterator`. When interoperating with JDBC, always close the SQLJ entity.

Note: As `ResultSetIterator` objects are only intended to be converted to JDBC result sets, they are not supported as host expressions in SQLJ.

8 Translator Command Line and Options

Once you have written your source code, you must translate it using the SQLJ translator. This chapter discusses the SQLJ translator command line, options, and properties files.

The following topics are discussed:

Translator Command Line and Properties Files

This section discusses general command-line syntax for the script `sqlj` that you use to run the SQLJ translator, and lists all of the options available. It then discusses SQLJ properties files, which can be used instead of the command line to set most options, and the `SQLJ_OPTIONS` environment variable, which can be used in addition to or instead of the command line for setting options. For detailed information about settings for the basic options, see "Basic Translator Options" on page 8-20. For information about more advanced options, see "Advanced Translator Options" on page 8-47 and "Translator Support and Options for Alternative Environments" on page 8-62.

The `sqlj` script invokes a Java VM and passes the class name of the SQLJ translator (`sqlj.tools.Sqlj`) to the VM. The VM invokes the translator and handles things 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".

This is the typical general syntax for the command line:

```
sqlj <optionlist> filelist
```

The *option list* is a list of SQLJ option settings separated by spaces. There are also prefixes to mark options to be passed to other executables.

The *file list* is the list of files, separated by spaces, to be processed by the SQLJ translator (they can be `.sqlj`, `.java`, `.ser`, or `.jar` files, as explained in "Command-Line Syntax and Operations" on page 8-10). The `*` wildcard entry can be used in file names. For example, `Foo*.sqlj` would find `Foo1.sqlj`, `Foo2.sqlj`, and `Foobar.sqlj`.

Do not include `.class` files in the file list but do be sure that your `CLASSPATH` is set so that the SQLJ translator can find any `.class` files it must have for type resolution of variables in your SQLJ source files.

Notes:

- Discussion of the SQLJ command line applies only to client-side translation, not server-side translation. There is a different mechanism for specifying options to SQLJ in the server. For information, see "Option Support in the Server Embedded Translator" on page 11-16.
- 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.

SQLJ Options, Flags, and Prefixes

This section discusses options supported by the SQLJ translator. Boolean options are referred to as *flags*. Also listed are *prefixes*, used to pass options to the Java VM, which the SQLJ script invokes, and to the Java compiler and SQLJ profile customizer, which the Java VM invokes.

Use an equals sign (=) to specify option and flag settings, although for simplicity you do not have to specify `=true` to turn on a flag; simply typing the flag name on the command line will suffice. You must, however, 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

Notes:

- 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 below, accept alternative syntax if specified on the command line to support compatibility with the Oracle `loadjava` utility.
- Several `javac` options are recognized directly by SQLJ if specified on the command line, as indicated in Table 8-1. All of these are passed to your Java compiler (presumably `javac`), and some also affect SQLJ operation.
- Most SQLJ options can also be set in a *properties file*. See "Properties Files for Option Settings" on page 8-13.
- The `SQLJ_OPTIONS` environment variable can be used in addition to or instead of the command line for setting options. See "SQLJ_OPTIONS Environment Variable for Option Settings" on page 8-17.
- If the same option appears more than once on the command line (or in the properties file), then the last value is used.
- In this document, boolean flags are usually discussed as being `true` or `false`, but they can also be enabled / disabled by setting them to `yes` / `no`, `on` / `off`, `1` / `0`.

For an example and discussion of command-line syntax and operations, see "Command-Line Syntax and Operations" on page 8-10.

Summary of SQLJ Options

Table 8-1 below lists options supported by the SQLJ translator, categorized as follows:

- Flags and options listed as "Basic" are discussed in "Basic Translator Options" on page 8-20.
- 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 situations where you are using a non-standard Java VM, compiler, or customizer.
- Options listed as "javac Compatible" are `javac` options that SQLJ supports and that are also passed directly to the Java compiler (presumably `javac`). These options are discussed in "Options for javac Compatibility" on page 8-9.

Table 8–1 SQLJ Translator Options

Option	Description	Default	Category
-C	prefix that marks options to pass to Java compiler	n/a	Advanced
-cache	flag for whether to cache the online semantics-checking results (to reduce trips to database)	false	Advanced
-classpath (command-line only)	option to specify `CLASSPATH` to Java VM and Java compiler (passed to `javac`)	none	Basic
-compile	flag to run the Java compilation step (for `.java` files generated during current SQLJ run, or previously generated `.java` files specified on the command line)	true	Advanced
-compiler-executable	option to specify the Java compiler to use	javac	Environment
-compiler-encoding-flag	flag to tell SQLJ whether to pass the `-encoding` setting (if that option is set) to the Java compiler	true	Environment
-compiler-output-file	option to specify a file to which the Java compiler output should be written (There is no default value; however, if this option is not set, SQLJ assumes that compiler output goes to standard output.)	none	Environment
-compiler-pipe-output-flag	flag instructing SQLJ whether to set `javac.pipe.output` system property, which determines whether the Java compiler outputs errors and messages to `STDOUT` instead of `STDERR`	true	Environment

Table 8–1 SQLJ Translator Options (Cont.)

Option	Description	Default	Category
-d	option to set output directory for profile (`.ser`) files generated by SQLJ and `.class` files generated by compiler (passed to `javac`)	empty (use directory of `.java` file; i.e., use same directory as -dir option)	Basic
-default-customizer	option to specify the profile customizer to use; specify a class name	oracle.sqlj.runtime.util.OraCustomizer	Environment
-default-url-prefix	option to set the default prefix for URL settings	jdbc:oracle:thin:	Basic
-depend (command-line only)	passed to `javac` only	n/a	`javac` Compatible
-dir	option to set output directory for SQLJ-generated `.java` files	empty (use directory of `.sqlj` input file)	Basic
-driver	option to specify JDBC driver class to register; specify a class name or comma-separated list of class names	oracle.jdbc.driver.OracleDriver	Basic
-encoding (also recognized as -e if on command line)	option to specify NLS encoding for SQLJ and compiler to use (passed to `javac`)	VM `file.encoding` setting	Basic
-explain	flag to request "cause" and "action" information to be displayed with translator error messages	false	Basic
-g (command-line only)	passed to `javac`; enables `-linemap`	n/a	`javac` Compatible
-help (also recognized as -h) -help-long -help-alias (all command-line only)	flags to display different levels of information about SQLJ option names, descriptions, and current values	not enabled	Basic
-J (command-line only)	prefix that marks options to pass to the Java VM	n/a	Advanced
-linemap	flag to enable mapping of line numbers between generated Java source code and original SQLJ code	false	Basic

Table 8–1 SQLJ Translator Options (Cont.)

Option	Description	Default	Category
-n (command-line only; alternatively -vm=echo)	flag instructing `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	n/a	Basic
-nowarn (command-line only)	passed to `javac`; sets `-warn=none`	n/a	`javac` Compatible
-O (command-line only)	passed to `javac`; disables `-linemap`	n/a	`javac` Compatible
-offline	option to specify offline checker to use for semantics-checking; specify a fully qualified class name	oracle.sqlj.checker. OracleChecker	Advanced
-online	option to specify online checker to use for semantics-checking; specify a fully qualified class name	oracle.sqlj.checker. OracleChecker	Advanced
-P	prefix that marks options to pass to SQLJ profile customizer	n/a	Advanced
-passes (command-line only)	flag instructing `sqlj` script to run SQLJ in two separate passes, with compilation in between	false	Environment
-password (also recognized as -p if on command line)	option to set user password for database connection for online semantics-checking	none	Basic
-profile	flag to run the profile customization step (for profile files generated during current SQLJ run)	true	Advanced
-props (command-line only)	option to specify properties file (alternative to command line for setting options); `sqlj.properties` is also still read	none	Basic
-ser2class	flag to instruct SQLJ to translate generated `.ser` profiles to `.class` files	false	Advanced
-status (also recognized as -v if on command line)	flag requesting SQLJ to display status messages as it runs	false	Basic

Table 8–1 SQLJ Translator Options (Cont.)

Option	Description	Default	Category
-url	option to set database URL for connection for online semantics-checking	jdbc:oracle:oci8:@	Basic
-user (also recognized as -u if on command line)	option to enable online semantics-checking and set username (and optionally password and URL) for database connection	none (no online semantics-checking)	Basic
-verbose (command-line only)	passed to `javac`; enables `-status`	n/a	`javac` Compatible
-version -version-long (both command-line only)	flag to display different levels of SQLJ and JDBC driver version information	not enabled	Basic
-vm (command-line only)	option to specify Java VM to use	java	Environment
-warn	comma-separated list of flags to enable or disable various SQLJ warnings—individual flags are `precision`/`noprecision`, `nulls`/`nonulls`, `portable`/`noportable`, `strict`/`nostrict`, and `verbose`/`noverbose`; global flag is `all`/`none`	precision nulls noportable strict noverbose	Basic

Options for loadjava Compatibility

For compatibility with the `loadjava` utility used to load Java and SQLJ applications into the Oracle8*i* server, the following alternative syntax is recognized for some options when specified on the command line (this is also noted in Table 8-1 above):

- `-e` (equivalent to `-encoding`)
- `-h` (equivalent to `-help`)
- `-p` (equivalent to `-password`)
- `-u` (equivalent to `-user`)
- `-v` (for verbose message output; equivalent to `-status`)

To maintain full consistency with `loadjava` syntax, you can use a space instead of "=" in setting these options, as in the following example:

```
-u scott/tiger -v -e SJIS
```

For general information about the `loadjava` utility, see the *Oracle8i Java Stored Procedures Developer's Guide*.

Note: This alternative option syntax is only recognized on the command line or in the `SQLJ_OPTIONS` environment variable, not in properties files.

Options for javac Compatibility

For compatibility with `javac`, the Java compiler provided with the Sun Microsystems JDK, the following `javac` options are accepted directly by SQLJ without the `-C` prefix if specified on the command line. As indicated: some also serve as SQLJ options; some are not SQLJ options per se but also set SQLJ options; some affect `javac` only. This is also indicated in Table 8-1 above. Refer to your `javac` documentation for information about `javac` option settings and functionality.

- `-classpath` (also a SQLJ option; sets the `CLASSPATH` for both `javac` and the Java VM)

  See "CLASSPATH for Java VM and Compiler (-classpath)" on page 8-21.

- `-d` (also a SQLJ option; sets the output directory for `.class` files and SQLJ profile files)

  See "Output Directory for Generated .ser and .class Files (-d)" on page 8-28.

- `-depend` (`javac` option only; compiles out-of-date files recursively)
- `-encoding` (also a SQLJ option; sets encoding for both SQLJ and `javac`)

  See "Encoding for Input and Output Source Files (-encoding)" on page 8-27.

- `-g` (generates `javac` debugging information; also sets SQLJ `-linemap=true`)

  See "Line-Mapping to SQLJ Source File (-linemap)" on page 8-45.

- `-nowarn` (instructs `javac` to generate no warnings; also sets SQLJ `-warn=none`)

  See "Translator Warnings (-warn)" on page 8-42.

- `-O` (instructs `javac` to optimize; also sets SQLJ `-linemap=false`)

See "Line-Mapping to SQLJ Source File (-linemap)" on page 8-45.

- `-verbose` (instructs `javac` to output real-time status messages; also sets SQLJ `-status=true`)

  See "Real-Time Status Messages (-status)" on page 8-44.

Profile Customizer Options

Profile customizer options, both options for the customizer harness front end and for the default Oracle customizer, are documented in "Customization Options and Choosing a Customizer" on page 10-11.

Command-Line Syntax and Operations

The general sequence of events triggered by running the script `sqlj` was discussed in "Translation Steps" on page 1-7. 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 Java VM, it passes all of its command-line arguments to the VM, which later passes them elsewhere (such as to the Java compiler or profile customizer) as appropriate.

Arguments from the Option List Option list arguments are used in the following ways:

- Options designated by the `-J` prefix are Java VM options and are used by the VM directly. Such options must be specified on the command line or in the `SQLJ_OPTIONS` environment variable.
- Options not designated by the `-J`, `-C`, or `-P` prefixes are SQLJ options and are passed to the SQLJ translator as the Java VM invokes it.
- Options designated by the `-C` prefix are Java compiler options and are passed to the compiler as the Java VM invokes it.

  Note that there are three SQLJ options that have the same name as Java compiler options and, if specified, are automatically passed to the Java compiler as well as being used by SQLJ:

  - `-d` is used by SQLJ to specify the output directory for its generated profile files, and is also passed to the compiler which uses it to specify the output directory for its generated `.class` files.
  - `-encoding` is used by SQLJ in reading `.sqlj` files and generating `.java` files, and is also passed to the Java compiler (unless the `-compiler-encoding-flag` is off), which uses it in reading `.java` files.
  - `-classpath` is passed by SQLJ to both the Java compiler and the Java VM to set the `CLASSPATH` for both. It must be specified on the command line or in the `SQLJ_OPTIONS` environment variable.

  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 must use the `-C` prefix (and the `-J` prefix) for `-classpath` if you want to set different `CLASSPATH` values for the compiler and VM.

- Options designated by the `-P` prefix are SQLJ profile customizer options and are passed to the customizer as the Java VM invokes it.

  Any profile customization other than what SQLJ performs automatically is considered an advanced feature and is covered in Chapter 10, "Profiles and Customization".

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:

- `.sqlj` files are processed by the SQLJ translator, Java compiler, and SQLJ profile customizer.

  In most cases an application requires only one `.sqlj` file, but you should use additional files if you declare additional public classes, such as iterator and connection context classes.

- `.java` files are processed by the Java compiler and are also used by the SQLJ translator for type resolution.
- profile files (`.ser` by default, or optionally `.class`) and `.jar` files are processed only by the profile customizer.

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. (See "Use of .jar Files for Profiles" on page 10-26 for details about how `.jar` files are handled.)

If you have `.sqlj` files and `.java` files with interdependencies (each requiring access to code in the others), you must 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 of the types.

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, all but the first reference are discarded from the command line. Also, if you list a `.java` file and `.sqlj` file with the same base name and in the same location without using the `-dir` option, only the `.sqlj` file is processed. This processing also applies to wild-card file name characters.

Consider the following command-line examples, presuming 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`

  This would result in the translator both writing to and reading from `Foo.java` in the same execution, so the translator discards `Foo.java` from the command line.

- `sqlj Foo.*`

  The translator would find both `Foo.sqlj` and `Foo.java`, which again would cause it to both write to and read from `Foo.java` in the same execution. Again, the translator discards `Foo.java` from the command line.

- `sqlj -dir=outdir Foo.sqlj Foo.java`

  This is okay, because the generated `Foo.java` will be in the `outdir` subdirectory, while the `Foo.java` being read is in the `/myhome/mypackage directory`.

This processing of the command line means that you can, for example, type the following command and have it execute without difficulty (with files references being automatically discarded as necessary):

```
sqlj *.sqlj *.java
```

This may be convenient in many situations.

Command Line Example and Results

Below is a sample command line. 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 Java VM, passes the class name of the SQLJ translator to the VM, then passes the command-line arguments to the VM (which later passes them to the translator, compiler, and customizer, as appropriate). If there are any options for the Java VM, as designated by `-J`, the script passes them to the VM 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 by hand).

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
```

(The above is all one line; it just wraps around.)

For more information about how Java VM options are handled, see "Options to Pass to Java VM (-J)" on page 8-47.

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.

For more information, see "Echo Command Line without Execution (-n)" on page 8-25.

Properties Files for Option Settings

You can use *properties files* instead of the command line to supply options to the SQLJ translator, Java compiler, and SQLJ profile customizer.

In addition, if your Java compiler will be running in a separate Java VM and you want to specify options to this VM regarding operation of the compiler, then you can use properties files to supply such options to this VM. Such options are passed to the VM at the time the compiler is run, after the SQLJ translation step. (It is more typical, however, to pass options to the compiler's VM 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 Java VM, for example, because properties files are read after the VM is invoked.

You also *cannot* do the following in properties files:

- Set `javac` options supported by SQLJ (`-depend`, `-g`, `-nowarn`, `-O`, `-verbose`)
- Use option abbreviations that are recognized on the command line for compatibility with `loadjava` (`-e`, `-h`, `-p`, `-u`, `-v`).

Notes: 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. For information, see "Option Support in the Server Embedded Translator" on page 11-16.

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 handled appropriately.)

Syntax for the various 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 `sqlj.` 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-`; only options that start with `compile.` are passed to the Java compiler. For example:

  ```
  compile.verbose
  ```

  (The Java compiler `-verbose` option outputs status messages during the compile.)

- General profile customization options (that apply regardless of the particular customizer you are using) are prefixed by `profile.` (including the period) instead of `-P-`; only options that start with `profile.` are passed to the profile customizer. For example:

  ```
  profile.backup
  ```

  (The profile customizer `backup` option saves a copy of the previous profile.)

  You can also specify options to a particular customizer by using `profile.C` as follows:

  ```
  profile.Csummary
  ```

  (For the Oracle customizer, the `summary` option displays a summary of the Oracle features used.)

  Any profile customization other than the default Oracle customization is considered an advanced feature and is covered in Chapter 10, "Profiles and Customization".

- 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/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: Always use the equals sign (=) in your option settings in a properties file, even though some options (such as `-user`, `-password`, and `-url`) allow use of a space instead of "=" on the command line.

Properties File: Simple Example The following are sample properties file entries:

```
# Set user and JDBC driver
sqlj.user=scott
sqlj.driver=oracle.jdbc.driver.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.driver.OracleDriver -C-verbose
```

Properties File: Non-Default Connection Context Classes Following is a sample properties file that specifies settings for a connection context class that you declared:

```
# JDBC driver
sqlj.driver=oracle.jdbc.driver.OracleDriver

# Oracle 8.0.4 on spock.natdecsys.com
sqlj.user@SourceContext=sde
sqlj.password@SourceContext=fornow
sqlj.url@SourceContext=jdbc:oracle:thin:@207.67.155.3:1521:nds

# Warning settings
sqlj.warn=all

# Cache
sqlj.cache=on
```

Default Properties Files

Regardless of whether a properties file is specified in 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.

Also see "Order of Precedence of Option Settings" on page 8-18.

SQLJ_OPTIONS Environment Variable for Option Settings

Oracle SQLJ supports use of 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 Java VM. And it is particularly useful for command-line-only options, such as `-classpath`, that you may 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: How to set environment variables is specific to your operating system. There may also be OS-specific restrictions. For example, in Windows 95 you use the `Environment` tab in the `System` control panel. Additionally, since Windows 95 does not support the "=" character in variable settings, SQLJ supports the use of "#" instead of "=" in setting `SQLJ_OPTIONS`. Consult your operating system documentation.

Order of Precedence of Option Settings

SQLJ takes option settings in the following order. At each step, it overrides any previous settings for any given option.

1. Sets options to default settings (where applicable).
2. Looks for a `sqlj.properties` file in the Java home directory; if one is found, options are set as specified there.
3. Looks for a `sqlj.properties` file in the user home directory; if one is found, options are set as specified there.
4. Looks for a `sqlj.properties` file in the current directory; if one is found, options are set as specified there.
5. Looks for option settings in the `SQLJ_OPTIONS` environment variable and effectively prepends them to the beginning of the command line. Sets options as specified in `SQLJ_OPTIONS`.
6. Looks for option settings on the command line; options are set as specified there. While SQLJ processes the command line, it looks in any file specified by the `-props` option and sets options as specified there.

Notes:

- In `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 inserts the file's option settings into the position on the command line where the `-props` option was specified.
- Options on the command line, with options from a `-props` file inserted, are read 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 allow you to run in a fairly standard mode of operation. For options that can also be specified in a properties file (such as `sqlj.properties`), that syntax is noted as well (see "Properties Files for Option Settings" on page 8-13).

More advanced command-line flags and options are discussed in "Advanced Translator Options" on page 8-47 and "Translator Support and Options for Alternative Environments" on page 8-62.

Basic Options for Command Line Only

The following basic options can be specified only on the SQLJ command line or, equivalently, in the `SQLJ_OPTIONS` environment variable. They cannot be specified in properties files.

- `-props`
- `-classpath`
- `-help`, `-help-long`, `-help-alias`, `-P-help`, `-C-help`
- `-version`, `-version-long`
- `-n`

The command-line-only flags (the `-help` flags, `-version` flags, and `-n`) do *not* support `=true` syntax. Enable them by typing only the flag name, as in the following example:

supported: `sqlj -version-long ...`

not supported: `sqlj -version-long=true ...`

Note: There are also more 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 (an alternative to specifying option settings on the command line).

See "Properties Files for Option Settings" on page 8-13 for information about the format of these files, the details of how they are used in relation to command-line options, and where SQLJ looks for default properties files.

Command-line syntax `-props=`*`filename`*

Command-line example `-props=myprops.properties`

Properties file syntax n/a

Properties file example n/a

Default value none

CLASSPATH for Java VM and Compiler (-classpath)

For compatibility with the syntax of most Java VMs 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 VMs or compilers, or "=", as with other SQLJ options. This is shown in the following examples (both on Solaris):

```
-classpath=.:./classes:/vobs/dbjava/classes/classes111.zip:/jdbc-1.2.zip
```

or:

```
-classpath .:./classes:/vobs/dbjava/classes/classes111.zip:/jdbc-1.2.zip
```

The `-classpath` option sets the Java CLASSPATH for both the Java VM and the Java compiler. If you do not want to use the same CLASSPATH for both, set them separately using the SQLJ `-J` and `-C` prefixes, described in "Prefixes that Pass Option Settings to Other Executables" on page 8-47.

> **Note:** As with other options described later in this chapter, if you do use "=" in setting the `-classpath` option, then it is stripped out when the option string is passed to the Java VM and compiler. This is because VMs and compilers do not support the "=" syntax in their option settings.

Command-line syntax `sqlj -classpath=<`*`class_path`*`>`

Command-line example `sqlj -classpath=/jdbc-1.2.zip:/classes/bin`

Properties file syntax n/a

Properties file example n/a

Default value none

SQLJ Option Information (-help)

The following three 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
```

or:

```
sqlj -help-long
```

or:

```
sqlj -help-alias
```

No input-file translation is done 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. It is assumed 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. These prefixes are discussed in "Prefixes that Pass Option Settings to Other Executables" on page 8-47. As with the `-help` flag, no translation is done if you request customizer or compiler help.

```
sqlj -P-help
```

```
sqlj -C-help
```

Like 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.

Notes:

- For compatibility with the `loadjava` utility, `-h` is recognized as equivalent to `-help` when specified on the command line. See "Options for loadjava Compatibility" on page 8-8.
- Multiple `-help` flag settings can be used 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 command-line-only.
- Help is also provided if you run SQLJ without specifying any files to process. This is equivalent to using the `-help` setting.

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 This 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 `boolean`, `integer`, or `String`)
- description
- current value
- how the current value was set (from command line, from 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-long` mode displays current settings of all options.)

The -help-alias Setting This setting provides a synopsis of the command-line abbreviations that are supported for compatibility with the `loadjava` utility.

Command-line syntax `sqlj` *`help_flag_settings`*

Command-line examples

```
sqlj -help
sqlj -help -help-alias
sqlj -help-long
sqlj -warn=none,null -help-long
sqlj -help-alias
```

Properties file syntax n/a

Properties file example n/a

Default value none

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
```

or:

```
sqlj -version-long
```

No input-file translation is done when you use the `-version` option, even if you include file names and other options on the command line. It is assumed 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.

Like other command-line-only flags `-version` does *not* support `=true` syntax. Enable it by typing only the flag name.

The -version Setting The `-version` setting displays the SQLJ release number, such as "Oracle SQLJ 8.1.5".

The -version-long Setting The `-version-long` setting displays SQLJ release and build version information and the JDBC driver release number if one can be found. For example, if an Oracle JDBC driver is used, this option would display something like "Oracle JDBC version 8.1 (8.1.5.0.0)".

Command-line syntax `sqlj` *`version_flag_settings`*

Command-line example

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

Properties file syntax n/a

Properties file example n/a

Default value none

Echo Command Line 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 Java VM 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 showing you 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.

Like other command-line-only flags `-n` does *not* support `=true` syntax. Enable it by typing only the flag name.

Consider the following 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
```

(This is all one wrap-around line.)

Note:

- As an alternative to `-n`, you can use `-vm=echo`
- 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`. See "SQLJ Option Information (-help)" on page 8-22.

Command-line syntax `-n`

Command-line example `-n`

Properties file syntax n/a

Properties file example n/a

Default value `false`

Options for Output Files and Directories

The following option specifies encoding for SQLJ input and output source files:

- `-encoding`

These options specify where SQLJ output files are placed:

- `-d`
- `-dir`

Encoding for Input and Output Source Files (-encoding)

The `-encoding` option specifies NLS encoding to be applied to `.sqlj` and `.java` input files and `.java` generated files. For compatibility with `javac`, you can use either a space or "=" in setting this option on the command line, as in the following examples:

```
-encoding=SJIS

-encoding SJIS
```

If setting `-encoding` in a properties file, however, 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.

Notes:

- As with the `-classpath` and `-d` options described below, if you do use an "=" in setting the `-encoding` option, then it is stripped out when the option string is passed to the Java VM and compiler. This is because VMs 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. See "Options for loadjava Compatibility" on page 8-8.

Command-line syntax `-encoding=`*`Java_character_encoding`*

Command-line example `-encoding=SJIS`

Properties file syntax `sqlj.encoding=`*`Java_character_encoding`*

Properties file example `sqlj.encoding=SJIS`

Default value setting in Java VM 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 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 in using the `-d` option. (For information about `-ser2class`, see "Conversion of .ser File to .class File (-ser2class)" on page 8-54.)

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 a package `a.b.c` and specify directory `/mydir`, output files will be placed in directory `/mydir/a/b/c`.

If you specify a relative directory path, 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
```

If setting `-d` in a properties file, however, 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 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 (the second example immediately below will go up a level, then back down to a parallel directory called `paralleldir`):

```
-d=.

-d=../paralleldir
```

If the `-d` option is not specified, then files are generated under the directory where the associated `.java` source file is generated, which is according to the `-dir` option. If you specifically want the output directory to be the same as that used by `-dir` (perhaps overriding other `-d` settings, such as in properties files), then you can use the `-d` option as follows:

```
-d=
```

Notes:

- Throughout this discussion, the forward-slash (/) was used as the file separator. It is important to note, however, 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 Java VM.
- As with the `-classpath` and `-encoding` options described above, if you do use an "=" in setting the `-d` option, then it is stripped out when the option string is passed to the Java VM and compiler. This is because VMs and compilers do not support the "=" syntax in their option settings.

Command-line syntax `-d=`*`directory_path`*

Command-line example `-d=/topleveldir/mydir`

Properties file syntax `sqlj.d=`*`directory_path`*

Properties file example `sqlj.d=/topleveldir/mydir`

Default value none (use directory of generated `.java` source file, per `-dir` option)

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 a package `a.b.c` and specify directory `/mydir`, then output files will be placed in directory `/mydir/a/b/c`.

If you specify a relative directory path 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
```

If your current directory is `/root/home/mydir` and you set the `-dir` option to the relative directory path `mysubdir/myothersubdir` as follows, then `/root/home/mydir/mysubdir/myothersubdir` will be the root directory for generated `.java` files:

```
-dir=mysubdir/myothersubdir
```

You can also use standard syntax such as a period for the current directory or two periods to go up a level (the second example immediately below will go up a level, then back down to a parallel directory called `paralleldir`):

```
-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: Throughout this discussion, the forward-slash (/) was used as the file separator. Be aware, however, that in specifying this or similar options, you must use the file separator of your operating system, as specified in the `file.separator` system property of your Java VM.

Command-line syntax `-dir=`*`directory_path`*

Command-line example `-dir=/topleveldir/mydir`

Properties file syntax `sqlj.dir=`*`directory_path`*

Properties file example `sqlj.dir=/topleveldir/mydir`

Default value none (use directory of `.sqlj` source file)

Connection Options

The following options are used for the database connection for online semantics-checking:

- `-user`
- `-password`
- `-url`
- `-default-url-prefix`
- `-driver`

There is no requirement that the SQLJ translator connect to the same database or schema as the application does at runtime. The connection information in application source code can be independent of the connection information in the SQLJ options.

A situation where you will probably want to use a different connection for translation than for runtime is if you are developing in a different environment than the one to which you will deploy.

Online Semantics-Checking and Username (-user)

Simple semantics-checking not involving a database connection is referred to as *offline checking*. The more thorough semantics-checking requiring a database connection is referred to as *online checking*. Online checking offers one of the prime advantages of the SQLJ strong-typing paradigm—type incompatibilities that would normally result in runtime SQL exceptions are caught during translation before users ever run the application.

The `-user` option enables online semantics-checking and specifies the username (schema name) for the database schema used for 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 it or disables it according to the presence or absence of the `-user` option.

Discussion of the `-user` option is split into two categories—1) effect of `-user` when you are using the default connection context class only; and 2) effect of `-user` when you are using non-default or multiple connection context classes. Non-default connection context classes are discussed in "Connection Contexts" on page 7-2.

General discussion of connection considerations, such as when to use multiple instances of the `DefaultContext` class and when to declare additional connection context classes, is in "Connection Considerations" on page 4-9.

Notes:

- For compatibility with the `loadjava` utility, `-u` is recognized as equivalent to `-user` when specified on the command line. See "Options for loadjava Compatibility" on page 8-8.
- Usernames cannot contain the characters "/" or "@".
- You are allowed to use a space instead of "=" in a username 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.

Effect of -user When Using Default Connection Context Class Only 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 of your SQLJ executable statements. This example results in online checking against the `scott` schema.

You can also specify the password or URL or both along with the username, using syntax as in the following examples (with "/" preceding the password and "@" preceding the URL):

```
-user=scott/tiger
```

or:

```
-user=scott@jdbc:oracle:oci8:@
```

or:

```
-user=scott/tiger@jdbc:oracle:oci8:@
```

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:

```
-user=
```

Again, when you are using only the default connection or other instances of the `DefaultContext` class, this will apply to all of 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 username, `URL.CONNECT`, which you can use when the URL specifies the user and password as well as the other details of the database connection. To see what the URL would look like in such a case, see "Connection URL for Online Semantics-Checking (-url)" on page 8-37.

Effect of -user When Using Non-Default or Multiple Connection Context Classes 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 username for online checking against a particular connection context class 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 object that is an 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:oci8:@
```

The `CtxClass` connection context class must be declared in your source code or previously compiled into a `.class` file. (See "Connection Contexts" on page 7-2 for more information.)

Employ the `-user` option separately for each connection context username you want to specify; these settings have no influence on each other:

```
-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 kinds of schemas.

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 username for that connection context. For example, after semantics checking is enabled, the following setting disables it for connection context `CtxClass2`:

```
-user@CtxClass2=
```

This disables online semantics-checking for any SQLJ executable statements that specify a connection object that is an instance of `CtxClass2`.

To disable online semantics-checking for the default connection context class and any other connection context classes for which you do not specify a username:

```
-user=
```

Command-line syntax `-user`*`<@conn_context_class>=username</password><@url>`*

Command-line examples

```
-user=scott
-user=scott/tiger
-user=scott@jdbc:oracle:oci8:@
-user=scott/tiger@jdbc:oracle:oci8:@
```

```
-user=
-user=URL.CONNECT
-user@Context=scott/tiger
-user@Context=
```

Properties file syntax `sqlj.user`*`<@conn _ocntext_class>=username</password><@url>`*

Properties file examples

```
sqlj.user=scott
sqlj.user=scott/tiger
sqlj.user=scott@jdbc:oracle:oci8:@
sqlj.user=scott/tiger@jdbc:oracle:oci8:@
sqlj.user=
sqlj.user=URL.CONNECT
sqlj.user@CtxClass=scott/tiger
sqlj.user@CtxClass=
```

Default value none (no online semantics-checking)

User Password for Online Semantics-Checking (-password)

The `-password` option specifies the user password to use 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. (See "Online Semantics-Checking and Username (-user)" on page 8-31.) 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`, 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 employ the `-user` option to specify additional 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 that was set 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 username `URL.CONNECT` is used, as discussed in "Online Semantics-Checking and Username (-user)" on page 8-31. In this case, username 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=
```

or:

```
-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
```

or:

```
-password@CtxClass2=EMPTY.PASSWORD
```

Oracle, however, does not permit an empty password.

Notes:

- `-p` is recognized as equivalent to `-password` when specified on the command line.
- 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
```

Command-line syntax -password<@*conn_context_class*>=*user_password*

Command-line examples

```
-password=tiger
-password=
-password=EMPTY.PASSWORD
-password@CtxClass=tiger
```

Properties file syntax `sqlj.password`<@*conn_context_class*>=*user_password*

Properties file examples

```
sqlj.password=tiger
sqlj.password=
sqlj.password=EMPTY.PASSWORD
sqlj.password@CtxClass=tiger
```

Default value none

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 Oracle database SID.

You can also specify the URL as part of the `-user` option setting. (See "Online Semantics-Checking and Username (-user)" on page 8-31.) 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`, the following example would set the URL to use for the database connection for checking all of your SQLJ statements:

```
-url=jdbc:oracle:oci8:@
```

Or, to include host name, port number, and SID:

```
-url=jdbc:oracle:thin:@hostname:1521:orcl
```

If you do not begin a URL setting with `jdbc:` then the setting is assumed to be of the form *`host`*`:`*`port`*`:`*`sid`* and by default is automatically prefixed with the following:

```
jdbc:oracle:thin:@
```

For a `-url` setting of `localhost:1521:orcl`, for example, this results in a URL of `jdbc:oracle:thin:@localhost:1521:orcl`.

You can remove or alter this default prefix with the `-default-url-prefix` option. See "Default URL Prefix (-default-url-prefix)" on page 8-39 for more information.

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:oci8:scott/tiger@ -user=URL.CONNECT
```

If you declare and use additional connection context classes, `CtxClass1` for example, you will presumably specify additional 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:oci8:@
```

Any connection context class without a URL setting, either through the `-url` setting or the `-user` setting, uses the URL that was set for the default connection context class, presuming a URL has been set for the default context class.

Notes:

- Remember that any connection context class with a URL setting must also have a username 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:oci8:@
-url@CtxClass jdbc:oracle:oci8:@
```

Command-line syntax `-url<@`*`conn_context_class`*`>=`*`URL`*

Command-line examples

```
-url=jdbc:oracle:oci8:@
-url=jdbc:oracle:thin:@hostname:1521:orcl
-url=jdbc:oracle:oci8:scott/tiger@
-url=hostname:1521:orcl
-url@CtxClass=jdbc:oracle:oci8:@
```

Properties file syntax `sqlj.url<@`*`conn_context_class`*`>=`*`URL`*

Properties file examples

```
sqlj.url=jdbc:oracle:oci8:@
sqlj.url=jdbc:oracle:thin:@hostname:1521:orcl
sqlj.url=jdbc:oracle:oci8:scott/tiger@
sqlj.url=hostname:1521:orcl
sqlj.url@CtxClass=jdbc:oracle:oci8:@
```

Default value `jdbc:oracle:oci8:@`

Default URL Prefix (-default-url-prefix)

The following is the default prefix for any URL setting you specify that does not already start with `jdbc:`

```
jdbc:oracle:thin:@
```

This allows you to use a shorthand in specifying a URL setting, either in the `-user` option or the `-url` option—you can specify only the host, port, and SID of the database. As an example, presume you set a URL as follows:

```
-url=myhost:1521:orcl
```

or:

```
-user=scott/tiger@myhost:1521:orcl
```

By default, the URL will be interpreted to be the following:

```
jdbc:oracle:thin:@myhost:1521:orcl
```

If you specify a full URL that starts with `jdbc:` then the default prefix will not be used, such as in the following example:

```
-url=jdbc:oracle:oci8:@orcl
```

Use the `-default-url-prefix` option to alter or remove the default prefix. For example, if you want your URL settings to default to the OCI 8 driver instead of the Thin driver, then set the default prefix as follows:

```
-default-url-prefix=jdbc:oracle:oci8:@
```

With this prefix, a setting of `-url=orcl` is equivalent to the `-url=jdbc:oracle:oci8:@orcl` setting above.

If you do not want any prefix, then set the `-default-url-prefix` option to an empty string, as follows:

```
-default-url-prefix=
```

Command-line syntax `-default-url-prefix=`*`url_prefix`*

Command-line examples

```
-default-url-prefix=jdbc:oracle:oci8:@
-default-url-prefix=
```

Properties file syntax `sqlj.default-url-prefix=`*`url_prefix`*

Properties file examples

```
sqlj.default-url-prefix=jdbc:oracle:oci8:@
sqlj.default-url-prefix=
```

Default value `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-separated list of classes.

The default, `OracleDriver`, supports the Oracle OCI 8, Thin, and server-side JDBC drivers for use with Oracle databases.

Command-line syntax `-driver=`*driver1*`<,`*driver2*`,`*driver3*`,...>`

Command-line examples

```
-driver=oracle.jdbc.driver.OracleDriver
-driver=oracle.jdbc.driver.OracleDriver,sun.jdbc.odbc.JdbcOdbcDriver
```

Properties file syntax `sqlj.driver=`*driver1*`<,`*driver2*`,`*driver3*`,...>`

Properties file examples

```
sqlj.driver=oracle.jdbc.driver.OracleDriver
sqlj.driver=oracle.jdbc.driver.OracleDriver,sun.jdbc.odbc.JdbcOdbcDriver
```

Default value `oracle.jdbc.driver.OracleDriver`

Reporting and Line-Mapping Options

The following options specify what kinds 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 option enables line-mapping from the `.java` output file back to the `.sqlj` source file, so that you can trace compilation and execution errors back to the appropriate location in your original source code.

- `-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 of the flags for this option must be combined into a single, comma-separated string.

Table 8-2 documents the conditions that can be tested, what a `true` flag value means, what the `true` and `false` flag values are for each condition, and which value is the default.

Table 8–2 Tests and Flags for SQLJ Warnings

Tests and Flag Functions	TRUE/FALSE Values
Data precision test—Enabling `precision` warns 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—Enabling `nulls` warns 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—Enabling `portable` checks SQLJ clauses for portability and warns if there are non-portable clauses. (Where *non-portable* 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—Enabling `strict` instructs 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—Enabling `verbose` provides additional informational messages about the translation process (such as what database connections were made for online checking).	`verbose` `noverbose` (default)
Enable or disable all warnings.	`all` `none`

The `verbose/noverbose` flag works differently than 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. The `-warn=verbose` flag simply results in additional reporting after the translation and 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 make sure all flags are off before you turn selected flags on, or vice versa. Essentially, `all` is equivalent to `precision, nulls, portable, strict, verbose` and `none` is equivalent to `noprecision, nonulls, noportable, nostrict, noverbose`. There is no default for `all/none`; there are only defaults for individual flags.

For example, use the following sequence to make sure only the `nulls` flag is on:

```
-warn=none,nulls
```

And the following sequence will have the same result, because the `verbose` setting will be overridden:

```
-warn=verbose,none,nulls
```

Or use the following to make sure everything except the portability flag is on:

```
-warn=all,noportable
```

And the following 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 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 precision, nullability, and strictness tests are part of online semantics-checking and require a database connection.

Command-line syntax `-warn=`*`comma-separated_list_of_flags`*

Command-line example `-warn=none,nulls,precision`

Properties file syntax `sqlj.warn=`*`comma-separated_list_of_flags`*

Properties file example `sqlj.warn=none,nulls,precision`

Default values `precision,nulls,noportable,strict,noverbose`

Real-Time Status Messages (-status)

The `-status` flag instructs SQLJ to output additional status messages throughout all aspects of the SQLJ process—translation, semantics-checking, compilation, and customization. Messages are output as each file is processed and at each stage of SQLJ operation.

Notes:

- 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 after the translation and about the translation phase only.
- For compatibility with the `loadjava` utility, `-v` is recognized as equivalent to `-status` when specified on the command line. See "Options for loadjava Compatibility" on page 8-8.

Command-line syntax `-status=`*`true/false`*

Command-line example `-status=true`

Properties file syntax `sqlj.status=`*`true/false`*

Properties file example `sqlj.status=false`

Default value `false`

Cause and Action with 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).

This is the same information that is provided in "Translation Time Messages" on page B-2.

Command-line syntax `-explain=`*`true/false`*

Command-line example `-explain=true`

Properties file syntax `sqlj.explain=`*`true/false`*

Properties file example `sqlj.explain=false`

Default value `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 of this, when Java runtime errors occur, the line number reported by the Java VM is the line number in the SQLJ source code, making it much easier to debug.

Normally, 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 doing 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

Notes:

- If you are processing a `.sqlj` file and the compilation step is skipped due to error, then no line-mapping can be done either, because no `.class` file is available for mapping.
- When the Java compiler is invoked from SQLJ (as is typical), 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.

Command-line syntax `-linemap=`*`true/false`*

Command-line example `-linemap=true`

Properties file syntax `sqlj.linemap=`*`true/false`*

Properties file example `sqlj.linemap=false`

Default value `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 employed to pass options to the Java VM, Java compiler, or SQLJ profile customizer. These options allow you to exercise any of the specialized features of Oracle SQLJ. For options that can also be specified in a properties file (such as `sqlj.properties`), that syntax is noted as well (see "Properties Files for Option Settings" on page 8-13 for more information).

Additional advanced options, intended specifically for situations where you are using alternative Java environments, are discussed in "Translator Support and Options for Alternative Environments" on page 8-62. More basic command line flags and options are discussed in "Basic Translator Options" on page 8-20.

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 Java interpreter)
- `-C` (mark options for Java compiler)
- `-P` (mark options for profile customizer)

Options to Pass to Java VM (-J)

The `-J` prefix, specified on the command line, marks options to be passed to the Java VM from which SQLJ was invoked. This prefix immediately precedes a Java VM option, with no spaces in between. After stripping off the `-J` prefix, the `sqlj` script passes the Java option to the VM.

For example:

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

After stripping the `-J` prefix, the `sqlj` script passes the `-Duser.language` argument as is to the VM. In the JDK, the flag `-Duser.language=ja` sets the system property `user.language` 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 Java VM from a properties file because properties files are read after the VM is invoked.

Note: It is not possible to use a properties file to pass options directly to the Java VM in which the SQLJ translator runs. It is possible, however, to use the `SQLJ_OPTIONS` environment variable for this purpose. See "SQLJ_OPTIONS Environment Variable for Option Settings" on page 8-17. It is also possible (if applicable) to use a properties file to pass options to the VM in which the Java compiler runs. See "Options to Pass to Java Compiler (-C)" on page 8-48 for information.

Command-line syntax `-J-`*`Java_option`*

Command-line example `-J-Duser.language=ja`

Properties file syntax n/a

Properties file example n/a

Default value n/a

Options to Pass to Java Compiler (-C)

The `-C` prefix marks options to pass to the Java compiler that is 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 (typically, but not necessarily, `javac`).

For example:

```
-C-nowarn
```

After stripping the `-C` prefix, the `sqlj` script passes the `-nowarn` argument as is to the compiler. The `-nowarn` flag is a `javac` option to suppress warning messages during compilation.

One Java compiler option, `-classpath`, is slightly modified when it is passed to the compiler. All other compiler options are passed without change. (Note that if you want the same `CLASSPATH` setting for the VM and compiler, then you can use the SQLJ `-classpath` option instead of `-J-classpath` and `-C-classpath`.)

Specify the `CLASSPATH` setting to the Java compiler using the following syntax:

```
-C-classpath=path
```

For example:

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

The equals sign is necessary for SQLJ parsing but is automatically replaced with a space when the option is passed to the Java compiler. After the `-C` is stripped off and the equals sign is replaced, the option is passed to the compiler as follows:

```
-classpath /user/jdk/bin
```

If the Java compiler runs in its own Java VM, then you can pass options to that VM through the compiler. This is accomplished by prefixing the VM option with `-C-J` with no spaces between this prefix combination and the VM option.

For example:

```
-C-J-Duser.language=de
```

Observe the following 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. (For information about the `-encoding` option, see "Encoding for Input and Output Source Files (-encoding)" on page 8-27.)
- 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 (`.ser`) files and is also passed to the Java compiler. This will ensure that `.class` files and `.ser` files are in the same directory. (For information about the `-d` option, see "Output Directory for Generated .ser and .class Files (-d)" on page 8-28.)

Notes:

- In the above `-classpath` discussion, the forward-slash (/) was used as the file separator. Be aware, however, that in specifying this or similar options you must use the file separator of your operating system, as specified in the `file.separator` system property of your Java VM.
- 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 a help option) 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 generally assumes that you want help or you want translation, but not both.)

Command-line syntax `-C-`*`Java_compiler_option`*

Command-line example `-C-nowarn`

Properties file syntax `compile.`*`Java_compiler_option`*

Properties file example `compile.nowarn`

Default value n/a

Options to Pass to Profile Customizer (-P)

During the customization phase, 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 the `-P` prefix by itself to pass generic options to the customizer harness that apply regardless of the customizer.
- Use the `-P-C` prefix 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 generic option:

```
-P-backup
```

The `-backup` flag is a generic customizer option to backup the previous customization before generating a new one.

Here 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.

Notes:

- Note that 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 generally assumes that you want help or you want translation, but not both.)
- 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.

For information about available generic and Oracle-specific customizer options, see "Customization Options and Choosing a Customizer" on page 10-11.

Command-line syntax `-P-<C>`*`profile_customizer_option`*

Command-line examples

```
-P-driver=oracle.jdbc.driver.OracleDriver
-P-Csummary
```

Properties file syntax `profile.<C>`*`profile_customizer_option`*

Properties file example

```
profile.driver=oracle.jdbc.driver.OracleDriver
profile.Csummary
```

Default value n/a

Flags for Special Processing

As mentioned above, `.sqlj` files are typically processed by the SQLJ translator, Java compiler, and SQLJ profile customizer. The following flags limit this processing, directing the SQLJ startup script to skip the indicated process:

- `-compile`
- `-profile`

In addition, the following flag directs special processing of the SQLJ profile after customization:

- `-ser2class`

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.

Notes:

- 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.
- There are situations where it is sensible for `-compile` to be set to `false` even when `.java` files are specified on the command line. You might do this, for example, if you are translating a `.sqlj` file and need to specify one or more `.java` files for type resolution during translation but want to compile all of your `.java` files later using a particular compiler. (An example of a situation where you must supply `.java` files for type resolution is if you are accessing Oracle8*i* objects in your SQLJ application and using the Oracle JPublisher utility to map these objects to custom Java types. The `.java` files produced by JPublisher must be supplied to the SQLJ translator for type resolution during translation. See "Compiling Custom Java Classes" on page 6-10 for more information.)

Command-line syntax `-compile=`*`true/false`*

Command-line example `-compile=false`

Properties file syntax `sqlj.compile=`*`true/false`*

Properties file example `sqlj.compile=false`

Default value `true` (enabled)

Profile Customization Flag (-profile)

The `-profile` flag enables or disables processing of generated profile (`.ser`) files by the SQLJ profile customizer. However, this only applies 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 behaves 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 there are other operations, such as line mapping, that you might want to perform 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: 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.

Command-line syntax -profile=*true/false*

Command-line example -profile=false

Properties file syntax sqlj.profile=*true/false*

Properties file example sqlj.profile=false

Default value true (enabled)

Conversion of .ser File to .class File (-ser2class)

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 is true of Netscape Navigator 4.x, for example.)

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. (This is discussed in "Naming of Class and Resource Schema Objects" on page 11-8.)

The conversion is performed after profile customization so that it includes your customizations.

The base names of converted files are the same as for the original files; the only difference in the file name is `.ser` being replaced by `.class`. For example:

```
Foo_SJProfile0.ser
```

is converted to:

```
Foo_SJProfile0.class
```

Notes:

- 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 rerun SQLJ and recreate the profile.
- Where encoding is necessary, the `-ser2class` option always uses `8859_1` encoding, ignoring the SQLJ `-encoding` setting.

Command-line syntax `-ser2class=`*`true/false`*

Command-line example `-ser2class=true`

Properties file syntax `sqlj.ser2class=`*`true/false`*

Properties file example `sqlj.ser2class=false`

Default value `false`

Semantics-Checking Options

The following options specify characteristics of online and offline semantics-checking:

- `-offline`
- `-online`
- `-cache`

Discussion of these options is preceded by a discussion of `OracleChecker`—the default front-end class for semantics-checking—and an introduction to the Oracle semantics-checkers.

Semantics-Checkers and the OracleChecker Front End (default checker)

The default checker (for both online and offline checking) is `oracle.sqlj.checker.OracleChecker`. 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 databases and JDBC drivers, there are the following categories of checkers (for both online and offline checking):

- Oracle8 checkers for Oracle8*i* types
- Oracle80 checkers for Oracle 8.0.5 types
- Oracle7 checkers for Oracle 7.3.4 types
- Oracle8To7 checkers for using an Oracle8*i* JDBC driver, but only with the subset of types that are compatible with an Oracle 7.3.4 database.

The Oracle80 and Oracle7 checkers are incompatible with the Oracle8*i* JDBC drivers, and the Oracle8 and Oracle8To7 checkers are incompatible with the Oracle 8.0.5 and Oracle 7.3.4 JDBC drivers. The Oracle8To7 checkers were created so that there is a way to use an Oracle8*i* JDBC driver and check against an Oracle 7.3.4 subset of types.

Online Checking with Oracle Database and JDBC Driver If you are using an 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 8-3 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 8–3 Oracle Online Semantics-Checkers Chosen by OracleChecker

Database Version	JDBC Version	Online Checker	Other Legal Checkers
Oracle8*i* or 8.0.5	Oracle8*i*	Oracle8JdbcChecker	Oracle8To7JdbcChecker
Oracle8*i* or 8.0.5	Oracle 8.0.5	Oracle80JdbcChecker	Oracle7JdbcChecker
Oracle8*i* or 8.0.5	Oracle 7.3.4	Oracle7JdbcChecker	none
Oracle 7.3.4	Oracle8*i*	Oracle8To7JdbcChecker	none

Table 8–3 Oracle Online Semantics-Checkers Chosen by OracleChecker (Cont.)

Database Version	JDBC Version	Online Checker	Other Legal Checkers
Oracle 7.3.4	Oracle 8.0.5	Oracle7JdbcChecker	none
Oracle 7.3.4	Oracle 7.3.4	Oracle7JdbcChecker	none

Offline Checking with Oracle JDBC Driver If you are using an Oracle JDBC driver with offline checking, then `OracleChecker` will choose a checker based on your JDBC driver version. Table 8-4 summarizes the possible choices. (Note that there is an `Oracle8To7OfflineChecker` but it can only be used by selecting it manually.)

Table 8–4 Oracle Offline Semantics-Checkers Chosen by OracleChecker

JDBC Version	Offline Checker	Other Legal Checkers
Oracle8*i*	Oracle8OfflineChecker	Oracle8To7OfflineChecker
Oracle 8.0.5	Oracle80OfflineChecker	Oracle7OfflineChecker
Oracle 7.3.4	Oracle7OfflineChecker	none

Not Using Oracle Database and JDBC Driver If `OracleChecker` detects that you do not use an Oracle JDBC driver, then it runs one of the following checkers:

- `sqlj.semantics.OfflineChecker` if online checking is not enabled
- `sqlj.semantics.JdbcChecker` if online checking is enabled

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 connection to the database—only SQL syntax and usage of Java types is checked. (For information about what offline and online semantics-checkers accomplish and how they function, see "Semantics-Checking" on page 9-2.)

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 discussed in "Semantics-Checkers and the OracleChecker Front End (default checker)" on page 8-56, will serve your needs unless you want to specify a particular checker which would not be chosen by `OracleChecker`. For example, you may run offline checking on a machine with an Oracle 8.0 JDBC driver but your application (or at least statements using a particular connection context class) will run against an Oracle 7.3 database. In this case you will want to check these statements using the Oracle7 checker.

The following example shows how to select the Oracle7 offline checker for a particular connection context (`CtxClass`):

```
-offline@CtxClass=oracle.sqlj.checker.Oracle7OfflineChecker
```

This results in SQLJ using `oracle.sqlj.checker.Oracle7OfflineChecker` for offline checking of any of your SQLJ executable statements that specify a connection object that is an instance of `CtxClass`.

The `CtxClass` connection context class must be declared in your source code or previously compiled into a `.class` file. (See "Connection Contexts" on page 7-2 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.Oracle7OfflineChecker
-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.Oracle7OfflineChecker
```

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.

Command-line syntax `-offline<@`*`conn_context_class`*`>=`*`checker_class`*

Command-line examples

```
-offline=oracle.sqlj.checker.Oracle80OfflineChecker
-offline@Context=oracle.sqlj.checker.Oracle80OfflineChecker
```

Properties file syntax `sqlj.offline<@`*`conn_context_class`*`>=`*`checker_class`*

Properties file examples

```
sqlj.offline=oracle.sqlj.checker.Oracle80OfflineChecker
sqlj.offline@Context=oracle.sqlj.checker.Oracle80OfflineChecker
```

Default value `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.

Remember 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. (For information about what offline and online semantics-checkers accomplish and how they function, see "Semantics-Checking" on page 9-2.)

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 that was established for online checking. (At analysis time, a connection is passed to each online checker, and the checker reports whether or not the database is recognized.)

The default `OracleChecker`, a front-end class discussed in "Semantics-Checkers and the OracleChecker Front End (default checker)" on page 8-56, will serve your needs unless you want to specify a particular checker that would not be chosen by `OracleChecker`. For example, you may run online checking on a machine with an Oracle 8.0 database and JDBC driver, but your application (or at least statements using a particular connection context class) will eventually run against an Oracle 7.3 database. In this case you will want to check these statements using the Oracle7 checker.

The following example shows how to select the Oracle7 online checker for the `DefaultContext` class (and any other connection context classes without a specified setting):

```
-online=oracle.sqlj.checker.Oracle7JdbcChecker
```

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.Oracle7JdbcChecker,sqlj.semantics.JdbcChecker
```

With this specification, if connection is made to an Oracle database, then SQLJ uses `oracle.sqlj.checker.Oracle7JdbcChecker`. If connection is made to any other kind of database, then SQLJ uses the generic `JdbcChecker`. This is similar functionally to what the default `OracleChecker` does but ensures that you use an Oracle7 checker instead of an Oracle8 checker if you connect to an Oracle database.

To specify the online checker for a particular connection context (`CtxClass`):

```
-online@CtxClass=oracle.sqlj.checker.Oracle7JdbcChecker
```

This results in the use of `oracle.sqlj.checker.Oracle7JdbcChecker` 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. (See "Connection Contexts" on page 7-2 for more information.)

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.Oracle80JdbcChecker
-online@CtxClass3=sqlj.semantics.JdbcChecker
```

Any connection context without an online checker setting uses the online checker setting of the default connection context, presuming you set an online checker for the default context.

Command-line syntax `-online<@`*`conn_context_class`*`>=`*`checker_class(list)`*

Command-line examples

```
-online=oracle.sqlj.checker.Oracle80JdbcChecker
-online=oracle.sqlj.checker.Oracle80JdbcChecker,sqlj.semantics.JdbcChecker
-online@Context=oracle.sqlj.checker.Oracle80JdbcChecker
```

Properties file syntax `sqlj.online<@`*`conn_context_class`*`>=`*`checker_class(list)`*

Properties file examples

```
sqlj.online=oracle.sqlj.checker.Oracle80JdbcChecker
sqlj.online=oracle.sqlj.checker.Oracle80JdbcChecker,sqlj.semantics.JdbcChecker
sqlj.online@Context=oracle.sqlj.checker.Oracle80JdbcChecker
```

Default value `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. Remove the `SQLChecker.cache` file to empty the cache.

Command-line syntax `-cache=`*`true/false`*

Command-line example `-cache=true`

Properties file syntax `sqlj.cache=`*`true/false`*

Properties file example `sqlj.cache=false`

Default value `false`

Translator Support and Options for Alternative Environments

By default, SQLJ is configured to run under the Sun Microsystems JDK 1.1.x (or later) and to use the Sun compiler `javac`. These are not requirements, however. You can configure SQLJ to work with alternative VMs or compilers. To do so, you must supply SQLJ with the following information:

- the name of the Java VM to use (`-vm` option)
- the name of the Java compiler to use (`-compiler-executable` option)
- any settings that the compiler requires

There is a set of SQLJ options that allow you to provide this information. These options are described in "Java and Compiler Options" on page 8-63.

SQLJ also defaults to the Oracle profile customizer but can work with alternative customizers as well. See "Customization Options" on page 8-69 for how to instruct SQLJ to use a different customizer.

Other SQLJ advanced flags and options are discussed in "Advanced Translator Options" on page 8-47. More basic flags and options are discussed in "Basic Translator Options" on page 8-20.

Note: With any operating system and environment you use, be aware of the limitations of the operating system and environment. In particular, the complete, expanded SQLJ command line must not exceed the maximum command-line size (for example, 250 characters for Windows 95 and 4000 characters for Windows NT). Consult your operating system documentation.

Required Compiler Behavior

If you use a Java compiler other than `javac`, then the following is required:

- The compiler must return a non-zero exit code to the operating system whenever a compilation error occurs.
- The line information that the compiler provides in any errors or messages must be in one of the following two formats (items in <> brackets are optional):
    - Sun `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.

Java and Compiler Options

The following options relate to the operation of the Java VM and Java compiler:

- `-vm` (specify the Java VM; command-line only)
- `-compiler-executable` (specify the Java compiler)
- `-compiler-encoding-flag`
- `-compiler-output-file`
- `-compiler-pipe-output-flag`

For some VM and compiler configurations, there may be problems with the way SQLJ normally invokes the compiler. The following option can be used to alleviate this:

- `-passes`

You can also pass options directly to the particular VM or compiler you use, through the `-J` and `-C` prefixes discussed in "Prefixes that Pass Option Settings to Other Executables" on page 8-47.

Note: The `-vm` 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. See "SQLJ_OPTIONS Environment Variable for Option Settings" on page 8-17.

Name of Java VM (-vm)

Use the `-vm` option if you want to specify a particular Java VM for SQLJ to use. Otherwise SQLJ uses the standard `java` from the Sun Microsystems JDK.

You must specify this command on the command line; you cannot specify it in a properties file because properties files are read after the VM is invoked.

If you do not specify a directory path along with the name of the VM executable file, then SQLJ looks for the executable according to the `PATH` setting of your operating system.

Note: Special functionality of this option, `-vm=echo`, is supported. This is equivalent to the `-n` option and instructs 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. For more information, see "Echo Command Line without Execution (-n)" on page 8-25.

Command-line syntax `-vm=`*`Java_VM_name`*

Command-line example `-vm=/myjavadir/myjavavm`

Properties file syntax n/a

Properties file example n/a

Default value `java`

Name of 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 `PATH` setting of your operating system.

The following is assumed of any Java compiler that you use:

- It can output error and status information to the standard output device (for example, `STDOUT` on a UNIX system) or, alternatively, to a file (as directed by the `-compiler-output-file` option, described below).
- It will understand the SQLJ `-d` option, which determines the root directory for class files.

Note: If you use a compiler that does not support an `-encoding` option, then disable the `-compiler-encoding-flag`, described in "Compiler Encoding Support (-compiler-encoding-flag)" on page 8-65.

Command-line syntax `-compiler-executable=`*`Java_compiler_name`*

Command-line example `-compiler-executable=/myjavadir/myjavac`

Properties file syntax `sqlj.compiler-executable=`*`Java_compiler_<path+>name`*

Properties file example `sqlj.compiler-executable=myjavac`

Default value `javac`

Compiler Encoding Support (-compiler-encoding-flag)

As mentioned in "Encoding for Input and Output Source Files (-encoding)" on page 8-27, it is typical that when you employ 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).

Command-line syntax `-compiler-encoding-flag=`*`true/false`*

Command-line example `-compiler-encoding-flag=false`

Properties file syntax `sqlj.compiler-encoding-flag=`*`true/false`*

Properties file example `sqlj.compiler-encoding-flag=false`

Default value `true`

Compiler Output File (-compiler-output-file)

If you have instructed the Java compiler to output 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 outputs to the standard output device (such as `STDOUT` on a UNIX system).

Note: You cannot use this option if you enable `-passes`, which requires output to `STDOUT`.

Command-line syntax `-compiler-output-file=`*`output_file_path+name`*

Command-line example `-compiler-output-file=/myjavadir/mycmploutput`

Properties file syntax `sqlj.compiler-output-file=`*`output_file_path+name`*

Properties file example `sqlj.compiler-output-file=/myjavadir/mycmploutput`

Default value none (standard output)

Compiler Message Output Pipe (-compiler-pipe-output-flag)

By default, the `javac` compiler provided with the Sun Microsystems JDK writes error and message output to `STDERR`. SQLJ, however, expects such compiler output to be written to `STDOUT` so that it can be captured reliably.

If SQLJ sets the `javac.pipe.output` system property to `true`, which is SQLJ's default behavior when it invokes the Java compiler, then compiler error and message output will be sent to `STDOUT`. You can specify `-compiler-pipe-output-flag=false`, however, to instruct SQLJ *not* to set this system property when it invokes the Java compiler. You should do this, for example, if the Java compiler you are using does not support the `javac.pipe.output` system property.

You can set this flag in a properties file as well as on the command line or in the `SQLJ_OPTIONS` environment variable.

Note: If you are using a Java compiler that originates from Sun Microsystems and that writes its output to `STDERR` by default, then you must leave `-compiler-pipe-output-flag` enabled if you enable `-passes`, which requires output to `STDOUT`.

Command-line syntax `-compiler-pipe-output-flag=`*`true/false`*

Command-line example `-compiler-pipe-output-flag=false`

Properties file syntax `sqlj.compiler-pipe-output-flag=`*`true/false`*

Properties file example `sqlj.compiler-pipe-output-flag=false`

Default value `true`

Run SQLJ in Two Passes (-passes)

By default, the following sequence occurs when you invoke the `sqlj` script:

1. The `sqlj` script invokes your Java VM, which runs the SQLJ translator.
2. The translator completes the semantics-checking and translation of your `.sqlj` files, outputting translated `.java` files.
3. The translator invokes your Java compiler, which compiles the `.java` files.
4. The translator processes the compiler output.
5. The translator invokes your profile customizer, which customizes your profiles.

For some VM and compiler configurations, however, the compiler invocation in step 3 will not return and 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 `-passes` 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.

Notes:

- If you enable `-passes`, then compiler output must go to `STDOUT`, so leave `-compiler-pipe-output-flag` enabled (which is its default). Also, you cannot use the `-compiler-output-file` option, which would result in output 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 sequence occurs when you invoke the `sqlj` script:

1. The `sqlj` script invokes your Java VM, which runs the SQLJ translator for its first pass.
2. The translator completes the semantics-checking and translation of your `.sqlj` files, outputting translated `.java` files.
3. The VM is terminated.
4. The `sqlj` script invokes the Java compiler, which compiles the `.java` files.
5. The `sqlj` script invokes your Java VM again, which runs the SQLJ translator for its second pass.
6. The translator processes compiler output.
7. The VM runs your profile customizer, which customizes your profiles.

With this sequence, you circumvent any problems the Java VM may have in invoking the Java compiler.

Command-line syntax -passes

Command-line example -passes

Properties file syntax n/a

Properties file example n/a

Default value `off`

Customization Options

The following options relate to the customization of your SQLJ profiles:

- `-default-customizer`
- options passed directly to the customizer

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 an Oracle database.

This option takes a Java class name as its argument.

Note: This option can be overridden using the `-P-customizer` option in your SQLJ command line (or properties file). For more information, see "Options to Pass to Profile Customizer (-P)" on page 8-50.

Command-line syntax `-default-customizer=`*`customizer_classname`*

Command-line example `-default-customizer=sqlj.myutil.MyCustomizer`

Properties file syntax `sqlj.default-customizer=`*`customizer_classname`*

Properties file example `sqlj.default-customizer=sqlj.myutil.MyCustomizer`

Default value `oracle.sqlj.runtime.util.OraCustomizer`

Note: When you use an Oracle database, Oracle recommends that you use the default `OraCustomizer` for your profile customization.

Options Passed Directly to the Customizer

As with the VM and compiler, you can pass options directly to the profile customizer harness using a prefix, in this case `-P`. This is discussed in "Options to Pass to 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 10-11.

9 Translator and Runtime Functionality

This chapter discusses internal operations and functionality of the Oracle SQLJ translator and runtime.

The following topics are discussed:

- Internal Translator Operations
- Functionality of Translator Errors, Messages, and Exit Codes
- SQLJ Runtime
- NLS Support in the Translator and Runtime

Internal Translator Operations

The following subsections summarize the operations executed by the SQLJ translator during a translation.

Code-Parsing and Syntax-Checking

In this first phase of SQLJ translation, a SQLJ parser and a Java parser are used to process all of 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. It does not check the grammar of the embedded SQL operations, however. SQL syntax is not checked until the semantics-checking step.

This syntax-check will discover errors such as missing semi-colons, mismatched curly braces, and obvious type mismatches (such as multiplying a number by a string).

If the parsers discover any syntax errors or type mismatches during this phase, the translation is aborted and the errors are reported to the user.

Semantics-Checking

Once the SQLJ application source code is verified as syntactically correct, the translator enters into the semantics-checking phase and invokes a semantics-checker according to user option settings. The semantics-checker verifies the validity of Java types in SQL operations (result expressions or host expressions) and optionally connects to a database to check compatibility between Java types and SQL types.

The `-user` option specifies online checking, and the `-password` and `-url` options finish specifying 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 is a checker front end called `OracleChecker`, which chooses the most appropriate checker according to whether you have enabled online checking and which JDBC driver you are using. For more information, see "Connection Options" on page 8-31 and "Semantics-Checking Options" on page 8-55.

The following two tasks are always performed during semantics-checking, whether offline or online:

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 that were also entered on the command-line, and any imported Java classes that 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 that is valid for the given situation and usage. For example, in 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.

   And in 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 (a `java.util.Vector`, for example).

   Note: Remember that 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.

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 };
```

The following two tasks are performed only if online checking is enabled:

3. SQLJ analyzes your embedded SQL operations and checks their syntax against the database.
4. SQLJ checks the types of Java expressions in SQLJ executable statements against: 1) SQL types of corresponding columns in the database; 2) SQL types of corresponding arguments and return variables of stored procedures and functions.

   In the process of doing this, SQLJ verifies that the schema objects 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 handle null data. (Nullability is not checked for stored procedure and function output parameters and return values, however.)

If the semantics-checker discovers any syntax or semantics errors during this phase, then the translation is aborted and the errors are reported.

Oracle supplies Oracle-specific offline checkers, a generic offline checker, Oracle-specific online checkers, and a generic online checker. For more information about checkers, see "Offline Semantics-Checker (-offline)" on page 8-57 and "Online Semantics-Checker (-online)" on page 8-59.

The generic checkers assume you are using only standard SQL92 and standard JDBC features. Oracle recommends that you use the Oracle-specific checkers when using an Oracle database.

Note: The following is *not* checked against the database during online semantics-checking:

- DDL statements (such as `CREATE`, `ALTER`, and `DROP`) and transaction-control statements (such as `COMMIT` and `ROLLBACK`)
- compatibility of data corresponding to weakly typed host expressions (those using the `oracle.sql` package `STRUCT`, `REF`, and `ARRAY` classes, which are discussed in "Weakly Typed Objects, References, and Collections" on page 6-62)
- mode compatibility (`IN`, `OUT`, or `IN OUT`) of expressions in PL/SQL anonymous blocks

---

Code Generation

For your `.sqlj` application source file, the SQLJ translator generates a `.java` file and at least one profile (either in `.ser` or `.class` files). A `.java` file is created for your translated application source code, class definitions for private iterators and connection contexts you declared, and a profile-keys class definition generated and used internally by SQLJ.

Notes: Profiles and a profile-keys class are not generated if you do not use any SQLJ executable statements in your code.

Generated Application Code in .java File

Once your application source code has passed the preceding syntax and semantics checks, it is translated and output to a `.java` file. SQLJ executable statements are replaced by calls to the SQLJ runtime, which in turn contains calls to the JDBC driver to access the database.

The generated `.java` file contains all of your generic Java code, your private iterator class and connection context class definitions, and calls to the SQLJ runtime.

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 class `Foo`, and the generated file will be `Foo.java`.

The location of the generated `.java` file depends on how the SQLJ `-dir` option is set. By default, the `.java` file will be placed in the directory of the `.sqlj` input file. (See "Output Directory for Generated .java Files (-dir)" on page 8-29 for more information.)

Generated Profile-Keys Class in .java File

During translation, SQLJ generates a *profile-keys* class that it uses internally during runtime to load and access the serialized profile. This class contains mapping information between the SQLJ runtime calls in your translated application and the SQL operations placed in the serialized profile. It also contains methods to access the serialized profile.

This 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` that is in the package `a.b` defines the following class:

```
a.b.Foo_SJProfileKeys
```

Generated Profiles in .ser or .class Files

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, stored procedures and functions to call.

Profiles are generated in `.ser` serialized resource files. If, however, you enable the SQLJ `-ser2class` option, they are automatically converted to `.class` files as part of the translation.

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 input file `Foo.sqlj`, 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 package `a.b`, 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:

```
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, the conversion to `.class` takes place *after* the customization step below.)

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. The default for this option is to use the same directory as the `-dir` option, which determines where generated `.java` files are placed. (See "Output Directory for Generated .ser and .class Files (-d)" on page 8-28 for more information.)

In a later step in the SQLJ process, your profiles are customized for use with your particular database. See "Profile Customization" on page 9-10.

More About Generated Calls to SQLJ Runtime

When your `#sql` statements are replaced by calls to the SQLJ runtime, these calls implement the following steps:

1. Get a SQLJ statement object (using information stored in the associated profile entry).
2. Bind inputs into the statement (using `setXXX()` methods of the statement object).
3. Execute the statement (using the `executeUpdate()` or `executeQuery()` method of the statement object).
4. Create iterator instances, if applicable.
5. Retrieve outputs from the statement (using `getXXX()` methods of the statement object).
6. Close the statement.

A SQLJ runtime uses SQLJ statement objects that are similar to JDBC statement objects, although a particular implementation of SQLJ may or may not employ JDBC statement classes directly. SQLJ statement classes add functionality that is 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 an 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 (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 9-5, the following `.class` files would be produced in the appropriate directory (given package information in the source code):

- `Foo.class`
- `Foo_SJProfileKeys.class`
- 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)

So that `.class` files generated by the compiler and profiles generated by SQLJ (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, SQLJ and the compiler will use the setting in the Java VM `file.encoding` property.

By default, SQLJ invokes the standard `javac` compiler of the Sun Microsystems 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, turn off the SQLJ `-compiler-encoding-flag` (otherwise SQLJ will attempt to pass the `-encoding` option to the compiler).

For information about compiler-related SQLJ options, see the following:

- "Output Directory for Generated .ser and .class Files (-d)" on page 8-28
- "Encoding for Input and Output Source Files (-encoding)" on page 8-27
- "Options to Pass to Java Compiler (-C)" on page 8-48
- "Compilation Flag (-compile)" on page 8-52
- "Compiler Encoding Support (-compiler-encoding-flag)" on page 8-65
- "Name of Java Compiler (-compiler-executable)" on page 8-64
- "Compiler Output File (-compiler-output-file)" on page 8-66
- "Compiler Message Output Pipe (-compiler-pipe-output-flag)" on page 8-66

Profile Customization

After Java compilation, the generated profiles (which contain information about your embedded SQL instructions) are customized so that your application can work efficiently with your database and use vendor-specific extensions.

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 by SQLJ option settings.

During customization, profiles are updated in two ways:

- to allow 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 an Oracle8*i* `Person` type that you had defined. You could then use `Person` as you would any other supported datatype.

You also have to use the Oracle customizer to utilize any of the `oracle.sql` type extensions.

Note: You can also 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 and compiled, but not both.

For more information about how `.jar` files are used, see "Use of .jar Files for Profiles" on page 10-26.

For more information about profile customization, see Chapter 10, "Profiles and Customization".

Also see the following for information about SQLJ options related to profile customization:

- "Default Profile Customizer (-default-customizer)" on page 8-69
- "Options to Pass to Profile Customizer (-P)" on page 8-50
- "Profile Customization Flag (-profile)" on page 8-53
- "Customization Options and Choosing a Customizer" on page 10-11

Functionality of Translator Errors, Messages, and Exit Codes

This section provides an overview of SQLJ translator messages and exit codes.

Translator Error, Warning, and Information Messages

There are three major levels of SQLJ messages you may encounter during the translation phase: *error*, *warning*, and *information*. Warning messages can be further broken down into two levels: *non-suppressable* and *suppressable*. Therefore, there are four message categories (in order of seriousness):

1. errors
2. non-suppressable warnings
3. suppressable warnings
4. information

You can control suppressable warnings and information by using the SQLJ `-warn` option, as described below.

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 almost certainly result in a runtime 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 (`.java` file or profiles) and compilation and customization are not executed.

Non-suppressable warning messages, prefixed by `Warning:`, indicate that one of the following has been encountered:

- a condition that would likely, but not necessarily, result in a runtime error if the code were executed (for example, a SELECT statement whose output is not assigned to anything)
- a condition that compromises SQLJ's ability to verify runtime 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 non-suppressable 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, offline checking is performed instead.

Note: For logistical reasons, the parser that the SQLJ translator employs to analyze SQL operations is not the same top-level parser that will be used at runtime. Therefore, errors may occasionally be detected during translation that will not actually cause problems when your application runs. Accordingly, such errors are reported as non-suppressable warnings rather than fatal errors.

Suppressable warning messages, also prefixed by `Warning:`, indicate that there is a problem with a particular aspect of your application, such as portability. For example, retrieving data from a nullable column into a primitive `int` variable. These messages can be suppressed by using the various `-warn` option flags:

- `precision/noprecision`—The `noprecision` setting suppresses warnings regarding possible loss of data precision during conversion.
- `nulls/nonulls`—The `nonulls` setting suppresses warnings about possible runtime errors that are due to nullable columns or types.
- `portable/noportable`—The `noportable` setting suppresses warnings regarding SQLJ code that uses Oracle-specific features or may otherwise be non-standard 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.

See "Translator Warnings (-warn)" on page 8-42 for more information about the `-warn` option and how to set the flags.

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 (non-suppressable) variety.

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. These messages can be suppressed by using the `-warn` option `verbose` flag, as follows:

- `verbose/noverbose`—The `noverbose` setting suppresses status messages that are merely informational and do not indicate error or warning conditions.

Notes: For information about particular error, warning, and information messages, see "Translation Time Messages" on page B-2 and "Runtime Messages" on page B-42.

Table 9-1 summarizes the categories of error and status messages generated by the SQLJ translator.

Table 9–1 SQLJ Translator Error Message Categories

Message Category	Prefix	Indicates	Suppressed By
error	`Error:`	fatal error that will cause compilation failure or runtime failure (translation is aborted)	n/a
non-suppressable warning	`Warning:`	condition that prevents proper translation or might cause runtime failure (translation is completed)	n/a
suppressable warning	`Warning:`	problem regarding a particular aspect of your application (translation is completed)	`-warn` option flags: `noprecision` `nonulls` `noportable` `nostrict`
informational/status message	`Info:`	information regarding the translation process	`-warn` option flags: `noverbose`

Translator Status Messages

In addition to the translator's error, warning, and information messages, SQLJ can produce status messages throughout all phases of SQLJ operation—translation, compilation, and customization. Status messages are output as each file is processed and at each phase of SQLJ operation.

You can control status messages by using the SQLJ `-status` option. This option is also discussed in "Real-Time Status Messages (-status)" on page 8-44.

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)

Notes:

- 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 Runtime

This section provides information about the Oracle SQLJ runtime, which is a thin layer of pure Java code that runs above the JDBC driver. When Oracle SQLJ translates your SQLJ source code, embedded SQL commands in your Java application are replaced by calls to the SQLJ runtime. Runtime classes act as wrappers for equivalent JDBC classes, providing special SQLJ functionality. When the end-user runs the application, the SQLJ runtime acts as an intermediary, reading information about your SQL operations from your profile and passing instructions along to the JDBC driver.

Note: A SQLJ runtime can be implemented to use any JDBC driver or vendor-proprietary means of accessing the database. The Oracle SQLJ runtime requires a JDBC driver but can use any standard JDBC driver. To use Oracle-specific database 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 an Oracle database and one of the Oracle JDBC drivers.

Runtime Packages

The Oracle SQLJ runtime consists of the packages listed below. There are classes you can import and use directly in your application (primarily in `sqlj.runtime`), but most of the runtime classes are for internal use by SQLJ. Packages whose names begin with `oracle` are for Oracle-specific SQLJ features.

- `sqlj.runtime`

  This package is the most likely to be imported and used directly in your application. It contains wrapper classes for various kinds of input streams (binary, ASCII, and Unicode), as well as interfaces and abstract classes that are implemented by Oracle SQLJ connection contexts and iterators.

  The interfaces and abstract classes in this package are implemented by classes in the `sqlj.runtime.ref` 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 almost certainly 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.

- `sqlj.runtime.profile`

  This package contains interfaces and abstract classes that define what SQLJ profiles look like. This includes the `EntryInfo` class and `TypeInfo` class. 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. It also provides the default JDBC-based runtime implementation.

- `sqlj.runtime.profile.util`

  This package contains utility classes used to access and manipulate profiles. For example, SQLJ uses the `CustomizerHarness` class in this package to invoke a specified profile customizer on a specified set of profiles, according to the SQLJ option settings you specify when you translate your SQLJ source code.

- `sqlj.runtime.error`

  This package, used internally by SQLJ, contains resource files for all generic (non-Oracle-specific) error messages that can be generated by the SQLJ translator.

- `oracle.sqlj.runtime`

  This package contains Oracle-specific runtime classes used by the Oracle implementation of SQLJ. For example, this package includes functionality to convert to and from Oracle type extensions. It also includes the `Oracle` class that you can use to instantiate the `DefaultContext` class and establish your default connection.

- `oracle.sqlj.runtime.util`

  This package contains utility classes used by SQLJ to access and manipulate Oracle-specific profiles, including the `OraCustomizer` class for customizing profiles.

- `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.

Note: The packages `sqlj.runtime.profile.util`, `oracle.sqlj.runtime.util`, and `oracle.sqlj.runtime.error` are not included in `runtime.zip`, which contains classes used only at runtime. Because the classes in these packages are used only by the translator, they are located only in `translator.zip`.

Categories of Runtime Errors

Runtime errors may be generated by any of the following:

- SQLJ runtime
- JDBC driver
- RDBMS

In any of these cases, a SQL exception is generated as an instance of the `java.sql.SQLException` class or a subclass (such as `sqlj.runtime.SQLNullException`).

Depending on where the error came from, there may be meaningful information that you can retrieve from an exception using the `getSQLState()`, `getErrorCode()`, and `getMessage()` methods. SQLJ errors, for example, include meaningful SQL state and message. For information, see "Retrieving SQL States and Error Codes" on page 4-22.

If errors are generated by the Oracle JDBC driver or RDBMS at runtime, look at the prefix and consult the appropriate documentation:

- *Oracle8i JDBC Developer's Guide and Reference* for JDBC errors
- *Oracle8i Error Messages* reference for RDBMS errors

For a list of SQLJ runtime errors, see "Runtime Messages" on page B-42.

NLS Support in the Translator and Runtime

Oracle SQLJ uses Java's built-in NLS capabilities. This section discusses the basics of SQLJ support for NLS and native character encoding, starting with background information covering some of the implementation details of character encoding and language support in Oracle SQLJ. This is followed by discussion of options available through the Oracle SQLJ command line that allow you to adjust your NLS configuration.

Some prior knowledge of Oracle NLS is assumed, particularly regarding character encoding and locales. For information, see the *Oracle8i National Language Support Guide*.

Character Encoding and Language Support

There are two main areas of SQLJ NLS support:

- character encoding

  There are three parts to this: 1) character encoding for reading and generating source files during SQLJ translation; 2) character encoding for generating error and status messages during SQLJ translation; and 3) character encoding for generating error and status messages when your 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 SQLJ runtime.

Notes:

- SQLJ fully supports Unicode 2.0 and Java Unicode escape sequences.
- Encoding and conversion of characters in your embedded SQL operations and characters read from or written to the database is handled by JDBC directly; SQLJ does not play a role in this. If online semantics-checking is enabled during translation, however, you will be warned if there are characters within your SQL DML operations that may not be convertible to the database character set.
- For information about JDBC NLS functionality, see the *Oracle8i JDBC Developer's Guide and Reference*.

Overview of Character Encoding

The character encoding setting for source files tells Oracle 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 Java VM `file.encoding` property. If your source files use other encodings, 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` property to specify encodings for source files. This may impact other aspects of your Java operation and may offer only a limited number of encodings, depending on platform or operating system considerations.

The character encoding setting for error and status messages determines how SQLJ messages are represented when output to the user, either during translation or during runtime 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 that is supported by your Java environment. If you are using the Sun Microsystems JDK, these are listed in the `native2ascii` documentation, which you can find at the following Web site:

`http://www.javasoft.com/products/jdk/1.1/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`.

Notes:

- 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 (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.

Overview of Language Support

SQLJ error and status reporting, either during translation or during runtime, uses the Java locale setting in the VM `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 which message resources are 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

Oracle SQLJ provides syntax that allows 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 runtime 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 runtime 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 output 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
```

Or it can be set 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 Java VM `file.encoding` property. This can also be set through the SQLJ command line, as discussed in "Setting Character Encoding and Locale for SQLJ Messages" on page 9-24.

For more information, see "Encoding for Input and Output Source Files (-encoding)" on page 8-27 and "Compiler Encoding Support (-compiler-encoding-flag)" on page 8-65.

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 above. For more information, see "Properties Files for Option Settings" on page 8-13.

Setting Character Encoding and Locale for SQLJ Messages

Character encoding and locale for SQLJ error and status messages output to the user, during both translation and runtime, are determined by the Java `file.encoding` and `user.language` properties. Although it is typically not necessary, you can set these and other VM properties in the SQLJ command line by using the SQLJ `-J` prefix. Options marked by this prefix are passed to the VM.

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 may 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 VM 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 8-17.
- 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 may have unforeseen consequences on other aspects of your Java operations; furthermore, any new setting must be compatible with your operating system.

For additional information about the SQLJ `-J` prefix, see "Command-Line Syntax and Operations" on page 8-10 and "Options to Pass to Java VM (-J)" on page 8-47.

SQLJ Command Line Example: Setting Encoding and Locale

Following is a complete SQLJ command line, including VM `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) encoding 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.

NLS Manipulation Outside of SQLJ

This section discusses ways to manipulate your NLS configuration outside of SQLJ.

Setting Encoding and Locale at Application Runtime

As with any end user running any Java application, end users running your SQLJ application can specify VM properties such as `file.encoding` and `user.language` directly as they invoke the VM 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.getProperties()` method (or the `getProperty()` method, if you specify a particular property).

The following is an 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.

You can get information about `java.lang.System` at the following Web site:

```
http://www.javasoft.com/products/jdk/1.1/docs/api/java.lang.System.html
```

Using native2ascii for Source File Encoding

If you are using the Sun JDK, there is an alternative to having SQLJ do the character encoding for your source files. You can use the utility `native2ascii` to convert sources with native encoding to sources in 7-bit ASCII with Unicode escape sequences.

Note: Before using `native2ascii`, you must ensure that the Java VM 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://www.javasoft.com/products/jdk/1.1/docs/tooldocs/solaris/native2ascii.html

10 Profiles and Customization

Profiles and profile customization are introduced in "SQLJ Profiles" on page 1-3.

This chapter goes into more technical detail and discusses customizer options and how to use customizers other than the default Oracle customizer.

The following topics are discussed:

- More About Profiles
- More About Profile Customization
- Customization Options and Choosing a Customizer
- Use of .jar Files for Profiles

More About Profiles

SQLJ profiles contain information about your embedded SQL operations, with a separate profile being created for each connection context class (or, equivalently, for each type of database schema) that your application uses. Profiles are created during the SQLJ translator's 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 re-customized) 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. Creates a profile object as an instance of the `sqlj.runtime.profile.Profile` class.
2. Inserts information about your embedded SQL operations (for SQLJ statements that use the relevant connection context class) into the profile object.
3. Serializes the profile object into a Java resource file, referred to as a *profile file*, with a `.ser` file name extension.

Note: Oracle SQLJ provides an option to have the translator automatically convert these `.ser` files to `.class` files (`.ser` files are not supported by some browsers). For information, see "Conversion of .ser File to .class File (-ser2class)" on page 8-54.

As discussed in "Code Generation" on page 9-5, 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 are used). 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 Runtime" on page 10-9.

Note: Referring to a "profile object" indicates that the profile is in its original non-serialized state. Referring to a "profile file" indicates that the profile is in its serialized state in a `.ser` file.

Sample Profile Entry

Below 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 ename = 'Smith';
Iter it;
...
#sql it = { select ENAME, SAL from EMP where ENAME = :ename };
```

Corresponding SQLJ Profile Entry

```
=================================================================
...
#sql { select ENAME, SAL from EMP where ENAME = ? };
...
PREPARED_STATEMENT executed via EXECUTE_QUERY
role is QUERY
descriptor is null
contains one parameters
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 "General Customizer Harness Options" on page 10-11.

More About Profile Customization

By default, running the `sqlj` script on a SQLJ source file includes an automatic customization process, where each profile created during the translator's 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 Oracle8*i* databases.

You can also run the `sqlj` script to customize profiles that were created previously. On the SQLJ command line, you can specify `.ser` files individually, `.jar` files containing `.ser` files, or both.

Note: You can run SQLJ to process `.sqlj` and/or `.java` files (translation, compilation, and customization), or to process `.ser` and/or `.jar` files (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 to the customizer harness (for general customization options) and your customizer (for customizer-specific options). In either case, you can specify these option either on the command line or in a properties file. This is discussed in "Customization Options and Choosing a Customizer" on page 10-11.

Implementation Details The following paragraphs into detail how Oracle implements the customizer harness and the Oracle customizer. This information is not necessary for most SQLJ developers.

The customizer harness is a command-line tool that is an instance of the class `sqlj.runtime.profile.util.CustomizerHarness`. A `CustomizerHarness` object is created and invoked each time you run the SQLJ translator. During the customization phase, the harness creates and invokes an object of the customizer class you are using (such as the default Oracle customizer), and loads your profiles.

The Oracle customizer is defined in the `oracle.sqlj.runtime.OraCustomizer` class. All customizers must be JavaBeans components that adhere to the JavaBeans API to expose their properties and must implement the `sqlj.runtime.profile.util.ProfileCustomizer` interface, which specifies a `customize()` method. It is the implementation of this method in a particular customizer that does the work of customizing profiles.

For each profile that is to be customized, the customizer harness calls the `customize()` method of the customizer object.

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. The customizer harness discovers and extracts the profile files within any `.jar` files (only applicable when you run SQLJ for customization only, specifying one or more `.jar` files on the command line).
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.

Note: The Oracle customizer does not currently use database connections.

6. For each profile, the harness calls the `customize()` method of the customizer object that was instantiated in step 2 (customizers used with Oracle 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 may 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. The customizer harness recreates the `.jar` contents, inserting each customized `.ser` file to replace the original corresponding uncustomized `.ser` file (only applicable when you run SQLJ for customization only, specifying one or more `.jar` files on the command line).

Notes:

- 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 JDBC `Connection` 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 Oracle customizer does not use connections, so a null `Connection` object is passed in this case.

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 that is 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.

Notes:

- 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.

Implementation Details The following paragraphs detail how Oracle implements the customization process. This information is not necessary for most SQLJ developers.

In the case of the Oracle customizer, the `customize()` method creates a data structure that has one entry for each entry in the original profile. The original entries are never changed, but the new entries are customized to take advantage of features of Oracle8*i*. For example, if you are using BLOBs, a generic `getObject()` call used to retrieve a BLOB in the original entry is replaced by a `getBLOB()` call.

These new entries are encapsulated in an object of a customization class that implements the `sqlj.runtime.profile.Customization` interface, and this customization object is installed into the profile object. (Customization objects, like profile objects, are serializable.)

The customizer harness then registers the customization, which is accomplished through functionality of the profile object. Registration allows a profile to keep track of the customizations that it contains.

Any errors encountered during customization are posted to the error log and reported by the customizer harness as appropriate.

A `Customization` object has an `acceptsConnection()` method that is called at runtime to determine if the customization can create a connected profile object for a given JDBC `Connection` object. A connected profile object—an instance of a class that implements the `sqlj.runtime.profile.ConnectedProfile` interface—represents a mapping between a profile object and a JDBC connection. It is equivalent to a JDBC `Connection` object, with the ability to create statements, but supports additional vendor-specific functionality.

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 suppressable, however. (See "Translator Error, Warning, and Information Messages" on page 9-12.)

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 allow 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 may 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 need to access error log contents directly.

Functionality of a Customized Profile at Runtime

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 runtime 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 runtime uses in applying vendor-specific features to the execution of your SQLJ application.

Implementation Details The following paragraphs details how the Oracle SQLJ runtime uses customized profiles. This information is not necessary for most SQLJ developers.

In executing a SQLJ statement, the SQLJ runtime uses methods of the connection context object associated with the statement, and the profile object associated with the connection context class, as follows:

1. When an end-user is running your application and a SQL operation is to be executed, the SQLJ runtime calls the connection context `getConnectedProfile()` method.
2. The connection context `getConnectedProfile()` method calls the `getConnectedProfile()` method of the profile object that is associated with the connection context class, passing it a connection. (This is the connection instance underlying the connection context instance used for the SQL operation.)
3. The profile object `getConnectedProfile()` method calls the `acceptsConnection()` method of each `Customization` object registered in the profile. The first `Customization` object that accepts the connection creates the connected profile that is passed back to the runtime.
4. In executing the SQL operation, the connected profile is used like a JDBC connection—creating statements to be executed—but implements special functionality of the customization.

Customization Options and Choosing a Customizer

This section discusses options for profile customization, which fall into three categories:

- general options you specify to the customizer harness, which apply regardless of which customizer you use
- 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.

To choose a customizer other than the default Oracle customizer, you can use either the customizer harness `-customizer` option (discussed in "General Customizer Harness Options" on page 10-11), or the SQLJ `-default-customizer` option (discussed in "SQLJ Options for Profile Customization" on page 10-25).

General Customizer Harness Options

The customizer harness provided with Oracle SQLJ offers several general options for your customizations. These apply regardless of the particular customizer you are using.

Customizer harness option settings on the SQLJ command line have the following syntax:

```
-P-option=value
```

Or, 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 Profile Customizer (-P)" on page 8-50 and "Properties File Syntax" on page 8-14.

The customizer harness supports the following 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 (only relevant if specifying `.jar` files to customize).
- `-help`—Display customizer options (SQLJ command-line-only).
- `-verbose`—Display status messages during customization.

The following options, related to database connections for customization, are supported by the customizer harness but are not used by the Oracle customizer:

- `-user`—Specify the username 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.

In addition, the following commands function as customizer harness options but are implemented through specialized customizers provided with Oracle SQLJ.

- `-print`—Output the contents of the specified profiles, in text format.
- `-debug`—Insert debugging information into the specified profiles, to be output at runtime.

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.

Command-line syntax `-P-backup<=`*`true/false`*`>`

Command-line example `-P-backup`

Properties file syntax `profile.backup<=`*`true/false`*`>`

Properties file example `profile.backup`

Default value `false`

Connection Context Option (-context)

Use the `-context` option to limit customizations to profiles that correspond to the specified connection context classes. Fully specify the classes and use a comma-separated list to specify multiple classes. For example:

```
-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 `-P-context=`*`ctx_class1`*`<,`*`ctx_class2`*`,...>`

Command-line example `-P-context=foo.bar.MyCtxtClass`

Properties file syntax `profile.context=`*`ctx_class1`*`<,`*`ctx_class2`*`,...>`

Properties file example `profile.context=foo.bar.MyCtxtClass`

Default value none (customize all profiles)

Customizer Option (-customizer)

Use the `-customizer` option to specify which customizer to use. Fully specify the class name, such as in the following example:

```
-P-customizer=sqlj.runtime.profile.util.AuditorInstaller
```

Note: If you do not set this option, then SQLJ will use the customizer specified in the SQLJ `-default-customizer` option.

Command-line syntax `-P-customizer=`*`customizer_class`*

Command-line example `-P-customizer=a.b.c.MyCustomizer`

Properties file syntax `profile.customizer=`*`customizer_class`*

Properties file example `profile.customizer=a.b.c.MyCustomizer`

Default value `oracle.sqlj.runtime.util.OraCustomizer`

.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, then use the `-digests` option to specify a comma-separated list of one or more algorithms. These are the algorithms that `jar` 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.

For information about `.jar` files and the `jar` utility, see the following Web site:

```
http://www.javasoft.com/products/jdk/1.1/docs/guide/jar/index.html
```

Command-line syntax `-P-digests=`*`algo1`*`<,`*`algo2`*`,...>`

Command-line example `-P-digests=SHA,MD5`

Properties file syntax `profile.digests=`*`algo1`*`<,`*`algo2`*`,...>`

Properties file example `profile.digests=SHA,MD5`

Default value `SHA,MD5`

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
```

Notes:

- You can use the `-P-help` option on the SQLJ command line only, not in a SQLJ properties file.
- No customizations are done if the `-P-help` flag is enabled, even if you specify profiles to customize on the command line.

Command-line syntax `-P-help <-P-customizer=`*`customizer_class`*`>`

Command-line example `-P-help`

Properties file syntax n/a

Properties file example n/a

Default value none

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 `-P-verbose<=`*`true/false`*`>`

Command-line example `-P-verbose`

Properties file syntax `profile.verbose<=`*`true/false`*`>`

Properties file example `profile.verbose`

Default value none

User Option (-user)

Set the `-user` option to specify a user schema if your customizer uses database connections.

Note: The Oracle customizer does not currently use database connections.

In addition to specifying the schema, you can optionally specify the password or 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:

```
-user=scott/tiger
-user=scott@jdbc:oracle:oci8:@
-user=scott/tiger@jdbc:oracle:oci8:@
```

Command-line syntax `-P-user=`*`username`*`</`*`password`*`><@`*`url`*`>`

Command-line examples

```
-P-user=scott
-P-user=scott/tiger
-P-user=scott/tiger@jdbc:oracle:oci8:@
```

Properties file syntax `profile.user=`*`username`*`</`*`password`*`><@`*`url`*`>`

Properties file examples

```
profile.user=scott
profile.user=scott/tiger
profile.user=scott/tiger@jdbc:oracle:oci8:@
```

Default value `null`

Password Option (-password)

Use the `-password` option if your customizer uses database connections.

Note: The Oracle customizer does not currently use database connections.

The password can also be set with the `-user` option, as described in "User Option (-user)" on page 10-16.

Command-line syntax `-P-password=`*`password`*

Command-line example `-P-password=tiger`

Properties file syntax `profile.password=`*`password`*

Properties file example `profile.password=tiger`

Default value `null`

URL Option (-url)

Use the `-url` option if your customizer uses database connections.

Note: The Oracle customizer does not currently use database connections.

The URL can also be set with the `-user` option, as described in "User Option (-user)" on page 10-16.

Command-line syntax `-P-url=`*`url`*

Command-line example `-P-url=jdbc:oracle:oci8:@`

Properties file syntax `profile.url=`*`url`*

Properties file example `profile.url=jdbc:oracle:oci8:@`

Default value `jdbc:oracle:oci8:@`

JDBC Driver Option (-driver)

Use the `-driver` option to register a comma-separated list of JDBC drivers if your customizer uses database connections. For example:

```
-P-driver=sun.jdbc.odbc.JdbcOdbcDriver,oracle.jdbc.driver.OracleDriver
```

There must be no space on either side of the comma.

> **Note:** The Oracle customizer does not currently use database connections.

Command-line syntax `-P-driver=`*`dvr_class1`*`<,`*`dvr_class2`*`,...>`

Command-line example `-P-driver=sun.jdbc.odbc.JdbcOdbcDriver`

Properties file syntax `profile.driver=`*`dvr_class1`*`<,`*`dvr_class2`*`,...>`

Properties file example `profile.driver=sun.jdbc.odbc.JdbcOdbcDriver`

Default value `oracle.jdbc.driver.OracleDriver`

Profile Print Option (specialized customizer) (-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.

Following are examples of how to specify the `-print` option:

```
sqlj -P-print Foo_SJProfile0.ser Bar_SJProfile0.ser
sqlj -P-print *.ser
```

The result of the `-print` option is that the customizer harness invokes a special print customizer that converts the profile information to text format and outputs the results.

For an example of what this output looks like, see "Sample Profile Entry" on page 10-3.

Note: Because the `-print` option invokes a customizer and only one customizer can run in a single execution of SQLJ, you cannot do any other customization when you use this option.

You also cannot use `-P-print` and `-P-debug` at the same time, because the `-debug` option also invokes a special customizer.

Command-line syntax `sqlj -P-print` *`profile_list`*

Command-line example `sqlj -P-print Foo_SJProfile*.ser`

Properties file syntax `profile.print` (must also specify profiles in file list)

Properties file example `profile.print` (must also specify profiles in file list)

Default value n/a

Profile Debug Option (specialized customizer) (-debug)

The `-debug` option runs a specialized customizer 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.

The debugging statements will execute during SQLJ runtime (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
```

The result of the `-debug` option is that the customizer harness invokes a special debug customizer to install the debugging statements into the profiles.

For more information about the `-debug` option, including additional options provided by the debug customizer, see "AuditorInstaller Customizer for Debugging" on page A-4.

Note: Because the `-debug` option invokes a customizer and only one customizer can run in a single execution of SQLJ, you cannot do any other customization when you use this option.

You also cannot use `-P-print` and `-P-debug` at the same time, because the `-print` option also invokes a special customizer.

Command-line syntax `sqlj -P-debug` *`profile_list`*

Command-line example `sqlj -P-debug Foo_SJProfile*.ser`

Properties file syntax `profile.debug` (must also specify profiles in file list)

Properties file example `profile.debug` (must also specify profiles in file list)

Default value n/a

Customizer-Specific Options (Oracle Customizer)

In addition to setting customizer harness options, you can use the SQLJ command line to specify option settings for the particular customizer you are using.

Set a customizer option on the SQLJ command line by preceding it with:

```
-P-C
```

Or set it in a SQLJ properties file by preceding it with:

```
profile.C
```

This option syntax is also discussed in "Options to Pass to Profile Customizer (-P)" on page 8-50 and "Properties File Syntax" on page 8-14.

The remainder of this section discusses features of the Oracle customizer, which supports several boolean options. Enable these options (flags) with:

```
-P-Coption
```

or:

```
-P-Coption=true
```

Boolean options are disabled by default but you can explicitly disable them with:

```
-P-Coption=false
```

The Oracle customizer supports the following options:

- `-compat`—Display version compatibility information.
- `-force`—Instruct the customizer to customize even if a valid customization already exists.
- `-showSQL`—Display SQL statement transformations.
- `-summary`—Display a summary of Oracle features used in your application.

To use these options, you must be using the default Oracle customizer (in other words, no other customizer is specified in the SQLJ `-default-customizer` setting or the customizer harness `-customizer` setting, and you are not using `-print` or `-debug`).

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 the Oracle database and Oracle JDBC drivers. This can be accomplished either during a full SQLJ translation run or on profiles previously created.

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 samples resulting from a `-P-Ccompat` setting when using the default Oracle customizer.

Example 1 The application can be used with all Oracle JDBC driver versions:

```
MyApp_SJProfile0.ser: Info: compatible with all Oracle JDBC drivers
```

Example 2 The application can be used only with 8.1 or later Oracle JDBC driver versions:

```
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 `-P-Ccompat<=`*`true/false`*`>`

Command-line example `-P-Ccompat`

Properties file syntax `profile.Ccompat<=`*`true/false`*`>`

Properties file example `profile.Ccompat`

Default value `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 of the `MyApp` profiles, whether or not 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 `-P-Cforce<=`*`true/false`*`>`

Command-line example `-P-Cforce`

Properties file syntax `profile.Cforce<=`*`true/false`*`>`

Properties file example `profile.Cforce`

Default value `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 Oracle8*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 Oracle8*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 `-P-CshowSQL<=`*`true/false`*`>`

Command-line example `-P-CshowSQL`

Properties file syntax `profile.CshowSQL<=`*`true/false`*`>`

Properties file example `profile.CshowSQL`

Default value `false`

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.

Example 1 No Oracle features used:

```
MyApp_SJProfile0.ser: Info: Oracle features used:
MyApp_SJProfile0.ser: Info: * none
```

Example 2 Oracle features used:

```
MyApp_SJProfile0.ser: Info: Oracle features used:
MyApp_SJProfile0.ser: Info: * oracle.sql.NUMBER: 2
MyApp_SJProfile0.ser: Info: * oracle.sql.RAW: 2
```

In this second sample, the application uses Oracle extended datatypes from the `oracle.sql` package.

Note: If customization does not take place because a valid previous customization is detected, the `-summary` option produces a summary regardless.

Command-line syntax `-P-Csummary<=`*`true/false`*`>`

Command-line example `-P-Csummary`

Properties file syntax `profile.Csummary<=`*`true/false`*`>`

Properties file example `profile.Csummary`

Default value `false`

SQLJ Options for Profile Customization

The following SQLJ 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 8-69.

- `-profile`—Specify whether to customize during this running of SQLJ.

  See "Profile Customization Flag (-profile)" on page 8-53.

Use of .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.

Notes:

- Remember that you can specify `.sqlj` and/or `.java` files on the SQLJ command line for normal SQLJ processing, or you can specify `.ser` and/or `.jar` files on the command line for customization only, but not both.
- 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 that is 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

When using a `.jar` file for profiles, the manifest entry for each profile must contain the line:

```
SQLJProfile: TRUE
```

Accomplish this by: 1) creating a plain text file with two lines for each profile that will be included in the `.jar` file—one line specifying the path or package and name, and one line as above; and 2) using the `jar` utility `-m` option to input this file.

The two lines must be consecutive (no blank line in between), and there must be a blank line preceding line-pairs for additional profiles.

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 Results

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 re-signing the new `.jar` file if the profiles require signing.

11 SQLJ in the Server

SQLJ applications can be stored and run in the 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 server's embedded translator.

This chapter discusses features and usage of SQLJ in the server, including additional considerations such as multithreading and recursive SQLJ calls.

Most of this chapter assumes you are writing stored procedures or stored functions, but additional vehicles such as Enterprise JavaBeans or CORBA objects are supported as well.

The following topics are discussed:

Introduction

SQLJ code, like any Java code, can run in the Oracle8*i* server in stored procedures, stored functions, triggers, Enterprise JavaBeans, or CORBA objects. Database access is through a server-side implementation of the SQLJ runtime in combination with the Oracle JDBC server-side driver. In addition, there is an embedded SQLJ translator in the Oracle8*i* server so that SQLJ source files for server-side use can optionally be translated 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 (`.java` and `.ser`) are transparent to the developer.

Creating SQLJ Code for Use in the Server

With few exceptions, writing SQLJ code for use in the Oracle8*i* server 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.

When writing SQLJ code for use in the server, be aware of the following:

- The SQLJ runtime packages are automatically available in the server.
- There are no explicit database connections for an application that executes in the server. Instead there is an implicit connection to the server itself.
- There are coding issues relating to the Oracle JDBC server-side driver, such as lack of auto-commit functionality.
- In the server, the default output device is the current trace file.

You should also be aware of class naming and class resolution considerations. These are discussed in "About Class Loading, Compiling, and Name Resolution" on page 11-27.

Database Connections in the Server

The concept of connecting to the server is different when your SQLJ code is running in the server itself. Note the following issues and characteristics when writing code for use in the server:

- There is no explicit database connection in the server as there is from a client. By default there is an implicit channel to the database for the current user, which is employed for any Java program running in the server. You do not have to initialize this "connection"; it is automatically initialized for SQLJ programs. You do not have to register or specify a driver, create a connection object, specify a default connection context, or specify any connection objects for any of your `#sql` statements.

  Any SQLJ connection context instance or JDBC connection instance in your program communicates to the database through the database session that is executing your Java code.
- Any attempt to close a SQLJ connection context instance or JDBC connection in a Java program running in the server is ignored. No exception is thrown and there is no effect on communication to the database.

- You cannot connect to a remote database from a Java program executing in the server.

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 Relating to the JDBC Server-Side Driver

Note the following characteristics of the Oracle JDBC server-side driver when writing code for use in the server:

- Result sets issued by the server-side driver persist across calls, and their finalizers do not release their database 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 server-side driver does not support auto-commit functionality—the auto-commit setting is ignored in the server. Use explicit `COMMIT` or `ROLLBACK` statements to implement or cancel your database 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. This applies particularly to Enterprise JavaBeans or CORBA objects, which typically use such transactions. See "Additional Vehicles for SQLJ in the Server" on page 11-35.

For more information about server-side JDBC and the server-side driver, see the *Oracle8i JDBC Developer's Guide and Reference*.

Server-Side Default Output Device

The default standard output device in the Oracle8*i* Java VM is the current trace file.

If you want to reroute all standard output from a program executing in the server (output from any `System.out.println()` calls, for example) to a user screen, then you can execute the `SET_OUTPUT()` procedure of the `DBMS_JAVA` package, as follows (inputting the buffer size in bytes):

```
sqlplus> execute dbms_java.set_output(10000);
```

If you want your code executing in the server to expressly output to the user screen, 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.

For more information about the `DBMS_OUTPUT` package, see the *Oracle8i Supplied Packages Reference*.

Translating SQLJ Source on Client and Loading Components

One approach to developing SQLJ code for the server is to first run the SQLJ translator on a client machine to take care of translation, compilation, and profile customization. Then load the resulting class and resource files into the server, typically using a Java archive (`.jar`) file.

If you are developing your source on a client machine, as is usually the case, and have a SQLJ translator available there, this approach is advisable. It allows the most flexibility in running the translator because option-setting and error-processing are not as convenient in the server.

It may 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. Be aware, however, that profiles converted to `.class` files cannot be further customized. To further customize, you would have to rerun the translator and regenerate the profiles. For information about the `-ser2class` option, see "Conversion of .ser File to .class File (-ser2class)" on page 8-54.

When you load `.class` files and `.ser` resource files (if any) into the server, either directly or using a `.jar` file, the resulting database library units are referred to as Java *class schema objects* (for Java classes) and Java *resource schema objects* (for Java resources). A separate schema object is created for each class and for each resource.

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 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 that is in the `.jar` file or on the command line.

The `loadjava` utility does not support compressed files. If you create a `.jar` file, specify the files it contains to be *uncompressed*, as in the following example (the "0" specifies no compression):

```
jar -cvf0 demo.jar *.class *.ser
```

The `loadjava` utility defaults to the JDBC OCI 8 driver (which does not require a URL as part of the `loadjava -user` option setting). Alternatively, you can use the Thin driver by using the `-thin` option and specifying an appropriate URL through the `-user` option, as shown in one of the examples below.

Consider an example where you: 1) translate and compile `Foo.sqlj`, which includes an iterator declaration for `MyIter`; 2) enable the `-ser2class` option when you translate `Foo.sqlj`; and 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):

```
loadjava -user scott/tiger Foo.jar
```

Or, to use the Thin driver for loading (specifying the `-thin` option and an appropriate URL):

```
loadjava -thin -user scott/tiger@localhost:1521:ORCL Foo.jar
```

Or, alternatively, use the original files (again with the OCI driver):

```
loadjava -user scott/tiger Foo.class MyIter.class Foo_SJProfileKeys.class Foo_SJProfile0.class
```

or:

```
loadjava -user scott/tiger Foo*.class MyIter.class
```

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.

For more information about the `loadjava` utility, see the *Oracle8i Enterprise JavaBeans and CORBA Developer's Guide*. For information about files generated by the SQLJ translator, see "Code Generation" on page 9-5 and "Java Compilation" on page 9-8.

Notes:

- Once you load a source file, you cannot subsequently load its classes and resources unless you first drop the source. See "Dropping Java Schema Objects" on page 11-30.
- By default, `loadjava` loads into the login schema specified by the `-user` option. Use the `-schema` option to specify a different schema to load into. This does not involve a login to that schema, but does require that you have sufficient permissions to alter it.
- When you load a profile into the server as a `.ser` file, it is first customized if it was not already customized on the client. If it was already customized, it is loaded as is.

Although the `loadjava` utility is recommended for loading your SQLJ and Java applications into the server, you can also use Oracle 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 the *Oracle8i SQL Reference* for more information about the `CREATE JAVA` commands.

Naming of 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.

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 only be one profile.

Notes: There are two forms of schema object names in the server: full names and short names. For information about these and about other file naming considerations, see "Full Names vs. Short Names in the Server" on page 11-24.

Full Names of Loaded Classes (including profiles if -ser2class enabled)

The full name of the class schema object that is 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 (so `loadjava` can find it, for example) or in the `.jar` file is irrelevant in determining the name of the schema object. For example, if `Foo.class` consists of a class `Foo` which was specified in the source code as being in package `x.y`, then the full name of the resulting class schema object is as follows:

```
x/y/Foo
```

Note that ".class" is dropped.

If `Foo.sqlj` declares an iterator `MyIter`, then the full name of its class schema object is:

```
x/y/MyIter
```

(Unless it is a nested class, in which case it will not have its own schema object.)

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 any resulting profile or profiles were generated in file `Foo_SJProfile0.class` (and `Foo_SJProfile1.class`, and so on); therefore, the full name of class schema objects are of the form:

```
x/y/Foo_SJProfile0
```

Full Names of Loaded Resources (including profiles if -ser2class not enabled)

This discussion is relevant only if you 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 than for 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 runtime, their names must include the correct path information. In the server, the correct full name of a resource is the same as the relative path and file name that Java would use to look it up on the client.

In the case of a SQLJ profile, this is a subdirectory under the `-d` option directory, according to the package name. If the translator `-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 package `abc.def`, then `.class` and `.ser` files generated during translation will be placed in the `/mydir/abc/def` directory.

At runtime, `/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:

```
cd /mydir
loadjava <...options...> abc/def/*.class abc/def/*.ser
```

Or, if you use a `.jar` file:

```
cd /mydir
jar -cvf0 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 above examples will correctly result in the following full name of the created resource schema object:

```
abc/def/App_SJProfile0.ser
```

Note that if you set `-d` to a directory whose hierarchy has no other contents (which is advisable), you can simply run `jar` as follows to recursively get your application components:

```
cd /mydir
jar -cvf0 myjar.jar *
loadjava <...options...> myjar.jar
```

Note: The "0" in `-cvf0` specifies no compression, which is required for `loadjava`.

For more information about the SQLJ `-d` option (including default value), see "Output Directory for Generated .ser and .class Files (-d)" on page 8-28.

Additional Steps After Loading Class and Resource Files

You can access the `USER_OBJECTS` view in your database schema to verify that your classes and resources are loaded properly. For more information, see "Checking Java Uploads" on page 11-24.

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 involves writing call descriptors, mapping datatypes, and setting parameter modes. For information, see the *Oracle8i Java Stored Procedures Developer's Guide.*

Steps in Running a Client Application in the Server

This section takes you through typical steps of running a client application in the server. As an example, it uses the `NamedIterDemo` sample application that is provided in "Named Iterator—NamedIterDemo.sqlj" on page 12-5.

Use the `connect.properties` file for your connection settings.

1. Update your runtime connection URL in `connect.properties` to use the server-side (KPRB) JDBC driver.

   For example, if you previously used the OCI 8 driver, update `connect.properties` as follows.

   old:

   ```
   sqlj.url=jdbc:oracle:oci8:@
   ```

new:

```
sqlj.url=jdbc:oracle:kprb:@
```

2. Create a `.jar` file for your application components and `connect.properties` file. For `NamedIterDemo`, the components include `SalesRec.class` as well as the application class and profile.

   You can create a `.jar` file `niter-server.jar` as follows (the 0 is to specify no compression, as required by `loadjava`):

```
jar cvf0 niter-server.jar Named*.class Named*.ser SalesRec.class connect.properties
```

3. Load the `.jar` file into the server.

   Use `loadjava`, as follows. This instructs `loadjava` to use the OCI 8 driver in loading the files. The `-resolve` option results in the class files being resolved.

   ```
   loadjava -oci8 -resolve -force -user scott/tiger niter-server.jar
   ```

4. 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 a trace file; the input parameter is the buffer size in bytes.

5. 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 employs the embedded SQLJ translator in the Oracle8*i* Java VM. This discussion still assumes you created the source on a client machine.

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, discussed below).

When you load `.sqlj` source files into the server, either directly or using a `.jar` file, the resulting database library units 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 and profiles are referred to as Java *class schema objects* (for classes) and Java *resource schema objects* (for resources, primarily profiles). A separate schema object is created for each class and for each profile or other resource.

Class schema objects are created when source is loaded; resource schema objects for profiles are created when source is translated.

Note: When you translate your SQLJ application in the server, profiles are always generated as resources, not classes, because there is no `-ser2class` option in SQLJ server-side translator.

Loading SQLJ Source Code into the Server

Use the Oracle `loadjava` client-side utility on a `.sqlj` file (instead of on `.class` and `.ser` files) 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, compilation, and customization of your application as it is loaded. Otherwise, the source is loaded into a source schema object without any translation. In this case, however, the source *is* implicitly translated, compiled, and customized the first time an attempt is made to use a class that is contained in the source. Such implicit translation may seem surprising at first because there is nothing comparable in client-side SQLJ.

The `loadjava` utility defaults to the JDBC OCI 8 driver, for which you do not specify a URL. Alternatively, you can use the Thin driver by using the `-thin` option and specifying an appropriate URL through the `-user` option, as shown in one of the examples below.

Notes:

- The `-resolve` option directs `loadjava` to resolve external references in each class after all classes on the command line (or in a `.jar` file) are loaded.
- If you enable `-resolve`, `loadjava` will output any translation or compilation error messages to your screen.
- If you do not enable `-resolve` but the source is later translated and compiled implicitly, then translation and compilation error messages are inserted in the user schema `USER_ERRORS` table.
- When you first load a source file, even without `-resolve`, 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.

For example, run `loadjava` as follows:

```
loadjava -user scott/tiger -resolve Foo.sqlj
```

Or, to use the Thin driver to load (specifying the `-thin` option and an appropriate URL):

```
loadjava -thin -user scott/tiger@localhost:1521:ORCL -resolve Foo.sqlj
```

Either of these will result in appropriate class schema objects and resource schema objects being created in addition to the source schema object. For information, see "Generated Output from loadjava and the Server Embedded Translator" on page 11-19.

Before running `loadjava`, however, you must first set SQLJ options appropriately. For more information, see "Option Support in the Server Embedded Translator" on page 11-16. Note that encoding is set on the `loadjava` command line instead of as a SQLJ 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.

For more information about the `loadjava` utility, see the *Oracle8i Enterprise JavaBeans and CORBA Developer's Guide*.

Notes:

- You cannot load a `.sqlj` file along with `.class` files or `.ser` 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 and profiles that it would also be trying to generate. (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.)
- Once you load classes and resources from a source file, you cannot subsequently load the classes and resources directly unless you first drop the source. (See "Dropping Java Schema Objects" on page 11-30.)
- You can put multiple `.sqlj` files into a `.jar` file and specify the `.jar` file to `loadjava`. When you create the `.jar` file, the files it contains must be *uncompressed*. The `loadjava` utility does not support compressed files.
- By default, `loadjava` loads into the login schema specified by the `-user` option. Use the `-schema` option to specify a different schema to load into. This does not involve a login to that schema, but does require that you have sufficient permissions to alter it.

Although the `loadjava` utility is recommended for loading your SQLJ and Java applications into the server, you can also use Oracle 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, compiled, and customized at that time, creating class schema objects and resource 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, compiled, and customized the first time an attempt is made to use a class that is contained in the source. Such implicit translation may seem surprising at first because there is nothing comparable in client-side SQLJ.

See the *Oracle8i SQL Reference* for more information about the `CREATE JAVA` commands.

Note: When you first load a source file, even without `AND COMPILE`, 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`

The encoding Option

This option determines any encoding (for example, `SJIS`) employed 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 or not 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 as discussed in "Loading SQLJ Source Code into the Server" on page 11-13. Any `loadjava` command-line setting for encoding overrides this `encoding` option.

See "Encoding for Input and Output Source Files (-encoding)" on page 8-27 for general information about this option.

See the *Oracle8i Enterprise JavaBeans and CORBA Developer's Guide* for more information about the `loadjava` utility.

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`.

The online Option

Setting this option to `TRUE` (the default value) enables online semantics-checking. Semantics-checking is performed relative to the schema in which the source is loaded.

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 depending on your JDBC driver version and database version. For information about `OracleChecker`, see "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, 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.

The `debug` option is used at the time the source is compiled. If you load it with the `loadjava -resolve` option enabled, this will occur immediately (right after SQLJ translation in the case of a `.sqlj` file). 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).

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 package `DBMS_JAVA`:

- `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, 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`, for example, 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 package `x.y`, then override that for the `Create` source by disabling online checking for that particular source.

Similarly, set encoding for package `x.y` to `SJIS` as follows:

```
sqlplus> execute dbms_java.set_compiler_option('x.y', 'encoding', 'SJIS');
```

Notes:

- 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 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.

Generated Output from loadjava and the Server Embedded Translator

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—a compiled class for each class you defined in the source, a compiled class for each iterator and connection context class, a compiled profile-keys class, and one or more customized profiles.

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 do not declare connection context classes in code that will run in the server)
- a class schema object for the profile-keys class (created by the translator, as on a client, presuming you use SQLJ executable statements in your code)
- a resource schema object for the profile (presumably there is just one profile)

The full names of these schema objects are determined as described in the following subsections.

Notes:

- There are two forms of schema object names in the server: full names and short names. For information about these and about other file naming considerations, see "Full Names vs. Short Names in the Server" on page 11-24.
- 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, whether or not 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 on the command line (so `loadjava` can find it) is irrelevant to the determination of the name of the schema object.

For example, if `Foo.sqlj` defines a class `Foo` in package `x.y` and defines or declares no other classes, then the full name of the resulting source schema object is:

```
x/y/Foo
```

If you define additional classes or declare iterator or connection context classes, then the source schema object is named according to the first class definition or declaration encountered. Whether any of the classes is public is irrelevant in naming the source schema object.

For example, if `Foo.sqlj` is still in package `x.y`, defines class `Bar` first and then class `Foo`, and has no 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
```

If, however, the declaration of 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

Classes 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" on page 11-20. Presume your source code specifies package `x.y`, defines class `Bar` then class `Foo`, then declares 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
```

The profile-keys class is named according to the name of the source schema object, appended by:

```
_SJProfileKeys
```

If the `Bar` definition precedes the `Foo` definition and `MyIter` declaration, then the class schema object for the profile-keys class is named:

```
x/y/Bar_SJProfileKeys
```

If the `MyIter` declaration precedes either of the class definitions, then the profile-keys class schema object is named:

```
x/y/MyIter_SJProfileKeys
```

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, they are named according to the conventions of the standard `javac` compiler.

Full Names of Generated Profiles

Resource schema objects for generated profiles are named in the same way as the profile-keys class schema object—based on the source schema object name, using package and class information from the source code in the same way. Any directory information specified on the command line (the `loadjava` command line, for example) or in a `.jar` file is irrelevant in determining the profile name.

When a source file is loaded and translated, the generated profiles use the source schema object name as a base name, followed by:

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

This is identical to what is appended to produce a profile name on the client.

Using the examples in "Full Name of Source" on page 11-20, where the source schema object was named either `x/y/Foo`, `x/y/Bar`, or `x/y/MyIter` (depending on the situation, as discussed), the name of the profile would be:

```
x/y/Foo_SJProfile0.ser
```

or:

```
x/y/Bar_SJProfile0.ser
```

or:

```
x/y/MyIter_SJProfile0.ser
```

Presume for this example that there is only one profile.

Note: Usually there will be no declared connection context classes, and therefore only one profile, in an application that runs in the server.

Error Output from the Server Embedded Translator

SQLJ error processing in the server is similar to general Java error processing. 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.

If you use `loadjava` to load your SQLJ source, however, `loadjava` also captures and outputs the error messages from the server-side translator.

Informational messages and suppressable 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 "Translator Warnings (-warn)" on page 8-42 for more information about the `-warn` option of the client-side translator.

Additional Steps After Loading Source Files

You can access the USER_OBJECTS view in your database schema to verify that your source file and the generated classes and profiles are loaded properly. See "Checking Java Uploads" on page 11-24 for more information.

Before using your SQLJ code in the server, you must publish your top-level methods as appropriate, as is true with any Java code you use in the server. This involves writing call descriptors, mapping datatypes, and setting parameter modes. For information, see the *Oracle8i Java Stored Procedures Developer's Guide*.

Checking Java Uploads

This section describes how to verify that your Java uploads are properly stored into schema objects. This requires a preliminary discussion of *short names* and *full names* of your Java schema objects in the server.

Full Names vs. Short Names in the Server

As mentioned previously, each Java source, class, and resource is stored in its own schema object in the server. The name of the schema object is derived from the fully qualified name, which includes relevant path or package information. Dots are replaced by slashes, however. These fully qualified names (with slashes)—used for loaded sources, loaded classes, loaded resources, generated classes, and generated resources—are referred to in this chapter as schema object *full names*.

Schema object names, however, have a maximum of only 31 characters, and all characters must be legal and convertible to characters in the database character set. If any full name is longer than 31 characters or contains illegal or inconvertible characters, then the Oracle8*i* server converts the full name to a *short name* to employ 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.

You can always specify a full name to the database by using the `SHORTNAME()` routine of the `DBMS_JAVA` package, which takes a full name as input and returns the corresponding short name. This is useful, for example, in querying the view `USER_OBJECTS`. (See "Using the USER_OBJECTS View" on page 11-24.)

In addition, there is a `LONGNAME()` routine in the `DBMS_JAVA` package, allowing you to specify a short name and have it converted to a full name.

Using the USER_OBJECTS View

You can query the database view `USER_OBJECTS` to obtain information about schema objects—including Java sources, classes, and resources—that you own. This allows you, for example, to verify that sources, classes, or resources that you load are properly stored into schema objects.

Columns in `USER_OBJECTS` include those listed in Table 11-1 below.

Table 11–1 Key USER_OBJECT Columns

Name	Datatype	Description
OBJECT_NAME	VARCHAR2(128)	name of the object
OBJECT_TYPE	VARCHAR2(15)	type of the object (such as JAVA SOURCE, JAVA CLASS, or JAVA RESOURCE)
STATUS	VARCHAR2(7)	status of the object (VALID or INVALID) (always VALID for JAVA RESOURCE)

An OBJECT_NAME in USER_OBJECTS is the full name, unless the full name exceeds 31 characters or contains a character that is illegal or inconvertible to the database character set. In either of these circumstances, a short name is used instead.

If the server has to use a short name for a schema object—meaning that the OBJECT_NAME in USER_OBJECTS is a short name—you can use the LONGNAME() routine of the server DBMS_JAVA package to receive it from a query in full name format without having to know the short name format or the conversion rules. For example:

```
SQL*Plus> SELECT dbms_java.longname(object_name) FROM user_objects
          WHERE object_type='JAVA SOURCE';
```

This will show you all of the Java source schema objects in full name format so you can see if all of your sources were loaded properly. Where no short name is used, no conversion occurs since the short name and full name are identical.

You can use the SHORTNAME() routine of the DBMS_JAVA package to use a full name as a query criterion without having to know whether it was converted to a short name in the database. For example:

```
SQL*Plus> SELECT object_type FROM user_objects
          WHERE object_name=dbms_java.shortname('known_fullname');
```

This will show you the OBJECT_TYPE of the schema object of the specified full name (presuming the full name is representable in the database character set). There will be an implicit conversion to the short name, if necessary. If no short name had to be used, then no conversion occurs because the full name and short name are actually the same in such cases.

STATUS is character string that indicates the validity of a Java source schema object or resource schema object. A source schema object is VALID if it compiled successfully; a class schema object is VALID if it was resolved successfully. A resource schema object is always VALID because resources are not resolved.

Examples: Accessing USER_OBJECTS The following SQL*Plus script accesses the USER_OBJECTS view to display information about uploaded Java sources, classes, and resources.

```
COL object_name format a30
COL object_type format a15
SELECT object_name, object_type, status
   FROM user_objects
   WHERE object_type IN ('JAVA SOURCE', 'JAVA CLASS', 'JAVA RESOURCE')
   ORDER BY object_type, object_name;
```

You can optionally use wildcards in querying USER_OBJECTS, as in the following example.

```
SELECT object_name, object_type, status
   FROM user_objects
   WHERE object_name LIKE '%Alerter';
```

This finds any OBJECT_NAME entries that end with the characters: Alerter

For more information about USER_OBJECTS, see the *Oracle8i Java Stored Procedures Developer's Guide.*

About Class Loading, Compiling, and Name Resolution

Class loading and name resolution in the server follow a very different paradigm than on a client, as the environments themselves are very different. This section discusses that paradigm.

SQL Names vs. 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 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 name `SCOTT.FIZZ` 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 handled 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 may be loaded at any execution site, with reasonable expectation that those classes will be classes that were used to compile the program.

Java Name Resolution

Java name resolution in the Oracle8*i* Java VM involves 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 and Java classes called by Java programs in the server

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 may reference each other recursively.)

When all of 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 pre-linked and ready to run.

A class schema object must be resolved before Java objects of the class can be instantiated or methods of the class can be executed.

Note: The `loadjava` utility resolves references to classes but not to resources. If you translated on the client, be careful how you load any resources into resource schema objects in the server, as discussed in "Naming of Class and Resource Schema Objects" on page 11-8. (You will typically have no resources, only classes, if you used the SQLJ `-ser2class` option.)

Oracle Resolver Specs

Many Java classes contain references to other classes. A conventional Java VM searches for classes in the directories, `.zip` files, and `.jar` files named in the `CLASSPATH`. By contrast, the Oracle8*i* Java VM searches schemas for class schema objects. Each class schema object has a *resolver spec*, which is equivalent to a `CLASSPATH`. The resolver spec for a hypothetical class `Alpha` is a list of schemas to search to find classes used by `Alpha`. Note that resolver specs apply to individual classes, whereas a `CLASSPATH` is global to all classes.

Oracle Resolver

In addition to a resolver spec, each class schema object has a list of interclass reference bindings. Each item in the reference list contains a reference to another class in addition to one of the following:

- the name of the class schema object to invoke when the reference is used
- a code indicating that the reference is unsatisfied; in other words, that the schema object is not known

Reference lists are maintained by an Oracle8*i* facility called the *resolver*. For each interclass reference in a class, the resolver searches the schemas specified by the class's resolver spec and looks for a class schema object that satisfies the reference. If all references are resolved, then the resolver marks the class valid. A class that has never been resolved, or has been resolved unsuccessfully, is marked invalid. A class that may use a schema object that becomes invalid is marked invalid; in other words, invalidation cascades upward from a class to the classes that use it and the classes that use them, and so on.

Resolving with loadjava

You can direct `loadjava` to resolve classes as you load them, or you can defer resolution to runtime. Deferring resolution is not recommended because unsuccessful resolution will likely cause problems for end users when they run your application.

The `loadjava` utility has the following resolution mode (in addition to "defer resolution"):

- Load-then-resolve (`-resolve` option, or `-r`): Load all classes specified on the command line, mark them invalid, and then resolve them. Use this mode when initially loading classes that refer to each other. By loading all classes and then resolving them, this mode avoids the problem of marking a class invalid if it refers to a class that will be loaded later in the execution of the command.

When you use `loadjava` to resolve a class schema object, you can choose one of the following resolver specs:

- Oracle resolver (`-oracleresolver` option)—definer's schema and the `PUBLIC` schema
- JDK resolver (`-jdkresolver` option) (default)—definer's schema, the `PUBLIC` schema, and the current schema at the time the class is resolved
- User-specified resolver (`-resolver` option)

For more information about these `loadjava` options, see the *Oracle8i Enterprise JavaBeans and CORBA Developer's Guide.*

Dropping Java Schema Objects

To complement the `loadjava` utility, Oracle provides the `dropjava` utility to remove (drop) 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 from the database. You can enter `.sqlj`, `.java`, `.class`, `.ser`, and `.jar` files on the command line in any order.

Alternatively, you can specify a schema object name (full name, not short name) directly to `dropjava`. A command-line argument that does not end in `.jar`, `.zip`, `.class`, `.java`, or `.sqlj` is presumed to be a schema object name. If you specify a schema object name that applies to multiple schema objects (such as a source schema object `Foo` and a class schema object `Foo`), all will be removed.

Dropping a class invalidates classes that depend on it, recursively cascading upwards. Dropping a source drops classes and resources that were generated from it.

Note: It is generally advisable to remove Java schema objects in the same way that you first load 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, as follows:

```
dropjava -user scott/tiger Foo.sqlj
```

If you translate your program on the client and load it using a `.jar` file containing the generated components, then use the same `.jar` file 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 (presume there were no iterator classes):

```
dropjava -user scott/tiger Foo*.class dir1/dir2/Foo_SJProfile*.ser
```

Care must be taken if you are removing a resource that was loaded directly into the server. (This includes profiles if you translated on the client without using the `-ser2class` option.) When dropping source or class schema objects, or resource schema objects that were generated by the server-side SQLJ translator, the schema objects will be found according to the package specification in the applicable `.sqlj` source file. However, the fully qualified schema object name of a resource that was generated on the client and loaded directly into the server depends on path information in the `.jar` file or on the command line at the time you loaded it. If you use a `.jar` file to load resources and use the same `.jar` file to remove resources, there will be no problem. If, however, you use the command line to load resources, then you must be careful to specify the same path information when you run `dropjava` to remove the resources. (See "Naming of Class and Resource Schema Objects" on page 11-8 for information about how resource schema objects are named when loaded directly into the server.)

For more information about `dropjava`, see the *Oracle8i Enterprise JavaBeans and CORBA Developer's Guide*.

Additional Considerations

This section discusses Java multithreading in the server and recursive SQLJ calls in the server.

Java Multithreading in the Server

Programs that use Java multithreading will execute in the Oracle8*i* server 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 the Oracle8*i* Java VM, as opposed to in client-side Java VMs:

- 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 the Oracle8*i* server with general Oracle server multithreading. The latter refers to simultaneous database 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 the Oracle server, not by Java.

For general information about Java multithreading in SQLJ, see "Multithreading in SQLJ" on page 7-19.

Recursive SQLJ Calls in the Server

As discussed in "Execution Context Synchronization" on page 7-15, 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.

This functionality would be less desirable in the Oracle server than on a client, however. 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 may 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 may 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 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: Be aware that if `proc2` has any method calls to the execution context instance to change control parameters, then this will affect operations subsequently executed in `proc1`. For this reason, be careful about using `ExecutionContext` control methods in stored procedures and functions or use separate execution context instances.

For information about `ExecutionContext` methods, see "ExecutionContext Methods" on page 7-16.

Verifying that Code is Running in the Server

There is a convenient way to verify that your code is actually running in the server. You can accomplish this by using the standard Java `System.getProperty()` method to retrieve the `oracle.server.version` Java property. If this property contains a version number, then you are running in the Oracle server. If it is `null`, then you are not. Here is an example:

```
...
if (System.getProperty("oracle.server.version") != null
{
   // (running in server)
}
...
```

Additional Vehicles for SQLJ in the Server

Most of the discussion throughout this chapter has presumed that SQLJ is being used for stored procedures or stored functions; there has been no special consideration of any other possibilities. Be aware, though, that you can also use SQLJ in the server in the following ways:

- in Enterprise JavaBeans
- in CORBA server objects

This section introduces the use of Enterprise JavaBeans and CORBA objects. For more information see the *Oracle8i Enterprise JavaBeans and CORBA Developer's Guide*. The `lookup` and `sqljimpl` examples in that manual use interface implementations that were developed with SQLJ.

Note: If your EJB or CORBA object uses any sort of XA transactions (`UserTransaction` in an EJB or JTS in a CORBA object, for example), you cannot use explicit SQLJ commit/rollback statements or JDBC commit/rollback methods. Any attempt to do so will result in an exception. You must instead execute your commits and rollbacks through the particular XA interface that you are using.

Enterprise JavaBeans

To use SQLJ in Enterprise JavaBeans (EJBs), develop and translate the SQLJ EJBs on a client and then load all resulting classes and resources into the server. To load and publish EJBs, however, you must use a utility called `deployejb` (`loadjava` is not used).

When you run the SQLJ translator for your EJB program, consider the following:

- It may be helpful to use the SQLJ `-ser2class` option so that your profiles are converted to `.class` files from `.ser` files. This simplifies the naming of the resulting schema objects in the server, as explained in "Naming of Class and Resource Schema Objects" on page 11-8; however, it prevents you from further customizing the profiles. (To further customize, you must rerun the SQLJ translator and regenerate the profiles.)
- It is also helpful to use the SQLJ `-d` option to direct all generated `.class` files (and `.ser` files, if any) into a specified directory.

Once you have translated your SQLJ EJB, gather everything into a `.jar` file. This includes:

- products of EJB development—`.class` files for home interface, remote interface, bean implementation, and any dependent classes; any required Java resources
- products of SQLJ development and translation—application `.class` files, profile-keys `.class` file, iterator `.class` files, and the profile (either in a `.class` file or a `.ser` file)

  "Summary of Translator Input and Output" on page 1-10 provides a summary of what SQLJ produces. (Note that you would presumably have no connection context classes in a program being loaded into the server because the only connection is to the server itself.)

After creating the `.jar` file, use the `deployejb` utility to load everything into the server, specifying the `.jar` file as input.

CORBA Server Objects

You can also use SQLJ for developing CORBA objects that have database DML statements. As with EJBs, you must be careful to include all classes and resource files that the SQLJ translator generates when you load files into the server. For CORBA objects, load and publish as you would for stored procedures.

Create a `.jar` file to hold all the SQLJ-generated files, as discussed in "Enterprise JavaBeans" on page 11-35, and use this file on the command line when you run `loadjava`.

12 Sample Applications

This chapter presents sample applications that highlight a range of SQLJ features from basic features to advanced features and Oracle extensions, categorized as follows:

- Properties Files
- Basic Samples
- Object, Collection, and CustomDatum Samples
- Advanced Samples
- Server-Side Samples
- JDBC versus SQLJ Sample Code

The examples in this chapter can be found in the following directory and its subdirectories:

```
[Oracle Home]/sqlj/demo
```

Properties files, basic samples, and advanced samples are in the top-level `demo` directory; object and collection samples are in the `demo/Objects` directory; server-side samples are in the `demo/server` directory.

Properties Files

This section consists of two properties files—one for the SQLJ runtime connection and one for translator option settings. These files are located in the following directory:

```
[Oracle Home]/sqlj/demo
```

Runtime Connection Properties File

The sample applications in this chapter use the `Oracle.connect()` method, a convenient way to create an instance of the `DefaultContext` class and establish it as your default connection. This method offers several signatures; the signature used in the samples takes a properties file—`connect.properties`—to specify connection parameters. Here are sample contents of that file:

```
# 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:ORCL
#sqlj.url=jdbc:oracle:oci8:@
#sqlj.url=jdbc:oracle:oci7:@
#
# User name and password here (edit to use different user/password)
sqlj.user=scott
sqlj.password=tiger
```

The version of this file in `[Oracle Home]/sqlj/demo` is configured to use the OCI 8 driver and `scott/tiger` schema. This is appropriate for the sample applications in this chapter, presuming you have a client installation as described in Chapter 2, "Getting Started".

For other uses, you must edit the file appropriately for your particular database connection.

SQLJ Translator Properties File

The following properties file, `sqlj.properties`, is used in translating the SQLJ demo applications. As is, this file will specify offline semantics-checking. To specify online semantics-checking, uncomment the `sqlj.user` entries or add new `sqlj.user` entries as appropriate. An appropriate checker, either offline or online as applicable, will be chosen for you by the default `OracleChecker` class.

For information about SQLJ properties files, see "Properties Files for Option Settings" on page 8-13.

```
###
### Settings to establish a database connection for online checking
###

### turn on checking by uncommenting user
### or specifying the -user option on the command line
#sqlj.user=scott
sqlj.password=tiger

### add additional drivers here
#sqlj.driver=oracle.jdbc.driver.OracleDriver<,driver2...>

### Oracle JDBC-OCI7 URL
#sqlj.url=jdbc:oracle:oci7:@

### Oracle JDBC-OCI8 URL
#sqlj.url=jdbc:oracle:oci8:@

### Oracle JDBC-Thin URL
#sqlj.url=jdbc:oracle:thin:@<host>:<port>:<oracle_sid>
#sqlj.url=jdbc:oracle:thin:@localhost:1521:orcl

# Julie’s database using thin driver
sqlj.url=jdbc:oracle:thin:@dlsun669:5521:815

### Warning settings
###  Note: All settings must be specified TOGETHER on a SINGLE line.

# Report portability warnings about Oracle-specific extensions to SQLJ
#sqlj.warn=portable

# Turn all warnings off
#sqlj.warn=none

# Turn informational messages on
#sqlj.warn=verbose

###
### Online checker
###

### Force Oracle 7.3 specific checker (with Oracle 8.0 JDBC and database):
#sqlj.online=oracle.sqlj.checker.Oracle7JdbcChecker
```

```
### JDBC-generic checker:
#sqlj.online=sqlj.semantics.JdbcChecker

###
### Offline checker
###

#sqlj.cache=on

### Force Oracle 7.3 specific checker (with Oracle 8.0 JDBC):
#sqlj.offline=oracle.sqlj.checker.Oracle7OfflineChecker

###
### Settings for the QueryDemo example
###
### shows how to set options for a particular connection context
###
#sqlj.user@QueryDemoCtx=scott
#sqlj.password@QueryDemoCtx=tiger
#sqlj.url@QueryDemoCtx=jdbc:oracle:oci8:@
#sqlj.url@QueryDemoCtx=jdbc:oracle:oci7:@
#sqlj.url@QueryDemoCtx=jdbc:oracle:thin:@<host>:<port>:<oracle_sid>
```

Basic Samples

This section presents examples that demonstrate some of the basic essentials of SQLJ. These samples are located in the following directory:

```
[Oracle Home]/sqlj/demo
```

Before beginning, connect to the database following the procedures described in "Set Up the Runtime Connection" on page 2-6. Note that this includes creating the following SALES table:

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

Named Iterator—NamedIterDemo.sqlj

This example demonstrates the use of a named iterator.

For information about named iterators (and positional iterators as well), see "Multi-Row Query Results—SQLJ Iterators" on page 3-35.

```
// ------------------ Begin of file NamedIterDemo.sqlj -----------------------
//
// Invoke the SQLJ translator with the following command:
//    sqlj NamedIterDemo.sqlj
// Then run as
//    java NamedIterDemo

/* Import useful classes.
**
** Note that java.sql.Date (and not java.util.Date) is being used.
*/

import java.sql.Date;
import java.sql.SQLException;
import oracle.sqlj.runtime.Oracle;

/* Declare an iterator.
**
** The comma-separated terms appearing in parentheses after the class name
** serve two purposes: they correspond to column names in the query results
** that later occupy instances of this iterator class, and they provide
** names for the accessor methods of the corresponding column data.
**
** The correspondence between the terms and column names is case-insensitive,
** while the correspondence between the terms and the generated accessor names
** is always case-sensitive.
*/

#sql iterator SalesRecs(
      int item_number,
      String item_name,
      Date sales_date,
      double cost,
      Integer sales_rep_number,
      String sales_rep_name );


class NamedIterDemo
{

  public static void main( String args[] )
  {
    try
    {
      NamedIterDemo app = new NamedIterDemo();
      app.runExample();
    }
    catch( SQLException exception )
    {
      System.err.println( "Error running the example: " + exception );
    }
  }


  /* Initialize database connection.
  **
  ** Before any #sql blocks can be executed, a connection to a database
  ** must be established.  The constructor of the application class is a
  ** convenient place to do this, since it is executed once, and only
  ** once, per application instance.
  */

  NamedIterDemo() throws SQLException
  {
```

```
  /* if you’re using a non-Oracle JDBC Driver, add a call here to
     DriverManager.registerDriver() to register your Driver
  */

  // set the default connection to the URL, user, and password
  // specified in your connect.properties file
  Oracle.connect(getClass(), "connect.properties");
}


void runExample() throws SQLException
{
  System.out.println();
  System.out.println( "Running the example." );
  System.out.println();


  /* Reset the database for the demo application.
  */

  #sql { DELETE FROM SALES };


  /* Insert a row into the cleared table.
  */

  #sql
  {
    INSERT INTO SALES VALUES(
          101,’Relativistic redshift recorder’,
          TO_DATE(’22-OCT-1997’,’dd-mon-yyyy’),
          10999.95,
          1,’John Smith’)
  };


  /* Insert another row in the table using bind variables.
  */

  int     itemID = 1001;
  String  itemName = "Left-handed hammer";
  double  totalCost = 79.99;

  Integer salesRepID = new Integer(358);
  String  salesRepName = "Jouni Seppanen";
```

```
Date    dateSold = new Date(97,11,6);

#sql { INSERT INTO SALES VALUES( :itemID,:itemName,:dateSold,:totalCost,
      :salesRepID,:salesRepName) };


/* Instantiate and initialize the iterator.
**
** The iterator object is initialized using the result of a query.
** The query creates a new instance of the iterator and stores it in
** the variable 'sales' of type 'SalesRecs'.  SQLJ translator has
** automatically declared the iterator so that it has methods for
** accessing the rows and columns of the result set.
*/

SalesRecs sales;

#sql sales = { SELECT item_number,item_name,sales_date,cost,
      sales_rep_number,sales_rep_name FROM sales };


/* 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( sales.next() )
{
  System.out.println( "ITEM ID: " + sales.item_number() );
  System.out.println( "ITEM NAME: " + sales.item_name() );
  System.out.println( "COST: " + sales.cost() );
  System.out.println( "SALES DATE: " + sales.sales_date() );
  System.out.println( "SALES REP ID: " + sales.sales_rep_number() );
  System.out.println( "SALES REP NAME: " + sales.sales_rep_name() );
  System.out.println();
}


/* Close the iterator.
**
** Iterators should be closed when you no longer need them.
*/
```

```
    sales.close() ;
  }

}
```

Positional Iterator—PosIterDemo.sqlj

This example demonstrates the use of a positional iterator.

For information about positional iterators (and named iterators as well), see "Multi-Row Query Results—SQLJ Iterators" on page 3-35.

```
// ---------------------- Begin of file PosIterDemo.sqlj ---------------------
//
// Invoke the SQLJ translator as follows:
//      sqlj PosIterDemo.sqlj
// Then run the program using
//      java PosIterDemo


import java.sql.* ;                   // JDBC classes
import oracle.sqlj.runtime.Oracle;    // Oracle class for connecting

/* Declare a ConnectionContext class named PosIterDemoCtx. Instances of this
   class can be used to specify where SQL operations should execute. */
#sql context PosIterDemoCtx;

/* Declare a positional iterator class named FetchSalesIter.*/
#sql iterator FetchSalesIter (int, String, Date, double);

class PosIterDemo {

  private PosIterDemoCtx ctx = null;  // holds the database connection info

  /* The constructor sets up a database connection. */
  public PosIterDemo() {
    try
    {
      /* if you're using a non-Oracle JDBC Driver, add a call here to
         DriverManager.registerDriver() to register your Driver
      */

      // get a context object based on the URL, user, and password
      // specified in your connect.properties file
      ctx =
            new PosIterDemoCtx(Oracle.getConnection(getClass(),
                                "connect.properties"));
    }
    catch (Exception exception)
    { System.err.println (
     "Error setting up database connection: " + exception);
    }
   }

  //Main method
  public static void main (String args[])
  {
    PosIterDemo posIter = new PosIterDemo();

    try
    {
      //Run the example
      posIter.runExample() ;

      //Close the connection
      posIter.ctx.close() ;
    }
    catch (SQLException exception)
    { System.err.println (
      "Error running the example: " + exception ) ;
    }
  } //End of method main


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

    /* Reset the database for the demo application.    */
    #sql [ctx] { DELETE FROM SALES
                   -- Deleting sales rows

               };

    insertSalesRecord
    ( 250, "widget1", new Date(97, 9, 9), 12.00,
      new Integer(218), "John Doe"
    ) ;
    insertSalesRecord
    ( 267, "thing1", new Date(97, 9, 10), 700.00,
      new Integer(218), "John Doe"
    ) ;

    insertSalesRecord
    ( 270, "widget2", new Date(97, 9, 10), 13.00,
      null, "Jane Doe" // Note: Java null is same as SQL null
    ) ;

    System.out.println("Sales records before delete") ;
    printRecords(fetchSales()) ;

    // Now delete some sales records
    Date delete_date;
    #sql [ctx] { SELECT MAX(sales_date) INTO :delete_date
                 FROM SALES };

    #sql [ctx] { DELETE FROM SALES WHERE sales_date = :delete_date };

    System.out.println("Sales records after delete") ;
    printRecords(fetchSales()) ;

  } //End of method runExample


  //Method to select all records from SALES through a positional iterator
  FetchSalesIter fetchSales() throws SQLException {
    FetchSalesIter f;

    #sql [ctx]  f = { SELECT item_number, item_name, sales_date, cost
                      FROM sales };
    return f;
  }

  //Method to print rows using a FetchSalesIter
  void printRecords(FetchSalesIter salesIter) throws SQLException
  {
    int item_number = 0;
    String item_name = null;
    Date sales_date = null;
    double cost = 0.0;

    while (true)
    {
```

```
            #sql { FETCH :salesIter
                   INTO :item_number, :item_name, :sales_date, :cost
                 };
            if (salesIter.endFetch()) break;

            System.out.println("ITEM NUMBER: " + item_number) ;
            System.out.println("ITEM NAME: " + item_name) ;
            System.out.println("SALES DATE: " + sales_date) ;
            System.out.println("COST: " + cost) ;
            System.out.println() ;
          }

          //Close the iterator since we are done with it.
          salesIter.close() ;

        } //End of method runExample


        //Method to insert one row into the database
        void insertSalesRecord(
          int item_number,
          String item_name,
          Date sales_date,
          double cost,
          Integer sales_rep_number,
          String sales_rep_name)

        throws SQLException
        {
          #sql [ctx] {INSERT INTO SALES VALUES
                      (:item_number, :item_name, :sales_date, :cost,
                       :sales_rep_number, :sales_rep_name
                      )
          } ;
        } //End of method insertSalesRecord

      } //End of class PosIterDemo

      //End of file PosIterDemo.sqlj
```

Host Expressions—ExprDemo.sqlj

This example demonstrates the use of host expressions.

For information about host expressions, see "Java Host Expressions, Context Expressions, and Result Expressions" on page 3-15.

```
import java.sql.Date;
import java.sql.SQLException;
import oracle.sqlj.runtime.Oracle;


class ExprDemo
{

  public static void main( String[] arg )
  {
    try
    {
      new ExprDemo().runExample();
    }
    catch( SQLException e )
    {
      System.err.println( "Error running the example: " + e );
    }
  }


  ExprDemo() throws SQLException
  {
    /* if you’re using a non-Oracle JDBC Driver, add a call here to
       DriverManager.registerDriver() to register your Driver
    */

    // set the default connection to the URL, user, and password
    // specified in your connect.properties file
    Oracle.connect(getClass(), "connect.properties");
  }


  int[] array;
  int indx;

  Integer integer;
```

```
class Demo
{
  int field = 0;
}


Demo obj = new Demo();

int total;


void printArray()
{
  System.out.print( "array[0.." + (array.length-1) + "] = { " );

  int i;

  for( i=0;i<array.length;++i )
  {
    System.out.print( array[i] + "," );
  }

  System.out.println( " }" );
}


void printIndex()
{
  System.out.println( "indx = " + indx );
}


void printTotal()
{
  System.out.println( "total = " + total );
}


void printField()
{
  System.out.println( "obj.field = " + obj.field );
}
```

```
void printInteger()
{
  System.out.println( "integer = " + integer );
}


void runExample() throws SQLException
{
  System.out.println();
  System.out.println( "Running the example." );
  System.out.println();

  /*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    Expressions 'indx++' and 'array[indx]' are evaluated in that order.
    Because 'indx++' increments the value of 'indx' from 1 to 2, the
    result will be stored in 'array[2]':

    Suggested Experiments:

     - Try preincrement operator instead of post-increment
     - See what happens if the array index goes out of bounds as a result
       of being manipulated in a host expression

  */

  array = new int[] { 1000,1001,1002,1003,1004,1005 };
  indx = 1;

  #sql { SELECT :(indx++) INTO :(array[indx]) FROM DUAL };

  printArray();

  System.out.println();

  /*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    Expressions 'array[indx]' and 'indx++' are evaluated in that order.
    The array reference is evaluated before the index is incremented,
    and hence the result will be stored in 'array[1]' (compare with the
    previous example):

  */

  array = new int[] { 1000,1001,1002,1003,1004,1005 };
```

```
indx = 1;

#sql { SET :(array[indx]) = :(indx++) };

printArray();

System.out.println();

/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  Expressions ’x.field’ and ’y.field’ both refer to the same variable,
  ’obj.field’.  If an attempt is made to assign more than one results
  in what is only one storage location, then only the last assignment
  will remain in effect (so in this example ’obj.field’ will contain 2
  after the execution of the SQL statement):

*/

Demo x = obj;
Demo y = obj;

#sql { SELECT :(1), :(2) INTO :(x.field), :(y.field) FROM DUAL };

printField();

System.out.println();

/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  All expressions are evaluated before any are assigned.  In this
  example the ’indx’ that appears in the second assignment will be
  evaluated before any of the assignments take place.  In particular,
  when ’indx’ is being used to assign to ’total’, its value has not
  yet been assigned to be 100.

  The following warning may be generated, depending on the settings
  of the SQLJ translator:

    Warning: Repeated host item indx in positions 1 and 3 in SQL
    block. Behavior is vendor-defined and non portable.

*/

indx = 1;
total = 0;
```

```
#sql
{
  BEGIN
    :OUT indx := 100;
    :OUT total := :IN (indx);
  END;
};

printIndex();
printTotal();

System.out.println();

/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  Expression 'indx++' in the following example is evaluated exactly
  once, despite appearing inside a SQL loop construct.  Its old value
  before increment is used repeatedly inside the loop, and its value
  is incremented only once, to 2.

*/

indx = 1;
total = 0;

#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;
};

printIndex();
printTotal();
```

```
System.out.println();

/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  In the next example there are two assignments to the same variable,
  each inside a different branch of an SQL ’if..then..else..end if’
  construct, so that only one of those will be actually executed at
  run-time.  However, assignments to OUT variable are always carried
  out, regardless of whether the SQL code that manipulates the return
  value has been executed or not.

  In the following example, only the first assignment is executed by
  the SQL; the second assignment is not executed.  When the control
  returns to Java from the SQL statement, the Java variable ’integer’
  is assigned twice: first with the value ’1’ it receives from the
  first SQL assignment, then with a ’null’ value it receives from the
  second assignment that is never executed.  Because the assignments
  occur in this order, the final value of ’integer’ after executing
  this SQL statement is undefined.

  The following warning may be generated, depending on the settings
  of the SQLJ translator:

    Warning: Repeated host item indx in positions 1 and 3 in SQL
    block. Behavior is vendor-defined and non portable.

  Suggested experiments:

   - Use a different OUT-variable in the ’else’-branch
   - Vary the condition so that the ’else’-branch gets executed

*/

integer = new Integer(0);

#sql
{
  BEGIN
    IF 1 > 0 THEN
      :OUT integer := 1;
    ELSE
      :OUT integer := 2;
    END IF;
  END;
};
```

```
    printInteger();

    System.out.println();

    /*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    */

  }
}
```

Object, Collection, and CustomDatum Samples

This section has examples that feature support of user-defined objects and collections and general use of the `oracle.sql.CustomDatum` interface. (This interface is discussed in "About Custom Java Classes and the CustomDatum Interface" on page 6-6.)

The object and collection samples are located in the following directory:

```
[Oracle Home]/sqlj/demo/Objects
```

Definition of Object and Collection Types

The following SQL script defines Oracle object types, Oracle collection types, and tables used in the object and collection sample applications below. In particular, it defines the following:

- object types `PERSON` and `ADDRESS` for the objects demo
- object types `MODULE_T` and `PARTICIPANT_T` for the nested tables demos
- nested table type `MODULETBL_T`
- VARRAY type `PHONE_ARRAY`

```
/*** Using UDTs in SQLJ ***/
SET ECHO ON;

/*** Clean up ***/
DROP TABLE EMPLOYEES
/
DROP TABLE PERSONS
/
DROP TABLE projects
/
DROP TABLE participants
/
DROP TYPE PHONE_ARRAY FORCE
/
DROP TYPE PERSON FORCE
/
DROP TYPE ADDRESS FORCE
/
DROP TYPE moduletbl_t FORCE
/
DROP TYPE module_t FORCE
```

```
/
DROP TYPE participant_t FORCE
/

/*** Create an address ADT ***/
CREATE TYPE address AS OBJECT
(
  street        VARCHAR(60),
  city          VARCHAR(30),
  state         CHAR(2),
  zip_code      CHAR(5)
)
/
show errors

/*** Create a person ADT containing an embedded Address ADT ***/
CREATE TYPE person AS OBJECT
(
  name    VARCHAR(30),
  ssn     NUMBER,
  addr    address
)
/
show errors

/*** Create a typed table for person objects ***/
CREATE TABLE persons OF person
/
show errors
CREATE TYPE PHONE_ARRAY IS VARRAY (10) OF varchar2(30)
/
show errors
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 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,
  phone_nums           phone_array
)
/
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 ;

CREATE TABLE participants  OF participant_t ;

/*** Now let’s put in some sample data
     Insert 2 objects into the persons typed table ***/

INSERT INTO persons VALUES (
            person(’Wolfgang Amadeus Mozart’, 123456,
            address(’Am Berg 100’, ’Salzburg’, ’AU’,’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, phone_nums) VALUES
            (1001,
             address('500 Oracle Parkway', 'Redwood City', 'CA', '94065'),
             50000,
             phone_array('(408) 555-1212', '(650) 555-9999'));
/

/** 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')
/

/* now we insert data into the PARTICIPANTS and PROJECTS tables */
INSERT INTO participants VALUES (
participant_T(7369,'ALAN SMITH','ANALYST',7902,
to_date('17-12-1980','dd-mm-yyyy'),800,20)) ;
INSERT INTO participants VALUES (
participant_t(7499,'ALLEN TOWNSEND','ANALYST',7698,
to_date('20-2-1981','dd-mm-yyyy'),1600,30));
INSERT INTO participants VALUES (
participant_t(7521,'DAVID WARD','MANAGER',7698,
to_date('22-2-1981','dd-mm-yyyy'),1250,30));
INSERT INTO participants VALUES (
participant_t(7566,'MATHEW JONES','MANAGER',7839,
to_date('2-4-1981','dd-mm-yyyy'),2975,20));
INSERT INTO participants VALUES (
participant_t(7654,'JOE MARTIN','MANAGER',7698,
to_date('28-9-1981','dd-mm-yyyy'),1250,30));
INSERT INTO participants VALUES (
participant_t(7698,'PAUL JONES','Director',7839,
to_date('1-5-1981','dd-mm-yyyy'),2850,30));
INSERT INTO participants VALUES (
participant_t(7782,'WILLIAM CLARK','MANAGER',7839,
to_date('9-6-1981','dd-mm-yyyy'),2450,10));
INSERT INTO participants VALUES (
participant_t(7788,'SCOTT MANDELSON','ANALYST',7566,
to_date('13-JUL-87','dd-mm-yy')-85,3000,20));
INSERT INTO participants VALUES (
participant_t(7839,'TOM KING','PRESIDENT',NULL,
to_date('17-11-1981','dd-mm-yyyy'),5000,10));
INSERT INTO participants VALUES (
participant_t(7844,'MARY TURNER','SR MANAGER',7698,
to_date('8-9-1981','dd-mm-yyyy'),1500,30));
INSERT INTO participants VALUES (
participant_t(7876,'JULIE ADAMS','SR ANALYST',7788,
to_date('13-JUL-87', 'dd-mm-yy')-51,1100,20));
INSERT INTO participants VALUES (
participant_t(7900,'PAMELA JAMES','SR ANALYST',7698,
to_date('3-12-1981','dd-mm-yyyy'),950,30));
INSERT INTO participants VALUES (
participant_t(7902,'ANDY FORD','ANALYST',7566,
to_date('3-12-1981','dd-mm-yyyy'),3000,20));
INSERT INTO participants VALUES (
participant_t(7934,'CHRIS MILLER','SR ANALYST',7782,
to_date('23-1-1982','dd-mm-yyyy'),1300,10));


INSERT INTO projects VALUES ( 101, 'Emarald', null, '10-JAN-98',  300,
     moduletbl_t( module_t ( 1011 , 'Market Analysis', null, '01-JAN-98', 100),
                module_t ( 1012 , 'Forecast', null, '05-FEB-98',20) ,
                module_t ( 1013 , 'Advertisement', null, '15-MAR-98', 50),
                module_t ( 1014 , 'Preview', null, '15-MAR-98',44),
                module_t ( 1015 , 'Release', null,'12-MAY-98',34) ) ) ;

update projects set owner=(select ref(p) from participants p where p.empno =
7839) where id=101 ;

update table ( select modules from projects a where a.id = 101 )
set  module_owner = ( select ref(p) from participants p where p.empno = 7844)
where module_id = 1011 ;

update table ( select modules from projects where id = 101 )
set  module_owner = ( select ref(p) from participants p where p.empno = 7934)
where module_id = 1012 ;

update table ( select modules from projects where id = 101 )
set  module_owner = ( select ref(p) from participants p where p.empno = 7902)
where module_id = 1013 ;

update table ( select modules from projects where id = 101 )
set  module_owner = ( select ref(p) from participants p where p.empno = 7876)
where module_id = 1014 ;
```

```
update table ( select modules from projects where id = 101 )
set  module_owner = ( select ref(p) from participants p where p.empno = 7788)
where module_id = 1015 ;

INSERT INTO projects VALUES ( 500, 'Diamond', null, '15-FEB-98', 555,
       moduletbl_t ( module_t ( 5001 , 'Manufacturing', null, '01-MAR-98', 120),
                   module_t ( 5002 , 'Production', null, '01-APR-98',100),
                   module_t ( 5003 , 'Materials', null, '01-MAY-98',200) ,
                   module_t ( 5004 , 'Marketing', null, '01-JUN-98',10) ,
                   module_t ( 5005 , 'Materials', null, '15-FEB-99',50),
                   module_t ( 5006 , 'Finance ', null, '16-FEB-99',12),
                   module_t ( 5007 , 'Budgets', null, '10-MAR-99',45))) ;

update projects set owner=(select ref(p) from participants p where p.empno =
7698) where id=500 ;

update table ( select modules from projects where id = 500 )
set  module_owner = ( select ref(p) from participants p where p.empno = 7369)
where module_id = 5001 ;

update table ( select modules from projects where id = 500 )
set  module_owner = ( select ref(p) from participants p where p.empno = 7499)
where module_id = 5002 ;

update table ( select modules from projects where id = 500 )
set  module_owner = ( select ref(p) from participants p where p.empno = 7521)
where module_id = 5004 ;

update table ( select modules from projects where id = 500 )
set  module_owner = ( select ref(p) from participants p where p.empno = 7566)
where module_id = 5005 ;

update table ( select modules from projects where id = 500 )
set  module_owner = ( select ref(p) from participants p where p.empno = 7654)
where module_id = 5007 ;

COMMIT
/
QUIT
```

Oracle Objects—ObjectDemo.sqlj

Following is the `ObjectDemo.sqlj` source code. This uses definitions from the SQL script in "Definition of Object and Collection Types" on page 12-20.

Use of objects is discussed in "Strongly Typed Objects and References in SQLJ Executable Statements" on page 6-41.

```
import java.sql.SQLException;
import java.sql.DriverManager;
import java.math.BigDecimal;
import oracle.sqlj.runtime.Oracle;

public class ObjectDemo
{

/* Global variables */

static String uid      = "scott";               /* user id */
static String password = "tiger";               /* password */
static String url      = "jdbc:oracle:oci8:@"; /* Oracle's OCI8 driver */

public static void main(String [] args)
{

  System.out.println("*** SQLJ OBJECT DEMO ***");

  try {

    /* if you're using a non-Oracle JDBC Driver, add a call here to
       DriverManager.registerDriver() to register your Driver
    */

    /* Connect to the database */
    Oracle.connect(ObjectDemo.class, "connect.properties");


  /* DML operations on single objects */

        selectAttributes(); /* Select Person attributes */
        updateAttributes(); /* Update Address attributes */

        selectObject();     /* Select a person object */
        insertObject();     /* Insert a new person object */
        updateObject();     /* Update an address object */
        selectRef();        /* Select Person objects via REFs */
        updateRef();        /* Update Person objects via REFs */

        #sql { rollback work };

  }
  catch (SQLException exn)
  {
     System.out.println("SQLException: "+exn);
  }
  finally
  {
    try
    {
      #sql { rollback work };
    }
    catch (SQLException exn)
    {
      System.out.println("Unable to roll back: "+exn);
    }
  }

  System.out.println("*** END OF SQLJ OBJECT DEMO ***");
}

/**
Iterator for selecting a person's data.
*/

#sql static iterator PData (String name, String address, int ssn);


/**
Selecting individual attributes of objects
*/
static void selectAttributes()

{
  /*
   Select individual scalar attributes of a person object
   into host types such as int, String
   */

        String name;
        String address;
        int ssn;

        PData iter;

        System.out.println("Selecting person attributes.");
  try {
    #sql iter =
    {
    select p.name as "name", p.ssn as "ssn",
           p.addr.street || ', ' || p.addr.city
                         || ', ' || p.addr.state
                         || ', ' || p.addr.zip_code as "address"
    from persons p
    where p.addr.state = 'AU' OR p.addr.state = 'CA' };

    while (iter.next())
    {
        System.out.println("Selected person attributes:\n" +
        "name = "    + iter.name() + "\n" +
        "ssn = "     + iter.ssn()  + "\n" +
        "address = " + iter.address() );
    }
  } catch (SQLException exn) {
    System.out.println("SELECT failed with "+exn);
  }
}


/**
Updating individual attributes of an object
*/

static void updateAttributes()
{

  /*
   * Update a person object to have a new address.  This example
   * illustrates the use of constructors in SQL to create object types
   * from scalars.
   */

  String name       = "Ludwig van Beethoven";
  String new_street = "New Street";
  String new_city   = "New City";
  String new_state  = "WA";
  String new_zip    = "53241";

  System.out.println("Updating person attributes..");

  try { #sql {

        update persons
        set addr = Address(:new_street, :new_city, :new_state, :new_zip)
        where name = :name };

        System.out.println("Updated address attribute of person.");

      } catch (SQLException exn) {

        System.out.println("UPDATE failed with "+exn);
  }
}


/**
Selecting an object
*/

static void selectObject()
{
    /*
     * When selecting an object from a typed table like persons
     * (as opposed to an object column in a relational table, e.g.,
     * office_addr in table employees), you have to use the VALUE
     * function with a table alias.
     */

    Person p;
    System.out.println("Selecting the Ludwig van Beethoven person object.");

    try { #sql {
            select value(p) into :p
               from persons p
            where p.addr.state = 'WA'  AND p.name = 'Ludwig van Beethoven' };

    printPersonDetails(p);

    /*
     * Memory for the person object was automatically allocated,
     * and it will be automatically garbage collected when this
```

```
      * method returns.
      */

    } catch (SQLException exn) {
      System.out.println("SELECT failed with "+exn);
    }
    catch (Exception exn)
    {
       System.out.println("An error occurred");
       exn.printStackTrace();
    }
}

/**
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
```

```
*/

static void updateObject()
{

        Address addr;
        Address new_addr;
        int empno = 1001;

try {
     #sql {
         select office_addr
         into :addr
         from employees
         where empnumber = :empno };
  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 = :empno };

       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 = :empno };
```

```
          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);
  }


    /* No need to free anything explicitly. */

  }

  /**
  Selecting an object via a REF
  */

  static void selectRef()
  {

    String name = "Ludwig van Beethoven";
    Person mgr;

    System.out.println("Selecting manager of "+name+" via a REF.");

    try {
       #sql {
        select deref(manager)
        into :mgr
        from employees e
        where e.person_data.name = :name
    } ;

       System.out.println("Current manager of "+name+":");
       printPersonDetails(mgr);

    } catch (SQLException exn) {
       System.out.println("SELECT REF failed with "+exn); }
  }


  /**
  Updating a REF to an object
  */

  static void updateRef()
  {

    int empno = 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 = :empno };

      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 = :empno
    } ;

       System.out.println("Current manager of "+empno+":");
       printPersonDetails(manager);

    } catch (SQLException exn) {
       System.out.println("SELECT REF failed with "+exn); }

  }
```

```
/**
Utility functions
*/

/**** Print the attributes of a person object ****/
   static void printPersonDetails(Person p) throws SQLException

   {
       if (p == null) {
       System.out.println("NULL Person");
       return;

       }

       System.out.print("Person ");
       System.out.print( (p.getName()==null) ? "NULL name" : p.getName() );
       System.out.print( ", SSN=" + ((p.getSsn()==null) ? "-1" :
                    p.getSsn().toString()) );
       System.out.println(":");
       printAddressDetails(p.getAddr());
   }

/**** Print the attributes of an address object ****/

   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 + "'\n" +
                        "City:   '" + city   + "'\n" +
                        "State:  '" + state  + "'\n" +
                        "Zip:    '" + zip_code + "'" );
}

/**** Populate a person object with data ****/
```

```
static Person createPersonData(int i) throws SQLException
{
   Person p = new Person();

   /* create and load the dummy data into the person */
   p.setName("Person " + i);
   p.setSsn(new BigDecimal(100000 + 10 * i));

   Address a = new Address();
   p.setAddr(a);
   a.setStreet("Street " + i);
   a.setCity("City " + i);
   a.setState("S" + i);
   a.setZipCode("Zip"+i);

  /* Illustrate NULL values for objects and individual attributes */

  if (i == 2)
    {
      /* Pick this person to have a NULL ssn and a NULL address */
      p.setSsn(null);
      p.setAddr(null);
    }
  return p;
}

}
```

Oracle Nested Tables—NestedDemo1.sqlj and NestedDemo2.sqlj

Following is the source code for `NestedDemo1.sqlj` and `NestedDemo2.sqlj`. These use definitions from the SQL script in "Definition of Object and Collection Types" on page 12-20.

Use of nested tables is discussed in "Strongly Typed Collections in SQLJ Executable Statements" on page 6-47.

NestedDemo1.sqlj

```
// --------------Begin of NestedDemo1.sqlj -------------------------

// Import Useful classes
```

```
import sqlj.runtime.*;
import sqlj.runtime.ref.*;
import java.sql.*;
import oracle.sql.* ;
import oracle.sqlj.runtime.Oracle;

public class NestedDemo1
{
 //  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 below:

  #sql public static iterator ProjIter(int id,
                                       String name,
                                       String owner,
                                       Date start_date,
                                       ModuleIter modules);

   public static void main(String[] args)
   {
     try {
       /* if you're using a non-Oracle JDBC Driver, add a call here to
          DriverManager.registerDriver() to register your Driver
       */

       /* Connect to the database */
       Oracle.connect(NestedDemo1.class, "connect.properties");

       listAllProjects();  // uses named iterator
     } catch (Exception e) {
          System.err.println( "Error running ProjDemo: " + e );
     }
   }


   public static void listAllProjects() throws SQLException
   {
     System.out.println("Listing projects...");
```

```
      // Instantiate and initilaise 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 below 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();
    }
  }
```

NestedDemo2.sqlj

```
// -------------Begin of NestedDemo2.sqlj ------------------------
// Demonstrate DML on Nested Tables in SQLJ
// Import Useful classes

import sqlj.runtime.*;
import sqlj.runtime.ref.*;
import java.sql.*;
import oracle.sql.*;
import oracle.sqlj.runtime.Oracle;


public class NestedDemo2
{
   #sql public static iterator ModIter(int, String, String) ;

   static ModuletblT mymodules=null;

   public static void main(String[] args)
   {
     try {
       /* if you're using a non-Oracle JDBC Driver, add a call here to
          DriverManager.registerDriver() to register your Driver
       */

       /* get connect to the database */
       Oracle.connect(NestedDemo2.class, "connect.properties");

       cleanupPreviousRuns();
       /*
       // insert new project into Projects table
       // get the owner details from 'participant'
       */

       String ProjName ="My project";
       int projid = 123;
       String Owner = "MARY TURNER";
       insertProject(projid, ProjName, Owner); // insert new project

       /*
       // Insert another Project
       // Both project details and Nested table details are inserted
       */
       projid = 600;
       insertProject2(projid);
       /* Insert a new module for the above project  */
       insertModules(projid);

       /* Update the nested table row */
       projid=600;
       String moduleName = "Module 1";
       String setownerto = "JULIE ADAMS";
       assignModule(projid, moduleName, setownerto);

      /* delete all the modules for the given project
      // which are unassigned
      */

       projid=600;
       deleteUnownedModules(projid);

      /* Display Modules  for 500 project */

       getModules(500) ;

       // Example to use nested table as host variable using a
       // JPub-generated SQL 'Array' type

       getModules2(600);


     } catch (Exception e) {
          System.err.println( "Error running ProjDemo: " + e );
     }
   }

  /* insertProject
  // inserts  into projects table
  */

  public static void insertProject(int id, String projectName, String ownerName)
        throws SQLException
  {
    System.out.println("Inserting Project '" + id + " "+projectName +
         "'  owner is '" + ownerName + "'");

    try {
      #sql { INSERT INTO Projects(id, name,owner,start_date,duration)
             SELECT :id, :projectName, ref(p), '12-JAN-97', 30
```

```
          FROM participants p WHERE ename = :ownerName };

  } catch ( Exception e) {
    System.out.println("Error:insertProject");
    e.printStackTrace();
  }
}

/* insert Project 2
// 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();
  }
}

/* insertModules
// Illustrates accessing the nested table using the TABLE construct
*/
public static void insertModules(int projId)  throws Exception
{
  System.out.println("Inserting Module 6009 for Project " + projId);
  try {
    #sql { INSERT INTO TABLE(SELECT modules FROM projects
                             WHERE id = :projId)
           VALUES (6009,'Module 1', null, '12-JAN-97', 10)};

  } catch(Exception e) {
    System.out.println("Error:insertModules");
    e.printStackTrace();
  }
}

/* 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();
    }
  }

  public static void getModules(int projId)
  throws Exception
  {
    System.out.println("Display modules for project " + projId ) ;

    try {
      ModIter  miter1 ;
      #sql miter1={SELECT m.module_id, m.module_name, m.module_owner.ename
                   FROM TABLE(SELECT modules
                              FROM projects WHERE id=:projId) m };
        int mid=0;
        String mname =null;
        String mowner =null;
      while (miter1.next())
      {
        #sql { FETCH :miter1 INTO :mid, :mname, :mowner } ;
        System.out.println ( mid + " " + mname + " "+mowner) ;
      }
    } catch(Exception e) {
      System.out.println("Error:getModules");
      e.printStackTrace();
    }
  }

  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();
    }
  }
  /* clean up database from any previous runs of this program */
  private static void cleanupPreviousRuns()
  {
    try {
      #sql {delete from projects where id in (123, 600)};
    } catch (Exception e) {
      System.out.println("Exception at cleanup time!") ;
      e.printStackTrace();
    }
  }
}
```

Oracle VARRAYs—VarrayDemo1.sqlj and VarrayDemo2.sqlj

Following is the source code for `VarrayDemo1.sqlj` and `VarrayDemo2.sqlj`. These use definitions from the SQL script in "Definition of Object and Collection Types" on page 12-20.

Use of VARRAYs is discussed in "Strongly Typed Collections in SQLJ Executable Statements" on page 6-47.

VarrayDemo1.sqlj

```
import java.sql.SQLException;
import java.sql.DriverManager;
import java.math.BigDecimal;
import oracle.sqlj.runtime.Oracle;

public class VarrayDemo1
{

/* Global variables */

static String uid      = "scott";              /* user id */
static String password = "tiger";              /* password */
static String url      = "jdbc:oracle:oci8:@"; /* Oracle's OCI8 driver */

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

  System.out.println("*** SQLJ VARRAY DEMO #1 ***");

  try {

    /* if you're using a non-Oracle JDBC Driver, add a call here to
       DriverManager.registerDriver() to register your Driver
    */

    /* Connect to the database */
    Oracle.connect(VarrayDemo1.class, "connect.properties");

    /* create a new VARRAY object and insert it into the DBMS */
    insertVarray();

    /* get the VARRAY object and print it */
    selectVarray();

  }
  catch (SQLException exn)
  {
     System.out.println("SQLException: "+exn);
  }
  finally
  {
    try
    {
      #sql { rollback work };
    }
    catch (SQLException exn)
    {
      System.out.println("Unable to roll back: "+exn);
    }
  }

  System.out.println("*** END OF SQLJ VARRAY DEMO #1 ***");
}

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]);

}

// 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);
}
```

```
}
```

VarrayDemo2.sqlj

```
import java.sql.SQLException;
import java.sql.DriverManager;
import java.math.BigDecimal;
import oracle.sqlj.runtime.Oracle;

#sql iterator StringIter (String s);
#sql iterator intIter(int value);

public class VarrayDemo2
{

/* Global variables */

static String uid      = "scott";              /* user id */
static String password = "tiger";              /* password */
static String url      = "jdbc:oracle:oci8:@"; /* Oracle's OCI8 driver */


public static void main(String [] args) throws SQLException
{
  System.out.println("*** SQLJ VARRAY  DEMO #2 ***");

  try {

    StringIter si = null;

    /* if you're using a non-Oracle JDBC Driver, add a call here to
       DriverManager.registerDriver() to register your Driver
    */

    /* Connect to the database */
    Oracle.connect(VarrayDemo2.class, "connect.properties");

    #sql si = {select column_value s from
               table(select phone_nums from employees where empnumber=1001)};

    while(si.next())
      System.out.println(si.s());
  }
  catch (SQLException exn)
  {
     System.out.println("SQLException: "+exn);
  }
  finally
  {
    try
    {
      #sql { rollback work };
    }
    catch (SQLException exn)
    {
      System.out.println("Unable to roll back: "+exn);
    }
  }

  System.out.println("*** END OF SQLJ VARRAY DEMO #2 ***");
}
}
```

General Use of CustomDatum—BetterDate.java

This example shows a class that implements the `CustomDatum` interface to provide a customized representation of Java dates.

Note: This is not a complete application—there is no `main()`.

```
import java.util.Date;
import oracle.sql.CustomDatum;
import oracle.sql.DATE;
import oracle.sql.CustomDatumFactory;
import oracle.jdbc.driver.OracleTypes;

// a Date class customized for user's preferences:
//      - months are numbers 1..12, not 0..11
//      - years are referred to via four-digit numbers, not two.

public class BetterDate extends java.util.Date
             implements CustomDatum, CustomDatumFactory {
  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 CustomDatumFactory getFactory() { return _BetterDateFactory;}

  // the current time...
  public BetterDate() {
    super();
  }

  public oracle.sql.Datum toDatum(oracle.jdbc.driver.OracleConnection conn) {
    return new DATE(toSQLDate());
  }

  public oracle.sql.CustomDatum 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 years 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 above) 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();
  }
}
```

Advanced Samples

This section presents examples that demonstrate some of the more advanced features of SQLJ. These samples are located in the following directory:

`[Oracle Home]/sqlj/demo`

REF CURSOR—RefCursDemo.sqlj

This example shows the use of a `REF CURSOR` type in an anonymous block, a stored procedure, and a stored function.

The PL/SQL code used to create the procedure and function is also shown.

For information about `REF CURSOR` types, see "Support for Oracle REF CURSOR Types" on page 5-33.

Definition of REF CURSOR Stored Procedure and Stored Function

This section contains the PL/SQL code that defines the following:

- a stored procedure that returns a `REF CURSOR` type as an `OUT` parameter
- a stored function that returns a `REF CURSOR` type as a result

```
create or replace package SQLJREFCURSDEMO as
  type EmpCursor is ref cursor;
  procedure REFCURSPROC( name VARCHAR,
                         no NUMBER,
                         empcur OUT EmpCursor);

  function REFCURSFUNC (name VARCHAR, no NUMBER)  return EmpCursor;

end SQLJREFCURSDEMO;
/

create or replace package body SQLJREFCURSDEMO is

  procedure REFCURSPROC( name VARCHAR,
                         no NUMBER,
                         empcur OUT EmpCursor)
     is begin
       insert into emp (ename, empno) values (name, no);
       open empcur for select ename, empno from emp
                       order by empno;
```

```
      end;

  function REFCURSFUNC (name VARCHAR, no NUMBER) return EmpCursor  is
    empcur EmpCursor;
    begin
       insert into emp (ename, empno) values (name, no);
       open empcur for select ename, empno from emp
                       order by empno;
       return empcur;
     end;
end SQLJREFCURSDEMO;
/
exit
/
```

REF CURSOR Sample Application Source Code

This application retrieves a REF CURSOR type from the following:

- an anonymous block
- a stored procedure (as an OUT parameter)
- a stored function (as a return value)

```
import java.sql.*;
import oracle.sqlj.runtime.Oracle;

public class RefCursDemo
{
  #sql public static iterator EmpIter (String ename, int empno);

  public static void main (String argv[]) throws SQLException
  {
    String name;  int no;
    EmpIter emps = null;

    dbConnect();

    name = "Joe Doe"; no = 8100;
    emps = refCursInAnonBlock(name, no);
    printEmps(emps);

    name = "Jane Doe"; no = 8200;
    emps = refCursInStoredProc(name, no);
    printEmps(emps);

    name = "Bill Smith"; no = 8300;
    emps = refCursInStoredFunc(name, no);
    printEmps(emps);
   }

  private static void dbConnect() throws SQLException {
    // set the default connection to the URL, user, and password
    // specified in your connect.properties file
    Oracle.connect(RefCursDemo.class, "connect.properties");
  }


  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;
  }

  private static EmpIter refCursInStoredProc (String name, int no)
    throws java.sql.SQLException {
    EmpIter emps = null;
    System.out.println("Using stored procedure for ref cursor..");
    #sql { call SQLJREFCURSDEMO.REFCURSPROC (:IN name, :IN no, :OUT emps)
         };
    return emps;
  }

  private static EmpIter refCursInStoredFunc (String name, int no)
    throws java.sql.SQLException {
    EmpIter emps = null;
    System.out.println("Using stored function for ref cursor..");
    #sql emps = {  VALUES (SQLJREFCURSDEMO.REFCURSFUNC(:name, :no))
                };
    return emps;
  }
```

```
  private static void printEmps(EmpIter emps)
    throws java.sql.SQLException {
    System.out.println("Employee list:");
    while (emps.next()) {
       System.out.println("\t Employee name: " + emps.ename() +
                          ", id : " + emps.empno());
    }
    System.out.println();
    emps.close();
  }
}
```

Multithreading—MultiThreadDemo.sqlj

The following is an example of a SQLJ application using multithreading. See "Multithreading in SQLJ" on page 7-19 for information about multithreading considerations in SQLJ.

```
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 {
      /* if you're using a non-Oracle JDBC Driver, add a call here to
         DriverManager.registerDriver() to register your Driver
      */

      // 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);
    }
  }

  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) {}
  }
}
```

Interoperability with JDBC—JDBCInteropDemo.sqlj

The following example uses JDBC to perform a dynamic query, casts the JDBC result set to a SQLJ iterator, and uses the iterator to view the results. It demonstrates how SQLJ and JDBC can interoperate in the same program.

For information about SQLJ-JDBC interoperability, see "SQLJ and JDBC Interoperability" on page 7-27.

```
import java.sql.*;
import sqlj.runtime.ref.DefaultContext;
import oracle.sqlj.runtime.Oracle;

public class JDBCInteropDemo
{
  // in this example, we use an iterator that is inner class
  #sql public static iterator Employees ( String ename, double sal ) ;

  public static void main(String[] args) throws SQLException
  {
    if (args.length != 1) {
      System.out.println("usage: JDBCInteropDemo <whereClause>");
      System.exit(1);
    }

    /* if you're using a non-Oracle JDBC Driver, add a call here to
       DriverManager.registerDriver() to register your Driver
    */

    // set the default connection to the URL, user, and password
    // specified in your connect.properties file
    Oracle.connect(JDBCInteropDemo.class, "connect.properties");
    Connection conn = DefaultContext.getDefaultContext().getConnection();

    // create a JDBCStatement object to execute a dynamic query
    Statement stmt = conn.createStatement();
    String query = "SELECT ename, sal FROM emp WHERE ";
    query += args[0];
    // use the result set returned by executing the query to create
    // a new strongly-typed SQLJ iterator
    ResultSet rs = stmt.executeQuery(query);
    Employees emps;
    #sql emps = { CAST :rs };

    while (emps.next()) {
      System.out.println(emps.ename() + " earns " + emps.sal());
    }
    emps.close();
    stmt.close();
  }
}
```

Multiple Connection Schemas—MultiSchemaDemo.sqlj

The following is an example of a SQLJ application using multiple connection contexts. It implicitly uses an object of the `DefaultContext` class for one type of schema and uses an instance of the declared connection context class `DeptContext` for another type of schema.

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. This example is repeated in "Connection Contexts" on page 7-2. You can refer to that section for information about multiple and non-default connection contexts.

```
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
  {
    /* if you're using a non-Oracle JDBC Driver, add a call here to
       DriverManager.registerDriver() to register your Driver
```

```
      */

      // 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();
    }
  }
```

Data Manipulation and Multiple Connection Contexts—QueryDemo.sqlj

This demo demonstrates constructs that you can use to fetch rows of data using SQLJ and also shows the use of multiple connection contexts.

```
import java.sql.SQLException;
import oracle.sqlj.runtime.Oracle;
import sqlj.runtime.ref.DefaultContext;

#sql context QueryDemoCtx ;

#sql iterator SalByName (double sal, String ename) ;

#sql iterator SalByPos (double, String ) ;

/**
  This sample program demonstrates the various constructs that may be
  used to fetch a row of data using SQLJ.  It also demonstrates the
  use of explicit and default connection contexts.
  **/
public class QueryDemo
{
  public static void main(String[] args) throws SQLException
  {
    if (args.length != 2) {
      System.out.println("usage: QueryDemo ename newSal");
      System.exit(1);
    }

    /* if you're using a non-Oracle JDBC Driver, add a call here to
       DriverManager.registerDriver() to register your Driver
    */

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

    QueryDemoCtx ctx =
      new QueryDemoCtx(DefaultContext.getDefaultContext().getConnection());
    String ename = args[0];
    int newSal = Integer.parseInt(args[1]);

    System.out.println("before update:");
    getSalByName(ename, ctx);
    getSalByPos(ename);
    updateSal(ename, newSal, ctx);

    System.out.println("after update:");
    getSalByCall(ename, ctx);
    getSalByInto(ename);
    ctx.close(ctx.KEEP_CONNECTION);
  }

  public static void getSalByName(String ename, QueryDemoCtx ctx)
    throws SQLException
  {
    SalByName iter = null;
    #sql [ctx] iter = { SELECT ename, sal FROM emp WHERE ename = :ename };
    while (iter.next()) {
      printSal(iter.ename(), iter.sal());
    }
    iter.close();
  }

  public static void getSalByPos(String ename) throws SQLException
  {
    SalByPos iter = null;
    double sal = 0;
    #sql iter = { SELECT sal, ename FROM emp WHERE ename = :ename };
    while (true) {
      #sql { FETCH :iter INTO :sal, :ename };
      if (iter.endFetch()) break;
      printSal(ename, sal);
    }
    iter.close();
  }

  public static void updateSal(String ename, int newSal, QueryDemoCtx ctx)
    throws SQLException
  {
    #sql [ctx] { UPDATE emp SET sal = :newSal WHERE ename = :ename };
  }

  public static void getSalByCall(String ename, QueryDemoCtx ctx)
    throws SQLException
  {
    double sal = 0;
    #sql [ctx] sal = { VALUES(get_sal(:ename)) };
    printSal(ename, sal);
  }
```

```
  public static void getSalByInto(String ename)
    throws SQLException
  {
    double sal = 0;
    #sql { SELECT sal INTO :sal FROM emp WHERE ename = :ename };
    printSal(ename, sal);
  }

  public static void printSal(String ename, double sal)
  {
    System.out.println("salary of " + ename + " is " + sal);
  }
}
```

Prefetch Demo—PrefetchDemo.sqlj

This sample shows the use of row-prefetching through SQLJ and insert-batching through JDBC.

For information about prefetching and batching, see "Row Prefetching" on page A-2.

```
import java.sql.SQLException;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import oracle.jdbc.driver.OracleConnection;
import oracle.jdbc.driver.OraclePreparedStatement;
import sqlj.runtime.ref.DefaultContext;
import oracle.sqlj.runtime.Oracle;

/**
 Before executing this demo, or compiling this demo with
 online checking, you must run the SQL script PrefetchDemo.sql.

 This demo shows how to set different prefetch values for
 SQLJ SELECT statements. It compares SQLJ and JDBC runs.

 Additionally, when creating the data in the PREFETCH_DEMO
 table, we  show how to batch INSERT statements in JDBC.
 Note that SQLJ currently does not support JDBC batching.
 However, it is always possible to interoperate with JDBC
 when it is necessary to exploit batching.
**/
```

```
public class PrefetchDemo
{

  #sql static iterator PrefetchDemoCur (int n);

  public static void main(String[] args) throws SQLException
  {
     System.out.println("*** Start of Prefetch demo ***");

     Oracle.connect(PrefetchDemo.class, "connect.properties");
     OracleConnection conn =
        (OracleConnection) DefaultContext.getDefaultContext().getConnection();
     System.out.println("Connected.");

     try
     {
       #sql { DELETE FROM PREFETCH_DEMO };
     }
     catch (SQLException exn)
     {
       System.out.println("A SQL exception occurred: "+exn);
       System.out.println
            ("You probably forgot to run the PrefetchDemo.sql script");
       System.out.println("Run the script and then try again.");
       System.exit(1);
     }

     System.out.println(">>> Inserting data into the PREFETCH_DEMO table <<<");

     // We batch _all_ rows here, so there is only a single roundtrip.
     int numRows = 1000;
     insertRowsBatchedJDBC(numRows,conn);

     System.out.println(">>> Selecting data from the PREFETCH_DEMO table <<<");

     System.out.println("Default Row Prefetch value is:  "
                        + conn.getDefaultRowPrefetch());

     // We show four row prefetch settings:
     //  1. every row fetched individually
     //  2. prefetching the default number of rows (10)
     //  4. prefetching the default number of rows (10)
     //  3. prefetching all of the rows at once
     //
```

```
   // each setting is run with JDBC and with SQLJ

   int[] prefetch = new int[] { 1, conn.getDefaultRowPrefetch(),
                                numRows / 10,  numRows };

   for (int i=0; i<prefetch.length; i++)
   {
      selectRowsJDBC(prefetch[i], conn);
      selectRowsSQLJ(prefetch[i], conn);
   }

}

public static void selectRowsSQLJ(int prefetch, OracleConnection conn)
                   throws SQLException
{
  System.out.print("SQLJ: SELECT using row prefetch "+prefetch+". ");
  System.out.flush();
  conn.setDefaultRowPrefetch(prefetch);

  PrefetchDemoCur c;

  long start = System.currentTimeMillis();
  #sql c = { SELECT n FROM PREFETCH_DEMO };
  while (c.next()) { };
  c.close();
  long delta = System.currentTimeMillis() - start;

  System.out.println("Done in "+(delta / 1000.0)+" seconds.");
}

public static void selectRowsJDBC(int prefetch, OracleConnection conn)
                   throws SQLException
{
  System.out.print("JDBC: SELECT using row prefetch "+prefetch+". ");
  System.out.flush();
  conn.setDefaultRowPrefetch(prefetch);

  long start = System.currentTimeMillis();
  PreparedStatement pstmt =
                    conn.prepareStatement("SELECT n FROM PREFETCH_DEMO");
  ResultSet rs = pstmt.executeQuery();
  while (rs.next()) { };
  rs.close();
  pstmt.close();
```

```
    long delta = System.currentTimeMillis() - start;

    System.out.println("Done in "+(delta / 1000.0)+" seconds.");
  }

  public static void insertRowsBatchedJDBC(int n, OracleConnection conn)
                     throws SQLException
  {
    System.out.print("JDBC BATCHED: INSERT "+n+" rows. ");
    System.out.flush();

    long start = System.currentTimeMillis();
    int curExecuteBatch = conn.getDefaultExecuteBatch();
    conn.setDefaultExecuteBatch(n);

    PreparedStatement pstmt = conn.prepareStatement
                              ("INSERT INTO PREFETCH_DEMO VALUES(?)");
    for (int i=1; i<=n; i++)
    {
      pstmt.setInt(1,i);
      pstmt.execute();
    }
    ((OraclePreparedStatement)pstmt).sendBatch();
    pstmt.close();
    conn.setDefaultExecuteBatch(curExecuteBatch);
    long delta = System.currentTimeMillis() - start;

    System.out.println("Done in "+(delta / 1000.0)+" seconds.");
  }

}
```

Server-Side Samples

This section contains samples that run in the server. The demo in this section is located in the following directory:

```
[Oracle Home]/sqlj/demo/server
```

SQLJ in the Server—ServerDemo.sqlj

This example demonstrates a SQLJ application that runs in the embedded Java VM of the Oracle8*i* server.

Before trying to run this server-side demo application, refer to `README.txt` in the following directory:

```
[Oracle Home]/sqlj/demo/server
```

```
//---------------- Start of file ServerDemo.sqlj ----------------
import java.sql.Date;
import java.sql.SQLException;

class ServerDemo
{
  public static void main (String argv[])
  {
    // Note: No explicit connection handling is required
    //       for server-side execution of SQLJ programs.

    try {
      putLine("Hello! I'm SQLJ in server!");
      Date today;
      #sql {select sysdate into :today from dual};
      putLine("Today is " + today);
      putLine("End of SQLJ demo.");
    } catch (Exception e) {
      putLine("Error running the example: " + e);
    }
  }

  static void putLine(String s) {
    try {
      #sql { call dbms_output.put_line(:s)};
    } catch (SQLException e) {}
  }
```

JDBC versus SQLJ Sample Code

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

In the sample, two methods are defined: `getEmployeeAddress()`, which selects into a table and returns an employee’s address 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:

- The `ObjectDemo.sql` SQL script has been run to create the schema in the database and populate the tables. The SQL for this script is in "Definition of Object and Collection Types" on page 12-20.
- A PL/SQL stored function `UPDATE_ADDRESS()`, which updates a given address, exists.
- 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.

Both versions of the sample code reference objects and tables created by the `ObjectDemo.sql` script.

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).

JDBC Version of the Sample Code

Following is 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 that 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.driver.*;
```

```
/**
  This is what we 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.getCustomDatum(1, Address.getFactory());
    //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.setCustomDatum(2, addr);
    }

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

Line 12: In the `getEmployeeAddress()` method definition, you must pass the connection object to the method definition explicitly.

Lines 16-20: Prepare a statement that selects an employee's address from the `employees` table on the basis of the employee number. The employee number is represented by a marker variable, which is set with the `setInt()` method. Note that because the prepared statement does not recognize "`INTO`" syntax, you must provide your own code to populate the address (`addr`) variable. Since the prepared statement is returning a custom object, cast the output to an Oracle result set.

Lines 21-23: Because the Oracle result set contains a custom object of type `Address`, use the `getCustomDatum()` method to retrieve it (the `Address` class can be created by JPublisher). The `getCustomDatum()` method requires you to use the factory method `Address.getFactory()` to materialize an instance of an `Address` object. Since `getCustomDatum()` returns a `Datum`, cast the output to an `Address` object.

Note that the routine assumes a one-row result set. The to-do items in the comment statements indicate that you must write additional code for the cases where the result set contains either no rows or more than one row.

Lines 25-27: Close the result set and prepared statement objects, then return the `addr` variable.

Line 29: In the `updateAddress()` definition, you must pass the connection object and the `Address` object explicitly.

The `updateAddress()` method passes an address object (`Address`) to the database for update, then fetches it back. The actual updating of the address is performed by the stored function `UPDATE_ADDRESS()` (the code for this function is not illustrated in this example).

Line 33-43: Prepare an Oracle callable statement that takes an address object (`Address`) and passes it to the `UPDATE_ADDRESS()` stored procedure. To register an object as an output parameter, you must know the object's SQL type code and SQL type name.

Before passing the address object (`addr`) as an input parameter, the program must determine whether `addr` has a value or is null. Depending on the value of `addr`, the program calls different setter methods. If `addr` is null, the program calls `setNull()`; if `addr` has a value, the program calls `setCustomDatum()`.

Line 44: Fetch the return result `addr`. Since the Oracle callable statement returns a custom object of type `Address`, use the `getCustomDatum()` method to retrieve it (the `Address` class can be created by JPublisher). The `getCustomDatum()` method requires you to use the factory method `Address.getFactory` to materialize an instance of an `Address` object. Because `getCustomDatum()` returns a `Datum`, cast the output to an `Address` object.

Lines 45, 46: Close the Oracle callable statement, then return the `addr` variable.

Coding Requirements of the JDBC Version

Note the following coding requirements for the JDBC version of the sample code:

- The `getEmployeeAddress()` and `updateAddress()` definitions must explicitly include the connection object.
- Long SQL strings must be concatenated with the SQL concatenation character ("+").
- You must explicitly manage resources (for example, close result set and statement objects).
- You must cast datatypes as needed.
- You must know the `_SQL_TYPECODE` and `_SQL_NAME` of the factory object and objects that you are registering as output parameters.
- Null data must be explicitly handled.
- Host variables must be represented by parameter markers in callable and prepared statements.

Maintaining JDBC Programs

JDBC programs have the potential of being expensive in terms of maintenance. For example, in the above code sample, if you add another `WHERE` clause, then you must change the `SELECT` string. If you append another host variable, then you must increment the index of the other host variables by one. A simple change to one line in a JDBC program may require changes in several other areas of the program.

SQLJ Version of the Sample Code

Following is 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.

```
import java.sql.*;

/**
  This is what we 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 23
    return addr;
  }
}
```

Line 10: The `getEmployeeAddress()` method does not require a connection object. SQLJ uses a default connection context instance, which would have been defined previously somewhere in your application.

Lines 13-15: The `getEmployeeAddress()` method retrieves an employee address according to employee number. Use standard SQLJ `SELECT INTO` syntax to select an employee's address from the employee table if their 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. (Host variables are sometimes also referred to as bind variables.)

Line 16: The `getEmployeeAddress()` method returns the `addr` object.

Line 19: The `updateAddress()` method also uses the default connection context instance.

Lines 19-23: The address is passed to the `updateAddress()` method, which passes it to the database. The database updates it and passes it back. The actual updating of the address is performed by the `UPDATE_ADDRESS()` stored function (the code for this function is not shown here). Use standard SQLJ function-call syntax to receive the address object (`addr`) output by `UPDATE_ADDRESS()`.

Line 24: The `updateAddress()` method returns the `addr` object.

Coding Requirements of the SQLJ Version

Note the following coding requirements for the SQLJ version of the sample code:

- An explicit connection is not required; SQLJ assumes that a default connection context has been defined previously in your application.
- No datatype casting is required.
- SQLJ does not require knowledge of `_SQL_TYPECODE`, `_SQL_NAME`, or factories.
- Null data is handled implicitly.
- No explicit code for resource management (for closing statements or results sets, for example) 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 OBDC-style escapes are not used.

A Performance and Debugging

This appendix discusses tips and utilities to enhance performance of your SQLJ application and to debug your SQLJ source code at runtime. The following topics are discussed:

Performance Enhancements

There are Oracle features to enhance your performance by making database access more efficient. This includes *row prefetching,* which retrieves query results in groups of rows instead of one at a time and is supported by both Oracle JDBC and Oracle SQLJ. This also includes *batch updates* (and batch inserts), a feature that sends database updates in batches instead of one at a time. This feature is not currently supported by Oracle SQLJ but is supported by Oracle JDBC, so you can use JDBC code if you require it. (Prepared statements are not cached in SQLJ, instead being re-prepared each time. This effectively disables batch updates.)

For information about JDBC support for row prefetching and batch updates, see the *Oracle8i JDBC Developer's Guide and Reference.*

Note: Neither Oracle SQLJ nor Oracle JDBC supports batch fetches, which is the fetching of sets of rows into arrays of values.

Additionally, you can employ the Oracle SQL optimizer for a SQLJ program.

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. Prefetching rows allows you to receive the results more efficiently, in groups of multiple rows each.

To specify the number of rows to prefetch for queries that use a given connection context instance, use the underlying JDBC `Connection` instance cast to an `OracleConnection`. Following is an example that sets the prefetch value to 20 for your default connection:

```
((OracleConnection)DefaultContext.getDefaultContext().getConnection()).setDefaultRowPrefetch(20);
```

Each additional connection context instance you use must be set separately, as desired. For example, if `ctx` is an instance of a declared connection context class, set its prefetch value as follows:

```
((OracleConnection)ctx.getConnection()).setDefaultRowPrefetch(20);
```

There is no maximum row-prefetch. The default value is 10 in JDBC, and this is inherited by SQLJ. This appears to be an effective value in typical circumstances, although you may want to increase it if you receive a large number of rows.

See "Prefetch Demo—PrefetchDemo.sqlj" on page 12-62 for a sample application showing row-prefetching through SQLJ and insert-batching through JDBC.

Oracle SQL Optimizer

Oracle SQL allows 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 runtime to the database 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 Oracle8*i*. 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.

For information about the Oracle optimizer, see the *Oracle8i SQL Reference*.

AuditorInstaller Customizer for Debugging

Oracle SQLJ provides a special customizer, `AuditorInstaller`, that will insert sets of debugging statements, called auditors, into profiles specified on the SQLJ command line. These profiles must already exist from previous customization.

The debugging statements will execute during SQLJ runtime (when someone runs your application), displaying a trace of method calls and values returned.

Use the customizer harness `-debug` option, preceded by `-P` like any general customization option, to insert the debugging statements. (Syntax for this option is also discussed in "Invoking AuditorInstaller with the Customizer Harness -debug Option" on page A-5.)

About 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 runtime functionality. The deepest layer uses straight Oracle JDBC calls and implements any of your SQLJ statements which can be executed through JDBC functionality. Each higher layer is a specialized layer for some kind of SQLJ functionality which is not supported by JDBC and so must be handled specially by the SQLJ runtime. For example, there is a layer for iterator conversion statements (`CAST`) used to convert JDBC result sets to SQLJ iterators. There is another layer for assignment statements (`SET`).

The functionality of the code layers is that at runtime each SQLJ executable statement is first passed to the shallowest layer, and then passed layer-by-layer until it reaches the layer that can handle 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 runtime, 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 runtime debugging output will be a trace of method calls resulting from all of your SQLJ executable statements. If you install an auditor at the deepest layer, its runtime output will be a trace of method calls from all of 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" on page A-6 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 class `sqlj.runtime.profile.util.AuditorInstaller`, which does the work of inserting the debugging statements.

The `-P-debug` option is equivalent to `-P-customizer=sqlj.runtime.profile.util.AuditorInstaller`, overriding the customizer specified in the SQLJ `-default-customizer` option.

Note:

- To run an application with auditors installed, the Oracle SQLJ file `translator.zip` must be in your `CLASSPATH` (normally, running a SQLJ application requires only `runtime.zip`, which is a subset of `translator.zip`).
- 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 do any other customization when you use this option.
- You also cannot use `-P-print` and `-P-debug` at the same time because the `-print` option also invokes a special customizer.

Command-line syntax `sqlj -P-debug` *`profile_list`*

Command-line example `sqlj -P-debug Foo_SJProfile*.ser`

Properties file syntax `profile.debug` (must also specify profiles in file list)

Properties file example `profile.debug` (must also specify profiles in file list)

Default value n/a

AuditorInstaller Options

Like any customizer, `AuditorInstaller` has its own options which can be set using the `-P-C` prefix on the SQLJ command line (or `profile.C`, instead of `-P-C`, in a SQLJ properties file).

`AuditorInstaller` supports the following options:

- `depth`—Specify how deeply you want to go into the layers of runtime functionality in your profiles.
- `log`—Specify the target file for runtime output of the debug statements of the installed auditor.
- `prefix`—Specify a prefix for each line of runtime output that will result from this installation of debug statements.
- `uninstall`—Removes the debug statements that were placed into the profiles during the most recent previous invocation of `AuditorInstaller` on those profiles.

AuditorInstaller depth Option

As discussed in "About Auditors and Code Layers" on page A-4, `AuditorInstaller` can install a set of debug statements, known as an auditor, at only a single layer of code during any one execution. The `AuditorInstaller` `depth` option allows 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 and 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 `-P-Cdepth=`*`n`*

Command-line example `-P-Cdepth=0`

Properties file syntax `profile.Cdepth=`*`n`*

Properties file example `profile.Cdepth=0`

Default value `-1` (deepest layer)

AuditorInstaller log Option

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—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 to have debug information from all layers of your application to go to the same log file in this way).

Command-line syntax `-P-Clog=`*`log_file`*

Command-line example `-P-Clog=foo/bar/mylog.txt`

Properties file syntax `profile.Clog=`*`log_file`*

Properties file example `profile.Clog=foo/bar/mylog.txt`

Default value empty (use standard output)

AuditorInstaller prefix Option

Use the `prefix` option to specify a prefix for each line of runtime output resulting from the debug statements that are installed during this invocation of `AuditorInstaller`.

This option is often used in conjunction with the `depth` option. By running `AuditorInstaller` a number of times with different prefixes for different layers, you can easily see at runtime what information is coming from which layers.

Command-line syntax `-P-Cprefix="`*`string`*`"`

Command-line example `-P-Cprefix="layer 2: "`

Properties file syntax `profile.Cprefix="`*`string`*`"`

Properties file example `profile.Cprefix="layer 2: "`

Default value empty

AuditorInstaller uninstall Option

Use the `uninstall` option to remove debug statements that were 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 `-P-Cuninstall`

Command-line example `-P-Cuninstall`

Properties file syntax `profile.Cuninstall`

Properties file example `profile.Cuninstall`

Default value `false`

Full Command-Line Examples

Following are some full SQLJ command-line examples showing the specification of AuditorInstaller options.

Insert a set of debug statements, or auditor, into the deepest 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 layer 0. Send runtime output to `log.txt`; prefix each line of runtime output with "`Layer 0: `" (the command below 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
```

AuditorInstaller Runtime Output

During runtime, debug statements placed by `AuditorInstaller` result in a trace of methods called and values returned. This is done for all profile layers that had debug statements installed. (There is no means of selective debug output at runtime.)

`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` runtime 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 showing the call and input parameters, and the second showing the return value.

Note: The classes you see in the `oracle.sqlj.runtime` package are SQLJ runtime classes with equivalent functionality to similarly named JDBC classes. For example, `OraRTResultSet` is the SQLJ runtime implementation of the JDBC `ResultSet` class, containing equivalent attributes and methods.

Additional SQLJ Debugging Considerations

In addition to the `AuditorInstaller` discussed in "AuditorInstaller Customizer for Debugging" on page A-4, there are other considerations to be aware of regarding debugging:

- If you are running SQLJ from the command line, there is a translator option (`-linemap`) that can aid in debugging your SQLJ code.
- The embedded server-side translator has an option that will aid in debugging your Java code in general, but not your SQLJ code in particular.
- SQLJ is integrated into the Oracle JDeveloper integrated development environment, allowing access to JDeveloper's debugging facilities.

SQLJ -linemap Flag

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 runtime errors occur, the line number reported by the Java VM is the line number in the SQLJ source code, making it much easier to debug.

See "Line-Mapping to SQLJ Source File (-linemap)" on page 8-45 for more information about this flag.

Server-Side debug Option

If you are loading SQLJ source into the server, using the server-side embedded translator to translate it, the 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.

See "Option Support in the Server Embedded Translator" on page 11-16 for more information about this option and information about how to set options in the server.

Developing and Debugging in JDeveloper

Oracle SQLJ is fully integrated into the Oracle JDeveloper visual programming tool.

JDeveloper also includes an integrated debugger that supports SQLJ. SQLJ statements, like standard Java statements, can be debugged in-line as your application executes. Reported line numbers map back to the line numbers in your SQLJ source code (as opposed to in the generated Java code).

See "SQLJ in JDeveloper and Other IDEs" on page 1-19 for an introduction to JDeveloper.

B
SQLJ Error Messages

This appendix lists error messages that can be produced by the SQLJ translator and SQLJ runtime. Cause and action information is also provided, as well as the SQL state for runtime errors.

Translation Time Messages

This section provides a list of error messages you may encounter from the SQLJ translator, including cause and action information.

Note: By enabling the SQLJ translator `-explain` flag, you can instruct the translator to provide "cause" and "action" information in real-time with its error message output. This is the same information that is provided in the error list below. See "Cause and Action with Translator Errors (-explain)" on page 8-45.

<<<NEW SQL>>>

Cause: The Oracle customizer translated a SQL operation into an Oracle-specific dialect, as shown in the remainder of the message. Messages of this nature are enabled with the Oracle customizer "showSQL" option.

Action: This is an informational message only. No further action is required.

[Connecting to user *user* at *connection*]

Cause: Informs user that SQLJ connects as user *user* to the database with URL *connection*.

[Preserving SQL checking info]

Cause: SQLJ will preserve analysis information obtained from online checking during this run.

[Querying database with "*sqlquery*"]

Cause: Informs user that database query was issued.

[Re-using cached SQL checking information]

Cause: Informs user that SQLJ is reusing cached analysis results from previous online checking runs.

[Registered JDBC drivers: *class*]

Cause: Lists the JDBC drivers that have been registered.

[SQL checking: read *m* of *n* cached objects.]

Cause: Analysis information cached from online checking has been retrieved.

[SQL function call "*sqlj call*" transformed into ODBC syntax "*jdbc call*"]

Cause: Informs user that SQLJ has converted SQLJ function call syntax to JDBC function call syntax.

A call to a stored function must return a value.

Cause: User ignores result returned by a stored function call.

A call to a stored procedure cannot return a value.

Cause: User tries to retrieve a return value from a stored procedure invocation.

A non-array type cannot be indexed.

Cause: Only array types can be used as the base operand of array access operator (’[]’).

Action: Check the type of the base operand.

A SQL quote was not terminated.

Action: Insert the terminating " or ’.

Access modifiers *modifier1* and *modifier2* are not compatible.

Cause: Named access modifiers cannot be applied to the same class, method, or member. For example, `private` and `public` are incompatible as access modifiers.

Action: Change or remove one of the conflicting access modifiers.

Ambiguous column names *columns* in SELECT list.

Cause: You may not use column names that are only distinguished by case.

Action: Use column aliases to distinguish column names.

Ambiguous constructor invocation.

Cause: More than one constructor declaration matches the arguments after standard conversions.

Action: Indicate with explicit cast which constructor argument types should be used.

Ambiguous method invocation.

Cause: More than one overloaded method declaration matches the arguments after standard conversions.

Action: Indicate with explicit cast which method argument types should be used.

an io error occured while generating output: *message*

Action: Ensure that you have appropriate permissions and sufficient space for SQLJ output.

Anonymous classes are not allowed in bind expressions.

Cause: Host expressions cannot contain anonymous classes.

Action: Move the expression that has anonymous class outside the #sql statement and store its value to a temporary variable of the correct type; then use that temporary variable in the host expression instead.

Argument #*n* of *name* must be a host variable, since this argument has mode OUT or INOUT.

Cause: Modes OUT and INOUT require the presence of variables or assignable expressions (such as array locations) in this argument position.

Argument #*n* of *name* requires mode IN.

Cause: The stored procedure or function *name* requires that the mode of the host expression #*n* be IN.

Action: Declare the host expression in the SQLJ statement as IN.

Argument #*n* of *name* requires mode INOUT.

Cause: The stored procedure or function *name* requires that the mode of the host expression #*n* be INOUT.

Action: Declare the host expression in the SQLJ statement as INOUT.

Argument #*n* of *name* requires mode OUT.

Cause: The stored procedure or function *name* requires that the mode of the host expression #*n* be OUT.

Action: Declare the host expression in the SQLJ statement as OUT.

Argument #*pos* is empty.

Cause: In the argument list of a stored function or procedure, you left the argument at position *pos* empty. For example: `proc(1, ,:x)`.

Action: Replace the empty argument with a host expression or a SQL expression.

Arithmetic expression requires numeric operands.

Cause: Both the left-hand side and the right-hand side of an arithmetic operation must have numeric types.

Action: Correct the types of the operands.

Array index must be a numeric type.

Cause: Array objects can only be indexed using a numeric index.

Action: Correct the type of the index operand.

Attributes *attribute1* and *attribute2* are not compatible.

Cause: The named attributes cannot be applied to the same class or method. For example, abstract and final are incompatible as attributes.

Action: Change or remove one of the conflicting attributes.

auditing layer added

Cause: An auditing customization was installed into the profile being customized.

Action: The profile will include audit calls when used. No further action required. Use the "uninstall" option to remove the auditor.

auditing layer removed

Cause: The last auditing customization previously installed into the profile was removed. If multiple auditors were installed, only the last to be installed is removed.

Action: Further "uninstall" calls may be required if you want to remove additional auditors.

backup created as *filename*

Cause: A backup file for the profile was created with the name *filename*. The backup file contains the original profile before customization.

Action: No further action required. The original profile can be restored by copying the backup file over the new profile.

bad filename: *filename*

Cause: The file *filename* could not be used as input to the customizer harness utility. Only filenames with ".ser" or ".jar" extensions are supported.

Action: Rename the file to have an accepted extension.

Bad octal literal '*token*'.

Cause: A numeric literal beginning with digit '0' is interpreted as an octal, and hence must not contain digits '8' or '9'.

Action: Modify the bad literal. If octal was intended, recalculate its value in base-8. If decimal was intended, remove all leading zeroes.

Badly placed #sql construct -- not a class declaration.

Cause: An executable SQLJ statement appears where a declaration was expected.

Action: Move the #sql construct to a legal position.

Bitwise operator requires boolean or numeric operands.

Cause: Bitwise operator can only operate on objects both of which are either boolean or numeric. A bitwise operation between two objects from different categories will fail.

Action: Check the types of operands.

Boolean operator requires boolean operands.

Cause: Boolean operators can only operate with boolean arguments.

Action: Check the types of operands.

cannot access option *option name*

Cause: The option named *option name* was not accessible to the customizer harness. This often indicates a non-standard customizer-specific option.

Action: Verify the intended use of the option. As a workaround, discontinue use of the option or use a different customizer.

Cannot analyze SQL statement online: unable to determine SQL types for *count* host items.

Cause: SQLJ determines a corresponding SQL type for each of the Java host expressions. These SQL types are required for checking the statement online.

Action: Use Java types that are supported by Oracle SQLJ.

Cannot determine default arguments for stored procedures and functions. May need to install SYS.SQLJUTL.

Cause: SQLJ cannot find the functions declared in the package `SYS.SQLJUTL`.

Action: Find the SQL file *[Oracle Home]*`/sqlj/lib/sqljutl.sql` and run it. Alternatively, if your stored functions or procedures do not use default arguments, you can ignore this message.

Cannot load JDBC driver class *class*.

Action: Check the name of the JDBC driver *class*.

cannot remove java file without first compiling it

Cause: The "nc" and "rj" options were specified at the same time to the profile conversion utility. The utility is unable to remove the Java file if it has not been compiled into a class file.

Action: Use only one of the "nc" and "rj" options.

Cannot resolve identifier because the enclosing class has errors.

Cause: Class that contains errors cannot be used in name resolution because access rights can be assigned to complete classes only.

Action: Fix the enclosing class, paying attention to correct spelling of base types, field types, method argument types and method return types. Also make sure that any external classes that are referenced by their base name only have been imported.

cannot specify both *option name* and *option name* options

Cause: Two incompatible options were specified at the same time to the profile conversion utility.

Action: Use only one of the specified options.

Class *class* does not implement the checker interface.

Cause: Checkers must implement `sqlj.framework.checker.SQLChecker`.

class cannot be constructed as an iterator: *class name*

Cause: The iterator class *class name* used in this SQL operation did not have the expected constructor. This indicates an iterator generated by a non-standard translator.

Action: Retranslate the iterator declaration using a standard translator.

class has already been defined: *classname*

Cause: Ensure that the class *classname* is only defined in one of the source files that you pass to SQLJ.

class implements both sqlj.runtime.NamedIterator and sqlj.runtime.PositionedIterator: *class name*

Cause: It could not be determined if the iterator class *class name* used in this SQL operation was a named iterator or positional iterator. This indicates an iterator that was generated by a non-standard translator or included an erroneous interface in its `implements` clause.

Action: Verify that the `implements` clause of the iterator declaration does not contain one of the problematic interfaces. Retranslate the iterator declaration using a standard translator.

Column *javatype column* not found in SELECT list.

Action: The column *column* could not be found in the result set returned by the query. Either fix the iterator declaration, or the SELECT statement, possibly by using an alias.

Column *name1 #pos1* will cause column *name2 #pos2* to be lost. Use a single stream column at the end of the select list.

Cause: You can have at most one stream column in a positional iterator, and this column must be the last one in the iterator.

Action: Move the stream column to the last position in the iterator. If you have more than one stream column, you can use a named iterator, ensuring that the stream columns (and other columns) are accessed in order.

Column *type column* is not compatible with database type *sqltype*

Cause: The Java and SQL types are not compatible.

Comparison operator requires numeric operands.

Cause: Only numeric values are meaningful in an operation that compares magnitudes.

Action: Check the types of operands.

compatible with the following drivers:

Cause: The Oracle customizer "compat" option was enabled. A list of Oracle JDBC driver versions that may be used with the current profile follows this message.

Action: Use one of the listed JDBC driver versions to run the program.

compiling *filename*

Cause: The profile in file *filename* was compiled into class file format by the profile conversion utility.

Action: No further action required.

Complement operator requires integral operand.

Cause: Only an integral value can be complemented bitwise.

Action: Check the types of operands.

Conditional expression requires boolean for its first operand.

Cause: Conditional expression uses its first operand to choose which one of the other two shall be executed; hence the first operand must have a boolean type.

Action: Check the type of the first operand.

Conditional expression result types must match.

Cause: The value of conditional expression is either its second or its third operand, both of which must be either boolean or numeric types, or object types at least one of which is assignable to the other.

Action: Check the types of operands.

Connection context expression does not have a Java type.

Cause: No valid Java type could be derived for your connection context expression.

Connection context must have been declared with #sql context ... It can not be declared as a ConnectionContext.

Action: Declare your connection context type with `#sql context` *ConnectionContext*`;`

ConnectionContext attribute *attribute* is not defined in the SQLJ specification.

Action: The `with`-clause attribute *attribute* is not explicitly part of the SQLJ specification. Check the spelling of your attribute name.

ConnectionContext cannot implement the *interface* interface.

Cause: In your SQLJ context declaration you specified an `implements` clause with the interface *interface*. However, connection contexts do not implement this interface.

Constructor not found.

Cause: The constructor that was invoked does not exist.

Action: Check the constructor arguments, or add a constructor with the desired arguments.

Context *context* ignored in FETCH statement.

Cause: Since a context is associated with a cursor object at the initialization of a cursor with a query, context information in FETCH statements is superfluous, and will be ignored by SQLJ.

converting profile *filename*

Cause: The profile in file *filename* was converted from serialized to Java source file format by the profile conversion utility.

Action: No further action required.

Cursor has *item count* items. Argument #*pos* of INTO-list is invalid.

Cause: Your INTO-list has more elements than the corresponding positional iterator from which you are fetching.

Action: Remove the extra INTO-list elements.

Cursor type in FETCH statement does not have a Java type.

Cause: No valid Java type could be derived for the iterator expression in the FETCH statement.

customized

Cause: The profile was successfully customized.

Action: No further action required.

customizer does not accept connection: *connection url*

Cause: The connection specified by *connection url* was established, but was either not needed or not recognized by the current customizer.

Action: Verify that the current customizer requires a connection. If not, omit the "user" option from the customizer harness. If so, verify that the database and schema connected to are compatible with the customizer.

Database error during signature lookup for stored procedure or function *name*: *message*

Cause: An error occurred when SQLJ tried to determine the existence and the signature of the function or procedure *name*.

Action: As a workaround you can translate your SQLJ program offline.

Database issued an error: *error.*

Cause: Database issued *error* when parsing a SQL statement against the exemplar schema.

Action: Check the validity of the SQL statement.

Database issued an error: *error sqltext*

Cause: Database issued an error when parsing the SQL statement against the exemplar schema.

Action: Check the validity of the SQL statement.

deleting *filename*

Cause: The intermediate file *filename* was removed by the profile conversion utility.

Action: No further action required.

Did not find a stored procedure or function *name* with *n* arguments.

Cause: No procedure or function *name* with *n* arguments appears in the database.

Action: Check the name of your stored procedure or function.

Did not find a stored procedure or function *name* with *n* arguments. *found functions/procedures with different numbers of arguments*

Cause: No procedure or function *name* with *n* arguments appears in the database. However, there is a procedure or function of this name with a different number of arguments.

Action: Check the name of your stored procedure/function, as well as for extraneous or missing arguments.

Did not find stored function *name* with *n* arguments.

Cause: SQLJ could not find a stored function of the desired name *name*.

Action: Check the name of your stored function.

Did not find stored function *proc* with *n* arguments. *found functions/procedures with different numbers of arguments*

Cause: No stored function *proc* with *n* arguments appears in the database. However, there is a procedure or function of this name with a different number of arguments.

Action: Check the name of your stored function, as well as for extraneous or missing arguments.

Did not find stored procedure *name* with *n* arguments.

Cause: SQLJ could not find a stored procedure of the desired name *name*.

Action: Check the name of your stored procedure.

Did not find stored procedure *proc* with *n* arguments. *found functions/procedures with different numbers of arguments*

Cause: No stored procedure *proc* with *n* arguments appears in the database. However, there is a procedure or function of this name with a different number of arguments.

Action: Check the name of your stored procedure, as well as for extraneous or missing arguments.

Do not know how to analyze this SQL statement.

Cause: An online connection is required to help SQLJ analyze this statement.

Do not understand this statement.

Cause: Unable to identify this statement, since it does not start with a SQL keyword (SELECT, UPDATE, DELETE, BEGIN, ...) or a SQLJ keyword (CALL, VALUES, FETCH, CAST, ...).

Duplicate access modifier.

Cause: The same access modifier appears more than once for the same class, method or member.

Action: Remove the superfluous access modifier.

Duplicate method *method*.

Cause: The method *method* was declared more than once.

Duplicate methods *method1* and *method2*.

Cause: Methods *method1* and *method2* map to the same SQL name. You cannot have two methods that map to the same SQL name in a named iterator declaration.

Equality operator operand types must match.

Cause: Equality operator can only compare objects both of which are either boolean or numeric types, or object types at least one of which is assignable to the other.

Action: Check the types of the operands to the equality operator.

error converting profile: *filename*

Cause: An error occurred while converting the profile in file *filename* from serialized to class file format. Details of the error were listed after this message.

Action: Consult the error details and fix as appropriate.

Error in Java compilation: *message*

Cause: An error occurred when SQLJ was invoking the Java compiler to compile `.java` source files.

Action: Ensure that the correct Java compiler is specified in the -compiler-executable flag, and that the compiler can be found on the PATH. Alternatively, you can use the -passes option, so that your Java compiler is called from the commandline rather than from SQLJ.

error loading customizer harness

Cause: The customizer harness utility could not be properly initialized. This indicates an incompatible Java runtime environment.

Action: Verify that the Java runtime environment is compatible with JRE 1.1 or later.

Expected "*token1*" and found "*token2*" instead.

Cause: The syntax of this statement requires a terminating token *token1* which was not found.

Expected 'FROM' to follow 'SELECT ... INTO ...'

Cause: The SELECT statement syntax is incorrect.

Action: Add FROM clause after the INTO clause.

Expected cast to be assigned to an iterator, found that cast was assigned to *type*.

Cause: The the left-hand-side of the CAST assignment must be a SQLJ iterator instance, not an expression of type *type*.

Expected cast to be assigned to an iterator.

Cause: The SQLJ CAST statement must be an assignment statement, with the left-hand-side of the assignment being a SQLJ iterator instance.

Expected cursor host variable.

Cause: A host variable representing an iterator type was expected here.

Expected cursor host variable. Encountered: "*token*"

Cause: A host variable representing an iterator type was expected here.

Expected end of cast statement. Found "*token*" ...

Cause: An unexpected token *token* was found after the CAST statement.

Expected end of FETCH statement. Encountered: "*token*"

Cause: No further tokens were expected in this FETCH statement.

Expected host variable of type java.sql.ResultSet, found "*token*" ...

Cause: You did not specify a host variable after the CAST keyword.

Expected host variable of type java.sql.ResultSet, found host variable of invalid Java type.

Cause: No valid Java type could be derived for the host expression.

Expected host variable of type java.sql.ResultSet, found host variable of type *type*.

Cause: The host expression has the Java type *type*, not `java.sql.ResultSet` as required.

Action: Use a host expression of type `java.sql.ResultSet`. If necessary, you can cast the expression to this type using a Java cast.

Expected host variable of type java.sql.ResultSet.

Cause: The SQLJ CAST statement assigns a `java.sql.ResultSet` to an iterator type. The type you are trying to convert is not a `java.sql.ResultSet`.

Action: You must use a host expression of type `java.sql.ResultSet`. If necessary, you can cast the expression to this type using a Java cast.

Expected INTO bind expression.

Cause: This statement should have a list of one or more INTO host expressions.

expected ODBC function call syntax "{ call func(...) }".

Cause: Invalid use of the JDBC escape syntax for calling stored procedures.

Expected stored function name. Found: *token*

Cause: The name of a stored function was expected here instead of the token *token*.

Expected stored function or procedure name. Found: *token*

Cause: The name of a stored function or a stored procedure was expected here instead of the token *token*.

Expected stored procedure name. Found: *token*

Cause: The name of a stored procedure was expected here instead of the token *token*.

Expected: FETCH :cursor INTO ...

Cause: The FETCH statement must have a cursor host variable, from which values are to be fetched.

Expected: WHERE CURRENT OF :hostvar. Found: WHERE CURRENT *token* ...

Action: Use proper syntax in the WHERE CURRENT OF clause.

Expected: WHERE CURRENT OF :hostvar. Found: WHERE CURRENT OF *token* ...

Action: Use proper syntax in the WHERE CURRENT OF clause.

field "*field name*" in *class name* is not a *class name* type

Cause: The field named *field name* in custom datum class *class name* did not have the expected type *class name.* A field of this type is required for proper conversion of the class to and from Oracle database types.

Action: Declare field *field name* to be the indicated type in the custom datum class.

field "*field name*" in *class name* is not accessible

Cause: The field named *field name* was not public in custom datum class *class name*. It is required for proper conversion of the class to and from Oracle database types.

Action: Declare field *field name* as `public` in the custom datum class.

field "*field name*" in *class name* is not uniquely defined

Cause: More than one field named *field name* was found in custom datum class *class name*. This can occur if *field name* is defined in two different interfaces that are both implemented by *class name*. A uniquely defined field is required for proper conversion of the class to and from Oracle database types.

Action: Update the custom datum class so that *field name* is defined only once.

field "*field name*" not found in *class name*

Cause: A field named *field name* could not be found in custom datum class *class name*. It is required for proper conversion of the class to and from Oracle database types.

Action: Declare the required field in the custom datum class.

Field not accessible.

Cause: This class has no access to the field.

Action: Check that the access rights of the field are set correctly.

file too large

Cause: A profile file contained in a JAR file was too large to be customized.

Action: Extract and customize the profile as a single file rather than as part of a JAR file.

filename must be a valid java identifier: *filename*

Cause: The filename is an illegal Java identifier. SQLJ creates additional class and resource definitions based on the name of the input file, so the name must be able to be used as a Java identifier.

Action: Rename the file so that it can be used as a Java identifier.

found incompatible types

Cause: The profile contained a combination of types that could not be supported by any one Oracle JDBC driver.

Action: Remove incompatible types from the program. Incompatible types are included in the types listed by the "summary" option.

Host item #*n* cannot be OUT or INOUT.

Cause: The host item at position #*n* is embedded in a SQL expression that constitutes an argument to a stored procedure or function. This argument position therefore must have the mode IN. This message is also given if you bind arguments by name.

Action: Change the mode of the argument to IN. If you are binding an OUT or INOUT argument by name, you should ignore this message.

Host item #*pos* must be an lvalue.

Cause: The OUT or INOUT host expression at position *pos* must be an assignable expression. Java variables, fields, and array elements are assignable expressions.

Host item *name* (at position #*n*) cannot be OUT or INOUT.

Cause: The host item *name* at position #*n* is embedded in a SQL expression that constitutes an argument to a stored procedure or function. This argument position therefore must have the mode IN. This message is also given if you bind arguments by name.

Action: Change the mode of the argument to IN. If you are binding an OUT or INOUT argument by name, you should ignore this message.

Identifier *identifier* may not begin with __sJT_.

Action: Ensure that you do not use identifiers that start with `__sJT_`.

ignoring context name *context name*

Cause: A profile was found with an associated connection context named *context name*. Since this context was not included in the customizer harness "context" option list, this profile was not customized.

Action: Rerun the customizer harness with a "context" setting that includes the named context, if desired.

Illegal entry for option *option*. Expected a boolean value, received: "*value*"

Action: Use a boolean value for *option* (such as `true`, `false`, `yes`, `no`, `0`, `1`).

Illegal INTO ... bind variable list: *error.*

Cause: One or more components of the INTO list do not have a valid Java type.

Illegal Java type in cursor for WHERE CURRENT OF

Cause: No valid Java type could be derived for the iterator in the WHERE CURRENT OF clause.

Illegal token '*token*' will be ignored.

Cause: Source file contains a sequence of characters that cannot be matched to any Java token.

Action: Modify the source file to fix the error and verify the source file contains valid Java source code.

illegal value: *option setting*

Cause: An option was set to a value that was out of range or invalid.

Action: Consult the message detail and correct the option value accordingly.

IN mode is not allowed for INTO-variables.

Cause: INTO variables return values in Java.

Action: Use OUT instead (which is the default, so if you want you can omit the specifier altogether).

Inaccessible Java type for host item #*n*: *type*.

Cause: The Java class *type* is not a publicly visible class, and thus cannot be instantiated by a driver.

Action: Use a `public` Java type in the host expression.

Inaccessible Java type for host item *name* (at position #*n*): *type*.

Cause: The host expression *name* has Java type *type*, which is not publicly visible, and thus cannot be instantiated by a driver.

Action: Use a `public` Java type in the host expression.

Inaccessible Java type for item #*pos* of INTO-list: *type*.

Cause: The Java class *type* of INTO-list item *pos* is not a publicly visible class, and thus cannot be instantiated by a driver.

Action: Use a `public` Java type in the INTO-list.

Increment/decrement operator requires numeric operand.

Cause: Increment and decrement operators can only operate on integer values.

Action: Check the type of the operand.

Initialization lists are not allowed in bind expressions.

Cause: Host expressions cannot have initialization lists.

Action: Move the expression that uses initialization list outside the #sql statement and store its value to a temporary variable of the correct type; then use that temporary variable in the host expression instead.

INOUT mode is not allowed for INTO-variables.

Cause: INTO variables return values in Java.

Action: Use OUT instead (which is the default, so if you want you can omit the specifier altogether).

Instanceof operator requires an object reference operand.

Cause: Instanceof operator can only operate on objects.

Action: Check the type of the operand.

INTERNAL ERROR SEM-*label*. Should not occur - please notify.

Action: Notify Oracle of the error message.

INTO-list item #*position* must be an lvalue.

Cause: The elements of an INTO-list must be assignable expression. Java variables, fields, and array elements are assignable expressions.

INTO-lists may only occur in SELECT and FETCH statements.

Cause: No INTO... bind list is permitted in the current SQL statement.

Invalid bind variable or expression.

Cause: A bind variable (i.e., host variable, context expression, or iterator expression when used to store the return value of a query) is not legal Java syntax.

Action: Fix the host variable or expression.

Invalid cursor type in FETCH statement: *type*.

Action: Iterator in the FETCH statement must implement `sqlj.runtime.FetchableIterator`.

Invalid CustomDatum implementation in *type*: *mesg*

Cause: You are employing a user-defined Java type *type* that implements the `oracle.sql.CustomDatum` interface. However, your type does not meet all of the requirements placed on user-defined type, as indicated by the message detail.

Action: Remedy the problem in your user-defined type. Alternatively, you may want to use the `jpub` utility to generate your user-defined type.

Invalid iterator declaration.

Cause: There is a syntax error in the SQL declaration.

Action: Check the SQL declaration syntax.

Invalid Java type for host item #*n*.

Cause: No valid Java type could be derived for host expression #*n*.

Invalid Java type for host item #*n*: *error*.

Cause: No valid Java type could be derived for host expression #*n*.

Invalid Java type for host item *name* (at position #*n*).

Cause: No valid Java type could be derived for host expression *name* (at position #*n*).

Invalid Java type for host item *name* (at position #*n*): *error*.

Cause: No valid Java type could be derived for host expression *name* (at position #*n*).

Invalid Java type for item #*pos* of INTO-list: *type*.

Cause: No valid Java type could be derived for INTO-item #*pos*: *type*.

invalid option "*option name*" set from *option origin*: *problem description*

Cause: The option *option name* had an invalid value.

Action: Correct the option value as needed for *problem description*.

invalid option: *option setting*

Cause: The option given by *option setting* was not recognized by the customizer harness.

Action: Correct or remove the unknown option.

invalid profile name: *profile name*

Cause: The JAR file MANIFEST file contained a SQLJ profile entry that was not contained in the JAR file.

Action: Add the named profile to the JAR file, or remove its entry from the MANIFEST file.

Invalid SQL iterator declaration.

Cause: An instance of a declared SQLJ type cannot be fully manipulated, because its declaration contains errors or ambiguities.

Action: Check the SQL iterator declaration, paying attention to the types that appear in the iterator column type list, and that those types are imported if they are referred to using their base name only.

Invalid SQL string.

Cause: There is a syntax error in the SQL statement.

Action: Check the SQL statement syntax, paying attention especially to missing delimiters (for example, closing parenthesis, braces, and brackets; quotation marks; comment delimiters, etc.).

Invalid type cast

Cause: An object cannot be cast to the indicated type.

Action: Check the type of the operand.

Item #*pos* of INTO-list does not have a Java type.

Cause: No valid Java type could be derived for INTO-item #*pos*.

iterator *class name* must implement either sqlj.runtime.NamedIterator or sqlj.runtime.PositionedIterator

Cause: The iterator class *class name* used in this SQL operation was neither a named iterator nor a positional iterator. This indicates an iterator that was generated by a non-standard translator.

Action: Retranslate the iterator declaration using a standard translator.

Iterator attribute *attribute* is not defined in the SQLJ specification.

Action: The `with`-clause attribute *attribute* is not explicitly part of the SQLJ specification. Check the spelling of your attribute name.

Iterator with attribute updateColumns must implement sqlj.runtime.ForUpdate

Action: Specify the `implements`-clause: `implements sqlj.runtime.ForUpdate` in your iterator declaration.

JAR does not contain MANIFEST file

Cause: A JAR file did not contain a MANIFEST file. The MANIFEST file is required to determine the profiles contained in the JAR file.

Action: Add a MANIFEST to the JAR file. The MANIFEST should include the line "SQLJProfile=TRUE" for each profile contained in the JAR file.

JAR MANIFEST file format unknown

Cause: A JAR file could not be customized because the JAR MANIFEST file was written using an unknown format.

Action: Recreate the JAR file with a MANIFEST file formatted according the JDK manifest file format specification. MANIFEST files created using the `jar` utility conform to this format.

Java type *javatype* for column *column* is illegal.

Cause: No valid Java class declaration could be found for *javatype*.

Java type *type* of iterator for WHERE CURRENT OF is not supported. It must implement sqlj.runtime.ForUpdate.

Cause: The iterator in the WHERE CURRENT OF clause must be declared as implementing the interface `sqlj.runtime.ForUpdate`.

JDBC does not specify that column *column type* is compatible with database type *sqltype*. Conversion is non-portable and may result in a runtime error.

Action: For maximum portability to different JDBC drivers, you should avoid this conversion.

JDBC reports a mode other than IN/OUT/INOUT/RETURN for *name* in position *n*.

Cause: Your JDBC reports an unknown mode for an argument of a stored procedure or function.

Action: Ensure that the stored function or procedure has been properly defined. Possibly update your JDBC driver.

JDBC reports an error during the retrieval of argument information for the stored procedure/function *name*: *error*.

Action: Because of the error, the modes for this function or procedure could not be determined. Repeat translation or translate offline if error persists.

JDBC reports more than one return value for *name*.

Cause: Your JDBC driver erroneously reports multiple return arguments for a stored procedure or function.

Action: Update your JDBC driver.

JDBC reports the return value for *function* in position *pos* instead of position 1.

Cause: Your JDBC driver does not properly report the return argument of a stored function first.

Action: Update your JDBC driver.

Left hand side of assignment does not have a Java type.

Cause: No valid Java type could be derived for the left-hand-side expression of the assignment statement.

list item value may not be empty

Cause: A list-valued option such as "driver" or "context" included an empty list item.

Action: Remove the empty item from the list.

Loss of precision possible in conversion from *sqltype* to column *column type*.

Cause: Conversion from a numeric SQL value to Java may result in a loss of precision.

Method name *method* is reserved by SQLJ.

Cause: SQLJ pre-defines several methods on iterators. You cannot use these names in your own methods.

Method not accessible.

Cause: This class has no access to the method.

Action: Check that the access rights of the method are set correctly.

Method not found.

Cause: The method does not exist.

Action: Check the method arguments, or add an overloaded method with the desired arguments.

Missing *count* elements in INTO list: *types*

Cause: The FETCH statement has fewer columns on the fetch cursor than required by the INTO bind variable list.

Missing closing ")" on argument list of stored procedure/function call.

Action: The argument list should be terminated with a ")".

Missing colon.

Cause: There was no colon where one was expected.

Action: Add the missing colon.

Missing comma.

Cause: There was no comma where one was expected.

Action: Add the missing comma.

Missing curly brace.

Cause: There was no opening curly brace where one was expected.

Action: Add the missing opening curly brace.

Missing dot operator.

Cause: There was no dot operator where one was expected.

Action: Add the missing dot operator.

Missing element in INTO list: *element*

Action: You must add *element* to the INTO list.

Missing equal sign in assignment.

Cause: A Java expression is in position of a return variable, but no equal sign follows the expression as required by assignment syntax.

Action: Add the missing assignment operator.

Missing parenthesis.

Cause: There was no opening parenthesis where one was expected.

Action: Add the missing opening parenthesis.

Missing semicolon.

Cause: There was no semicolon where one was expected.

Action: Add the missing semicolon.

Missing square bracket.

Cause: There was no opening square bracket where one was expected.

Action: Add the missing opening square bracket.

Missing terminating "*token*".

Cause: No matching token *token* was found in the SQL statement.

Mode of left-hand-side expression in SET statement was changed to OUT.

Cause: In a `SET :`*x* = ... statement you specified the mode of the host expression *x* as IN or INOUT. This is incorrect.

Action: Either omit the mode, or specify the mode as OUT.

Modifier *modifier* not allowed in declaration.

Cause: Not all modifiers are permitted in a SQLJ class declaration.

Modifier *modifier* not allowed in top-level declarations.

Cause: Not all modifiers are permitted in a SQLJ class declaration.

More than one INTO ... bind list in SQL statement.

Action: Eliminate superfluous INTO ... bind lists.

moving *original filename* to *new filename*

Cause: A backup of the profile was created by the profile conversion utility. The backup file is named *new filename*.

Action: No further action required.

Name '*illegal identifier*' cannot be used as an identifier.

Cause: The string '*illegal identifier*' cannot be used as an identifier because it represents some other language element (for example, operator, punctuation, control structure, etc.).

Action: Use some other name for the identifier.

Negation operator requires boolean operand.

Cause: Negation operator can operate only on a boolean operand.

Action: Check the type of the operand.

No ";" permitted after stored procedure/function call.

Cause: SQLJ does not permit a terminating semicolon after a stored procedure or function invocation.

No connect string specified for context *context*.

Cause: No JDBC connection URL was given for *context*.

Action: Specify a JDBC URL in the `-url@`*context* option, or in the `-user@`*context* option.

No connect string specified.

Cause: No JDBC connection URL was given.

Action: Specify a JDBC URL in the `-url` option, or in the `-user` option.

No connection specified for context *context*. Will attempt to use connection *defaultconnection* instead.

Cause: If no explicit connection information is given for the online checking of *context*, SQLJ will use the values for the default online exemplar schema.

no customizer specified

Cause: Profile customization was requested but no customizer was specified.

Action: Set the profile customizer using the "customizer" or "default-customizer" option.

No instrumentation: class already instrumented.

Cause: This class file was already instrumented with the source locations from the original `.sqlj` file.

No instrumentation: no line info in class.

Cause: This class file does not have any line information and thus cannot be instrumented. Most likely, this happened because you used the -O (optimize) flag to the Java compiler, which will strip line information from the class file.

No INTO variable for column #*pos*: "*name*" *type*

Cause: In a SELECT-INTO statement, the column *name* at position *pos* of type *type* does not have a corresponding Java host expression.

Action: Either expand your INTO-list, or change your SELECT statement.

No offline checker specified for context *context*.

Cause: No offline analysis can be performed for *context*.

No offline checker specified.

Cause: No offline analysis can be performed.

No online checker specified for context *context*. Attempting to use offline checker instead.

Cause: The *context* will be checked offline, even though online checking was requested.

No online checker specified. Attempting to use offline checker instead.

Cause: Offline checking will be performed, even though online checking was requested.

No SQL code permitted after stored procedure/function call. Found: "*token*" ...

Cause: SQLJ does not permit additional statements after a stored procedure or function invocation.

No suitable online checker found for context *context*. Attempting to use offline checker instead.

Cause: None of the online checkers is capable to check *context*.

No suitable online checker found. Attempting to use offline checker instead.

Cause: None of the online checkers is capable to check the default context.

No user specified for context *context*. Will attempt to connect as user *user.*

Cause: If a user is specified for the default context, SQLJ will attempt to check online for all contexts.

not a directory: *name*

Cause: You have directed SQLJ via the `-d` or the `-dir` option to create output files into a directory hierarchy starting with the root directory *name*. Ensure that the root directory exists and is writable.

not a valid input filename: *filename*

Cause: Input files to SQLJ must have the extension ".sqlj", ".java", ".ser", or ".jar".

Not an interface: ***name***

Cause: The name *name* was used in the `implements` clause. However, it does not represent a Java interface.

Not an original sqlj file - no instrumentation.

Cause: The Java file from which the class file was compiled was not generated by the SQLJ translator.

Not found: ***name*****. There is no stored procedure or function of this name.**

Cause: A stored function or procedure could not be found.

option is read only: ***option name***

Cause: An option value was specified for the read-only option named *option name*.

Action: Verify the intended use of the option.

Oracle features used:

Cause: The Oracle customizer "summary" option was enabled. A list of Oracle specific types and features used by the current profile follows this message.

Action: If wider portability is desired, types and features listed may need to be removed from the program.

PLEASE ENTER PASSWORD FOR ***user*** **AT** ***connection*** **>**

Action: You are requested to enter a user password and hit <enter>.

positioned update/delete not supported

Cause: Select and use a ROWID to refer to a particular table row.

Action: A SQL positioned update or delete operation was contained in the profile. This operation cannot be executed by Oracle at runtime.

Premature end-of-file.

Cause: The source file ended before the class declaration was completed.

Action: Check the source file, paying attention to missing quotation marks; correct placement or possible omission of enclosing parenthesis, brackets, or braces; missing comment delimiters; and that it contains at least one valid Java class.

Public class ***class name*** **must be defined in a file called** ***filename*****.sqlj or** ***filename*****.java**

Cause: Java requires that the class name must match with the base name of the source file that contains its definition.

Action: Rename the class or the file.

Public declaration must reside in file with base name *name*, not in the file *file*.

Action: Ensure that the name of the SQLJ file name and the public class name match.

re-installing Oracle customization

Cause: An older version of the Oracle customization was previously installed into the profile being customized. The old customization was replaced with a more recent version.

Action: The profile is ready for use with Oracle. No further action required.

recursive iterators not supported: *iterator name*

Cause: A SQL operation used a recursively defined iterator type. A recursively defined iterator type "A" is an iterator which eventually contains "A" as one of its column types. An iterator is said to eventually contain "A" if it has a column type that is either "A" or an iterator that itself eventually contains "A".

Action: Use an iterator that is not recursive.

registering Oracle customization

Cause: The Oracle customization was installed into the profile being customized.

Action: The profile is ready for use with Oracle. No further action required.

Repeated host item *name* in positions *pos1* and *pos2* in SQL block. Behavior is vendor-defined and non portable.

Cause: The host variable *name* appeared in more than one position with the mode OUT, or INOUT, or it appears with the mode IN as well as OUT or INOUT.

Action: Be aware that host variables are not passed by reference, but each occurrence is passed individually by value-result. To avoid this message, use separate host variables for each OUT or INOUT position.

Result expression must be an lvalue.

Cause: The left-hand side of a SQLJ assignment statement must be an assignable expression. Java variables, fields, and array elements are assignable expressions.

Return type *javatype* of stored function is not legal.

Cause: The stored function returns a Java type *javatype*, which does not refer to a valid Java class.

Return type *type* is not a visible Java type.

Cause: The type *type* is not a publicly visible Java type, and thus no instances of this type can be created and returned from a database driver.

Action: Declare type *type* as public.

Return type *type* is not supported in Oracle SQL.

Cause: The Java type *type* cannot be returned by a SQL statement.

Return type *type* of stored function is not a JDBC output type. This will not be portable.

Cause: Use types as per the JDBC specification for maximum portability.

Return type *type* of stored function is not a visible Java type.

Cause: The type *type* is not a publicly visible Java type, and thus no instances of this type can be created and returned from a database driver.

Action: Declare type *type* as public.

Return type incompatible with SELECT statement: *type* is not an iterator type.

Action: SQL queries that return a value must be assigned to a `java.sql.ResultSet`, or to a positional or named iterator object.

Select list has only *n* elements. Column *type #pos* is not available.

Cause: The database query returns fewer columns than required by the iterator or by an INTO host variable list.

Action: Either change the query, or remove elements from the INTO-list.

Select list has only one element. Column *type #pos* is not available.

Cause: The database query returns fewer columns than required by the iterator or by an INTO host variable list.

Action: Either change the query, or remove elements from the INTO-list.

Shift operator requires integral operands.

Cause: Shift operator can operate only on numeric operands.

Action: Check the types of operands.

Sign operator requires numeric operand.

Cause: Sign operator can operate only on a numeric operand.

Action: Check the type of the operand.

SQL checker did not categorize this statement.

Cause: The specified SQL checker did not determine the nature of this SQL statement.

Action: Your SQL checker should be categorizing every SQL statement. Check the SQL checker that is being used (`-online` and `-offline` options).

SQL checking did not assign mode for host variable #*n* - assuming IN.

Cause: The specified SQL checker did not assign mode information for this host variable. The mode IN is assumed.

Action: Your SQL checker should be assigning modes to all host expressions. Check the SQL checker that is being used (`-online` and `-offline` options).

SQL checking did not assign mode for host variable #*n*.

Cause: The specified SQL checker did not assign mode information for this host variable. The mode IN is assumed.

Action: Your SQL checker should be assigning modes to all host expressions. Check the SQL checker that is being used (`-online` and `-offline` options).

SQL checking did not assign mode for host variable *name* (at position #*n*) - assuming IN.

Cause: The specified SQL checker did not assign mode information for this host variable. The mode IN is assumed.

Action: Your SQL checker should be assigning modes to all host expressions. Check the SQL checker that is being used (`-online` and `-offline` options).

SQL checking did not assign mode for host variable *name* (at position #*n*).

Cause: The specified SQL checker did not assign mode information for this host variable. The mode IN is assumed.

Action: Your SQL checker should be assigning modes to all host expressions. Check the SQL checker that is being used (`-online` and `-offline` options).

SQL statement could not be categorized.

Cause: This SQL statement did not begin with a recognizable SQL or SQLJ keyword, such as SELECT, UPDATE, DELETE, ..., CALL, VALUES, FETCH, CAST, etc.

Action: Check the syntax of your SQL statement.

SQL statement does not return a value.

Cause: The program contained an assignment statement that was neither a query nor a stored function call. Only queries and functions can return immediate results.

SQL statement with INTO ... bind variables can not additionally return a value.

Action: Either remove INTO ... bind list, or remove assignment to an iterator.

SQLJ declarations cannot be inside method blocks.

Cause: Method blocks cannot contain SQLJ declarations.

Action: Move the SQLJ declaration from the method block scope to the class scope or file scope instead (renaming the declared type and all references to it if necessary to avoid ambiguity).

Statement execution expression does not have a Java type.

Cause: No valid Java type could be derived for your execution context expression.

Stored function or procedure syntax does not follow SQLJ specification.

Cause: Stored functions use the VALUES(...) syntax, while stored procedures use the CALL ... syntax.

Action: SQLJ understands your function / procedure syntax. However, if you want your SQLJ program to be maximally portable, you may want to use the documented syntax.

Stored function syntax does not follow SQLJ specification.

Cause: Stored functions use the VALUES(...) syntax.

Action: SQLJ understands your function syntax. However, if you want your SQLJ program to be maximally portable, you may want to use the documented syntax.

Stream column *name #pos* not permitted in SELECT INTO statement.

Cause: You cannot use stream types, such as `sqlj.runtime.AsciiStream`, in a SELECT INTO statement.

Action: For a single stream column, you can use a positional iterator and place the stream column at the end. Alternatively, you can use a named iterator, ensuring that the stream columns (and other columns) are accessed in order.

Syntax [<connection context>, <execution context>, ...] is illegal. Only two context descriptors are permitted.

Action: Use #sql [<connection context>, <execution context>] { ... }; for specifying both connection and execution contexts.

The class prefix is *prefix*, which has the SQLJ reserved shape <file>_SJ.

Cause: You should avoid class names of the form <*file*>`_SJ`<*suffix*>, which are reserved for SQLJ-internal use.

The column *column type* is not nullable, even though it may be NULL in the select list. This may result in a runtime error.

Cause: Nullability in Java does not reflect nullability in the database.

The option value -warn=*value* is invalid. Permitted values are: all, none, nulls, nonulls, precision, noprecision, strict, nostrict, verbose, noverbose.

Action: Use only permitted values in your `-warn` option.

The result set column "*name*" *type* was not used by the named cursor.

Cause: The column *name* of type *type* was selected by the query. However, this column is not required by the named iterator.

Action: Change the query or ignore this message (you can turn it off with the `-warn=nostrict` option).

The tag *tag* in option *option* is invalid. This option does not permit tags.

Action: Only the `-user`, `-url`, `-password`, `-offline`, and `-online` options are used with tags. Specify the option as `-`*option* not as `-`*option@tag*.

The type of the context expression is *type*. It does not implement a connection context.

Cause: A connection context must implement `sqlj.runtime.ConnectionContext`.

The type of the statement execution context is *type*. It does not implement an ExecutionContext.

Cause: An execution context must be an instance of class `sqlj.runtime.ExecutionContext`.

This type is not legal as an IN argument.

Cause: The Java type is supported as an OUT argument but not as an IN argument by your JDBC driver.

This type is not legal as an OUT argument.

Cause: The Java type is supported as an IN argument but not as an OUT argument by your JDBC driver.

Type *type* for column *column* is not a JDBC type. Column declaration is not portable.

Action: Use types as per the JDBC specification for maximum portability.

Type *type* for column *column* is not a valid Java type.

Cause: No valid Java class declaration could be found for *type*.

Type *type* of column *column* is not publicly accessible.

Cause: The Java class *type* of SELECT-list column *column* is not a publicly visible class, and thus cannot be instantiated by a driver.

Action: Use a `public` Java type in the SELECT-list.

Type *type* of host item #*n* is not permitted in JDBC. This will not be portable.

Action: Use types as per the JDBC specification for maximum portability.

Type *type* of host item *item* (at position #*n*) is not permitted in JDBC. This will not be portable.

Action: Use types as per the JDBC specification for maximum portability.

Type *type* of INTO-list item *n* is not publicly accessible.

Cause: The Java class *type* of INTO-list item *n* is not a publicly visible class, and thus cannot be instantiated by a driver.

Action: Use a `public` Java type in the INTO-list.

Type cast operator requires non-void operand.

Cause: A void type cannot be cast to any actual type.

Action: Correct the type of the operand, or remove the cast operation altogether.

Type mismatch in argument #*n* of INTO-list. Expected: *type1* Found: *type2*

Cause: The Java type *type2* of your host expression #*n* in the INTO-list does not match the Java type *type1* prescribed by the positional iterator.

Unable to check SQL query. Error returned by database is: *error*

Cause: The database issued an error message when checking a SQL query against the exemplar schema.

Action: Verify whether the SQL query is correct.

Unable to check SQL statement. Could not parse the SQL statement.

Cause: An error occurred during parsing of a SQL statement, making it impossible to determine the contents of the select list.

Action: Verify the syntax of your SQL query.

Unable to check SQL statement. Error returned by database is: *error*

Cause: The database issued an error message when checking a SQL statement against the exemplar schema.

Action: Verify whether the SQL statement is correct.

Unable to check WHERE clause. Error returned by database is: *error*

Cause: When determining the shape of a query from an exemplar schema, the database issued an error message.

Action: Verify the syntax of your SQL query.

unable to create backup file

Cause: A backup file for the current profile could not be created. This indicates that a new file could not be created in the directory containing the profile. The original profile remains unchanged.

Action: Verify that the directory containing the profile has the proper permissions and rerun the customizer harness. Omit the "backup" option to customize the profile without creating a backup file.

unable to create output file *file*

Action: Ensure that SQLJ has the appropriate permissions to create the file *file*.

unable to create package directory *directory*

Cause: You have directed SQLJ via the `-d` or the `-dir` option to create output files into a directory hierarchy. Ensure that SQLJ is able to create appropriate subdirectories.

unable to delete *filename*

Cause: The profile file *filename* could not be removed by the profile conversion utility.

Action: Verify that the file given by *filename* has the proper permissions.

unable to find input file *filename*

Action: Ensure that file *filename* exists.

Unable to instantiate the offline checker *class*.

Cause: Class *class* does not have a `public` default constructor.

Unable to instantiate the online checker *class*.

Cause: Class *class* does not have a `public` default constructor.

Unable to instrument *args*: *message*

Cause: SQLJ could not instrument the classfile *args* due to some error that occurred during instrumentation.

Action: Ensure that the class file is present, that it is not corrupt, and that it is writable.

unable to load class *class name*: *error description*

Cause: A parameter or iterator column with type *class name* used in this SQL statement could not be loaded by the customizer. To perform customization, the customizer must be able to load all classes used in the SQL operation.

Action: Verify the type *class name* exists in ".class" format, and can be found on the CLASSPATH. Examine *error description* for details of the problem.

Unable to load the offline checker *class*.

Cause: The Java class *class* could not be found.

Unable to load the online checker *class*.

Cause: The Java class *class* could not be found.

unable to move *original filename* to *new filename*

Cause: The profile file *original filename* could not be renamed as *new filename* by the profile conversion utility.

Action: Verify that the files and output directory have the proper permissions.

Unable to obtain DatabaseMetaData to determine the online checker to use for context *context*. Attempting to use offline checker instead.

Cause: JDBC database meta data was unavailable, or did not supply information on the database name and version.

Action: Ensure that you have a proper JDBC driver available.

Unable to obtain description of stored function or procedure: *error.*

Cause: An error occurred when trying to characterize a stored function or procedure invocation.

Action: Ensure that you are calling a proper stored procedure or function. Ensure that you are using an appropriate JDBC driver to check your SQLJ program.

unable to open temporary output file *filename*

Action: Ensure that you can create a temporary file *filename*, and that the directory is writable.

Unable to perform online type checking on weakly typed host item *untypables*

Cause: For each of the Java host expressions, SQLJ determines a corresponding SQL type. These SQL types are required for checking the statement online. When you are using "weak types", SQLJ cannot check your SQL statement online in may cases.

Action: Replace weak types with user-defined types.

Unable to perform semantic analysis on connection *connectionUrl* by user *user.* Error returned by database is: *error*

Cause: SQLJ failed in establishing a connection for online checking.

unable to read input file *filename*

Action: Ensure that the file *filename* exists, and that you have read permissions on it.

Unable to read password from user: *error.*

Cause: An error occurred when reading a user password.

unable to read property file *property file*

Action: You specified a property file in the -props=*property file* option. Ensure that this file exists and is readable.

Unable to read translation state from *file*: *message*

Action: Ensure that SQLJ can create and subsequently read a temporary file *file*.

Unable to remove file *file1* or *file2*

Cause: SQLJ was unable to remove temporary files that it created during translation.

Action: Check the default permissions for newly created files.

unable to remove file *filename*

Cause: During profile customization, a temporary file named *filename* was created that was unable to be removed.

Action: Verify the default permissions for newly created files. Manually remove the temporary file.

unable to rename file *original filename* to *new filename*

Cause: During profile customization, a temporary file named *original filename* could not be renamed *new filename*. This indicates that the customizer harness was unable to replace the original profile or `.jar` file with the customized version.

Action: Verify that the original profile or jar file is writable.

unable to rename output file from *original filename* to *new filename*

Action: Ensure that *new filename* is writable.

Unable to resolve stored function *function* - *n* declarations match this call.

Cause: The stored function invocation matches more than one stored function signature in the database.

Action: Use Java host expressions rather than SQL expressions in the arguments to the stored function to enable signature resolution.

Unable to resolve stored procedure *procedure* - *n* declarations match this call.

Cause: The stored procedure invocation matches more than one stored procedure signature in the database.

Action: Use Java host expressions rather than SQL expressions in the arguments to the stored procedure to enable signature resolution.

Unable to resolve type of WITH attribute *attribute*.

Cause: You used a WITH attribute with your iterator or context declaration. The value of the WITH attribute was not a literal or symbolic constant, which made it impossible for SQLJ to determine the Java type of the attribute.

Action: Use a literal constant or a symbolic constant to specify the value of the WITH attribute.

Unable to write Java compiler command line to *file*: *message*

Action: Ensure that SQLJ can create and subsequently read a temporary file *file*.

Unable to write translation state to *file*: *message*

Action: Ensure that SQLJ can write to a temporary file *file*.

Unbalanced curly braces.

Cause: There was no closing curly brace where one was expected.

Action: Add the missing closing curly brace.

Unbalanced parenthesis.

Cause: There was no closing parenthesis where one was expected.

Action: Add the missing closing parenthesis.

Unbalanced square brackets.

Cause: There was no closing square bracket where one was expected.

Action: Add the missing closing square bracket.

unchanged

Cause: The profile was not modified by the customization process.

Action: Correct errors that prevented customization, if any. Note that some customizers (such as the profile printer) intentionally leave the profile unchanged; in such cases, this is the expected message.

unexpected error occurred...

Action: An unexpected error occurred during SQLJ translation. Contact Oracle if this error persists.

Unexpected token '*unexpected token*' in Java statement.

Cause: Java statement cannot have token '*unexpected token*' in the position in which it appears in the source code.

Action: Check the syntax of the statement.

unknown digest algorithm: *algorithm name*

Cause: An unknown `jar` message digest algorithm was specified in the customizer harness "digests" option.

Action: Verify that *algorithm name* is a valid message digest algorithm and that the corresponding `MessageDigest` implementation class exists in the CLASSPATH.

Unknown identifier '*unknown identifier*'.

Cause: The identifier '*unknown identifier*' has not been defined.

Action: Check the identifier for typing errors, and/or make sure that it has been defined.

Unknown identifier.

Cause: The identifier has not been defined.

Action: Check the identifier for typing errors, and/or make sure that it has been defined.

unknown option found in *location*: *name*

Action: Ensure that you are using a valid SQLJ option. Run sqlj `-help-long` to obtain a list of supported options.

unknown option type: *option name*

Cause: The option named *option name* could not be handled by the customizer harness. This often indicates a non-standard, customizer-specific option for which an appropriate JavaBeans property editor could not be found.

Action: Verify that property editors associated with the current customizer are accessible on the CLASSPATH. As a workaround, discontinue use of the option or use a different customizer.

Unknown target type in cast expression.

Cause: The target type of the cast operation has not been defined.

Action: Verify the type name and/or make sure that it has been defined.

unrecognized option: *option*

Cause: An unknown option was given to the profile conversion utility.

Action: Verify that the option is spelled correctly.

Unrecognized SET TRANSACTION syntax at "*token*" ...

Cause: SQLJ was not able to understand this SET TRANSACTION statement.

Action: If you rely on SQLJ to recognize this particular SET TRANSACTION clause, you should use the documented syntax.

Unrecognized SET TRANSACTION syntax.

Cause: SQLJ was not able to understand this SET TRANSACTION statement.

Action: If you rely on SQLJ to recognize this particular SET TRANSACTION clause, you should use the documented syntax.

Unrecognized SQL statement: *keyword*

Cause: The SQL statement was introduced with the keyword *keyword*. Neither SQLJ nor the JDBC driver recognized it as a SQL keyword.

Action: Check your SQL statement. If this is a vendor-specific keyword that neither your JDBC driver nor your SQL checker knows about, you can ignore this message.

Unsupported file encoding

Action: Ensure that the encoding specified in the `-encoding` option is supported by your Java VM.

Unsupported Java type for host item #*n*: *type*.

Cause: The Java type *type* is not supported as a host item by your JDBC driver.

Action: Use a different Java type in your host expression. Possibly update your JDBC driver.

Unsupported Java type for host item *name* (at position #*n*): *type*.

Cause: The Java type *type* is not supported as a host item by your JDBC driver.

Action: Use a different Java type in your host expression. Possibly update your JDBC driver.

Unsupported Java type for item #*pos* of INTO-list: *type*.

Cause: The Java class *type* of INTO-list item *pos* is not supported by your JDBC driver.

Action: Use supported Java types in the INTO-list. Possibly update your JDBC driver.

Unterminated comment.

Cause: The source file ended in a comment before the class declaration was completed.

Action: Check the source file for a missing comment delimiter.

valid Oracle customization exists

Cause: A valid Oracle customization was previously installed into the profile being customized. The profile was not modified.

Action: The profile is ready for use with Oracle. No further action required.

Value of iterator attribute *attribute* must be a boolean.

Action: This iterator `with`-clause attribute requires a boolean value. Specify one of: *attribute*=`true`, or *attribute*=`false`.

Value of iterator attribute updateColumns must be a String containing a list of column names.

Action: Declare the `updateColumns` attribute in your iterators `with`-clause as follows: `updateColumns="col1,col2,col3"` where the column names represent the updatable columns.

Value of the iterator with-clause attribute sensitivity must be one of SENSITIVE, ASENSITIVE, or INSENSITIVE.

Action: To set `sensitivity`, specify one of: `sensitivity=SENSITIVE`, `sensitivity=ASENSITIVE`, or `sensitivity=INSENSITIVE` on the `with`-clause of your iterator declaration.

Value returned by SQL query is not assigned to a variable.

Cause: User is ignoring the result returned by a query.

Action: Verify your SQL statement, and that it is your intention to discard the result of the SELECT.

Value returned by SQL stored function is not assigned to a variable.

Cause: User is ignoring the result returned by a stored function call.

Action: Verify your SQL statement, and that it is your intention to discard the result of a stored function call.

You are using a non-Oracle JDBC driver to connect to an Oracle database. Only JDBC-generic checking will be performed.

Cause: In order to perform Oracle-specific checking, an Oracle JDBC driver is required.

You are using an Oracle 8.0 JDBC driver, but connecting to an Oracle7 database. SQLJ will use Oracle7 specific SQL checking.

Cause: Translation with an online connection will automatically be limited to the features of the database that you are connected to.

Action: If you use the Oracle 8.0 JDBC driver but also want to connect to Oracle7 databases, you may want to explicitly specify `oracle.sqlj.checker.Oracle7OfflineChecker` and `oracle.sqlj.checker.Oracle7JdbcChecker` for offline and online checking, respectively.

You are using an Oracle 8.1 JDBC driver, but are not connecting to an Oracle8 or Oracle7 database. SQLJ will perform JDBC-generic SQL checking.

Cause: This version of SQLJ does not recognize the database you are connecting to.

Action: Connect to an Oracle7 or Oracle8 database.

You are using an Oracle 8.1 JDBC driver, but connecting to an Oracle7 database. SQLJ will use Oracle7 specific SQL checking.

Cause: Translation with an online connection will automatically be limited to the features of the database that you are connected to.

Action: If you use the Oracle 8.1 JDBC driver but also want to connect to Oracle7 databases, you may want to explicitly specify `oracle.sqlj.checker.Oracle8To7OfflineChecker` and `oracle.sqlj.checker.Oracle8To7JdbcChecker` for offline and online checking, respectively.

You are using an Oracle JDBC driver, but connecting to an non-Oracle database. SQLJ will perform JDBC-generic SQL checking.

Cause: This version of SQLJ does not recognize the database you are connecting to.

Action: Connect to an Oracle7 or Oracle8 database

You cannot specify both, source files (.sqlj,.java) and profile files (.ser,.jar)

Cause: Either use SQLJ to translate, compile, and customize `.sqlj` and `.java` source files, or use SQLJ to customize profile files by specifying `.ser` files and `.jar` archives containing `.ser` files, but not both.

Runtime Messages

This section provides a list of error messages your users may encounter from the SQLJ runtime, including SQL state, cause, and action information.

See "Retrieving SQL States and Error Codes" on page 4-22 for information about SQL states.

java.io.InvalidObjectException: invalid descriptor: *descriptor value*

Cause: In the loading of a profile object, it was determined that the descriptor object of one of the SQL operations was invalid. This suggests that the profile does not conform to the standard, or was read from a corrupted file.

Action: Recreate the profile by retranslating the original source file.

java.io.InvalidObjectException: invalid execute type: *type value*

Cause: In the loading of a profile object, it was determined that the method used to execute one of the SQL operations was invalid. This suggests that the profile does not conform to the standard, or was read from a corrupted file.

Action: Recreate the profile by retranslating the original source file.

java.io.InvalidObjectException: invalid modality: *mode value*

Cause: In the loading of a profile object, it was determined that the modality of one of the SQL operation parameters was invalid. This suggests that the profile does not conform to the standard, or was read from a corrupted file.

Action: Recreate the profile by retranslating the original source file.

java.io.InvalidObjectException: invalid result set type: *type value*

Cause: In the loading of a profile object, it was determined that the type of result produced by of one of the SQL operations was invalid. This suggests that the profile does not conform to the standard, or was read from a corrupted file.

Action: Recreate the profile by retranslating the original source file.

java.io.InvalidObjectException: invalid role: *role value*

Cause: In the loading of a profile object, it was determined that the contents of one of the SQL operations was invalid. This suggests that the profile does not conform to the standard, or was read from a corrupted file.

Action: Recreate the profile by retranslating the original source file.

java.io.InvalidObjectException: invalid statement type: *type value*

Cause: In the loading of a profile object, it was determined that the statement type of one of the SQL operations was invalid. This suggests that the profile does not conform to the standard, or was read from a corrupted file.

Action: Recreate the profile by retranslating the original source file.

java.lang.ClassNotFoundException: not a profile: *profile name*

Cause: The object created as the profile named *profile name* cannot be used as a profile. This error suggests that the file containing the profile has unknown data or has been corrupted.

Action: Recreate the profile by retranslating the original source file.

java.lang.ClassNotFoundException: unable to instantiate profile *profile name*

Cause: The profile named *profile name* exists but could not be instantiated. This suggests that the profile contains invalid data or was read from a corrupted file.

Action: Recreate the profile by retranslating the original source file.

java.lang.ClassNotFoundException: unable to instantiate serialized profile *profile name*

Cause: The profile named *profile name* exists as type `sqlj.runtime.SerializedProfile`, but could not be instantiated. A profile of this type usually indicates that the profile has been converted to `.class` format. This error suggests that the profile contains invalid data or was read from a corrupted file.

Action: Recreate the profile by retranslating the original source file. Use the `ser2class` option if the profiles should be created in `.class` format.

java.sql.SQLException: expected *x* columns in select list but found *y*

SQL State: 42122

Cause: The query executed selects *x* items, but has *y* INTO-list items or is assigned to an iterator containing *y* columns.

Action: Correct the program so that the number of INTO-list items or iterator columns matches the number of items selected.

java.sql.SQLException: expected instance of ForUpdate iterator at parameter *x*, found class *class name*

SQL State: 46130

Cause: A positional SQL operation contained a host expression with runtime type *class name* as the target of the CURRENT OF clause. The *class name* must be an instance of the `sqlj.runtime.ForUpdate` interface.

Action: Update the declaration of the iterator type passed as the target of the CURRENT OF clause. Include the `ForUpdate` interface in the implements clause.

java.sql.SQLException: expected statement with no OUT parameters: {*statement*}

SQL State: 46130

Cause: A SQL operation unexpectedly contained one or more OUT or INOUT parameters. This indicates an operation that does not conform to the SQLJ runtime standard, and may require a special customization to be executed. Alternatively, the profile may have been read from a corrupted file.

Action: Verify the original SQL operation is valid. Retranslate the source file or install a customization that supports the extended functionality.

java.sql.SQLException: expected statement with OUT parameters: {*statement*}

SQL State: 46130

Cause: A SQL operation contained no OUT or INOUT parameters when it was expected to have at least one. This indicates an operation that does not conform to the SQLJ runtime standard, and may require a special customization to be executed. Alternatively, the profile may have been read from a corrupted file.

Action: Verify the original SQL operation is valid. Retranslate the source file or install a customization that supports the extended functionality.

java.sql.SQLException: expected statement {*statement*} to be executed via executeQuery

SQL State: 46130

Cause: A SQL operation was unexpectedly requested to produce an update count instead of a result set. This indicates an operation that does not conform to the SQLJ runtime standard, and may require a special customization to be executed. Alternatively, the profile may have been read from a corrupted file.

Action: Verify the original SQL operation is valid. Retranslate the source file or install a customization that supports the extended functionality.

java.sql.SQLException: expected statement {*statement*} to be executed via executeUpdate

SQL State: 46130

Cause: A SQL operation was unexpectedly requested to produce a result set instead of an update count. This indicates an operation that does not conform to the SQLJ runtime standard, and may require a special customization to be executed. Alternatively, the profile may have been read from a corrupted file.

Action: Verify the original SQL operation is valid. Retranslate the source file or install a customization that supports the extended functionality.

java.sql.SQLException: expected statement {*statement*} to use *x* parameters, found *y*

SQL State: 46130

Cause: A SQL operation that was expected to contain *y* host expressions was found to contain *x* host expressions instead. This indicates an operation that does not conform to the SQLJ runtime standard, and may require a special customization to be executed. Alternatively, the profile may have been read from a corrupted file.

Action: Verify the original SQL operation is valid. Retranslate the source file or install a customization that supports the extended functionality.

java.sql.SQLException: found null connection context

SQL State: 08003

Cause: The connection context instance used in an executable SQL statement was null.

Action: Initialize the connection context instance to a non-null value. If the SQL statement uses an implicit connection context, it is initialized using the static `setDefaultContext` method of the `sqlj.runtime.ref.DefaultContext` class.

java.sql.SQLException: found null execution context

SQL State: 08000

Cause: The execution context instance used in an executable SQL statement was null.

Action: Initialize the execution context instance to a non-null value.

java.sql.SQLException: Invalid column name

SQL State: 46121

Cause: There was a mismatch between a column name declared in the named iterator used in this SQL operation and a column name contained in the underlying result set. Each column of a named iterator must uniquely case-insensitive match the name of a column in the underlying result set.

Action: Change either the name of the column in the named iterator, or the name of the column in the associated query, so that they match.

java.sql.SQLException: invalid iterator type: *type name*

SQL State: 46120

Cause: An object returned or used by this SQL operation with type *type name* was not a valid iterator type. This may indicate that the iterator class was produced by a non-standard translator.

Action: Verify the original SQL operation and the iterator types it uses are valid. Retranslate the source files as needed.

java.sql.SQLException: key is not defined in connect properties: *key name*

SQL State: 08000

Cause: The key named *key name* was not defined in the connection properties resource file. Information contained in the connection properties resource file is used to establish a database connection, and must include a key named *key name*.

Action: Add the key *key name* to the connection properties file with an appropriate value for the desired connection.

java.sql.SQLException: multiple rows found for select into statement

SQL State: 21000

Cause: The execution of a SELECT INTO statement produced a result that contained more than one row.

Action: Correct the SELECT INTO query or queried data so that exactly one row is selected.

java.sql.SQLException: no rows found for select into statement

SQL State: 02000

Cause: The execution of a SELECT INTO statement produced a result that contained no rows.

Action: Correct the SELECT INTO query or queried data so that exactly one row is selected.

java.sql.SQLException: null connection

SQL State: 08000

Cause: A null SQLJ connection context or JDBC connection object was passed to the constructor of a connection context class.

Action: If a JDBC connection is used, establish a database connection with the JDBC connection object before passing it to the connection context constructor. For Oracle JDBC drivers, this is done using one of the static `getConnection` methods of the `java.sql.DriverManager` class. If a connection context object is used, make sure it has been properly initialized before passing it to the constructor. If the default connection context is used, call `setDefaultContext` before using the default context.

java.sql.SQLException: profile *profile name* not found: *error description*

SQL State: 46130

Cause: The profile named *profile name* could not be found or instantiated. The problem is further explained by *error description*.

Action: Consult the recommended action for the problem detail given by *error description*.

java.sql.SQLException: unable to convert database class *found type* to client class *expected type*

SQL State: 22005

Cause: The default mapping from a database type into a Java object produced class *found type* when class *expected type* was required by the host expression. This often indicates a failed conversion to the client-side class `java.math.BigDecimal`. It may also indicate a failed conversion to a non-standard class that is only supported when a particular customization is installed.

Action: Verify that the database type selected has a default mapping assignable to the type of host variable or iterator column fetched into. This may require the use of a different client-side type. Verify that the customization required to support the client-side type, if any, is installed.

java.sql.SQLException: Unable to create CallableStatement for RTStatement

SQL State: 46110

Cause: Execution of this SQL operation requires the use of a JDBC `CallableStatement` object at runtime. However, such an object was not available from the customization used to execute the operation. This indicates that incompatible customizations may have been installed into your application, or that the operation may require the use of a special customization.

Action: Retranslate the source file or install a customization that supports the extended functionality.

java.sql.SQLException: Unable to create PreparedStatement for RTStatement

SQL State: 46110

Cause: Execution of this SQL operation requires the use of a JDBC `PreparedStatement` object at runtime. However, such an object was not available from the customization used to execute the operation. This indicates that incompatible customizations may have been installed into your application, or that the operation may require the use of a special customization.

Action: Retranslate the source file or install a customization that supports the extended functionality.

java.sql.SQLException: unable to load connect properties file: *filename*

SQL State: 08000

Cause: The connection properties file named *filename* could not be loaded as a resource file. It is used to establish a database connection. Since it is loaded as an application resource file, it must be packaged with the application classes. This message indicates that the file does not exist in the expected location or is not readable.

Action: Verify that the connection properties file is readable and packaged with the application classes.

java.sql.SQLException: unexpected call to method *method name*

SQL State: 46130

Cause: The execution of a SQL operation unexpectedly involved a call to method *method name*. This indicates an operation that does not conform to the SQLJ runtime standard, and may require a special customization to be executed. It may also indicate the use of a non-standard SQLJ translator.

Action: Verify the original SQL operation is valid. Retranslate the source file or install a customization that supports the extended functionality.

java.sql.SQLException: unexpected exception raised by constructor *constructor name* : *exception description*

SQL State: 46120

Cause: The construction of a runtime result or output parameter resulted in a runtime exception being thrown by the constructor.

Action: Examine the contents of *exception description* to determine the cause of the exception.

java.sql.SQLException: unexpected exception raised by method *method name* : *exception description*

SQL State: 46120

Cause: The conversion of a host expression to or from a database type involved in a call to method *method name*, which raised an exception other than a SQLException.

Action: Examine the contents of *exception description* to determine the cause of the exception.

sqlj.runtime.SQLNullException: cannot fetch null into primitive data type

SQL State: 22002

Cause: Attempted to store a SQL NULL into Java primitive iterator column type, result, OUT parameter, or INOUT parameter.

Action: Use a nullable Java wrapper type instead of the primitive type.

Index