

Oracle9i

SQLJ Developer's Guide and Reference

Release 2 (9.2)

March 2002

Part No. A96655-01

ORACLE

Oracle9i SQLJ Developer's Guide and Reference, Release 2 (9.2)

Part No. A96655-01

Copyright © 1999, 2002 Oracle Corporation. All rights reserved.

Primary Author: Brian Wright

Contributing Authors: Janice Nygard, Ekkehard Rohwedder

Contributors: Brian Becker, Alan Thiesen, Lei Tang, Julie Basu, Pierre Dufour, Quan Wang, Jerry Schwarz, Risto Lankinen, Cheuk Chau, Vishu Krishnamurthy, Rafiul Ahad, Jack Melnick, Tim Smith, Thomas Pfaeffle, Tom Portfolio, Ellen Barnes, Susan Kraft, Sheryl Maring, Angie Long

The Programs (which include both the software and documentation) contain proprietary information of Oracle Corporation; they are provided under a license agreement containing restrictions on use and disclosure and are also protected by copyright, patent and other intellectual and industrial property laws. Reverse engineering, disassembly or decompilation of the Programs, except to the extent required to obtain interoperability with other independently created software or as specified by law, is prohibited.

The information contained in this document is subject to change without notice. If you find any problems in the documentation, please report them to us in writing. Oracle Corporation does not warrant that this document is error-free. Except as may be expressly permitted in your license agreement for these Programs, no part of these Programs may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Oracle Corporation.

If the Programs are delivered to the U.S. Government or anyone licensing or using the programs on behalf of the U.S. Government, the following notice is applicable:

Restricted Rights Notice Programs delivered subject to the DOD FAR Supplement are "commercial computer software" and use, duplication, and disclosure of the Programs, including documentation, shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement. Otherwise, Programs delivered subject to the Federal Acquisition Regulations are "restricted computer software" and use, duplication, and disclosure of the Programs shall be subject to the restrictions in FAR 52.227-19, Commercial Computer Software - Restricted Rights (June, 1987). Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065.

The Programs are not intended for use in any nuclear, aviation, mass transit, medical, or other inherently dangerous applications. It shall be the licensee's responsibility to take all appropriate fail-safe, backup, redundancy, and other measures to ensure the safe use of such applications if the Programs are used for such purposes, and Oracle Corporation disclaims liability for any damages caused by such use of the Programs.

Oracle is a registered trademark, and Oracle9i, Oracle8i, Oracle8, Oracle7, PL/SQL, SQL*Plus, and Oracle Store are trademarks or registered trademarks of Oracle Corporation. Other names may be trademarks of their respective owners.

Contents

Send Us Your Comments	xv
Preface.....	xvii
Intended Audience	xviii
Documentation Accessibility	xviii
Organization.....	xix
Related Documentation	xxi
Conventions.....	xxiv
1 Overview	
Introduction to SQLJ.....	1-2
Basic Concepts.....	1-2
Oracle-Specific Code Generation Versus ISO Standard Code Generation	1-3
Java and SQLJ Versus PL/SQL.....	1-3
Overview of SQLJ Components	1-5
SQLJ Translator and SQLJ Runtime.....	1-5
SQLJ Profiles (ISO Standard Code).....	1-6
Overview of Oracle Extensions to the SQLJ Standard	1-8
Basic Translation Steps and Runtime Processing.....	1-10
SQLJ Translation Steps	1-10
Summary of Translator Input and Output	1-13
SQLJ Runtime Processing.....	1-16
JDBC Versus SQLJ Sample Code.....	1-18
JDBC Version of the Sample Code	1-19

SQLJ Version of the Sample Code.....	1-22
Alternative Deployment Scenarios	1-25
Running SQLJ in Applets	1-25
Introduction to SQLJ in the Server	1-29
Using SQLJ with Oracle9i Lite	1-30
Alternative Development Scenarios	1-32
SQLJ Globalization Support.....	1-32
SQLJ in Oracle9i JDeveloper and Other IDEs.....	1-32
Windows Considerations	1-33

2 Getting Started

Assumptions and Requirements	2-2
Assumptions About Your Environment	2-2
Requirements for Using Oracle SQLJ	2-3
Oracle SQLJ Environment: Key Scenarios and Guidelines	2-4
Environment Issues and Limitations	2-6
Oracle SQLJ Backward Compatibility	2-9
Checking the Installation and Configuration	2-11
Check for Installed Directories and Files	2-11
Set the Path and Classpath	2-12
Verify Installation of sqljutil Package	2-13
Testing the Setup	2-14
Set Up the Runtime Connection	2-14
Create a Table to Verify the Database.....	2-16
Verify the JDBC Driver	2-16
Verify the SQLJ Translator and Runtime	2-17
Verify the SQLJ Translator Connection to the Database.....	2-17

3 Basic Language Features

Overview of SQLJ Declarations	3-2
Rules for SQLJ Declarations	3-2
Iterator Declarations.....	3-3
Connection Context Declarations.....	3-4
Declaration IMPLEMENTS Clause	3-5
Declaration WITH Clause.....	3-6

Overview of SQLJ Executable Statements	3-9
Rules for SQLJ Executable Statements	3-9
SQLJ Clauses	3-10
Specifying Connection Context Instances and Execution Context Instances	3-11
Executable Statement Examples.....	3-12
PL/SQL Blocks in Executable Statements.....	3-14
Java Host Expressions, Context Expressions, and Result Expressions	3-16
Overview of Host Expressions	3-16
Basic Host Expression Syntax.....	3-17
Examples of Host Expressions.....	3-20
Overview of Result Expressions and Context Expressions.....	3-21
Evaluation of Java Expressions at Runtime	3-22
Examples of Evaluation of Java Expressions at Runtime (ISO Code Generation)	3-24
Restrictions on Host Expressions	3-33
Single-Row Query Results: SELECT INTO Statements	3-34
SELECT INTO Syntax	3-34
Examples of SELECT INTO Statements.....	3-35
Examples with Host Expressions in SELECT-List.....	3-35
SELECT INTO Error Conditions	3-36
Multi-Row Query Results: SQLJ Iterators	3-37
Iterator Concepts	3-37
General Steps in Using an Iterator	3-41
Named Iterators Versus Positional Iterators Versus Result Set Iterators.....	3-42
Using Named Iterators	3-43
Using Positional Iterators	3-48
Using Iterators and Result Sets as Host Variables.....	3-52
Using Iterators and Result Sets as Iterator Columns.....	3-55
Assignment Statements (SET)	3-58
Stored Procedure and Function Calls	3-60
Calling Stored Procedures.....	3-60
Calling Stored Functions	3-61
Using Iterators and Result Sets as Stored Function Returns.....	3-63

4 Key Programming Considerations

Selection of the JDBC Driver	4-2
---	-----

Overview of the Oracle JDBC Drivers	4-2
Driver Selection for Translation.....	4-4
Driver Selection and Registration for Runtime	4-5
Connection Considerations	4-6
Single Connection or Multiple Connections Using DefaultContext	4-6
Closing Connections.....	4-11
Multiple Connections Using Declared Connection Context Classes	4-12
More About the Oracle Class	4-12
More About the DefaultContext Class	4-14
Connection for Translation.....	4-17
Connection for Customization.....	4-18
Null-Handling	4-19
Wrapper Classes for Null-Handling	4-19
Examples of Null-Handling	4-20
Exception-Handling Basics	4-22
SQLJ and JDBC Exception-Handling Requirements	4-22
Processing Exceptions.....	4-23
Using SQLException Subclasses.....	4-25
Basic Transaction Control	4-26
Overview of Transactions.....	4-26
Automatic Commits Versus Manual Commits	4-26
Specifying Auto-Commit as You Define a Connection.....	4-27
Modifying Auto-Commit in an Existing Connection	4-28
Using Manual COMMIT and ROLLBACK	4-28
Effect of Commits and Rollbacks on Iterators and Result Sets	4-29
Using Savepoints	4-30
Summary: First Steps in SQLJ Code	4-31
Import Required Classes.....	4-31
Register JDBC Drivers and Set Default Connection	4-32
Set Up Exception Handling.....	4-32
Set Up Host Variables, Execute SQLJ Clause, Process Results	4-33
Example of Single-Row Query using SELECT INTO	4-34
Set Up a Named Iterator	4-35
Example of Multiple-Row Query Using Named Iterator	4-36
Oracle-Specific Code Generation (No Profiles)	4-39

Advantages and Disadvantages of Oracle-Specific Code Generation.....	4-39
Environment Requirements for Oracle-Specific Code Generation	4-40
Code Considerations and Limitations with Oracle-Specific Code Generation.....	4-41
SQLJ Usage Changes with Oracle-Specific Code Generation.....	4-42
Server-Side Considerations with Oracle-Specific Code Generation	4-44
Requirements and Restrictions for Naming	4-45
Java Namespace: Local Variable and Class Naming Restrictions.....	4-45
SQLJ Namespace	4-46
SQL Namespace.....	4-47
File Name Requirements and Restrictions.....	4-47
Considerations for SQLJ in the Middle Tier	4-48

5 Type Support

Supported Types for Host Expressions	5-2
Summary of Supported Types.....	5-2
Supported Types and Requirements for JDBC 2.0	5-8
Using PL/SQL BOOLEAN, RECORD Types, and TABLE Types.....	5-9
Backward Compatibility for Previous Oracle JDBC Releases.....	5-11
Support for Streams	5-14
General Use of SQLJ Streams.....	5-14
Key Aspects of Stream Support Classes.....	5-15
Using SQLJ Streams to Send Data.....	5-16
Retrieving Data into Streams: Precautions	5-19
Using SQLJ Streams to Retrieve Data.....	5-20
Stream Class Methods.....	5-22
Examples of Retrieving and Processing Stream Data	5-24
SQLJ Stream Objects as Output Parameters and Function Return Values	5-26
Support for JDBC 2.0 LOB Types and Oracle Type Extensions	5-29
Package oracle.sql.....	5-30
Support for BLOB, CLOB, and BFILE.....	5-30
Support for Oracle ROWID.....	5-37
Support for Oracle REF CURSOR Types	5-40
Support for Other Oracle9i Datatypes.....	5-42
Extended Support for BigDecimal	5-42

6 Objects, Collections, and OPAQUE Types

Oracle Objects and Collections	6-2
Introduction to Objects and Collections	6-2
Oracle Object Fundamentals	6-4
Oracle Collection Fundamentals	6-4
Object and Collection Datatypes	6-5
Custom Java Classes	6-6
Custom Java Class Interface Specifications	6-6
Custom Java Class Support for Object Methods	6-10
Custom Java Class Requirements	6-11
Compiling Custom Java Classes	6-17
Reading and Writing Custom Data	6-17
Additional Uses for ORADData Implementations	6-18
User-Defined Types	6-23
Creating Object Types	6-23
Creating Collection Types	6-25
JPublisher and the Creation of Custom Java Classes	6-28
What JPublisher Produces	6-29
Generating Custom Java Classes	6-32
JPublisher INPUT Files and Properties Files	6-42
Creating Custom Java Classes and Specifying Member Names	6-45
JPublisher Implementation of Wrapper Methods	6-46
JPublisher Custom Java Class Examples	6-47
Extending Classes Generated by JPublisher	6-51
Strongly Typed Objects and References in SQLJ Executable Statements	6-55
Selecting Objects and Object References into Iterator Columns	6-55
Updating an Object	6-57
Inserting an Object Created from Individual Object Attributes	6-59
Updating an Object Reference	6-60
Strongly Typed Collections in SQLJ Executable Statements	6-62
Accessing Nested Tables: TABLE syntax and CURSOR syntax	6-62
Inserting a Row that Includes a Nested Table	6-63
Selecting a Nested Table into a Host Expression	6-64
Manipulating a Nested Table Using TABLE Syntax	6-65
Selecting Data from a Nested Table Using a Nested Iterator	6-67

Selecting a VARRAY into a Host Expression	6-69
Inserting a Row that Includes a VARRAY	6-70
Serialized Java Objects	6-71
Serializing Java Classes to RAW and BLOB Columns	6-71
SerializableDatum: an ORADData Implementation	6-74
SerializableDatum in SQLJ Applications	6-77
SerializableDatum (Complete Class)	6-78
Weakly Typed Objects, References, and Collections	6-80
Support for Weakly Typed Objects, References, and Collections	6-80
Restrictions on Weakly Typed Objects, References, and Collections	6-81
Oracle OPAQUE Types	6-82

7 Advanced Language Features

Connection Contexts	7-2
Connection Context Concepts	7-2
Connection Context Logistics	7-4
More About Declaring and Using a Connection Context Class	7-5
Example of Multiple Connection Contexts	7-7
Implementation and Functionality of Connection Context Classes	7-9
Using the IMPLEMENTS Clause in Connection Context Declarations	7-11
Semantics-Checking of Your Connection Context Usage	7-12
Standard Data Source Support	7-13
SQLJ-Specific Data Sources	7-16
SQLJ-Specific Connection JavaBeans for JavaServer Pages	7-20
Execution Contexts	7-24
Relation of Execution Contexts to Connection Contexts	7-25
Creating and Specifying Execution Context Instances	7-26
Execution Context Synchronization	7-27
Execution Context Methods	7-28
Relation of Execution Contexts to Multithreading	7-33
Multithreading in SQLJ	7-35
Iterator Class Implementation and Advanced Functionality	7-38
Implementation and Functionality of Iterator Classes	7-38
Using the IMPLEMENTS Clause in Iterator Declarations	7-40
Support for Subclassing of Iterator Classes	7-40

Result Set Iterators	7-41
Scrollable Iterators	7-42
Advanced Transaction Control	7-49
SET TRANSACTION Syntax	7-49
Access Mode Settings	7-50
Isolation Level Settings	7-50
Using JDBC Connection Class Methods.....	7-51
SQLJ and JDBC Interoperability	7-53
SQLJ Connection Context and JDBC Connection Interoperability	7-53
SQLJ Iterator and JDBC Result Set Interoperability	7-58
Support for Dynamic SQL	7-63
Meta Bind Expressions.....	7-63
SQLJ Dynamic SQL Examples	7-65

8 Translator Command Line and Options

Translator Command Line and Properties Files	8-2
SQLJ Options, Flags, and Prefixes.....	8-3
Command-Line Syntax and Operations.....	8-12
Properties Files for Option Settings	8-16
SQLJ_OPTIONS Environment Variable for Option Settings	8-19
Order of Precedence of Option Settings.....	8-20
Basic Translator Options	8-22
Basic Options for the Command Line Only.....	8-22
Options for Output Files and Directories.....	8-29
Connection Options.....	8-34
Options for Reporting and Line-Mapping.....	8-45
Options for Code Generation, Optimizations, and CHAR Comparisons	8-51
Advanced Translator Options	8-60
Prefixes that Pass Option Settings to Other Executables	8-60
Flags for Special Processing	8-65
Semantics-Checking and Offline-Parsing Options	8-71
Translator Support and Options for Alternative Environments	8-80
Java and Compiler Options.....	8-80
Customization Options.....	8-87

9 Translator and Runtime Functionality

Internal Translator Operations	9-2
Java and SQLJ Code-Parsing and Syntax-Checking.....	9-2
SQL Semantics-Checking and Offline Parsing.....	9-2
Code Generation.....	9-5
Java Compilation.....	9-9
Profile Customization (ISO Code Generation).....	9-10
Functionality of Translator Errors, Messages, and Exit Codes	9-12
Translator Error, Warning, and Information Messages.....	9-12
Translator Status Messages.....	9-15
Translator Exit Codes.....	9-15
SQLJ Runtime	9-16
SQLJ Runtime Packages.....	9-16
Categories of Runtime Errors.....	9-18
Globalization Support in the Translator and Runtime	9-19
Character Encoding and Language Support.....	9-19
SQLJ and Java Settings for Character Encoding and Language Support.....	9-22
Oracle SQLJ Extended Globalization Support.....	9-25
Manipulation Outside of SQLJ for Globalization Support.....	9-29

10 Performance and Debugging

Performance Enhancement Features	10-2
Row Prefetching.....	10-3
Statement Caching.....	10-4
Update Batching.....	10-12
Column Definitions.....	10-23
Parameter Size Definitions.....	10-25
SQLJ Debugging Features	10-28
SQLJ -linemap Flag for Debugging.....	10-28
Server-Side debug Option.....	10-29
Introduction to the AuditorInstaller Specialized Customizer.....	10-29
Introduction to Developing and Debugging in Oracle9i JDeveloper.....	10-29

11 SQLJ in the Server

Introduction to Server-Side SQLJ	11-2
Creating SQLJ Code for Use in the Server	11-4
Database Connections within the Server	11-4
Coding Issues within the Server.....	11-5
Default Output Device in the Server.....	11-6
Name Resolution in the Server	11-7
SQL Names Versus Java Names.....	11-8
Translating SQLJ Source on a Client and Loading Components	11-9
Loading Classes and Resources into the Server	11-9
Naming of Loaded Class and Resource Schema Objects.....	11-11
Publishing the Application After Loading Class and Resource Files	11-14
Summary: Running a Client Application in the Server	11-14
Loading SQLJ Source and Translating in the Server	11-16
Loading SQLJ Source Code into the Server	11-16
Option Support in the Server Embedded Translator	11-18
Naming of Loaded Source and Generated Class and Resource Schema Objects.....	11-22
Error Output from the Server Embedded Translator	11-24
Publishing the Application After Loading Source Files	11-24
Dropping Java Schema Objects	11-25
Additional Considerations	11-26
Java Multithreading in the Server	11-26
Recursive SQLJ Calls in the Server.....	11-26
Verifying that Code is Running in the Server	11-28

A Customization and Specialized Customizers

More About Profiles	A-2
Creation of a Profile During Code Generation.....	A-2
Sample Profile Entry.....	A-3
More About Profile Customization	A-5
Overview of the Customizer Harness and Customizers	A-5
Steps in the Customization Process	A-6
Creation and Registration of a Profile Customization	A-7
Customization Error and Status Messages	A-9
Functionality of a Customized Profile at Runtime	A-9

Customization Options and Choosing a Customizer	A-11
Overview of Customizer Harness Options.....	A-12
General Customizer Harness Options.....	A-14
Customizer Harness Options for Connections.....	A-18
Customizer Harness Options that Invoke Specialized Customizers	A-21
Overview of Customizer-Specific Options	A-23
Oracle Customizer Options.....	A-24
Options for Other Customizers	A-37
SQLJ Translator Options for Profile Customization.....	A-37
JAR Files for Profiles	A-38
JAR File Requirements.....	A-38
JAR File Logistics.....	A-39
SQLCheckerCustomizer for Profile Semantics-Checking	A-40
Invoking SQLCheckerCustomizer with the Customizer Harness verify Option	A-40
SQLCheckerCustomizer Options.....	A-41
AuditorInstaller Customizer for Debugging	A-44
Overview of Auditors and Code Layers	A-44
Invoking AuditorInstaller with the Customizer Harness debug Option.....	A-45
AuditorInstaller Runtime Output.....	A-46
AuditorInstaller Options	A-47
Full Command-Line Examples.....	A-51
B SQLJ Error Messages	
SQLJ Translation Time Messages.....	B-2
SQLJ Runtime Messages.....	B-47

Index

Send Us Your Comments

Oracle9i SQLJ Developer's Guide and Reference, Release 2 (9.2)

Part No. A96655-01

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this document. 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?

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

- Electronic mail: jpgcomment_us@oracle.com
- FAX: (650) 506-7225 Attn: Java Platform Group, Information Development Manager
- Postal service:

Oracle Corporation
Java Platform Group, Information Development Manager
500 Oracle Parkway, Mailstop 4op9
Redwood Shores, CA 94065
USA

If you would like a reply, please give your name, address, telephone number, and (optionally) electronic mail address.

If you have problems with the software, please contact your local Oracle Support Services.

Preface

This preface introduces you to the *Oracle9i SQLJ Developer's Guide and Reference*, discussing the intended audience, structure, and conventions of this document. A list of related Oracle documents is also provided.

This preface contains these topics:

- [Intended Audience](#)
- [Documentation Accessibility](#)
- [Organization](#)
- [Related Documentation](#)
- [Conventions](#)

Note: For space considerations, the Sample Applications chapter has been removed from this document. Please try the demo applications that are provided with Oracle SQLJ for examples of the many programming features described here. Also see the OTN link at the end of "[Related Documentation](#)".

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 Documentation](#)" below for the names of Oracle documents that discuss SQL and JDBC.

Documentation Accessibility

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

<http://www.oracle.com/accessibility/>

Accessibility of Code Examples in Documentation JAWS, a Windows screen reader, may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, JAWS may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation This documentation may contain links to Web sites of other companies or organizations that Oracle Corporation does not own or control. Oracle Corporation neither

evaluates nor makes any representations regarding the accessibility of these Web sites.

Organization

The two major aspects of using SQLJ are:

- creating your SQLJ source code
- running the SQLJ translator

Chapters 3 through 7 provide information about programming features, with chapters 3 and 4 covering the most important aspects.

Chapter 8 provides information about translator options and features.

In all, this document contains:

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 an 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, exception-handling, and Oracle-specific code generation.

Chapter 5, "Type Support"

Lists Java types that Oracle SQLJ supports, discusses the use of stream types, and discusses Oracle datatype extensions and the Java types that correspond to them.

Chapter 6, "Objects, Collections, and OPAQUE Types"

Discusses Oracle SQLJ support of user-defined object and collection types, including use of the Oracle JPublisher utility to generate corresponding Java types. There is also a brief discussion of support for Oracle OPAQUE 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 globalization support.

Chapter 10, "Performance and Debugging"

Discusses standard and Oracle-specific performance tuning features, and general SQLJ debugging considerations.

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.

Appendix A, "Customization and Specialized Customizers"

Describes SQLJ profiles, used in implementing SQL operations for ISO standard code generation; documents options you can specify during translation regarding the customization of your profiles for particular environments; discusses specialized customizers, including one for semantics-checking for profiles and one for installing "auditors" for debugging.

Appendix B, "SQLJ Error Messages"

Lists Oracle SQLJ translator and runtime error messages, their causes, and what actions you should take in response.

Related Documentation

Also available from the Oracle Java Platform group, for Oracle9i releases:

- *Oracle9i Java Developer's Guide*

This book introduces the basic concepts of Java in Oracle9i and provides general information about server-side configuration and functionality. Information that pertains to the Oracle database Java environment in general, rather than to a particular product such as JDBC or SQLJ, is in this book.

- *Oracle9i JDBC Developer's Guide and Reference*

This book covers programming syntax and features of the Oracle implementation of the JDBC standard (for Java Database Connectivity). This includes an overview of the Oracle JDBC drivers, details of the Oracle implementation of JDBC 1.22, 2.0, and 3.0 features, and discussion of Oracle JDBC type extensions and performance extensions.

- *Oracle9i JPublisher User's Guide*

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

- *Oracle9i Support for JavaServer Pages Reference*

This book covers the use of JavaServer Pages technology to embed Java code and JavaBean invocations inside HTML pages. Both standard JSP features and Oracle-specific features are described. Discussion covers considerations for the Oracle9i release 2 Apache JServ environment, but also covers features for servlet 2.2 environments and emulation of some of those features by the Oracle JSP container for JServ.

- *Oracle9i Java Stored Procedures Developer's Guide*

This book discusses Java stored procedures—programs that run directly in the Oracle9i database. With stored procedures (functions, procedures, triggers, and SQL methods), Java developers can implement business logic at the server level, thereby improving application performance, scalability, and security.

The following OC4J documents, for Oracle9i Application Server releases, are also available from the Oracle Java Platform group.

- *Oracle9iAS Containers for J2EE User's Guide*

This book provides some overview and general information for OC4J; primer chapters for servlets, JSP pages, and EJBs; and general configuration and deployment instructions.

- *Oracle9iAS Containers for J2EE Support for JavaServer Pages Reference*

This book provides information for JSP developers who want to run their pages in OC4J. It includes a general overview of JSP standards and programming considerations, as well as discussion of Oracle value-added features and steps for getting started in the OC4J environment.

- *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*

This book provides conceptual information and detailed syntax and usage information for tag libraries, JavaBeans, and other Java utilities provided with OC4J.

- *Oracle9iAS Containers for J2EE Servlet Developer's Guide*

This book provides information for servlet developers regarding use of servlets and the servlet container in OC4J. It also documents relevant OC4J configuration files.

- *Oracle9iAS Containers for J2EE Services Guide*

This book provides information about basic Java services supplied with OC4J, such as JTA, JNDI, and the Oracle9i Application Server Java Object Cache.

- *Oracle9iAS Containers for J2EE Enterprise JavaBeans Developer's Guide and Reference*

This book provides information about the EJB implementation and EJB container in OC4J.

The following documents are from the Oracle Server Technologies group:

- *Oracle9i XML Database Developer's Guide - Oracle XML DB*
- *Oracle9i XML Developer's Kits Guide - XDK*
- *Oracle9i Application Developer's Guide - Fundamentals*
- *Oracle9i Application Developer's Guide - Large Objects (LOBs)*
- *Oracle9i Application Developer's Guide - Object-Relational Features*
- *Oracle9i Supplied Java Packages Reference*
- *Oracle9i Supplied PL/SQL Packages and Types Reference*

- *PL/SQL User's Guide and Reference*
- *Oracle9i SQL Reference*
- *Oracle9i Net Services Administrator's Guide*
- *Oracle Advanced Security Administrator's Guide*
- *Oracle9i Database Globalization Support Guide*
- *Oracle9i Database Reference*
- *Oracle9i Database Error Messages*
- *Oracle9i Sample Schemas*

The following documents from the Oracle9i Application Server group may also be of interest:

- *Oracle9i Application Server Administrator's Guide*
- *Oracle Enterprise Manager Administrator's Guide*
- *Oracle HTTP Server Administration Guide*
- *Oracle9i Application Server Performance Guide*
- *Oracle9i Application Server Globalization Support Guide*
- *Oracle9iAS Web Cache Administration and Deployment Guide*
- *Oracle9i Application Server: Migrating from Oracle9i Application Server 1.x*

The following are available from the Oracle9i JDeveloper group:

- JDeveloper online help
- JDeveloper documentation on the Oracle Technology Network:
<http://otn.oracle.com/products/jdev/content.html>

In North America, printed documentation is available for sale in the Oracle Store at
<http://oraclestore.oracle.com/>

Customers in Europe, the Middle East, and Africa (EMEA) can purchase documentation from

<http://www.oraclebookshop.com/>

Other customers can contact their Oracle representative to purchase printed documentation.

To download free release notes, installation documentation, white papers, or other collateral, please visit the Oracle Technology Network (OTN). You must register online before using OTN; registration is free and can be done at

<http://otn.oracle.com/admin/account/membership.html>

If you already have a username and password for OTN, then you can go directly to the documentation section of the OTN Web site at

<http://otn.oracle.com/docs/index.htm>

To access the database documentation search engine directly, please visit

<http://tahiti.oracle.com>

For documentation of SQLJ standard features and syntax, refer to ANSI specification X3.135.10-1998:

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

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

<http://www.ansi.org/>

(Click "Electronic Standards Store" and search for the above specification number.)

The following location has SQLJ sample applications:

http://otn.oracle.com/sample_code/tech/java/sqlj_jdbc/content.html

Conventions

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

- [Conventions in Text](#)
- [Conventions in Code Examples](#)

Conventions in Text

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

Convention	Meaning	Example
<i>Italics</i>	Italic typeface indicates book titles or emphasis, or terms that are defined in the text.	<i>Oracle9i Database Concepts</i> Ensure that the recovery catalog and target database do <i>not</i> reside on the same disk.
UPPERCASE monospace (fixed-width) font	Uppercase monospace typeface indicates elements supplied by the system. Such elements include parameters, privileges, datatypes, RMAN keywords, SQL keywords, SQL*Plus or utility commands, packages and methods, as well as system-supplied column names, database objects and structures, usernames, and roles.	You can specify this clause only for a NUMBER column. You can back up the database by using the BACKUP command. Query the TABLE_NAME column in the USER_TABLES data dictionary view. Use the DEMS_STATS.GENERATE_STATS procedure.
lowercase monospace (fixed-width) font	Lowercase monospace typeface indicates executables, filenames, directory names, and sample user-supplied elements. Such elements include computer and database names, net service names, and connect identifiers, as well as user-supplied database objects and structures, column names, packages and classes, usernames and roles, program units, and parameter values. Note: Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.	Enter sqlplus to open SQL*Plus. The password is specified in the orapwd file. Back up the data files and control files in the /disk1/oracle/dbs directory. The department_id, department_name, and location_id columns are in the hr.departments table. Set the QUERY_REWRITE_ENABLED initialization parameter to true. Connect as oe user. The JRepUtil class implements these methods.
<i>lowercase italic monospace (fixed-width) font</i>	Lowercase italic monospace font represents place holders or variables.	You can specify the <i>parallel_clause</i> . Run <i>old_release.SQL</i> where <i>old_release</i> refers to the release you installed prior to upgrading.

Conventions in Code Examples

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

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

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

Convention	Meaning	Example
< >	In this document, angle brackets are used instead of regular brackets to enclose one or more optional items. Do not enter the angle brackets. (Regular brackets are not used due to SQLJ syntax considerations.)	DECIMAL (<i>digits</i> < , <i>precision</i> >)
	A vertical bar represents a choice of two or more options within brackets or braces. Enter one of the options. Do not enter the vertical bar.	{ENABLE DISABLE} [COMPRESS NOCOMPRESS]
...	Horizontal ellipsis points indicate either: <ul style="list-style-type: none"> That we have omitted parts of the code that are not directly related to the example That you can repeat a portion of the code 	CREATE TABLE ... AS <i>subquery</i> ; SELECT <i>col1</i> , <i>col2</i> , ... , <i>coln</i> FROM employees;
Other notation	You must enter symbols other than brackets, braces, vertical bars, and ellipsis points as shown.	acctbal NUMBER(11,2); acct CONSTANT NUMBER(4) := 3;
<i>Italics</i>	Italicized text indicates place holders or variables for which you must supply particular values.	CONNECT SYSTEM/ <i>system_password</i> DB_NAME = <i>database_name</i>
UPPERCASE	Uppercase typeface indicates elements supplied by the system. We show these terms in uppercase in order to distinguish them from terms you define. Unless terms appear in brackets, enter them in the order and with the spelling shown. However, because these terms are not case sensitive, you can enter them in lowercase.	SELECT last_name, employee_id FROM employees; SELECT * FROM USER_TABLES; DROP TABLE hr.employees;
lowercase	Lowercase typeface indicates programmatic elements that you supply. For example, lowercase indicates names of tables, columns, or files. Note: Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.	SELECT last_name, employee_id FROM employees; sqlplus hr/hr CREATE USER mjones IDENTIFIED BY ty3MU9;

Overview

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

- [Introduction to SQLJ](#)
- [Overview of SQLJ Components](#)
- [Overview of Oracle Extensions to the SQLJ Standard](#)
- [Basic Translation Steps and Runtime Processing](#)
- [JDBC Versus SQLJ Sample Code](#)
- [Alternative Deployment Scenarios](#)
- [Alternative Development Scenarios](#)

Introduction to SQLJ

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

Basic Concepts

SQLJ enables applications programmers to embed 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 SQL statements that comply with the ISO standard SQLJ Language Reference syntax. Oracle9i SQLJ supports the ISO SQLJ standard specification. The standard covers only *static SQL* operations—those that are predefined and do not change in real-time as a user runs the application (although the data values that are transmitted can change dynamically). Oracle SQLJ also offers extensions to support *dynamic SQL* operations—those that are *not* predefined, where the operations themselves can change in real-time. (It is also possible to use dynamic SQL operations through JDBC code or PL/SQL code within a SQLJ application.) Typical applications contain much more static SQL than dynamic SQL.

SQLJ consists of both a translator and a runtime component and is smoothly integrated into your development environment. The developer runs the translator, with translation, compilation, and customization (for ISO standard code) taking place in a single step when the `sqlj` front-end utility is run. The translation process replaces embedded SQL with calls to the SQLJ runtime, which implements the SQL operations. In ISO standard SQLJ this is typically, but not necessarily, performed through calls to a JDBC driver. To access 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.

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

- syntax of the embedded SQL
- SQL constructs, against a specified database schema to ensure consistency within a particular set of SQL entities (optional)
It verifies table names and column names, for example.
- datatypes, to ensure that the data exchanged between Java and SQL have compatible types and proper type conversions

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.

Oracle-Specific Code Generation Versus ISO Standard Code Generation

While the Oracle SQLJ implementation supports the ISO SQLJ standard, it also offers the option of Oracle-specific code generation, where Oracle JDBC calls are generated directly into the code. As of Oracle9i release 2, this is the default behavior. In the case of Oracle-specific code generation, be aware of the following:

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

Much of the SQLJ introductory discussion in this chapter mentions features of ISO standard code, so be aware of these key differences in Oracle-specific code.

For more information, see "[Oracle-Specific Code Generation \(No Profiles\)](#)" on page 4-39.

Java and SQLJ Versus PL/SQL

Java (including SQLJ) in Oracle applications does not replace PL/SQL. Java and PL/SQL are complementary to each other in the needs they serve.

While PL/SQL and Java can both be used to build database applications, the two languages were designed with different intents and, as a result, are suited for different kinds of applications:

- PL/SQL is a better solution for SQL-intensive applications. PL/SQL is optimized for SQL, and so SQL operations are faster in PL/SQL than in Java. Also, PL/SQL uses SQL datatypes directly, while Java applications must convert between SQL datatypes and Java types.
- Java, with its superior programming model, is a better solution for logic-intensive applications. Furthermore, the more general type system of Java is better suited than PL/SQL for component-oriented applications.

Oracle provides easy interoperability between PL/SQL and Java, ensuring that you can take advantage of the strengths of both languages. PL/SQL programs can

transparently call Java stored procedures, enabling you to build component-based Enterprise JavaBeans applications. PL/SQL programs can have transparent access to a wide variety of existing Java class libraries through PL/SQL call specifications.

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

Note: Using PL/SQL anonymous blocks within SQLJ statements is one way to support dynamic SQL in a SQLJ application. However, Oracle9i SQLJ includes extensions to support dynamic SQL directly. (See "[Support for Dynamic SQL](#)" on page 7-63.)

Overview of SQLJ Components

This section introduces the main SQLJ components and the concept of SQLJ profiles. (Profiles are for ISO code generation only.)

SQLJ Translator and SQLJ Runtime

Oracle SQLJ consists of two major components:

- Oracle SQLJ **translator**—This component is a 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. Here is a sample SQLJ statement:

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

The translator produces a `.java` file and, for ISO standard SQLJ code generation, 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.

Note: By default as of Oracle9i release 2, there is an Oracle-specific code generation setting that results in translation directly into Oracle JDBC code. In this case, no profiles are produced. See "[Oracle-Specific Code Generation \(No Profiles\)](#)" on page 4-39.

- Oracle SQLJ **runtime**—This component, also written in pure Java, is invoked automatically each time an end user runs a SQLJ application.

For ISO standard code generation, the SQLJ runtime implements the desired actions of your SQL operations, accessing the database using a JDBC driver. The generic ISO SQLJ standard does not require that a SQLJ runtime use a JDBC driver to access the database; however, Oracle SQLJ does require a JDBC driver, and, in fact, requires an Oracle JDBC driver if your application is customized with the default Oracle customizer (see below).

For Oracle-specific code generation (the default), Oracle JDBC calls are generated directly into the translated code and the SQLJ runtime plays a much smaller role.

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** that plays a role if you use ISO standard code generation. A customizer tailors SQLJ profiles for a particular database implementation and vendor-specific features and datatypes. By default, for ISO standard code, the Oracle SQLJ front end invokes an Oracle customizer to tailor your profiles for an Oracle database and Oracle-specific features and datatypes.

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

SQLJ Profiles (ISO Standard Code)

With ISO standard SQLJ code generation, SQLJ *profiles* are serialized Java resources (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).

Note: By default, as of Oracle9i release 2, Oracle-specific code generation is used. In this case, the translator generates Oracle JDBC calls directly, and details of your embedded SQL operations are embodied in the JDBC calls. There are no profiles. See ["Oracle-Specific Code Generation \(No Profiles\)"](#) on page 4-39.

Overview of Profiles

SQLJ profiles are used in ISO standard code 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.

For ISO code generation, SQLJ generates a profile for each connection context class in your application, where, typically, each connection context class corresponds to a particular set of SQL entities you use in your database operations. (There is one default connection context class, and you can declare additional classes.) The ISO

SQLJ standard requires that the profiles be of standard format and content. Therefore, for your application to use vendor-specific extended features, your profiles must be customized. By default, this occurs automatically, with your profiles being customized to use Oracle-specific extended features.

Profile customization allows 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 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. Other serialized objects can use 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

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

Overview of Oracle Extensions to the SQLJ Standard

Oracle9i SQLJ supports the ISO SQLJ specification. Because the ISO SQLJ standard is a superset of the ANSI SQLJ standard, it requires a JDK 1.2 or later environment that complies with J2EE. The ANSI SQLJ standard requires only JDK 1.1.x. The Oracle SQLJ translator accepts a broader range of SQL syntax than the ANSI SQLJ standard specifies.

The ANSI standard addresses only the SQL92 dialect of SQL, but allows extension beyond that. Oracle SQLJ supports the Oracle SQL dialect, which is a superset of SQL92. If you need to create SQLJ programs that work with other DBMS vendors, avoid using SQL syntax and SQL types that are not in the standard and, therefore, may not be supported in other environments. (On your product CD, the directory `[Oracle_Home]/sqlj/demo/components` includes a semantics-checker that you can use to verify that your SQLJ statements contain only standard SQL.)

For general information about Oracle SQLJ extensions, see [Chapter 5, "Type Support"](#), and [Chapter 6, "Objects, Collections, and OPAQUE Types"](#).

Oracle SQLJ Type Extensions

Oracle SQLJ supports the Java types listed below 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 8-45.

Using any of the following extensions requires Oracle-specific code generation or Oracle customization during translation, as well as the Oracle SQLJ runtime and an Oracle JDBC driver when your application runs.

- instances of `oracle.sql.*` classes as wrappers for SQL data
See ["Support for JDBC 2.0 LOB Types and Oracle Type Extensions"](#) on page 5-29.
- custom Java classes (classes that implement the `oracle.sql.ORAData` interface or the JDBC standard `java.sql.SQLData` interface), typically produced by the Oracle9i JPublisher utility to correspond to SQL objects, object references, and collections
See ["Custom Java Classes"](#) on page 6-6. Note, however, that the `SQLData` interface is standard. Classes that implement it are likely supported by other vendors' JDBC drivers and databases.

- stream instances—`BinaryStream` and `CharacterStream`, the latter of which replaces the deprecated `AsciiStream` and `UnicodeStream`, used as output parameters (see ["Support for Streams"](#) on page 5-14)
- iterator and result set instances as input or output parameters anywhere
The SQLJ standard specifies them only in result expressions or cast statements; see ["Using Iterators and Result Sets as Host Variables"](#) on page 3-52 and ["Using Iterators and Result Sets as Stored Function Returns"](#) on page 3-63.
- Unicode character types—`NString`, `NCHAR`, `NCLOB`, and `NcharCharacterStream`, the latter of which replaces the deprecated `NcharAsciiStream` and `NcharUnicodeStream` (see ["Oracle SQLJ Extended Globalization Support"](#) on page 9-25)

Oracle SQLJ Functionality Extensions

Oracle SQLJ also supports the following extended functionality:

- Oracle-specific code generation
This generates JDBC code directly. No profiles are produced and much of the SQLJ runtime functionality is bypassed during program execution. See ["Oracle-Specific Code Generation \(No Profiles\)"](#) on page 4-39.
- dynamic SQL in SQLJ statements
See ["Support for Dynamic SQL"](#) on page 7-63.
- scrollable result set iterators with additional navigation methods, and `FETCH` syntax from result set iterators and scrollable result set iterators
See ["Scrollable Iterators"](#) on page 7-42.
- optimization flags for column and parameter size definitions
See ["Column Definitions"](#) on page 10-23, ["Parameter Size Definitions"](#) on page 10-25, and ["Options for Code Generation, Optimizations, and CHAR Comparisons"](#) on page 8-51.
- flags for modified translator behavior—binding host expressions by identifier, accounting for blank padding in `CHAR` comparisons for `WHERE` clauses
See ["Binding Host Expressions by Identifier \(-bind-by-identifier\)"](#) on page 8-70 and ["CHAR Comparisons with Blank Padding \(-fixedchar\)"](#) on page 8-58.
- SQLJ statement caching on connection contexts
See ["Statement Caching"](#) on page 10-4.

Basic Translation Steps and Runtime Processing

This section introduces the following:

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

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. The file name must be a legal Java identifier. 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 should match the first defined class.

SQLJ Translation Steps

After you have written your `.sqlj` file, you must run SQLJ to process the files. (For coding the `.sqlj` file, basic SQLJ programming features and key considerations are discussed in [Chapter 3](#) and [Chapter 4](#).) The following 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 virtual machine (JVM), and passes arguments to it. The JVM 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 8-12.

From this point the following sequence of events occurs (presuming each step completes without fatal error). See "[Internal Translator Operations](#)" on page 9-2 for more detailed information.

1. The JVM invokes the SQLJ translator.
2. The translator parses the SQLJ and Java code in the `.sqlj` file, checking for proper SQLJ syntax and looking for type mismatches between your declared SQL datatypes and corresponding Java host variables. (Host variables are local Java variables used as input or output parameters in your SQL operations. "[Java Host Expressions, Context Expressions, and Result Expressions](#)" on page 3-16 describes them.)
3. Depending on SQLJ option settings, the translator invokes the online semantics-checker, the offline parser, neither, or both. This is to verify syntax of embedded SQL and PL/SQL statements and, for online checking, to check the use of database elements in your code against an appropriate database schema. Even when neither is specified, some basic level of checking is performed.

When online checking is specified, SQLJ will connect to a specified database schema to verify that the database supports all the database tables, stored procedures, and SQL syntax that the application uses, and that the host variable types in the SQLJ application are compatible with datatypes of corresponding database columns.

4. For Oracle-specific SQLJ code generation (the default `-codegen=oracle`), SQL operations are converted directly into Oracle JDBC calls, and no profiles are produced. See "[Oracle-Specific Code Generation \(No Profiles\)](#)" on page 4-39.

For ISO standard code generation (`-codegen=iso`), 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 class is typically used for each interrelated set of SQL entities that you use in your operations.

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

- any class definitions and Java code from your `.sqlj` source file
 - class definitions created as a result of your SQLJ iterator and connection context 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 (for ISO standard SQLJ code generation only)

- calls to Oracle JDBC (for Oracle-specific code generation) or to the SQLJ runtime (for ISO standard code generation) to implement the actions of your embedded SQL operations

Generated profiles (for ISO standard code generation only) contain information about all 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 passes them to the JDBC driver.

By default, profiles (if applicable) 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 JVM invokes the Java compiler, which is usually, but not necessarily, the standard `javac` provided with the Sun Microsystems JDK.
6. The compiler compiles the Java source file generated in step 4 and produces Java `.class` files as appropriate. This will include a `.class` file for each class you defined, a `.class` file for each of your SQLJ declarations, and a `.class` file for the profile-keys class (for ISO code generation).
7. For ISO standard SQLJ code generation, the JVM invokes the Oracle SQLJ customizer or other specified customizer to customize the profiles generated in step 4.

General SQLJ Notes Consider the following when translating and running SQLJ applications:

- The preceding 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.
- For Oracle-specific code generation, your application will require an Oracle JDBC driver when it runs, even if your code does not use Oracle-specific features.
- For ISO code generation, SQLJ generates profiles and the profile-keys class only if your source code includes SQLJ executable statements.
- Also for ISO code, if you use the Oracle customizer during translation, your application will require the Oracle SQLJ runtime and an Oracle JDBC driver when it runs, even if your code does not use Oracle-specific features. You can

avoid this by specifying `-profile=false` when you translate, to bypass Oracle-specific customization.

Summary of Translator Input and Output

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

Translator 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 any, or else on the first class it defines. 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 must 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.

Translator Output

The translation step produces a Java source file for each `.sqlj` file in your application, and, for ISO standard code generation, 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 output `.java` file also contains class definitions for any iterators or connection context classes you declare.

- The application profile files, if applicable, contain information about the SQL operations of your SQLJ application. There will be one profile for each connection class that you use 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 name extension, the naming is the same.

The compilation step compiles the Java source file into multiple class files. There is one `.class` file for each class you define in your `.sqlj` source file (minimum of one), and, for ISO code, one for a class known as the *profile-keys* 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.) Also, separate `.class` files will be produced for any inner classes or anonymous classes in your code.

For Oracle-specific code generation (the default), no profiles or profile-keys class are produced. For information about Oracle-specific code generation, see "[Oracle-Specific Code Generation \(No Profiles\)](#)" on page 4-39.

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, the translator output file `MyClass.java` is compiled into the `MyClass.class` class file.

- The profile-keys class (if applicable) 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`, there will be a `MyIter.class` class file.

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 (if any) 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 8-29.

SQLJ Runtime Processing

This section discusses runtime processing during program execution, considering both Oracle-specific code generation and ISO standard SQLJ code generation.

Processing for Oracle-Specific Generated Code

When you translate with the default setting `-codegen=oracle`, your program at runtime will execute the following:

- Oracle-specific APIs in the SQLJ runtime that ensure batching support and proper creation and closing of Oracle JDBC statements
- direct calls into the Oracle JDBC APIs for registering, passing, and retrieving parameters and result sets

For general information about Oracle-specific code generation, see "[Oracle-Specific Code Generation \(No Profiles\)](#)" on page 4-39.

Processing for ISO Standard Generated Code

For ISO standard SQLJ applications, 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 a 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 or output for SQL operations.

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.

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

In the sample, two methods are defined: `getEmployeeAddress()`, which selects from 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:

- A SQL script (not shown here) has been run to create the schema in the database and populate the tables. Both versions of the sample code refer to objects and tables created by this script.
- A PL/SQL stored function `UPDATE_ADDRESS()` exists, and updates a given address.
- The `Connection` object (for JDBC) and default connection context (for SQLJ) have been created previously by the caller.
- Exceptions are handled by the caller.
- The value of the address argument (`addr`) passed to the `updateAddress()` method can be null.

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

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.*;

/**
 * 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.getORADData(1, Address.getORADDataFactory());
    //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 {  
        pstmt.setORADData(2, addr);  
    }  
  
    pstmt.executeUpdate(); // line 43  
    addr = (Address)pstmt.getORADData(1, Address.getORADDataFactory());  
    pstmt.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, based on 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. Because 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 `getORADData()` method to retrieve it. The `Address` class can be created by `JPublisher`. The `getORADData()` method requires a "factory" object that it can use to create additional custom objects (additional `Address` objects in this case) as it retrieves the data to populate them. Use the static factory method `Address.getORADDataFactory()` to materialize an `Address` factory object for the `getORADData()` method to use.

Because `getORADData()` 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 provided 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 SQL type code and SQL type name of the object.

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 `setORAData()`.

Line 44: Fetch the return result `addr`. Because the Oracle callable statement returns a custom object of type `Address`, use the `getORAData()` method to retrieve it. The `Address` class can be created by `JPublisher`. The `getORAData()` method requires you to use the factory method `Address.getORADataFactory` to materialize an instance of an `Address` object. Because `getORAData()` returns a `Datum` object, 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` values of the factory object and any objects that you are registering as output parameters.
- Null data must be explicitly processed.
- Host variables must be represented by parameter markers in callable and prepared statements.

- If you want to reuse statement objects, for example if you want to repeatedly call `getEmployeeAddress()` and `updateAddress()`, then you must code this appropriately. Both Oracle SQLJ and Oracle JDBC support statement caching.

Maintaining JDBC Programs

JDBC programs are potentially expensive to maintain. 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 might 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 22
        return addr;
    }
}
```


Line 10: The `getEmployeeAddress()` method does not require an explicit connection object. SQLJ can use a default connection context instance, which would have been initialized previously somewhere in the 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 the employee number matches the one (`empno`) passed in to `getEmployeeAddress()`. This requires a declaration of the Address object (`addr`) that will receive the data. The `empno` and `addr` variables are used as input host variables.

Line 16: The `getEmployeeAddress()` method returns the `addr` object.

Line 19: The `updateAddress()` method also uses the default connection context instance.

Lines 19-22: The address is passed to the `updateAddress()` method, which passes it to the database. The database updates 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 23: The `updateAddress()` method returns the `addr` object.

Coding Requirements of the SQLJ Version

Note the following coding requirements (and lack of requirements) for the SQLJ version of the sample code:

- An explicit connection is not required—SQLJ can use a default connection context that has been initialized previously in the application.
- No datatype casting is required.
- SQLJ does not require knowledge of `_SQL_TYPECODE`, `_SQL_NAME`, or factories.
- Null data is processed 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 ODBC-style escapes are not used.
- You do not have to implement your own statement cache. By default, SQLJ will automatically cache `#sql` statements. This results in improved performance, for example, if you repeatedly call `getEmployeeAddress()` and `updateAddress()`.

Alternative Deployment Scenarios

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

- from an applet
- in the server (optionally running the SQLJ translator in the server as well)
- against Oracle9i Lite

Running SQLJ in Applets

Because the SQLJ runtime is pure Java, you can use SQLJ source code in applets as well as applications. There are, however, a few considerations, as discussed below.

For applet issues that apply more generally to the Oracle JDBC drivers, see the *Oracle9i JDBC Developer's Guide and Reference*, which includes discussion of firewalls and security issues as well.

General Development and Deployment Considerations

The following general considerations apply to the use of Oracle SQLJ applets.

- You must package all 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 (for ISO code generation):

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

These classes are included with your Oracle installation in one of several runtime libraries in the `[Oracle_Home]/lib` directory. (See "[Requirements for Using Oracle SQLJ](#)" on page 2-3.)

- You must specify a pure Java JDBC driver, such as the Oracle JDBC Thin driver, for your database connection.
- You must explicitly specify a connection context instance for each SQLJ executable statement in an applet. This is a requirement because you could

conceivably run two SQLJ applets in a single browser and, thus, in the same JVM. (For information about connections, see "[Connection Considerations](#)" on page 4-6.)

- The default translator setting `-codegen=oracle` generates Oracle-specific code. This will eliminate the use of Java reflection at runtime, thereby increasing portability across different browser environments. For information about the `-codegen` option, see "[Code Generation \(-codegen\)](#)" on page 8-52. For general information about Oracle-specific code generation, see "[Oracle-Specific Code Generation \(No Profiles\)](#)" on page 4-39.

General End User Considerations

When end users run your SQLJ applet, classes in their classpath may conflict with classes that are downloaded with the applet.

Oracle, therefore, recommends that end users clear their classpath before running the applet.

Java Environment and the Java Plug-in

Here are some additional considerations regarding the Java environment and use of Oracle-specific features.

- SQLJ requires the runtime environment of JDK 1.1.x or higher. Users cannot run SQLJ applets in browsers employing JDK 1.0.x, such as Netscape Navigator 3.x and Microsoft Internet Explorer 3.x, without a plug-in or some other means of using JRE 1.1.x instead of the default JRE of the browser.

One option is to use a Java plug-in offered by Sun Microsystems. For information, refer to the following Web site:

<http://www.javasoft.com/products/plugin>

- Some browsers, such as Netscape Navigator 4.x, do not support resource files with a `.ser` extension, which is the extension employed by the SQLJ serialized object files that are used for profiles (relevant for ISO standard code only). The Sun Microsystems Java plug-in, however, supports `.ser` files.

Alternatively, if you do not want to use the plug-in, Oracle SQLJ offers the `-ser2class` option to convert `.ser` files to `.class` files during translation. See "[Conversion of .ser File to .class File \(-ser2class\)](#)" on page 8-68 for more information.

Note: These considerations do not apply to the default Oracle-specific code generation, where no profiles are produced.

- Applets using Oracle-specific features require the Oracle SQLJ runtime to work. The Oracle runtime consists of the classes in the SQLJ runtime library file under `oracle.sqlj.*`. The Oracle SQLJ runtime library requires the Java Reflection API (`java.lang.reflect.*`); the `runtime11`, `runtime12`, and `runtime12ee` runtime libraries must use the Reflection API only in the circumstances outlined below. Most browsers do not support the Reflection API or impose security restrictions, but the Sun Microsystems Java plug-in provides support for the Reflection API.

Note: The term "Oracle-specific features" refers to the use of Oracle type extensions (discussed in [Chapter 5, "Type Support"](#)) and the use of SQLJ features that require Oracle-specific code generation or, for ISO code generation, require your application to be customized to work against an Oracle database. (For example, this is true of the `SET` statement, discussed in [Chapter 3, "Basic Language Features"](#).)

With ISO standard SQLJ code generation, the following SQLJ language features always require the Java Reflection API (`java.lang.reflect.*`), regardless of the version of the SQLJ runtime you are using:

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

Notes:

- An exception to the preceding is if you use SQLJ in a mode that is fully compatible with ISO. That is, if you use SQLJ in an environment that complies with J2EE and you translate and run your program with the SQLJ `runtime12ee` library, and you employ connection context type maps as specified by ISO. In this case, instances of `java.sql.Ref`, `Struct`, `Blob`, `Clob`, and `SQLData` are being retrieved without the use of reflection.
 - If you use Oracle-specific code generation (the default translator setting `-codegen=oracle`), you will eliminate the use of reflection in all of the instances listed above.
-
-

- Consider using the `runtime11` library for your applets, or `runtime12/runtime12ee` if your browser supports JDK 1.2. Doing so permits you to use Oracle-specific features and Oracle-specific customization.
- If your applet does not use any Oracle-specific features, you can distribute it with the generic SQLJ runtime library, `runtime-nonoracle`. To support this, do not use Oracle-specific code generation and do not customize the applet during translation. Set `-codegen=iso` and `-profile=false` when you translate the code. If you neglect to set `-profile=false`, then the default Oracle customizer will load Oracle-specific runtime classes. This will result in your applet requiring the Oracle runtime even though it does not use Oracle-specific features.

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

- Distribute your applet with the `runtime11` and `classes111` libraries. In this case, the SQLJ and JDBC versions must match. For example, to use the SQLJ 9.0.0 runtime, you must have the Oracle 9.0.0 JDBC driver.
- If you use object types, JDBC 2.0 types, REF CURSORS, or the CAST statement in your SQLJ statements, then you must adhere to your choice of the following:
 - Use the default `-codegen=oracle` setting when you translate your applet.or:
 - Ensure that the browser in which you run supports JDK 1.1 or higher and permits reflection.

or:

- Run your applet through a browser Java plug-in.
- If your applet does not use Oracle-specific features, then you can compile it using ISO standard code generation (`-codegen=iso`) without customization (`-profile=false`) and distribute it with the generic SQLJ runtime, `runtime-nonoracle`.

Introduction to SQLJ in the Server

In addition to its use in client applications, SQLJ code can run within a target Oracle9i database in stored procedures, stored functions, or triggers. Server-side access occurs through an Oracle JDBC driver that runs inside the server itself. Additionally, the Oracle9i database has an embedded SQLJ translator so that SQLJ source files for server-side use can optionally be translated directly in the server.

The two main areas to consider, which [Chapter 11, "SQLJ in the Server"](#), discusses in detail, are the following:

- creating SQLJ code for use within the server

Coding a SQLJ application for use within the target Oracle9i database is similar to coding for 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 have only one connection.
- The connection is to the database in which the code is running.
- The connection is implicit (does not have to be explicitly initialized, unlike on a client).
- The connection cannot be closed—any attempt to close it will be ignored.

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

Note: There is also a server-side Thin driver for connecting to one server from code that runs in another. This case is effectively the same as using a Thin driver from a client and is coded in the same way. See "[Overview of the Oracle JDBC Drivers](#)" on page 4-2.

- 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 translation and compilation are done automatically. In general, `loadjava` or SQL commands can be used for class and resource files or for source files. From a user perspective `.sqlj` files are treated the same as `.java` files, with translation taking place implicitly.

See "[Loading SQLJ Source and Translating in the Server](#)" on page 11-16 for information about using the embedded server-side translator.

Note: The server-side translator does not support the Oracle SQLJ `-codegen` option and generates Oracle-specific code. To use ISO standard code in the server, you must translate on a client and load the individual components into the server. Also note restrictions on interoperability when running code generated with different settings. For more information, see "[Translating SQLJ Source on a Client and Loading Components](#)" on page 11-9 and "[Oracle-Specific Code Generation \(No Profiles\)](#)" on page 4-39.

Using SQLJ with Oracle9i Lite

You can use SQLJ on top of Oracle9i Lite. This section provides an overview of this functionality. For more information, refer to the *Oracle9i Lite Java Developer's Guide*.

Overview of Oracle9i Lite and Java Support

Oracle9i 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.

Oracle9i Lite is compatible with Oracle9i, Oracle8i, Oracle8, and Oracle7. It provides comprehensive support for Java, including JDBC, SQLJ, and Java stored procedures.

There are two alternatives for access to Oracle9i Lite from Java programs, as follows:

- 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 Oracle9i Lite object model, allowing them to access persistent information stored in Oracle9i Lite, 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 Oracle9i Lite schema. This can be generated by Oracle9i 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 Oracle9i 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 Oracle9i Lite

Note the following requirements if you intend to run a Java program on top of Oracle9i Lite:

- Windows NT 3.51 or higher, Windows 95, or Windows 98
- Oracle9i Lite 3.0 or higher
- JDK 1.1.x or higher
- Java Runtime Environment (JRE) that supports Java Native Interface (JNI)

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

Support for Oracle Extensions

Oracle9i Lite 4.0.x and higher includes 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 globalization support for deployment to other languages. This section introduces these topics:

- globalization support
- SQLJ in IDEs
- Windows considerations

SQLJ Globalization Support

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

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

For information, see ["Globalization Support in the Translator and Runtime"](#) on page 9-19.

SQLJ in Oracle9i JDeveloper and Other IDEs

Oracle SQLJ includes a programmatic API so that it can be embedded in integrated development environments (IDEs) such as Oracle9i JDeveloper. The IDE takes on a role similar to that of the front-end `sqlj` script, invoking the translator, semantics-checker, compiler, and customizer (as applicable).

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

The Oracle JDBC OCI and Thin drivers are included with JDeveloper, as well as drivers to access Oracle9i Lite.

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://otn.oracle.com/products/jdev/content.html>

Windows Considerations

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

- This manual uses UNIX syntax. Use platform-specific file names and directory separators (such as "\" on Windows) that are appropriate for your platform, because your JVM expects file names and paths in the platform-specific format. This is true even if you are using a shell (such as `ksh` on NT) that permits a different file name syntax.
- For UNIX, 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.
- How to set environment variables is specific to the operating system. There may also be OS-specific restrictions. In Windows 95, use the `Environment` tab in the `System` control panel. Additionally, since Windows 95 does not support the "=" character in variable settings, SQLJ supports the use of "#" instead of "=" in setting `SQLJ_OPTIONS`, an environment variable that SQLJ can use for option settings. Consult your operating system documentation regarding settings and syntax for environment variables, and be aware of any size limitations.
- As with any operating system and environment you use, be aware of specific limitations. In particular, the complete, expanded SQLJ command line must not exceed the maximum command-line size, which is 250 characters for Windows 95 and 4000 characters for Windows NT. Consult your operating system documentation.

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

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, you should also verify your JDBC installation according to the *Oracle9i JDBC Developer's Guide and Reference*.

This chapter discusses the following topics:

- [Assumptions and Requirements](#)
- [Checking the Installation and Configuration](#)
- [Testing the Setup](#)

Assumptions and Requirements

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

- [Assumptions About Your Environment](#)
- [Requirements for Using Oracle SQLJ](#)
- [Oracle SQLJ Environment: Key Scenarios and Guidelines](#)
- [Environment Issues and Limitations](#)
- [Oracle SQLJ Backward Compatibility](#)

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 a Sun Microsystems JDK, but other implementations of Java will work. Make sure you can run Java (typically `java`) and your Java compiler (typically `javac`).

To translate and run Oracle SQLJ applications on a Sun JDK, you must use a JDK 1.2.x (or higher) or JDK 1.1.x version, with an appropriate JDBC driver. There are Oracle JDBC Thin and OCI driver versions that work with any of these JDK scenarios.

For more information, see "[Oracle SQLJ Environment: Key Scenarios and Guidelines](#)" on page 2-4.

Note: A Java runtime environment (JRE), such as the one installed with Oracle9i, is not by itself sufficient for translating SQLJ programs. A JRE *is* sufficient, however, for running SQLJ programs that have already been translated and compiled.

- 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 *Oracle9i JDBC Developer's Guide and Reference*. You can also refer to Chapter 1, "Overview", of

that document for information about the Oracle JDBC drivers and how to decide which is appropriate for your situation.

Notes: If you are using a non-Oracle JDBC driver, you must do the following:

- Modify `connect.properties`, as discussed in "[Set Up the Runtime Connection](#)" on page 2-14.
 - Modify the demo applications, as discussed in "[Driver Selection and Registration for Runtime](#)" on page 4-5, so that your driver is registered before the call to the `Oracle.connect()` method.
-

Requirements for Using Oracle SQLJ

The following are required to use Oracle SQLJ:

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

Oracle SQLJ works with any standard JDBC driver.

- a database system that is accessible using your JDBC driver
- class files for the SQLJ translator

Translator-related classes are available in the file:

`[Oracle_Home]/sqlj/lib/translator.jar` (or `.zip`)

- class files for the SQLJ runtime

Several SQLJ runtime versions are available. You must select a runtime version that is compatible with your Java environment and JDBC driver (these are all in `[Oracle_Home]/sqlj/lib`).

- `runtime12.jar` (or `.zip`)—for use with Oracle9i JDBC drivers under JDK 1.2.x or higher, providing full ISO SQLJ functionality
- `runtime12ee.jar` (or `.zip`)—for use with Oracle9i JDBC drivers in a J2EE environment (using JDK 1.2.x or higher), providing full ISO SQLJ functionality
- `runtime11.jar` (or `.zip`)—for use with Oracle9i JDBC drivers under JDK 1.1.x

- `runtime.jar` (or `.zip`)—for use with older Oracle JDBC drivers and any JDK environment (intended for Oracle JDBC release 8.1.7 and prior)
- `runtime-nonoracle.jar` (or `.zip`)—for use with non-Oracle JDBC drivers and any JDK environment

Notes: Also be aware of the following:

- In SQLJ release 8.1.6 and earlier, there was only one runtime library, and the `runtime` JAR/ZIP file was a subset of the `translator` JAR/ZIP file. In Oracle9i this is no longer the case. You must now specify both a runtime file and the translator file in your classpath.
 - The `runtime-nonoracle` library provides portability across different Java and JDBC environments, but does not support Oracle-specific functionality.
 - The `runtime` library provides flexibility across different Java and Oracle JDBC environments, but does not support all ISO SQLJ functionality.
 - For ISO SQLJ-compliant support for JDBC 2.0 types such as `java.sql.Ref`, `Clob`, `Blob`, `Struct`, and `SQLData`, use the `runtime12` or `runtime12ee` library with JDK 1.2 or J2EE and an Oracle9i JDBC driver.
 - If you will be running only SQLJ applications that have already been translated, compiled, and customized, you will not need the `translator` JAR/ZIP file.
 - The `translator` and `runtime` JAR files are compressed. The ZIP files are *uncompressed*, however, for maximum portability.
-

Oracle SQLJ Environment: Key Scenarios and Guidelines

To ensure you have a fully working environment, you must consider several aspects of your environment—SQLJ and its code generation mode, JDBC, and the JDK. This section first discusses the two main Oracle scenarios of supported combinations, and then discusses some important general guidelines.

Also see "[Environment Issues and Limitations](#)" on page 2-6 for related information.

Note: Code generation is determined by the Oracle SQLJ `-codegen` option. See "[Code Generation \(-codegen\)](#)" on page 8-52 for more information.

Scenario 1: Oracle-Specific Code

This section documents a typical environment setup for Oracle-specific code generation. Note that in this case, the SQLJ generic runtime library is not an option.

- SQLJ code generation: `-codegen=oracle` (default)
- SQLJ library during translation:
 - Oracle SQLJ `runtime11` library with JDK 1.1
 - or:
 - Oracle SQLJ `runtime12` or `runtime12ee` library with JDK 1.2 or higher
- JDBC and JDK: Oracle JDBC version 9.0.1 or higher with JDK 1.1 or higher
If you might be running against either the 9.0.1 or 9.2 JDBC driver, translate against 9.0.1. In general, compile against the oldest driver you might use.
- SQLJ library during runtime: same library as for translation

Scenario 2: ISO Standard Code

This section documents a typical environment setup for ISO standard code generation.

1. SQLJ code generation: `-codegen=iso`
2. SQLJ library during translation:
 - SQLJ `runtime11` library with JDK 1.1
 - or:
 - SQLJ `runtime12` or `runtime12ee` library with JDK 1.2 or higher
 - or, for Oracle JDBC versions 8.1.7 and prior:
 - SQLJ generic `runtime` library with any JDK 1.1 or higher
3. JDBC and JDK: any Oracle JDBC version with any JDK 1.1 or higher
4. SQLJ library during runtime: same library as for translation

Environment Scenarios: Key Guidelines

Regarding your environment for running Oracle SQLJ, be aware of the following important guidelines and considerations:

- In general, use the same versions of the SQLJ library, JDBC library, and JDK in translating and compiling all components of your application.
- Always be aware of the following cross-compatibility considerations:
 - If you want to be able to run the same compiled code in either a JDK 1.1 environment or a JDK 1.2 or higher environment, then translate against the `runtime11` library under JDK 1.1. This allows the option of using the `runtime12` or `runtime12ee` library under JDK 1.2 at runtime. If you translate against JDK 1.2, several JDBC 2.0 APIs that are not supported under JDK 1.1 are compiled into your class files.
 - If you want to be able to run against either a version 9.0.1 or a version 9.2 Oracle JDBC driver, then translate against a 9.0.1 driver. This allows you to use either driver version at runtime.

Generated code is optimized toward the JDBC driver in the classpath during translation.

- For maximal cross-compatibility, avoid using declared connection context classes. If you use `JPublisher`, use the default settings for the `-compatible` and `-context` options. See the *Oracle9i JPublisher User's Guide* for information about these options.

You can also consider using the `runtime` library for cross-compatibility, but this library has disadvantages (such as not supporting the Oracle9i `oracle.jdbc.OracleXXX` interfaces, which causes problems in the middle tier).

Environment Issues and Limitations

This section discusses a key environmental issue—JDK migration regarding type maps—and lists resulting limitations and some additional environment issues and limitations. These are among the considerations in planning the typical scenarios outlined in "[Oracle SQLJ Environment: Key Scenarios and Guidelines](#)" on page 2-4.

Notes:

- Where the generic `runtime` library is discussed, an Oracle JDBC environment is still assumed (version 8.1.7 or prior). For non-Oracle JDBC environments, substitute the `runtime-nonoracle` library.
 - The `runtime` and `runtime-nonoracle` libraries are intended mainly for backward compatibility. They do not support Oracle-specific features.
-

JDK Migration Issues Regarding Type Maps

The type for JDBC type maps changed between JDK 1.1.x and JDK 1.2.x, from `java.util.Dictionary` to `java.util.Map`.

The `getTypeMap()` method of all SQLJ connection context classes returns a type map instance. The Java type system requires that an implemented method must return exactly the type specified in an underlying interface (in this case, the JDBC `Connection` interface). Consider the following method signature:

```
java.util.Hashtable getTypeMap() { ... }
```

This would seem to implement both the JDK 1.1 and JDK 1.2 specifications, because `Hashtable` extends `Dictionary` and implements `Map`; however, it is not acceptable to the Java type system.

This incompatibility between the JDK 1.1 and JDK 1.2 interfaces has the following consequences if your SQLJ code declares connection context types:

- If you compile under JDK 1.1.x, including when you translate against the SQLJ `runtime11` library, or when you compile under any JDK when translating against the SQLJ generic `runtime` library, the generated code will return `Dictionary` instances. The code should run under both JDK 1.1.x and JDK 1.2.x or higher. (Under JDK 1.2, the SQLJ runtime uses Java Reflection as necessary.)
- If you compile under JDK 1.2.x or higher, including when you translate against the SQLJ `runtime12` or `runtime12ee` library, you will not be able to run under JDK 1.1.x (because `java.util.Map`, among other things, was not defined in JDK 1.1.x).
- If you translate `.sqlj` files under JDK 1.1.x, the generated `.java` files will not compile under JDK 1.2.x or higher. (This is relevant if you run SQLJ with the

-compile=false setting, in order to translate and compile separately.) Likewise, if you translate under JDK 1.2.x or higher, the generated .java files will not compile under JDK 1.1.x.

Note: If you use JPublisher, the default setting is to use DefaultContext instances for connections, in which case the preceding issues do not arise. But with JPublisher nondefault -context option settings, or with a -compatible option setting of both8i or 8i, JPublisher-generated .sqlj source code declares connection context types. See the *Oracle9i JPublisher User's Guide* for more information.

Other Environment Issues and Limitations

The following list notes additional environmental issues and limitations, mostly related to the type map issues discussed above.

- With the default -codegen=oracle setting, you cannot use the generic runtime library. Use the runtime11 library (for JDK 1.1) or the runtime12 or runtime12ee library (for JDK 1.2 or higher).
- For Oracle JDBC version 8.1.7 or prior, you must use -codegen=iso and the generic runtime library (not runtime11, runtime12, or runtime12ee).
- For any SQLJ application using declared connection context classes, all modules must be translated against JDK 1.1 (using the runtime or runtime11 library) or all modules must be translated against JDK 1.2 or higher (using the runtime12 or runtime12ee library). There cannot be a mixture. This includes situations where you use JPublisher with a nondefault -context option setting.
- You can run against a JDK version that is at least as high as the version you translated against. If you translate under JDK 1.1.x, you can run the application under either JDK 1.1.x or JDK 1.2 or later. This assumes that you do not have any JDBC code that uses the oracle.jdbc2 package, which Oracle SQLJ does not support. Oracle JDBC used this package to support JDBC 2.0 types under JDK 1.1.x. If you translate under JDK 1.2, you can run the application under JDK 1.2 or higher.

Notes:

- Oracle JDBC releases 8.1.5 and prior do not support JDK 1.2.x.
 - In Oracle9i, Oracle SQLJ and Oracle JDBC do not support JDK 1.0.2. (Release 8.1.6 was the last Oracle JDBC release to support JDK 1.0.2, while Oracle SQLJ has never supported JDK 1.0.2.) This includes applets running in browsers that use JDK 1.0.2 except where special preparations have been made. (This chapter does not discuss applets. Refer to "[Running SQLJ in Applets](#)" on page 1-25.)
-

Oracle SQLJ Backward Compatibility

Be aware of the following regarding Oracle SQLJ backward compatibility:

- Code generated with an earlier release of the SQLJ translator will continue to do the following (subject to cross-compatibility limitations discussed in "[Environment Issues and Limitations](#)" on page 2-6):
 - run against current runtime (.jar or .zip) libraries
 - be compilable against current runtime (.jar or .zip) libraries
- Oracle-specific translator output (code generated with the default `-codegen=oracle` setting) must be created and executed using the `runtime11`, `runtime12`, or `runtime12ee` library. Furthermore:
 - Such code will be executable under future releases of Oracle JDBC and SQLJ.
 - Such code, however, will *not* be executable under previous releases of Oracle JDBC and the Oracle SQLJ runtime. In these circumstances, you will have to retranslate the code.

Also remember that Oracle-specific code is not portable.

- ISO standard generated code (`-codegen=iso`) can be created and executed against an earlier Oracle JDBC release using the current `runtime (.jar or .zip)` library.

Note: Regarding Oracle JDBC backward compatibility to prior database releases, any given Oracle JDBC driver release is compatible with any Oracle database release from 7.3.4 up to the release number of the JDBC driver. For more information see the *Oracle9i JDBC Developer's Guide and Reference*.

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 *Oracle9i JDBC Developer's Guide and Reference* for information about JDBC files that should be installed on your system.

Directories for Oracle SQLJ

Installing the Oracle9i Java environment will include, among other things, installing a `sqlj` directory under your `[Oracle_Home]` directory. The `sqlj` directory contains the following subdirectories:

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

In addition, 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`.

The structure is similar if you download SQLJ from a Web site, such as the Oracle Technology Network <http://otn.oracle.com> address. The `bin` directory, with both SQLJ and JPublisher executable files, is directly under the `sqlj` directory.

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, you will need the Oracle JDBC classes JAR/ZIP file that is appropriate for your environment.

JDK 1.1-compatible classes are in `classes111.jar` or `.zip`; JDK 1.2 (or higher) compatible classes are in `classes12.jar` or `.zip`; JDK 1.4-compatible classes are in `ojdbc14.jar` or `.zip`. Presuming you use a Sun Microsystems JDK, make sure the appropriate JAR/ZIP file name is in your classpath setting. There may also be alternative JDBC driver libraries available, such as `classes12_g.jar`, which permits driver debugging information to be printed.

For more information about libraries and required path and classpath settings for Oracle JDBC, refer to the *Oracle9i JDBC Developer's Guide and Reference*.

Path and Classpath for Oracle SQLJ

Set your `PATH` and `CLASSPATH` variables as follows for Oracle SQLJ.

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 (either `.jar` or `.zip`):

```
[Oracle_Home]/sqlj/lib/translator.jar
```

Use backward slashes for Windows. Replace `[Oracle_Home]` with your actual Oracle home directory.

In addition, you must include *one* of the following runtime libraries in your classpath (either `.jar` or `.zip`):

```
[Oracle_Home]/sqlj/lib/runtime12.jar  
[Oracle_Home]/sqlj/lib/runtime12ee.jar
```



```
[Oracle_Home]/sqlj/lib/runtime11.jar  
[Oracle_Home]/sqlj/lib/runtime.jar  
[Oracle_Home]/sqlj/lib/runtime-nonoracle.jar
```

See "[Requirements for Using Oracle SQLJ](#)" on page 2-3 regarding which runtime library to use for your JDBC driver and Java environment.

Important: You will not be able to run the SQLJ translator if you do not add a runtime library. You must specify a runtime library as well as the translator library in your classpath.

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

```
sqlj -version-long
```

Verify Installation of sqljutil Package

Note: This step is relevant only for online checking during translation, and is applicable only if you are using SQLJ stored procedures or functions with a pre-8.1.5 Oracle database (or an 8.1.5 or later database that was installed without a server-side JVM).

The package `sqljutil` is required for online checking of stored procedures and functions in an Oracle database. For Oracle release 8.1.5 and later, it should have been installed automatically under the `SYS` schema during installation of your database server-side JVM. To verify the installation of `sqljutil`, issue the following SQL command (from `SQL*Plus`, for example):

```
describe sys.sqljutil
```

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

```
[Oracle_Home]/sqlj/lib/sqljutil.sql
```

(The `sqljutil` package is installed in the `SYS` schema.)

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/compile them as appropriate (as explained in the following subsections).

The demo applications provided with the Oracle SQLJ installation refer to tables on an Oracle 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 Oracle 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:oci:@
#
# User name and password here
```

```
sqlj.user=scott  
sqlj.password=tiger
```

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

Connecting with an Oracle JDBC Driver

With Oracle9i, use "oci" in the connect string for the Oracle JDBC OCI driver in any new code. For backward compatibility, however, "oci8" is still accepted, so you do not have to change existing code. (Also, "oci7" is accepted for Oracle JDBC release 7.3.4.)

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

Connecting with 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 performed automatically in the demo and test programs if you use an Oracle JDBC driver. See "[Driver Selection and Registration for Runtime](#)" on page 4-5.

In addition, in the SQLJ demo programs, you must replace the following code:

```
Oracle.connect(url, user, password);
```

with the following:

```
DriverManager.registerDriver(new yourdriver());  
Connection conn = DriverManager.getConnection(url, user, password);  
conn.setAutoCommit(false);  
DefaultContext.setDefaultContext(new DefaultContext(conn));
```

Create a Table to Verify the Database

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

```
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 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.

In a UNIX environment, 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 JAR/ZIP file has the class files for the SQLJ translator and runtime. It 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 the `connect.properties` file. 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 OCI driver. The user name will be discussed next.

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

Online semantics-checking is enabled as soon as you specify a user name for the translation-time connection. You can specify the user name either by uncommenting the `sqlj.user` line in the `sqlj.properties` file or by using the `-user` command-line option. The user, password, URL, and driver options all can be set either on the command line or in the properties file. This is explained in "[Connection Options](#)" on page 8-34.

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

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

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

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

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

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

If everything works, this prints:

```
Hello, SQLJ Checker!
```

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 (designed to help you strongly type your connections according to the sets of SQL entities being used); 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](#)
- [Overview of SQLJ Executable Statements](#)
- [Java Host Expressions, Context Expressions, and Result Expressions](#)
- [Single-Row Query Results: SELECT INTO Statements](#)
- [Multi-Row Query Results: SQLJ Iterators](#)
- [Assignment Statements \(SET\)](#)
- [Stored Procedure and Function Calls](#)

Overview of SQLJ Declarations

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

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

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

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

- `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 anywhere that a class definition would be allowed in standard Java. The only limitation is that you cannot have a declaration inside a method block under JDK 1.1.x. For example:

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

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



```

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

void func()
{
    SQLJ declaration; // OK in JDK 1.2.x; ILLEGAL in JDK 1.1.x (method block)
}
}

```

Note: As with standard Java, any public class should be declared in one of the following ways (this is a requirement if you are using the standard `javac` compiler provided with the Sun Microsystems JDK):

- Declare it in a separate source file. The base name of the file should be the same as the class name.

or:

- Declare it at class-level scope or nested-class-level scope. In this case, it may be advisable to use `public static` modifiers.
-

Iterator Declarations

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

Basic iterator declarations use the following syntax:

```
#sql <modifiers> iterator iterator_classname (type declarations);
```

Modifiers are optional and can be any standard Java class modifiers such as `public`, `static`, and so on. Type declarations are separated by commas.

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

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 3-37.

Connection Context Declarations

A connection context declaration creates a connection context class, whose instances are typically used for database connections that use a particular set of SQL entities.

Basic connection context declarations use the following syntax:

```
#sql <modifiers> context context_classname;
```

As for iterator declarations, modifiers are optional and can be any standard Java class modifiers. 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 schemas that include a desired set of entities, such as tables, views, and stored procedures. Different instances of `MyContext` might be used to connect to different schemas, but each schema might be expected, for example, to include an `EMP` table, a `DEPT` table, and a `TRANSFER_EMPLOYEE` stored procedure.

Declared connection context classes are an advanced topic and are not necessary for basic SQLJ applications that use only one interrelated set of SQL entities. In basic scenarios, you can use multiple connections by creating multiple instances of the `sqlj.runtime.ref.DefaultContext` class, which does not require any connection context declarations.

See "[Connection Considerations](#)" on page 4-6 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

When you declare any iterator class or connection context class, you can specify one or more interfaces to be implemented by the generated class.

Use the following syntax for an iterator class:

```
#sql <modifiers> iterator iterator_classname implements intfc1,..., intfcN
    (type declarations);
```

The portion `implements intfc1,..., intfcN` is known as the `implements` clause. Note that in an iterator declaration, the `implements` clause precedes the iterator type declarations.

Here is the syntax for a connection context declaration:

```
#sql <modifiers> context context_classname implements intfc1,..., intfcN;
```

The `implements` clause is potentially useful in either an iterator declaration or a connection context declaration, but is more likely to be useful in iterator declarations—particularly in implementing the `sqlj.runtime.Scrollable` or `sqlj.runtime.ForUpdate` interface. Scrollable iterators are supported in Oracle SQLJ (see ["Scrollable Iterators"](#) on page 7-42); positioned updates or deletes are not currently supported.

For more information about the `implements` clause, see ["Using the IMPLEMENTS Clause in Iterator Declarations"](#) on page 7-40 and ["Using the IMPLEMENTS Clause in Connection Context Declarations"](#) on page 7-11.

Note: The SQLJ `implements` clause corresponds to the Java `implements` clause.

The following example uses an `implements` clause in declaring a named iterator class. Presume you have created a package, `mypackage`, that includes an iterator interface, `MyIterIntfc`.

```
#sql public iterator MyIter implements mypackage.MyIterIntfc
    (String ename, int empno);
```

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

This next example declares a connection context class that implements an interface named `MyConnCtxtIntfc`. Presume that it, too, is in the package `mypackage`.

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

Declaration WITH Clause

In declaring a connection context class or iterator class, you can use a `with` clause to specify and initialize one or more constants to be included in the definition of the generated class. Most of this usage is standard, although Oracle adds one kind of extended functionality for iterator declarations.

Standard WITH Clause Usage

In using a `with` clause, the constants that are produced are always `public static final`. Use the following syntax for an iterator class:

```
#sql <modifiers> iterator iterator_classname with (var1=value1,..., varN=valueN)
      (type declarations);
```

The portion `with (var1=value1,..., varN=valueN)` is the `with` clause. Note that in an iterator declaration, the `with` clause precedes the iterator type declarations.

Where there is both a `with` clause and an `implements` clause, the `implements` clause must come first. Note that parentheses are used to enclose `with` lists, but not `implements` lists.

Here is the syntax for a connection context declaration that uses a `with` clause:

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

And here is an example:

```
#sql public context MyContext with (typeMap="MyPack.MyClass");
```

The declared class `MyContext` will define the attribute `typeMap` that will be `public static final` of the type `String` and initialized to the value `"MyPack.MyClass"`. This value is the fully qualified class name of a `ListResourceBundle` implementation that provides the mapping between SQL and Java types for statements executed on instances of the `MyContext` class.

Here is another example (see the note about sensitivity below):

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

This declaration sets the cursor `sensitivity` to `ASENSITIVE` for a named iterator class.

The following example uses both an `implements` clause and a `with` clause. (See the note about `holdability` immediately below.)

```
#sql public iterator MyScrollableIterator implements sqlj.runtime.Scrollable
    with (holdability=true) (String ename, int empno);
```

The `implements` clause must precede the `with` clause.

This declaration implements the interface `sqlj.runtime.Scrollable` and enables the cursor `holdability` for a named iterator class. (But `holdability` is not currently meaningful to Oracle9i.)

The following standard constants on iterator declarations are not supported in Oracle SQLJ. They mostly involve cursor states and can take only particular values, as follows:

- `holdability` (`true/false`)
- `updateColumns` (a `String` literal containing a comma-delimited list of column names)

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

Oracle SQLJ supports the following standard constants on connection context declarations.

- `sensitivity` (`SENSITIVE/ASENSITIVE/INSENSITIVE`, to define the sensitivity of a scrollable iterator)
- `returnability` (`true/false`, to define whether an iterator can be returned from a Java stored procedure or function)
- `typeMap` (a `String` literal defining the name of a type map properties resource)
- `dataSource` (a `String` literal defining the name under which a data source is looked up in the `InitialContext`)

See "[Standard Data Source Support](#)" on page 7-13 for information about SQLJ support for data sources.

The following standard constants on connection context declarations are not currently supported in Oracle SQLJ:

- `path` (a `String` literal defining the name of a path to be prepended for resolution of Java stored procedures and functions)
- `transformGroup` (a `String` literal defining the name of a SQL transformation group that may be applied to SQL types)

Note: A predefined set of standard SQLJ constants can be defined in a `with` clause; however, not all of these constants are meaningful to Oracle9i or to the Oracle SQLJ runtime. Attempts to define constants other than the standard constants (as in the example above) is legal with Oracle9i, 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 8-45.

Oracle-Specific WITH Clause Usage

In addition to standard `with` clause usage in a connection context declaration to associate a type map with the connection context class, Oracle allows you to use a `with` clause to associate a type map with the iterator class in an iterator declaration. Here is an example:

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

If you use Oracle-specific code generation (through the default translator setting `-codegen=oracle`) and you use type maps in your application, then your iterator and connection context declarations must use the same type map(s). See ["Code Considerations and Limitations with Oracle-Specific Code Generation"](#) on page 4-41 for more information.

Overview of SQLJ Executable Statements

A SQLJ executable statement consists of the `#sql` token followed by a *SQLJ clause*, which uses syntax that follows a specified standard for embedding executable SQL statements in Java code. The embedded SQL operation of a SQLJ executable statement can be any SQL operation supported by 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 operation must be enclosed in curly braces: `{ . . . }`.
- It must be terminated with a semi-colon (`;`).

Notes:

- It is recommended that you *not* close the SQL operation (inside the braces) with a semi-colon. The parser will detect the end of the operation when it encounters the closing curly brace of the SQLJ clause.
 - Everything inside the curly braces of a SQLJ executable statement is treated as SQL syntax and must follow SQL rules, with the exception of Java host expressions (which are described in "[Java Host Expressions, Context Expressions, and Result Expressions](#)" on page 3-16).
 - During offline parsing of SQL operations, all SQL syntax is checked. During online semantics-checking (done through a database connection), however, only DML operations (such as `SELECT`, `UPDATE`, `INSERT`, and `DELETE`) can be parsed and checked. DDL operations (such as `CREATE . . .`, or `ALTER . . .`), transaction-control operations (such as `COMMIT` and `ROLLBACK`), or any other kinds of SQL operations cannot be.
-
-

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` in this example:

```
#sql { SQL operation }; // For a statement with no output, like INSERT
...
#sql result = { SQL operation }; // For a statement with output, like SELECT
```

A clause without a result expression, such as in the first 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` (where the iterator can be an instance of an iterator class or subclass).

A SQL operation in a SQLJ statement can use standard SQL syntax only, or can use a clause with syntax specific to SQLJ (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 two entries in [Table 3-1](#) are general categories for statement clauses that use standard SQL syntax or Oracle PL/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 3-34
FETCH clause	Fetch data from a positional iterator.	"Using Positional Iterators" on page 3-48
COMMIT clause	Commit changes to the data.	"Using Manual COMMIT and ROLLBACK" on page 4-28
ROLLBACK clause	Cancel changes to the data.	"Using Manual COMMIT and ROLLBACK" on page 4-28
SET SAVEPOINT RELEASE SAVEPOINT ROLLBACK TO clauses	Set a savepoint for future rollbacks, release a specified savepoint, roll back to a savepoint.	"Using Savepoints" on page 4-30

Table 3–1 SQLJ Statement Clauses (Cont.)

Category	Functionality	More Information
SET TRANSACTION clause	Use advanced transaction control for access mode and isolation level.	"Advanced Transaction Control" on page 7-49
procedure clause	Call a stored procedure.	"Calling Stored Procedures" on page 3-60
assignment clause	Assign values to Java host expressions.	"Assignment Statements (SET)" on page 3-58
SQL clause	Use standard SQL syntax and functionality: UPDATE, INSERT, DELETE.	<i>Oracle9i SQL Reference</i>
PL/SQL block	Use BEGIN . . END or DECLARE . . BEGIN . . END anonymous block inside SQLJ statement.	"PL/SQL Blocks in Executable Statements" on page 3-14 <i>PL/SQL User's Guide and Reference</i>

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 3-37
function clause	Call a stored function.	"Calling Stored Functions" on page 3-61
iterator conversion clause	Convert a JDBC result set to a SQLJ iterator.	"Converting from Result Sets to Named or Positional Iterators" on page 7-58

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 4-6 discusses connection context instances.

If you have defined one or more *execution context* instances (of the class `sqlj.runtime.ExecutionContext`) 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 7-24 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 };
```

Notes:

- 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 specific to SQLJ.

Consider an employee table EMP with the following rows:

```
CREATE TABLE EMP (
    ENAME VARCHAR2(10),
    SAL NUMBER(7,2) );
```

Use the following SQLJ executable statement (that uses only standard SQL syntax) to insert Joe as a new employee into the EMP table, specifying his name and salary:

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

Elementary INSERT with Connection Context or Execution Context Instances

The following examples use `ctx` as a connection context instance (an instance of either the default `sqlj.runtime.ref.DefaultContext` or a class that you have previously declared in a connection context declaration) and `execctx` as an execution context instance:

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

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

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

A Simple SQLJ Method

This example demonstrates a simple method using SQLJ code, demonstrating how SQLJ statements interrelate with and are interspersed with Java statements. The SQLJ statement uses standard `INSERT INTO table VALUES` syntax supported by 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 3-16.

```
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 4-22.
 - SQLJ function calls also use a `VALUES` token, but these situations are not related semantically.
-

PL/SQL Blocks in Executable Statements

PL/SQL blocks can be used within the curly braces of a SQLJ executable statement just as SQL operations can, as in the following example:

```
#sql {  
  DECLARE  
    n NUMBER;  
  BEGIN  
    n := 1;  
    WHILE n <= 100 LOOP  
      INSERT INTO emp (empno) VALUES(2000 + n);  
      n := n + 1;  
    END LOOP;  
  END  
};
```

This example goes through a loop that inserts new employees in the `EMP` table, creating employee numbers 2001-2100. (It presumes data other than the employee number will be filled in later.)

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

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

Using PL/SQL anonymous blocks within SQLJ statements is one way to use dynamic SQL in your application. You can also use dynamic SQL directly through Oracle SQLJ extensions, or through JDBC code within a SQLJ application. (See ["Support for Dynamic SQL"](#) on page 7-63 and ["SQLJ and JDBC Interoperability"](#) on page 7-53.)

Notes:

- It is recommended that you *not* close a PL/SQL block with a semi-colon after the `END`. The parser will detect the end of the block when it encounters the closing curly brace of the SQLJ clause.
 - 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

This section discusses three categories of Java expressions used in SQLJ code: *host expressions*, *context expressions*, and *result expressions*. Host expressions are the most frequently used and merit the most discussion. (Another category of expressions, called *meta bind expressions*, are used specifically for dynamic SQL operations and use syntax similar to that of host expressions. See "[Support for Dynamic SQL](#)" on page 7-63.)

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 the following:

- arithmetic expressions
- Java method calls with return values
- Java class field values
- array elements
- conditional expressions (a ? b : c)
- logical expressions
- bitwise expressions

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

- local variables
- declared parameters
- class fields (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 or 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. 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 put white space (space, tab, newline, or comment) between the mode specifier and the host variable, as in the following example:

```
:INOUT hostvar
```

The white space is required to distinguish between the mode specifier and the variable name.

- For any other host expression, enclose the expression in parentheses and place it after the mode specifier, or after the colon if there is no mode specifier, as in the following examples:

```
:IN(hostvar1+hostvar2)
```

```
:(hostvar3*hostvar4)
```

```
:(index--)
```

White space is not required after the mode specifier in 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()
```

Syntax 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—any of the following in the SQL namespace:
 - SQL comments after the colon and before the mode specifier
 - SQL comments after the colon and before the host expression if there is no mode specifier
 - SQL comments after the mode specifier and before the host expressionor the following in the Java namespace:
 - Java comments within the host expression (inside the parentheses)
- The `IN`, `OUT`, and `INOUT` syntax used for host variables and expressions is not case sensitive; these tokens can be uppercase, lowercase, or mixed.

- Do not confuse the `IN`, `OUT`, and `INOUT` syntax of SQLJ host expressions with similar `IN`, `OUT`, and `IN OUT` syntax used in PL/SQL declarations to specify the mode of parameters passed to PL/SQL stored functions and procedures.

Usage Notes

- A simple host variable can appear multiple times in the same SQLJ statement, as follows ("output" refers to `OUT` or `INOUT` variables, as applicable):
 - If the host variable appears only as an input variable, then there are no restrictions or complications.
 - If at least one appearance of the host variable is as an output variable in a PL/SQL block, then you will receive a portability warning if the translator `-warn=portability` flag is set. SQLJ runtime behavior in this situation is vendor-specific. The Oracle SQLJ runtime uses value semantics (as opposed to reference semantics) for all occurrences of the host variable. For information about the `-warn=portability` flag, see "[Translator Warnings \(-warn\)](#)" on page 8-45.
 - If at least one appearance of the host variable is as an output variable in a stored procedure call, stored function call, `SET` statement, or `INTO`-list, then you will *not* receive any warning. SQLJ runtime behavior in this situation is standardized, using value semantics.
- If a host expression that is a simple host variable appears multiple times in a SQLJ statement, then by default each appearance is treated completely independently of the other appearances, using value semantics. However, if you use the SQLJ translator `-bind-by-identifier=true` setting, then this is not the case. With a `true` setting, multiple appearances of the same host variable in a given SQLJ statement or PL/SQL block are treated as a single bind occurrence. See "[Binding Host Expressions by Identifier \(-bind-by-identifier\)](#)" on page 8-70.
- When binding a string host expression into a `WHERE` clause for comparison against `CHAR` data, be aware that there is a SQLJ option, `-fixedchar`, that accounts for blank padding in the `CHAR` column when the comparison is made. See "[CHAR Comparisons with Blank Padding \(-fixedchar\)](#)" on page 8-58.

For examples of Oracle SQLJ runtime evaluation of host expressions, see "[Examples of Evaluation of Java Expressions at Runtime \(ISO Code Generation\)](#)" on page 3-24.

Examples of Host Expressions

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 3-34.)

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.

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

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

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

As you can see, ":" 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 empno = 28959 };
```

`OUT` is the default for an `INTO`-list, but you can state it explicitly as well:

```
#sql { SELECT ename INTO :OUT empname FROM emp WHERE empno = 28959 };
```

This looks in the `EMPNO` column of the `EMP` table for employee number 28959, selects the name in the `ename` column of that row, and outputs it to the `empname` output host variable, which is a Java string.

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 shows use of the output of a method call as an input host expression and also uses an input host variable. This statement uses the value returned by `getNewSal()` to update the SAL column in the EMP table for the employee (in the ENAME column) who is specified by the Java `empname` variable. Java code initializing the host variables is also shown.

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

Overview of Result Expressions and Context Expressions

A context expression is an input expression that specifies the name of a connection context instance or an execution context instance to be used in a SQLJ executable statement. Any legal Java expression that yields such a name can be used.

A result expression is an output expression used for query results or a function return. It can be any legal Java expression that is *assignable*, meaning that it can logically appear on the left side of an equals sign (this is sometimes referred to as an *l-value*).

The following examples can be used for either result expressions or context expressions:

- local variables
- declared parameters
- class fields (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 these kinds of expressions:

```
#sql [connctx_exp, execctx_exp] result_exp = { SQL with host expression };
```

Java expressions can be used as any of the following, as appropriate:

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

For ISO standard code generation (the `-codegen=iso` setting), the evaluation of Java expressions is well-defined, even for the use of any side effects that depend on the order in which expressions are evaluated. Examples of such side effects are shown in "[Examples of Evaluation of Java Expressions at Runtime \(ISO Code Generation\)](#)" on page 3-24.

For Oracle-specific code generation (the default `-codegen=oracle` setting), evaluation of Java expressions follows the ISO standard when there are no side effects (except when the `-bind-by-identifier` option is enabled), but is implementation-specific and subject to change when there are side effects.

Important: The following discussion and the related examples later do *not* apply to Oracle-specific code generation. If you use side effects as described here, request ISO code generation during translation.

The following is a summary, for ISO code, of the overall order of evaluation, execution, and assignment of Java expressions for each statement that executes during 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 \(ISO Code Generation\)](#)" on page 3-24 provides examples that clarify this sequence, highlights key points, 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 result expressions as follows.

1. `OUT` and `INOUT` host expressions are assigned output in order from left to right.
2. The result expression, if there is one, is assigned output last.

Note that during runtime all host expressions are treated as distinct values, even if they share the same name or reference the same object. The execution of each SQL operation is treated as if invoking a remote method, and each host expression is taken as a distinct parameter. Each input or input-output parameter is evaluated and passed as it is first encountered, before any output assignments are made for that statement, and each output parameter is also taken as distinct and is assigned exactly once.

It is also important to remember that each host expression is evaluated only once. An `INOUT` expression is evaluated when it is first encountered. When the output assignment is made, the expression itself is not re-evaluated, nor are any side-effects repeated.

Examples of Evaluation of Java Expressions at Runtime (ISO Code Generation)

For ISO code generation (`-codegen=iso`), this section discusses some of the subtleties of how Java expressions are evaluated when your application executes. Do *not* count on these effects if you use Oracle-specific code generation (the default `-codegen=oracle` setting). Request ISO code generation during translation if you depend on such effects.

Numerous examples are included here. Some of these examples use `SELECT INTO` statements, which are described in "[Single-Row Query Results: SELECT INTO Statements](#)" on page 3-34; some use assignment statements, which are described in "[Assignment Statements \(SET\)](#)" on page 3-58; and some use stored procedure and function calls, which are described in "[Stored Procedure and Function Calls](#)" on page 3-60.

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 example 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 example is evaluated as follows:

```
#sql { ... :OUT (array[1]) ... :IN (2) ... };
```

The variable `indx` is incremented to 2 after the first expression is evaluated, but before the second expression is evaluated. It is incremented to 3 after the second expression is evaluated and will have that value the next time it is encountered.

Example 3: prefix and postfix operators

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

This example is evaluated as follows:

```
#sql { ... :OUT (array[2]) ... :IN (2) ... };
```

The variable `indx` is incremented to 2 before the first expression is evaluated. It is incremented to 3 after the second expression is evaluated and will have that value the next time it is encountered.

Example 4: postfix operator

```
int grade = 0;
int count1 = 0;
...
```

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

This example 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 example 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 example 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 versus INOUT versus 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 matters is its position in the left-to-right order.

Example 7: `IN` versus `INOUT` versus `OUT`

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

This example is evaluated as follows:

```
#sql { SET :OUT (array[0]) = 1 };
```

One might expect input expressions to be evaluated before output expressions, but that is not the case. The expression `:OUT (array[n])` is evaluated first because it is the left-most expression. Then `n` is incremented prior to evaluation of `++n`, because it is being operated on by a prefix operator. Then `++n` is evaluated as 1. The result will be assigned to `array[0]`, not `array[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 example is evaluated as follows:

```
#sql { BEGIN :OUT x := 10; :OUT z := 3; END };
```

Therefore, it would print "x=10, z=3".

All expressions in a PL/SQL block are evaluated before any are executed. In this example, the host expressions in the second statement, `:OUT z` and `:x`, are evaluated before the first statement is executed. In particular, the second statement is evaluated while `x` still has its original value of 3, before it has been assigned the value 10.

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 example is evaluated as follows:

```
#sql { BEGIN
      :OUT x := 4 + 1;
      :OUT z := 1;
      END };
```

The postfix increment operator is executed after `(y++)` is evaluated, so the expression is evaluated as 4 (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 versus 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 example is evaluated as follows:

```
#sql { BEGIN :OUT y := 1 + 1; :OUT x := 1 + 1; END };
```

The `y` in the second statement is evaluated before either statement is executed, 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 Only

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 (because 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 3-60.

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, or in 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, 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, SQLJ allows you to assign selected items directly to Java host expressions inside SQL syntax. This is done using the `SELECT INTO` statement.

SELECT INTO Syntax

The syntax for a `SELECT INTO` statement is as follows:

```
#sql { SELECT expression1, ..., expressionN INTO :host_exp1, ..., :host_expN
      FROM table <optional_clauses> };
```

where:

- The items *expression1* through *expressionN* are expressions specifying what is to be selected from the database. These can be any expressions valid for any `SELECT` statement. This list of expressions is referred to as the *SELECT-list*.

In a simple case, these would be names of columns from a database table.

It is also legal to include a host expression in the *SELECT-list*. See ["Examples of SELECT INTO Statements"](#) below.

- The items *host_exp1* through *host_expN* are target host expressions, such as variables or array elements. This list of host expressions is referred to as the *INTO-list*.
- The item *table* is the name of the database table, view, or snapshot from which you are selecting the data.
- The item *optional_clauses* is for any additional clauses you want to include that are valid in a `SELECT` statement, such as a `WHERE` clause.

A `SELECT INTO` statement must return one, and only one, row of data, otherwise an error will be generated at 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 *table*, and the optional clauses is the same as for any SQL SELECT statement. For information about what is permissible in Oracle SQL, see the *Oracle9i 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 use an employee table EMP with the following rows:

```
CREATE TABLE EMP (
  EMPNO NUMBER(4),
  ENAME VARCHAR2(10),
  HIREDATE DATE );
```

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

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

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

```
String empname;
Date hdate;
#sql { SELECT ename, hiredate INTO :empname, :hdate FROM emp
      WHERE empno=28959 };
```

Examples with Host Expressions in SELECT-List

It is legal to use Java host expressions in the SELECT-list as well as in the INTO-list.

For example, you can select directly from one host expression into another (though this is of limited usefulness):

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

More realistically, you might want to perform an operation or concatenation on the data selected, as in the following examples. Assume Java variables were previously declared and assigned, as necessary.

```
...
#sql { SELECT sal + :raise INTO :newsal FROM emp WHERE empno=28959 };
...

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

```

In the second example, presume MYEMP is a table much like the EMP table but with an EMP_LAST_NAME column instead of an ENAME column. In the SELECT statement, `firstname` is prepended to " " (a single space), using a Java host expression and 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.

SELECT INTO Error Conditions

Remember that `SELECT INTO` statements are intended for queries that return exactly one row of data only.

A `SELECT INTO` query that finds zero rows or multiple rows will result in an exception, as follows:

- A `SELECT INTO` finding now rows will return an exception with a SQL state of 2000, representing a "no data" condition.
- A `SELECT INTO` finding multiple rows will return an exception with a SQL state of 21000, representing a cardinality violation.

These exceptions are listed under "[SQLJ Runtime Messages](#)" on page B-47. You can retrieve the SQL state through the `getSQLState()` method of the `java.sql.SQLException` class, as described in "[Retrieving SQL States and Error Codes](#)" on page 4-24.

This is vendor-independent behavior that is specified in the ISO SQLJ standard. There is no vendor-specific error code in these cases—the error code is always 0.

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`. If you declared it at the class level or nested-class level, then it might be advisable to declare it as `public static`.

For information about usage as stored function returns, see ["Using Iterators and Result Sets as Stored Function Returns"](#) on page 3-63, 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 7-38. This section discusses how iterator classes are implemented and what advanced functionality is available, such as interoperability with JDBC result sets and subclassing of iterators.

Iterator Concepts

Using a SQLJ iterator declaration, as described in ["Overview of SQLJ Declarations"](#) on page 3-2, results in a strongly typed iterator. This is the typical usage for iterators, and takes particular advantage of SQLJ semantics-checking features during translation.

It is also possible, and at times advantageous, to use weakly typed iterators. There are generic classes you can instantiate in order to use a weakly typed iterator.

This section primarily introduces features of strongly typed iterators, but concludes with a brief introduction to weakly typed iterators.

Introduction to Strongly Typed Iterators

Before using a strongly typed iterator object, you must declare an iterator class. An iterator declaration specifies a Java class that SQLJ constructs for you, where the class attributes define the 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 a standard `java.sql.ResultSet` instance and can, in principle, contain any number of columns of any type.

When you declare an iterator, you specify either just the 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 retrieved into an iterator object are converted to the Java types specified in the iterator declaration.

When you query to populate a named iterator object, the 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 `ENAME` column in a database table can be selected and put into an iterator `ename` column. Alternatively, you can use an alias to map a database column name to an iterator column name if the names differ. 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. (These last two cases are discussed in ["Instantiating and Populating Named Iterators"](#) on page 3-45.)

Because SQLJ iterators are strongly typed, they offer the benefit of Java type-checking during the SQLJ semantics-checking phase.

As an example, consider the following table:

```
CREATE TABLE EMP_SAL (
    EMPNO NUMBER(4),
    ENAME VARCHAR2(10),
    OLDSAL NUMBER(10),
    RAISE NUMBER(10) );
```

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

Declaration:

```
#sql iterator SalNamedIter (int empno, String ename, float raise);
```

Executable code:

```
class MyClass {
    void func() throws SQLException {
        ...
        SalNamedIter niter;
        #sql niter = { SELECT ename, empno, raise FROM emp_sal };

        ... process niter ...
    }
}
```

This is a simple case where the iterator column names match the table column names. Note that the order of items in the `SELECT` statement does not matter when you use a named iterator—data is matched by name, not position.

When you query to populate a positional iterator object, the data is retrieved according to the order in which you select the columns. Data from the first column selected from the database table is placed into the first column of the iterator, and so on. The 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 `EMP_SAL` 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;
        #sql piter = { SELECT empno, ename, raise FROM empсал };

        ... process piter ...
    }
}
```

Note that the order of the data items in the SELECT statement must be the same as in the iterator.

The processing differs between named iterators and positional iterators, as described in ["Accessing Named Iterators"](#) on page 3-46 and ["Accessing Positional Iterators"](#) on page 3-49.

General Iterator Notes In addition to the preceding concepts, be aware of the following general notes about iterators:

- "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 8-45.
- 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 7-58.

- Generally speaking, the contents of an iterator is determined only by the state of the database at the time of execution of the `SELECT` statement that populated it. Subsequent `UPDATE`, `INSERT`, `DELETE`, `COMMIT`, or `ROLLBACK` operations have no effect on the iterator or its contents. This is further discussed in ["Effect of Commits and Rollbacks on Iterators and Result Sets"](#) on page 4-29.

The exception to this is if you declare an iterator to be scrollable and "sensitive" to changes in the data. See ["Declaring Scrollable Iterators"](#) on page 7-42 and ["Scrollable Iterator Sensitivity"](#) on page 7-42.

Introduction to Weakly Typed Iterators

In case you would rather not declare an iterator class, Oracle SQLJ permits you to use a weakly typed kind of iterator. Such iterators are known as *result set iterators*. To use a plain (non-scrollable) result set iterator, instantiate the `sqlj.runtime.ResultSetIterator` class. To use a scrollable result set iterator, instantiate the `sqlj.runtime.ScrollableResultSetIterator` class. (Scrollable iterators are described in ["Scrollable Iterators"](#) on page 7-42.)

The drawback to using result set iterators, compared to strongly typed iterators, is that SQLJ cannot perform as much semantics-checking for your queries.

For more information, see ["Result Set Iterators"](#) on page 7-41.

General Steps in Using an Iterator

Five general steps are involved in using SQLJ named or positional iterator:

1. Use a SQLJ declaration to define the iterator class (in other words, to define the iterator type).
2. Declare a variable of the iterator class.
3. Populate the iterator variable with the results from a SQL query, using a `SELECT` statement.
4. Access the query columns in the iterator. How to accomplish this differs between named iterators and positional iterators, as explained below.
5. When you finish processing the results of the query, close the iterator to release its resources.

Named Iterators Versus Positional Iterators Versus Result Set Iterators

There are advantages and appropriate situations for each kind of SQLJ iterator.

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 do not 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 available row of the iterator before retrieving the next set of values.

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 `ename()` method to retrieve data from an `ename` 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.

Finally, if you do not want to declare strongly typed iterator classes for your queries, you can choose the alternative of using weakly typed result set iterators. Result set iterators are most convenient when converting JDBC code to SQLJ code. You must balance this consideration against the fact that result set iterators (either `ResultSetIterator` instances or `ScrollableResultSetIterator` instances) do not allow complete SQLJ semantics-checking during translation. With named or positional iterators, SQLJ verifies that `SELECT`-list types match the Java types into which the data will be materialized. With result set iterators, this is not possible. See "[Result Set Iterators](#)" on page 7-41 for more information.

Comparative Iterator Notes Be aware of the following notes regarding SQLJ iterators:

- In populating a positional iterator, the number of columns you select from the database must equal the number of columns in the iterator. In populating a

named iterator, the number of columns you select from the database can never be less than the number of columns in the iterator, but can be greater than the number of columns in the iterator if you have the translator `-warn=nostrict` flag set. Unmatched columns are ignored in this case. For information about this flag, see "[Translator Warnings \(-warn\)](#)" on page 8-45.

- 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, depending on the row-prefetch value. This is because you are fetching data from the iterator, not the database. If the row-prefetch value is 1, however, then each fetch does involve a separate trip to the database. (The row-prefetch value determines how many rows are retrieved with each trip to the database. See "[Row Prefetching](#)" on page 10-3.)
- Result set iterators use the same `FETCH INTO` syntax used with positional iterators, and are subject to the same restriction at runtime—the size (number of data items) of the `SELECT`-list must match the number of variables that are assigned data in the `FETCH` statement.

Using Named Iterators

When you declare a named iterator class, you declare the name as well as the 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 `ENAME` would match `ENAME`).
- 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 names are identical to the column names. (Unlike most accessor method names in Java, accessor method names in named iterator classes do not start with "get".) For example, a named iterator object with a column `sal` would have a `sal()` accessor method.

Note: The following restrictions apply in naming the columns of a named iterator:

- Column names cannot use Java reserved words.
 - Column names cannot have the same name as utility methods provided in named iterator classes, such as the `next()`, `close()`, `getResultSet()`, and `isClosed()` methods. For scrollable named iterators, this includes additional methods such as `previous()`, `first()`, and `last()`. (See "[The Scrollable Interface](#)" on page 7-43 and "[Scrollable Named Iterators](#)" on page 7-44.)
-
-

Declaring Named Iterator Classes

Use the following syntax to declare a named iterator class:

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

In this syntax, *modifiers* is an optional sequence of legal Java class modifiers, *classname* is the desired class name for the iterator, and *type-name-list* is a list of the Java types and names equivalent to (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.

Now consider the following table:

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

You might declare the following named iterator for use with this table:

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

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

Note: As with standard Java, any public class should be declared in one of the following ways. This is a requirement if you are using the standard `javac` compiler provided with the Sun Microsystems JDK:

- Declare it in a separate source file. The base name of the file should be the same as the class name.

or:

- Declare it at class-level scope or nested-class-level scope, with `public static` modifiers.
-
-

Instantiating and Populating Named Iterators

Declare a variable of the `ProjIter` positional iterator type from the preceding section and populate it with a `SELECT` statement.

Continuing to use the `PROJECTS` table and `ProjIter` iterator defined in the preceding section, note that there are columns in the table whose names and datatypes match the `id` and `projname` columns of the iterator, but you must use an alias and perform an operation to populate the `deadline` column of the iterator. Here is an example:

```
ProjIter projsIter;
```

```
#sql projsIter = { SELECT start_date + duration AS deadline, projname, id  
                  FROM projects WHERE start_date + duration >= sysdate };
```

This calculates a deadline for each project by adding its duration to its start date, then aliases the results as `deadline` to match the `deadline` iterator column. It also uses a `WHERE` clause so that only future deadlines are processed (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 function `MAXIMUM()` that takes a `DURATION` entry and an integer as input and returns the maximum of the two. For example, you could input a 3 to make sure each project has at least a three-month duration in your application.

Now presume you are declaring your iterator as follows:

```
#sql public iterator ProjIter2 (String projname, int id, float duration);
```

You could use the `MAXIMUM()` function in your query, with an alias for the result, as follows.

```
ProjIter2 projsIter2;  
  
#sql projsIter2 = { SELECT id, projname, maximum(duration, 3) AS duration  
                   FROM projects };
```

Generally, you must use an alias in your query for any 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 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 3-45.

Note: Each iterator has a `close()` method that you must always call when you finish retrieving data from the iterator. This is necessary to close the iterator and free its resources.

Presume the following iterator class declaration:

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

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

```
// Declare the iterator variable  
ProjIter projsIter;  
  
// Instantiate and populate iterator; order of SELECT doesn't matter
```

```
#sql projsIter = { SELECT start_date + duration AS deadline, projname, id
                  FROM projects WHERE start_date + duration >= sysdate };

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

// Close the iterator
projsIter.close();
...
```

Note the convenient use of the `projname()`, `id()`, and `deadline()` accessor methods to retrieve the data. Note also that the order of the `SELECT` items does not matter, nor does the order in which the accessor methods are used.

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.

Using Positional Iterators

When you declare a positional iterator class, you declare the datatype of each column but not the column name. The Java types into which the columns of the SQL query results are selected must be compatible with the 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 3-49).

Declaring Positional Iterator Classes

Use the following syntax to declare a positional iterator class:

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

In this syntax, *modifiers* is an optional sequence of legal Java class modifiers, and the *position-list* is a list of Java types compatible with the column types in a database table.

The *implements clause* and *with clause* are optional, specifying interfaces to implement and variables to define and initialize, respectively. These are discussed in ["Declaration IMPLEMENTS Clause"](#) on page 3-5 and ["Declaration WITH Clause"](#) on page 3-6.

Now consider an employee table `EMP` with the following rows:

```
CREATE TABLE EMP (
    EMPNO NUMBER(4),
    ENAME VARCHAR2(10),
    SAL NUMBER(7,2) );
```

And consider the following positional iterator declaration:

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

This example defines Java class `EmpIter` with unnamed `String`, `int`, and `float` columns. Note that the table columns and iterator columns are in a different order—the `String` corresponds to `ENAME` and the `int` corresponds to `EMPNO`. The

order of the iterator columns determines the order in which you must select the data, as shown in "[Instantiating and Populating Positional Iterators](#)" below.

Note: As with standard Java, any public class should be declared in one of the following ways. This is a requirement if you are using the standard `javac` compiler provided with the Sun Microsystems JDK:

- Declare it in a separate source file. The base name of the file should be the same as the class name.

or:

- Declare it at class-level scope or nested-class-level scope, with `public static` modifiers.
-
-

Instantiating and Populating Positional Iterators

Declare a variable of the `EmpIter` positional iterator type from the preceding section and populate it with a `SELECT` statement.

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.

The three datatypes in the `EmpIter` iterator class are compatible with the types of the `EMP` table, but be careful how you select the data, because the order is different. The following will work, because the `SELECT`-fields are in the same order as the iterator columns, as declared above in "[Declaring Positional Iterator Classes](#)":

```
EmpIter empIter;  
  
#sql empIter = { SELECT ename, empno, sal FROM emp };
```

Remember that in populating a positional iterator, the number of columns you select from the database must equal the number of columns in the iterator.

Accessing Positional Iterators

Access the columns defined by a positional iterator using `SQL FETCH INTO` syntax.

The `INTO` part of the command specifies Java host variables that receive the results columns. The host variables must be in the same order as the corresponding iterator columns. Use the `endFetch()` method provided with all positional iterator classes to determine whether the last fetch reached the end of the data.

Notes:

- The `endFetch()` method initially returns `true` before any rows have been fetched, then returns `false` once a row has been successfully retrieved, then returns `true` again when a `FETCH` finds no more rows to retrieve. Therefore, you must perform the `endFetch()` test *after* the `FETCH INTO` statement. If your `endFetch()` test precedes the `FETCH INTO` statement, then you will never retrieve any rows, because `endFetch()` would be `true` before your first `FETCH` and you would immediately break out of the `while` loop.
 - The `endFetch()` test must be *before* the results are processed, however, because the `FETCH` does not throw a SQL exception when it reaches the end of the data, it just triggers the next `endFetch()` call to return `true`. If there is no `endFetch()` test before results are processed, then your code will try to process null or invalid data from the first `FETCH` attempt after the end of the data had been reached.
 - Each iterator has a `close()` method that you must always call once you finish retrieving data from it. This is necessary to close the iterator and free its resources.
-

The following is an example, repeating the declaration, instantiation, and population used under "[Instantiating and Populating Positional Iterators](#)" above.

Note that the Java host variables in the `SELECT` statement are in the same order as the columns of the positional iterator, which is mandatory.

First, presume the following iterator class declaration:

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

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

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

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



```
#sql empIter = { SELECT ename, empno, sal FROM emp };

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

// Close the iterator
empIter.close();
...
```

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

Do not use the `next()` method for a positional iterator. A `FETCH` operation calls it implicitly to move to the next row.

Note: Host variables in a `FETCH INTO` statement must always be initialized because they are assigned in one branch of a conditional statement. Otherwise, you will get a compiler error indicating they may never be assigned. `FETCH` can assign the variables only if there was a row to be fetched.

Positional Iterator Navigation with the `next()` Method

The positional iterator `FETCH` clause discussed in the previous section performs a movement—an implicit `next()` call—before it populates the host variables (if any). As an alternative, Oracle SQLJ supports using a special `FETCH` syntax in conjunction with explicit `next()` calls in order to use the same movement logic as with JDBC result sets and SQLJ named iterators. Using this special `FETCH` syntax, the semantics differ—there is no implicit `next()` call before the `INTO`-list is populated.

See ["FETCH CURRENT Syntax: from JDBC Result Sets to SQLJ Iterators"](#) on page 7-46 for more information.

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 3-63.
 - 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.

For the examples in this section, consider the following department and employee tables:

```
CREATE TABLE DEPT (  
    DEPTNO NUMBER(2),  
    DNAME VARCHAR2(14) );
```

```
CREATE TABLE EMP (  
    EMPNO NUMBER(4),  
    ENAME VARCHAR2(10),  
    SAL NUMBER(7,2),  
    DEPTNO NUMBER(2) );
```

Example: Use of Result Set as OUT Host Variable This example uses a JDBC result set as an output host variable.

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

```

while (rs.next())
{
    String empname = rs.getString(1);
    int empnum = rs.getInt(2);
}
rs.close();
...

```

This example opens the result set `rs` in a PL/SQL block to receive data from a `SELECT` statement, selects data from the `ENAME` and `EMPNO` columns of the `EMP` table, then loops through the result set to retrieve data into local variables.

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

Declaration:

```
#sql public <static> iterator EmpIter (String ename, int empno);
```

The `public` modifier is required, and `static` may be advisable if your declaration is at class level or nested-class level.

Executable code:

```

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

while (iter.next())
{
    String empname = iter.ename();
    int empnum = iter.empno();

    ...process/output empname and empnum...
}
iter.close();
...

```

This example opens the iterator `iter` in a PL/SQL block to receive data from a `SELECT` statement, selects data from the `ENAME` and `EMPNO` columns of the `EMP` table, then loops through the iterator to retrieve data into local variables.

Example: Use of Iterator as OUT Host Variable for SELECT INTO This example uses a named iterator as an output host variable, taking data through a `SELECT INTO` statement. `OUT` is the default for host variables in an INTO-list. For information about `SELECT INTO` statements and syntax, see ["Single-Row Query Results: SELECT INTO Statements"](#) on page 3-34.

Declaration:

```
#sql public <static> iterator ENameIter (String ename);
```

The `public` modifier is required, and `static` may be advisable if your declaration is at class level or nested-class level.

Executable code:

```
...
ENameIter enamesIter;
String deptname;
...

#sql { SELECT dname, cursor
      (SELECT ename FROM emp WHERE deptno = dept.deptno)
      INTO :deptname, :enamesIter FROM dept WHERE deptno = 20 };

System.out.println(deptname);
while (enamesIter.next())
{
    System.out.println(enamesIter.ename());
}
enamesIter.close();
...
```

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

- Select the name of department number 20 from the `DEPT` table, selecting it into the output host variable `deptname`.
- Query the `EMP` table to select all employees whose department number is 20, selecting the resulting cursor into the output host variable `enamesIter`, which is a named iterator.
- Print the department name.
- Loop through the named iterator printing employee names. This prints the names of all employees in the department.

In most cases, using `SELECT INTO` is more convenient than using nested iterators if you are retrieving a single row in the outer `SELECT`, although that option is also available as discussed below (such as in "[Example: Named Iterator Column in a Positional Iterator](#)" on page 3-57). Also, with nested iterators, you would have to process the data to determine how many rows there are in the outer `SELECT`. With `SELECT INTO` you are assured of just one row.

Using Iterators and Result Sets as Iterator Columns

Oracle SQLJ includes extensions that allow iterator declarations to specify columns of type `ResultSet` or columns of other iterator types 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—each uses a nested result set or iterator (result sets or iterators in a column within an iterator) to print all the employees in each department in the `DEPT` table. The first example uses result sets within a named iterator, the second example uses named iterators within a named iterator, and the third example uses named iterators within a positional iterator.

Here are the steps:

1. Select each `DNAME` (department name) from the `DEPT` table.
2. Do a nested `SELECT` into a cursor to get all employees from the `EMP` table for each department.
3. Put the department names and sets of employees into the outer iterator (`iter`), which has a name column and an iterator column. The cursor with the employee information for any given department goes into the iterator column of that department's row of the outer iterator.
4. Go through a nested loop that, for each department, prints the department name and then loops through the inner iterator to print all employee names for that department.

Example: Result Set Column in a Named Iterator This example uses a column of type `ResultSet` in a named iterator.

Declaration:

```
#sql iterator DeptIter (String dname, ResultSet emps);
```

Executable code:

```
...
DeptIter iter;
...
#sql iter = { SELECT dname, cursor
              (SELECT ename FROM emp WHERE deptno = dept.deptno)
              AS emps FROM dept };
while (iter.next())
{
    System.out.println(iter.dname());
    ResultSet enamesRs = iter.emps();
    while (enamesRs.next())
    {
        String empname = enamesRs.getString(1);
        System.out.println(empname);
    }
    enamesRs.close();
}
iter.close();
...
```

Example: Named Iterator Column in a Named Iterator This example uses a named iterator that has a column whose type is that of a previously defined named iterator (nested iterators).

Declarations:

```
#sql iterator ENameIter (String ename);
#sql iterator DeptIter (String dname, ENameIter emps);
```

Executable code:

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

while (iter.next())
{
    System.out.println(iter.dname());
    ENameIter enamesIter = iter.emps();
    while (enamesIter.next())
```

```

        {
            System.out.println(enamesIter.ename());
        }
        enamesIter.close();
    }
    iter.close();
    ...

```

Example: Named Iterator Column in a Positional Iterator This example uses a positional iterator that has a column whose type is that of a previously defined named iterator (nested iterators). This uses the `FETCH INTO` syntax of positional iterators. This example is functionally equivalent to the previous two.

Note that because the outer iterator is a positional iterator, there does not have to be an alias to match a column name, as was required when the outer iterator was a named iterator in the previous example.

Declarations:

```

#sql iterator ENameIter (String ename);
#sql iterator DeptIter (String, ENameIter);

```

Executable code:

```

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

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

```

Assignment Statements (SET)

SQLJ 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 PL/SQL code:

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

Here is a simple example of an assignment statement:

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

This statement assigns to `x` the sum of the return values of `foo1()` and `foo2()` and assumes that the type of `x` is compatible with the type of the sum of the outputs of these functions.

Consider the following additional examples:

```
int i2;
java.sql.Date dat;
...
#sql { SET :i2 = TO_NUMBER(substr('750 etc.', 1, 3)) +
        TO_NUMBER(substr('250 etc.', 1, 3)) };
...
#sql { SET :dat = sysdate };
...

```

The first statement will assign to `i2` the value 1000 (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 3-60. 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. These procedures and functions could be written in Java, PL/SQL, or any other language supported by the database.

A stored function requires a result expression in your SQLJ executable statement to accept the return value, and 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 6-28. For additional information, see the *Oracle9i 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 white space and then the procedure name. There must be a space after the CALL token to differentiate it from the procedure name. There *cannot* be a set of outer parentheses around the procedure call. This differs from the syntax for function calls, as explained in "[Calling Stored Functions](#)" on page 3-61.

```
#sql { CALL PROC(<PARAM_LIST>) };
```

PROC is the name of the stored procedure, which can optionally take a list of input, output, and input-output parameters. *PROC* can include a schema or package name as well, such as SCOTT.MYPROC().

Presume that you have defined the following PL/SQL stored procedure:

```
CREATE OR REPLACE PROCEDURE MAX_DEADLINE (deadline OUT DATE) IS
BEGIN
    SELECT MAX(start_date + duration) INTO deadline FROM projects;
END;
```

This reads the 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 white space between the `VALUES` token and the begin-parenthesis. (A `VALUES` token can also be used in `INSERT INTO table VALUES` syntax supported by Oracle SQL, but these situations are unrelated semantically and syntactically.)

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

In this syntax, *result* is the result expression, which takes the function return value. *FUNC* is the name of the stored function, which can optionally take a list of

input, output, and input-output parameters. *FUNC* can include a schema or package name, such as `SCOTT.MYFUNC()`.

Referring back to the example in "Calling Stored Procedures" on page 3-60, consider defining the stored procedure as a stored function instead, as follows:

```
CREATE OR REPLACE FUNCTION GET_MAX_DEADLINE RETURN DATE IS
  deadline DATE;
BEGIN
  SELECT MAX(start_date + duration) INTO deadline FROM projects;
  RETURN deadline;
END;
```

In SQLJ, you can call this `GET_MAX_DEADLINE` function as follows:

```
java.sql.Date maxDeadline;
...
#sql maxDeadline = { VALUES(GET_MAX_DEADLINE) };
```

The result expression must have a type compatible with the return type of the function.

In 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.

Note: If you want your stored function to be portable to non-Oracle environments, then you should use only input parameters in the calling sequence, not output or input-output parameters.

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, if the function returns a REF CURSOR type.

The following example uses an iterator to take a stored function return. Using a result set is similar.

Example: Iterator as Stored Function Return This example uses an iterator as a return type for a stored function, using a REF CURSOR type in the process. REF CURSOR types are described in "[Support for Oracle REF CURSOR Types](#)" on page 5-40.

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.

Declaration:

```
#sql public <static> iterator EmpIter (String ename, int empno);
```

The `public` modifier is required, and `static` may be advisable if your declaration is at class level or nested-class level.

Executable code:

```
EmpIter iter;
...
#sql iter = { VALUES(sqlj_refcursor.job_listing('SALES')) };
```

```
while (iter.next())
{
    String empname = iter.ename();
    int empnum = iter.empno();

    ... process empname and empnum ...
}
iter.close();
...
```

This example calls the `job_listing()` function to return an iterator that contains the name and employee number of each employee whose job title is "SALES". It then retrieves this data from the iterator.

Key Programming Considerations

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

- [Selection of the JDBC Driver](#)
- [Connection Considerations](#)
- [Null-Handling](#)
- [Exception-Handling Basics](#)
- [Basic Transaction Control](#)
- [Summary: First Steps in SQLJ Code](#)
- [Oracle-Specific Code Generation \(No Profiles\)](#)
- [Requirements and Restrictions for Naming](#)
- [Considerations for SQLJ in the Middle Tier](#)

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.

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

Overview of the Oracle JDBC Drivers

Oracle provides the following JDBC drivers:

- **OCI driver** for client-side use with an Oracle client installation
- **Thin driver**, a 100% Java driver for client-side use, particularly with applets (does not require an Oracle client installation)
- **server-side Thin driver**, which is functionally the same as the client-side Thin driver, but is for code that runs inside an Oracle server and needs to access a remote server
- **server-side internal driver** for code that runs inside the target server (that is, inside the Oracle server that it must access)

Oracle provides client-side drivers compatible with JDK 1.1, JDK 1.2 (or higher), and JDK 1.4. The versions in the Oracle9i database are compatible with JDK 1.2 or higher. (The Oracle9i release 2 database includes a JDK 1.3 J2SE Java environment.)

The rest of this section provides a brief overview of each driver. For more information about the drivers and about which might be most appropriate for your particular situation, see the *Oracle9i 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 driver at translation time for semantics-checking, but the Oracle JDBC Thin driver at runtime.

Core JDBC Functionality

The core functionality of all these drivers is the same. They support the same feature set, syntax, programming interfaces, and Oracle extensions. All Oracle JDBC drivers are supported by the `oracle.jdbc.OracleDriver` class.

JDBC OCI Driver

The Oracle JDBC OCI driver accesses the database by calling the Oracle Call Interface (OCI) directly from Java, providing the highest compatibility with the different Oracle 7, 8, 8*i*, and 9*i* versions. These drivers support all installed Oracle9*i* Net adapters, including IPC, named pipes, TCP/IP, and IPX/SPX.

The use of native methods to call C entry points makes the OCI driver dependent on the Oracle platform, requiring an Oracle client installation that includes Oracle9*i* Net. Therefore it is not suitable for applets.

"Connect strings" for the OCI driver is of the following form (where *tns* is an optional TNS alias or full TNS specification):

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

(For backward compatibility, "oci8" is still acceptable instead of "oci". Also, "oci7" is accepted for Oracle JDBC release 7.3.4.)

JDBC Thin Driver

The Oracle JDBC Thin driver is a platform-independent, 100% pure Java implementation that uses Java sockets to connect directly to the Oracle server from any Oracle or non-Oracle client. It can be downloaded into a browser simultaneously with the Java applet being run.

The Thin driver supports only TCP/IP protocol and requires a TNS listener to be listening on TCP/IP sockets from the database server. When the Thin driver is used with an applet, the client browser must have the capability to support Java sockets.

Connect strings for the Thin driver are typically of the following form (though there is also a longer form):

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

JDBC Server-Side Thin Driver

The Oracle JDBC server-side Thin driver offers the same functionality as the client-side Thin driver, but runs inside Oracle9*i* and accesses a remote server. This is useful in accessing one Oracle server from inside another, such as from a Java stored procedure.

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

Note: In order to leave the originating database when using the server-side Thin driver, the user account must have `SocketPermission` assigned. See the *Oracle9i JDBC Developer's Guide and Reference* for more information. See the *Oracle9i Java Developer's Guide* for general information about `SocketPermission` and other permissions.

JDBC Server-Side Internal Driver

The Oracle JDBC server-side internal driver provides support for any Java code that runs inside the target Oracle9i instance where the SQL operations are to be performed. The server-side internal driver allows the Oracle JVM to communicate directly with the SQL engine. This driver is the default JDBC driver for SQLJ code running as a stored procedure, stored function, or trigger in Oracle9i.

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

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

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

Driver Selection for Translation

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

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

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

For information about these options, see "[Connection Options](#)" on page 8-34.

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 class with the driver manager. One driver class might be used for multiple driver protocols (such as `OracleDriver`, which is used for all of the Oracle JDBC protocols).

Driver Selection and Registration for Runtime

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

If you are using an Oracle JDBC driver and create a default connection using the `Oracle.connect()` method (discussed below, under "[Single Connection or Multiple Connections Using DefaultContext](#)" on page 4-6), then SQLJ handles this automatically—`Oracle.connect()` registers the `oracle.jdbc.OracleDriver` class.

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

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

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

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

In any case, you must also set your connection URL, user name, and password. This is described in "[Single Connection or Multiple Connections Using DefaultContext](#)" on page 4-6. That section also further discusses the `Oracle.connect()` method.

Note: As an alternative to using the JDBC driver manager in establishing JDBC connections, you can use *data sources*. You can specify a data source in a *with* clause, as described in "[Declaration WITH Clause](#)" on page 3-6. For general information about data sources, see the *Oracle9i JDBC Developer's Guide and Reference*.

Connection Considerations

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

- Will you need just one database connection or multiple connections?
- If using multiple connections (possibly to multiple schemas), will each connection use SQL entities of the same name—tables of the same name, columns of the same name and datatypes, stored procedures of the same name and signature, and so on?
- Will you need different connections 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 was previously set as the default).

Note: If your operations will use different sets of SQL entities, then you will typically want to declare and use additional connection context classes. This 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 that use SQL entities with the same names and datatypes.

Single Connection

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

You can use the `connect()` method of the `oracle.sqlj.runtime.Oracle` class to accomplish this. Calling this method automatically initializes the default connection context instance.

This method has several signatures, including ones that allow you to specify user name, password, and URL, either directly or using a properties file. In the following example, the properties file `connect.properties` is used:

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

Assume `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 2-14.

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

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

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

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

In this 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 (Oracle session ID) of the database to connect to on that machine.

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

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

Note that in using a Thin driver, the URL must include the hostname, port number, and SID, as in the preceding example, and the database must have a listener running at the specified port. In using the OCI driver, you can specify an SID, or no SID if you intend to use the client's default account. Alternatively, you can use name-value pairs (see the *Oracle9i JDBC Developer's Guide and Reference* for more information).

The first example here will connect to the database with SID `orcl`; the second example will connect to the default account of the client:

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

Notes:

- `Oracle.connect()` will not set your default connection if one had already been set. In that case, it returns `null`. (This functionality allows you to use the same code on a client or in the server.) If you do want to override your default connection, use the static `setDefaultContext()` method of the `DefaultContext` class, as described in the next section.
- The `Oracle.connect()` method defaults to a `false` setting of the auto-commit flag; however, it also has signatures to set it explicitly. See "[More About the Oracle Class](#)" on page 4-12. For general information about auto-commit functionality, see "[Basic Transaction Control](#)" on page 4-26. (In Oracle JDBC, the auto-commit flag defaults to `true`.)
- You can optionally specify `getClass()`, instead of `MyClass.class`, in the `Oracle.connect()` call, as long as you are not calling `getClass()` from a static method. The `getClass()` method is used in some of the SQLJ demo applications.
- You can access the static `DefaultContext` instance, which corresponds to your default connection, as follows:

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

Multiple Connections

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

This is actually an understood shortcut for the following:

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

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

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

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

The following statements establish multiple connections to the same schema (in case you want to use multiple Oracle 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 SID `orcl1` on the machine `localhost1`, using the Oracle JDBC Thin driver.

Now consider a case where you 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 SID `orcl1` on the machine `localhost1`, while the `ctx2` object connects to `bill/lion` on 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: 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 method initializes the default connection context instance. This way, you can avoid specifying connections in your SQLJ statements.

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

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

Closing Connections

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

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

In addition, the `oracle.sqlj.runtime.Oracle` class has a static `close()` method to close the default connection only.

In the following example, presume `ctx` is an instance of any connection context class:

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

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

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

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

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

Always commit or roll back any pending changes before closing the connection. Whether there would be an implicit `COMMIT` operation as the connection is closed is

not specified in the JDBC standard and may vary from vendor to vendor. For Oracle, there is an implicit `COMMIT` when a connection is closed, and an implicit `ROLLBACK` when a connection is garbage-collected without being closed, but it is not advisable to rely on these mechanisms.

Note: It is also possible to close a connection context instance without closing the underlying connection (in case the underlying connection is shared). See "[Closing Shared Connections](#)" on page 7-57.

Multiple Connections Using Declared Connection Context Classes

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

This situation is somewhat advance, however. 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 initializes the default connection context instance—instantiating a `DefaultContext` object and installing it as your default connection. You do not need to assign or use the `DefaultContext` instance returned by `connect()`. If you had already established a default connection, then `connect()` returns `null`.

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

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

The static `close()` method closes the default connection.

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

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

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

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

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

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

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

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

`DefaultContext`, you must specify the auto-commit setting. In Oracle JDBC, the default for the auto-commit flag is `true`.

The auto-commit flag is discussed in ["Basic Transaction Control"](#) on page 4-26.

Some examples of `connect()` and `getConnection()` calls are under ["Single Connection or Multiple Connections Using DefaultContext"](#) on page 4-6.

Optional `Oracle.close()` Method Parameters

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

To keep the underlying physical connection open:

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

To close the underlying physical connection (default behavior):

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

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

For more information about using these parameters and about shared connections, see ["Closing Shared Connections"](#) on page 7-57.

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 7-9.)

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

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

DefaultContext Methods

The `DefaultContext` class has four methods of note:

- `getConnection()`—Gets the underlying JDBC connection object. This is useful if you want to have JDBC code in your application (which is one way to use dynamic SQL operations, for example). 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; it 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 the `Oracle.connect()` method).
- `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()` method).
- `close()`—Like any connection context class, the `DefaultContext` class includes a `close()` method to close the connection context instance.

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

Note: On a client, `getDefaultContext()` returns null if `setDefaultContext()` was not previously called. However, if a data source object has been bound under "jdbc/defaultDataSource" in JNDI, then the client will use this data source object as its default connection. (For information about Oracle SQLJ support for data sources and JNDI, see ["Standard Data Source Support"](#) on page 7-13.)

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

DefaultContext Constructors

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 input parameters as follows.

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

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

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

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

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

Following is an example of constructing a DefaultContext instance:

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

Notes About Connection Context Constructors:

- It is important to note that connection context class constructors, unlike the `Oracle.connect()` method, require an auto-commit setting.
- To use any of the first three constructors 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 4-5.
- Connection context classes that you declare generally have the same constructor signatures as the `DefaultContext` class. However, if you declare a connection context class to be associated with a data source, a different set of constructors is provided. (See ["Standard Data Source Support"](#) on page 7-13 for more information.)
- When using the constructor that takes a JDBC connection object, do not initialize the connection context instance with a null JDBC connection.
- The auto-commit setting determines whether SQL operations are automatically committed. For more information, see ["Basic Transaction Control"](#) on page 4-26.

Optional `DefaultContext` `close()` Method Parameters

When you close a connection context instance (of the `DefaultContext` class or any other class), you have the option of specifying whether or not to close the underlying physical connection. By default it is closed. This is relevant if you are sharing the physical connection between multiple connection objects, either SQLJ connection context instances or JDBC connection instances. The following examples presume a `DefaultContext` instance `defctx`.

To keep the underlying physical connection open:

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

To close the underlying physical connection (default behavior):

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

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

For more information about using these parameters and about shared connections, see "[Closing Shared Connections](#)" on page 7-57.

Connection for Translation

If you want to use online semantics-checking during translation, you must specify a database connection for SQLJ to use—these are referred to as *exemplar schemas* and are further discussed in "[Connection Context Concepts](#)" on page 7-2

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

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

For information about these options, see "[Connection Options](#)" on page 8-34.

Connection for Customization

Generally speaking, Oracle customization does not require a database connection; however, Oracle SQLJ does support customizer connections. This is useful in two circumstances:

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

For information about the Oracle customizer `optcols` option (for ISO standard code generation), see "[Oracle Customizer Column Definition Option \(optcols\)](#)" on page A-27. (For Oracle-specific code generation, the SQLJ translator has an `-optcols` option with the same functionality.)

The `SQLCheckerCustomizer` is invoked through the Oracle customizer harness `verify` option. See "[SQLCheckerCustomizer for Profile Semantics-Checking](#)" on page A-40.

Use the customizer harness `user`, `password`, `url`, and `driver` options to specify connection parameters for whatever customizer you are using, as appropriate. See "[Customizer Harness Options for Connections](#)" on page A-18.

Null-Handling

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

Wrapper Classes for Null-Handling

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

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

To avoid the possibility of null 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`

In case you must convert back to a primitive value, each of these wrapper classes has an `xxxValue()` method. For example, `intValue()` returns an `int` value from an `Integer` object and `floatValue()` returns a `float` value from a `Float` object. Do this as in the following example, presuming `intobj` is an `Integer` object:

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

Notes:

- `SQLException` is a subclass of the standard `java.sql.SQLException` class. See ["Using SQLException Subclasses"](#) on page 4-25.
 - 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.

Example:

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

#sql { UPDATE emp SET comm = :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 `EMP` table whose salary is at least \$50,000, the employee name (`ENAME`) and commission (`COMM`) are selected into the iterator. Then each row is tested to determine if the `COMM` field is, in fact, null. If so, it is processed accordingly.

Presume the following declaration:

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

Example:

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

while (ei.next())
```

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

Note: To execute a WHERE-clause comparison against null values, use the following SQL syntax:

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

Exception-Handling Basics

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

SQLJ and JDBC Exception-Handling Requirements

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

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

Example: Exception Handling This example demonstrates the kind of basic exception-handling 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.* ;
import oracle.sqlj.runtime.Oracle;

// iterator for the select

#sql iterator MyIter (String ITEM_NAME);

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

Errors originating in the SQLJ runtime are listed in ["SQLJ Runtime Messages"](#) on page B-47.

Errors originating in the Oracle JDBC driver are listed in the *Oracle9i JDBC Developer's Guide and Reference*. Errors originating in the Oracle RDBMS are listed in the *Oracle9i Database Error Messages* reference.

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.

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

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

Retrieving SQL States and Error Codes

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

- `String getMessage()`

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

- `int getErrorCode()`

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

- `String getSQLState()`

If the error originates in the SQLJ runtime, this method returns a string with a five-digit code indicating the SQL state. If the error originates in the JDBC driver, it returns no meaningful information. If the error originates in the RDBMS, it returns the five-digit SQL state. Your code should be prepared to handle a null return.

The following example 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);
}
...
```

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.)

For batch-enabled environments, there is also the standard `java.sql.BatchUpdateException` subclass. See ["Error Conditions During Batch Execution"](#) on page 10-22 for further discussion.

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 data updates.

For information about SQLJ support for more advanced transaction control functions—access mode and isolation level—see "[Advanced Transaction Control](#)" on page 7-49.

Overview of Transactions

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

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

A transaction ends with a COMMIT or ROLLBACK operation.

Note: In Oracle9i, all DDL commands (such as CREATE and ALTER) include an implicit COMMIT. This will commit not only the DDL command, but any preceding DML commands (INSERT, DELETE, UPDATE) that had not yet been committed or rolled back.

Automatic Commits Versus Manual Commits

In using SQLJ or JDBC, you can either have your data updates automatically committed, or commit them manually. In either case, each COMMIT operation starts a new transaction. You can specify that changes be committed automatically by enabling the auto-commit flag, 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. In addition, some SQLJ or JDBC features are incompatible with auto-commit mode. For example, you must disable the auto-commit flag for update batching or SELECT FOR UPDATE syntax to work properly.

Specifying Auto-Commit as You Define a Connection

When you use the `Oracle.connect()` or `Oracle.getConnection()` method to create a `DefaultContext` instance and define a connection, the auto-commit flag is set to `false` by default. 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 mode:

```
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 4-12.

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"](#) below.

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

Modifying Auto-Commit in an Existing Connection

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

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

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

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

or:

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

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

Using Manual COMMIT and ROLLBACK

If you disable the auto-commit flag, then you must manually commit any data updates.

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

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

To roll back (cancel) any changes that have been executed since the last `COMMIT` operation, 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:

- You can also roll back to a specified savepoint. See ["Using Savepoints"](#) on page 4-30.
 - All DDL statements in Oracle SQL include an implicit `COMMIT` operation. There is no special SQLJ functionality in this regard; such statements follow standard Oracle SQL rules.
 - If auto-commit mode is off and you close a connection context instance from a client application, then any changes since your last `COMMIT` will be committed, unless you close the connection context instance with `KEEP_CONNECTION` (explained in ["Closing Shared Connections"](#) on page 7-57).
-

Effect of Commits and Rollbacks on Iterators and Result Sets

`COMMIT` operations (either automatic or manual) and `ROLLBACK` operations do not affect open result sets and iterators. The result sets and iterators will still be open, and usually all that is relevant to their content is the state of the database at the time of execution of the `SELECT` statements that populated them.

Note: An exception to this is if you declared an iterator class with `sensitivity=SENSITIVE`. In this case, changes to the underlying result set may be seen whenever the iterator is scrolled outside of its window size. For more information about scrollable iterators, see ["Scrollable Iterators"](#) on page 7-42. For more information about the underlying scrollable result sets, see the *Oracle9i JDBC Developer's Guide and Reference*

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

Consider a situation where you `SELECT`, then `UPDATE`, then `COMMIT`. A non-sensitive 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 non-sensitive result set or iterator populated by the `SELECT` will still contain the updated data, regardless of the subsequent `ROLLBACK`.

Using Savepoints

The JDBC 3.0 specification adds support for *savepoints*. A savepoint is a defined point in a transaction which you can roll back to, if desired, instead of rolling back the entire transaction. Oracle SQLJ and JDBC support savepoints as of Oracle9i release 2, for use in any JDK of version 1.1 or higher.

SQLJ supports the following statements for savepoints:

```
#sql { SET SAVEPOINT :savepoint };  
...  
#sql { ROLLBACK TO :savepoint };  
...  
#sql { RELEASE :savepoint };
```

The savepoint is the point in the transaction where the `SET SAVEPOINT` statement appears. The *savepoint* host expression specifies the name of the savepoint, as a Java string. Later you can roll back to a specified savepoint or release (remove) a savepoint.

Savepoints are saved into the SQLJ execution context, which has methods that parallel the functionality of the three statements above. See "[Savepoint Methods](#)" on page 7-31.

Because any `COMMIT` operation ends the transaction, this also releases all savepoints of the transaction. This includes manual `COMMIT` operations, automatic `COMMIT` operations, and DDL statements (which result in an automatic `COMMIT`).

Note: As of Oracle9i release 2, Oracle9i and Oracle9i JDBC do not support release-savepoint functionality.

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 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 2-12.

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

Register JDBC Drivers and Set Default Connection

Declare the `SimpleExample` class with a constructor that uses the static `Oracle.connect()` method to set the default connection. This also registers the Oracle JDBC drivers. 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, user name, and password from the `connect.properties` file. An example of this file is in the directory `[Oracle_Home]/sqlj/demo` and also in ["Set Up the Runtime Connection"](#) on page 2-14.

```
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 in ["Set Up Exception Handling"](#) 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 in ["Set Up Host Variables, Execute SQLJ Clause, Process Results"](#) below.

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

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

Set Up Host Variables, Execute SQLJ Clause, Process Results

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

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

Here is the code:

```
void runExample() throws SQLException {

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

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

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

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

This example declares `salary` and `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 `empname` 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 ol = new SimpleExample();
            ol.runExample();
        }
        catch (SQLException ex) {
            System.err.println("Error running the example: " + ex);
        }
    }

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

    void runExample() throws SQLException {

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



```

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

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

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

```

Set Up a Named Iterator

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, use Integer.)
    String ename,
    String job,
    Integer mgr,
    Date hiredate,
    Float sal,
    Float comm,
    int deptno);

```

Later, 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 EmpDemolApp {

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

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

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

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

        EmpRecs employees;

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

        // Print the result using the iterator.

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

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

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

Oracle-Specific Code Generation (No Profiles)

Throughout this manual there is general and standard discussion of the SQLJ runtime layer and SQLJ profiles. As of Oracle9i release 2, however, Oracle SQLJ by default generates Oracle-specific code with direct calls to Oracle JDBC, instead of generating ISO standard code that calls the SQLJ runtime for SQL operations, which in turn contains calls to Oracle JDBC. With Oracle-specific code generation, there are no profile files, and the role of the SQLJ runtime layer is greatly reduced during program execution.

Oracle-specific code supports all Oracle-specific extended features.

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

See "[Code Generation \(-codegen\)](#)" on page 8-52 for information about syntax for this option.

The remainder of this section covers the following topics:

- [Advantages and Disadvantages of Oracle-Specific Code Generation](#)
- [Environment Requirements for Oracle-Specific Code Generation](#)
- [Code Considerations and Limitations with Oracle-Specific Code Generation](#)
- [SQLJ Usage Changes with Oracle-Specific Code Generation](#)
- [Server-Side Considerations with Oracle-Specific Code Generation](#)

Advantages and Disadvantages of Oracle-Specific Code Generation

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

- Applications run more efficiently. The code calls JDBC APIs directly, placing runtime performance directly at the JDBC level. The role of the intermediate SQLJ runtime layer is greatly reduced during program execution.
- Applications are smaller in size.
- No profile files (`.ser`) are produced. This is especially convenient if you are loading a translated application into the database or porting it to another system—there are fewer components.
- Translation is faster, because there is no profile customization step.

- During runtime, Oracle SQLJ and Oracle JDBC use the same statement cache resources, so partitioning resources between the two is unnecessary.
- Having the SQL-specific information appear in the Java class files instead of in separate profile files avoids potential security issues.
- You will not have to rewrite your code to take advantage of possible future Oracle JDBC performance enhancements, such as enhancements being considered for execution of static SQL code. Future releases of the Oracle SQLJ translator will handle this automatically.
- The use of Java reflection at runtime is eliminated, thereby providing full portability to browser environments.

There are relatively few disadvantages:

- Oracle-specific generated code does not adhere to SQLJ standards and is not portable to generic JDBC platforms.
- Profile-specific functionality is not available. For example, you cannot perform customizations at a later date to use the Oracle customizer harness `-debug`, `-verify`, and `-print` options. (These options are described in "[Customizer Harness Options that Invoke Specialized Customizers](#)" on page A-21. The `AuditorInstaller` invoked by the `-debug` option is described in "[AuditorInstaller Customizer for Debugging](#)" on page A-44.)

Environment Requirements for Oracle-Specific Code Generation

Be aware of the following requirements of your environment if you use Oracle-specific code generation:

- You must use an Oracle9i JDBC driver, because Oracle-specific code generation requires JDBC statement caching functionality. None of the Oracle8i (or prior) JDBC releases will work.
- The generic SQLJ runtime libraries, `runtime` and `runtime-nonoracle`, are not supported for Oracle-specific code generation. You must have one of the following Oracle SQLJ runtime libraries in your classpath:
 - `runtime11.jar` (or `.zip`)
 - `runtime12.jar` (or `.zip`)
 - `runtime12ee.jar` (or `.zip`)

These runtime libraries are further discussed in "[Requirements for Using Oracle SQLJ](#)" on page 2-3.

Code Considerations and Limitations with Oracle-Specific Code Generation

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

- To use a nondefault statement cache size, you must include appropriate method calls in your code, because the Oracle customizer `stmtcache` option is unavailable. See ["SQLJ Usage Changes with Oracle-Specific Code Generation"](#) on page 4-42.
- Do not mix Oracle-specific generated code with ISO standard generated code in the same application.

However, if Oracle-specific code and ISO standard code *must* share the same connection, do one of the following:

- Ensure that the Oracle-specific code and ISO standard code use different SQLJ execution context instances. (See ["Execution Contexts"](#) on page 7-24 for information about SQLJ execution contexts.)

or:

- Place a transaction boundary—a manual `COMMIT` or `ROLLBACK` statement—between the two kinds of code.

This limitation regarding mixing code is especially significant for server-side code, because all Java code running in a given session uses the same JDBC connection and SQLJ connection context. (Also see ["Server-Side Considerations with Oracle-Specific Code Generation"](#) on page 4-44.)

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

For example, avoid statements such as the following:

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

or:

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

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

Side effects are discussed, and examples shown, in ["Evaluation of Java Expressions at Runtime"](#) on page 3-22 and ["Examples of Evaluation of Java Expressions at Runtime \(ISO Code Generation\)"](#) on page 3-24.

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

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

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

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

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

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

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

Type maps are discussed in ["Custom Java Class Requirements"](#) on page 6-11. For general information about `with` clauses, see ["Declaration WITH Clause"](#) on page 3-6.

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

SQLJ Usage Changes with Oracle-Specific Code Generation

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

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

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

and the following instance methods:

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

By default, statement caching is enabled.

See "[Connection Context Methods for Statement Caching \(Oracle-Specific Code\)](#)" on page 10-5 for more information. (This is a subsection under "[Statement Caching](#)" on page 10-4, which provides an overview of statement caching.)

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

- `-optcols`—Enable iterator column type and size definitions to optimize performance.
- `-optparams`—Enable parameter size definitions to optimize JDBC resource allocation (used in conjunction with `optparamdefaults`).
- `-optparamdefaults`—Set parameter size defaults for particular datatypes (used in conjunction with `optparams`).
- `-fixedchar`—Enable CHAR comparisons with blank padding for WHERE clauses.

See "[Options for Code Generation, Optimizations, and CHAR Comparisons](#)" on page 8-51 for more information about these options.

Be aware of the following:

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

Server-Side Considerations with Oracle-Specific Code Generation

Note the following considerations if your SQLJ code will run in the server:

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

Therefore, to use ISO standard generated code in the server, you must translate and compile the SQLJ code on a client and then load the individual components into the server. (See ["Translating SQLJ Source on a Client and Loading Components"](#) on page 11-9.)

- The caution against mixing Oracle-specific generated code with ISO standard generated code (described in ["Code Considerations and Limitations with Oracle-Specific Code Generation"](#) on page 4-41) applies to server-side Java code that calls a Java stored procedure or stored function, even if the stored procedure is invoked through a PL/SQL wrapper. This constitutes a recursive call-in—by default, the `ExecutionContext` object is shared by both the calling module and the called module. Therefore, both modules should be translated with the same `-codegen` setting.

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

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

Important: To avoid resource leakage when using an explicit `ExecutionContext` instance, be sure to use the `close()` method, as shown in this example.

Requirements and Restrictions for Naming

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

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

Java Namespace: Local Variable and Class Naming Restrictions

The *Java namespace* applies to all 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 3-33.

Local Variable Naming Restrictions

Some of the functionality of the SQLJ translator results in minor restrictions in naming local variables.

The SQLJ translator replaces each SQLJ executable statement with a statement block, where the SQLJ executable statement is of the standard syntax:

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

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

```
__sJT_
```

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

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

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

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

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

Class Naming Restrictions

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

- You must not declare class names that may conflict with SQLJ internal classes. In particular, a top-level class cannot have a name of the following form 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`, then SQLJ generates a profile-keys class called `Foo_SJProfileKeys`. Do not declare a class name that conflicts with this.

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

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

SQLJ Namespace

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

Note: Restrictions particular to the naming of iterator columns are discussed in "[Using Named Iterators](#)" on page 3-43.

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 inside the curly braces. Normal SQL naming restrictions apply here. See the *Oracle9i SQL Reference* for more information.

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

File Name Requirements and Restrictions

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

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

```
MySource.sqlj
```

Note: These file naming requirements follow the Java Language Specification and are not SQLJ-specific. These requirements do not directly apply in Oracle9i, but it is still advisable to adhere to them.

Considerations for SQLJ in the Middle Tier

There are special considerations if you run SQLJ in the middle tier, such as in an Oracle9iAS Containers for J2EE (OC4J) environment.

With release 9.0.1 and later, the Oracle JDBC drivers provide Oracle-specific interfaces in the `oracle.jdbc` package. The Oracle SQLJ libraries `runtime11`, `runtime12`, and `runtime12ee` make full use of these interfaces. This is the reason why these libraries are not compatible with Oracle JDBC releases 8.1.7 and prior.

In the Oracle9iAS product, connections are established through data sources, which typically return instances of the `oracle.jdbc.OracleConnection` interface instead of the older `oracle.jdbc.driver.OracleConnection` class. This is necessary for certain connection functionality, such as distributed transactions (XA). To support such features, connection objects must implement the new interface.

This has the following consequences, relevant in an Oracle9iAS middle-tier environment, or any situation where data sources are used:

- For maximum portability and flexibility of your code, use `oracle.jdbc.OracleXXX` types instead of `oracle.jdbc.driver.OracleXXX` types.
- For custom Java types (typically for SQL objects and collections), implement `oracle.sql.ORAData` instead of the deprecated `oracle.sql.CustomDatum` interface.
- Do not use the SQLJ runtime library. Use `runtime11`, `runtime12`, or `runtime12ee` instead (depending on your JDK environment). The runtime library is backward compatible with older JDBC drivers, such as release 8.1.7, so supports the `oracle.jdbc.driver.OracleXXX` types, not the `oracle.jdbc.OracleXXX` types.

However, if you must use the runtime library for some reason, then set the option `-profile=false` during translation. In this case, your program will not use Oracle-specific customization and therefore will not fail if passed an `oracle.jdbc.OracleConnection` instance instead of an `oracle.jdbc.driver.OracleConnection` instance. In this circumstance, Oracle-specific features will *not* be supported.

To facilitate management of connections obtained through data sources and connection JavaBeans (for SQLJ JavaServer Pages), Oracle9i SQLJ provides a number of APIs in the `runtime12ee` library.

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

- ["Standard Data Source Support"](#) on page 7-13
- ["SQLJ-Specific Data Sources"](#) on page 7-16
- ["SQLJ-Specific Connection JavaBeans for JavaServer Pages"](#) on page 7-20

Type Support

This chapter documents datatypes supported by Oracle SQLJ, listing supported SQL types and the Java types that correspond to them, including information about backward 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, Collections, and OPAQUE Types"](#).

This chapter covers the following topics:

- [Supported Types for Host Expressions](#)
- [Support for Streams](#)
- [Support for JDBC 2.0 LOB Types and Oracle Type Extensions](#)

Supported Types for Host Expressions

This section summarizes the types supported by Oracle SQLJ, including information about new support for JDBC 2.0 types, and backward compatibility for the 8.0.x and 7.3.x Oracle JDBC drivers.

For a complete list of legal Java mappings for each Oracle SQL type, see the reference information in the *Oracle9i JDBC Developer's Guide and Reference*.

Note: SQLJ (and SQL) perform implicit conversions between SQL and Java types. Although this is generally useful and helpful, it can produce unexpected results. Do not rely on translation-time type-checking alone to ensure the correctness of your code.

Summary of Supported Types

[Table 5-1](#) lists the Java types that you can use in host expressions when employing the Oracle JDBC drivers. This table also documents the correlation between Java types, SQL types whose *typecodes* are defined in the class `oracle.jdbc.OracleTypes`, and datatypes in Oracle9i.

Note: The `OracleTypes` class simply defines a typecode, which is an integer constant, for each Oracle datatype. For standard JDBC types, the `OracleTypes` value is identical to the standard `java.sql.Types` value.

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

Table 5–1 Type Mappings for Supported Host Expression Types

Java Type	OracleTypes Definition	Oracle SQL Datatype
STANDARD JDBC 1.x TYPES		
boolean	BIT	NUMBER
byte	TINYINT	NUMBER
short	SMALLINT	NUMBER
int	INTEGER	NUMBER
long	BIGINT	NUMBER
float	REAL	NUMBER
double	FLOAT, DOUBLE	NUMBER
java.lang.String	CHAR VARCHAR LONGVARCHAR	CHAR VARCHAR2 LONG
byte[]	BINARY VARBINARY LONGVARBINARY	RAW RAW LONGRAW
java.sql.Date	DATE	DATE
java.sql.Time	TIME	DATE
java.sql.Timestamp	TIMESTAMP TIMESTAMP	DATE TIMESTAMP
java.math.BigDecimal	NUMERIC DECIMAL	NUMBER NUMBER
STANDARD JDBC 2.0 TYPES		
java.sql.Blob	BLOB	BLOB
java.sql.Clob	CLOB	CLOB
java.sql.Struct	STRUCT	object types
java.sql.Ref	REF	reference types
java.sql.Array	ARRAY	collection types
custom object classes implementing java.sql.SQLData	STRUCT	object types
JAVA WRAPPER CLASSES		
java.lang.Boolean	BIT	NUMBER

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

Java Type	OracleTypes Definition	Oracle SQL Datatype
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.CharacterStream	LONGVARCHAR	LONG
sqlj.runtime.ASCIIStream (deprecated ; use CharacterStream)	LONGVARCHAR	LONG
sqlj.runtime.UnicodeStream (deprecated ; use CharacterStream)	LONGVARCHAR	LONG
ORACLE EXTENSIONS		
oracle.sql.NUMBER	NUMBER	NUMBER
oracle.sql.CHAR	CHAR	CHAR
oracle.sql.RAW	RAW	RAW
oracle.sql.DATE	DATE	DATE
oracle.sql.TIMESTAMP	TIMESTAMP	TIMESTAMP
oracle.sql.TIMESTAMPTZ	TIMESTAMPTZ	TIMESTAMP-WITH-TIMEZONE
oracle.sql.TIMESTAMPLTZ	TIMESTAMPLTZ	TIMESTAMP-WITH-LOCAL-TIMEZONE
oracle.sql.ROWID	ROWID	ROWID
oracle.sql.BLOB	BLOB	BLOB
oracle.sql.CLOB	CLOB	CLOB
oracle.sql.BFILE	BFILE	BFILE
oracle.sql.STRUCT	STRUCT	object types
oracle.sql.REF	REF	reference types

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

Java Type	OracleTypes Definition	Oracle SQL Datatype
oracle.sql.ARRAY	ARRAY	collection types
oracle.sql.OPAQUE	OPAQUE	OPAQUE types
custom object classes implementing oracle.sql.ORADData	STRUCT	object types
custom reference classes implementing oracle.sql.ORADData	REF	reference types
custom collection classes implementing oracle.sql.ORADData	ARRAY	collection types
custom classes implementing oracle.sql.ORADData for OPAQUE types (for example, oracle.xdb.XMLType)	OPAQUE	OPAQUE types
other custom Java classes implementing oracle.sql.ORADData (to wrap any oracle.sql type)	any	any
SQLJ object Java types (can implement either SQLData or ORADData)	JAVA_STRUCT	SQLJ object SQL types (JAVA_STRUCT behind the scenes; automatic conversion to an appropriate Java class)
JAVA TYPES FOR PL/SQL TYPES		
scalar indexed-by table (JDBC OCI driver only), represented by a Java numeric array or an array of <code>String</code> , <code>oracle.sql.CHAR</code> , or <code>oracle.sql.NUMBER</code>	n/a	n/a Note: There is a <code>PLSQL_INDEX_TABLE</code> type, but it does not appear to be used externally.
GLOBALIZATION SUPPORT		
oracle.sql.NCHAR	CHAR	CHAR
oracle.sql.NString	CHAR VARCHAR LONGVARCHAR	CHAR VARCHAR2 LONG
oracle.sql.NCLOB	CLOB	CLOB
oracle.sqlj.runtime.NcharCharacterStream	LONGVARCHAR	LONG
oracle.sqlj.runtime.NcharAsciiStream (deprecated ; use <code>NcharCharacterStream</code>)	LONGVARCHAR	LONG

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

Java Type	OracleTypes Definition	Oracle SQL Datatype
oracle.sql.runtime.NcharUnicodeStream (deprecated ; use NcharCharacterStream)	LONGVARCHAR	LONG
QUERY RESULT OBJECTS		
java.sql.ResultSet	CURSOR	CURSOR
SQLJ iterator objects	CURSOR	CURSOR

You can refer to the *Oracle9i JDBC Developer's Guide and Reference* for more information about Oracle type support.

The following points relate to type support for standard features:

- JDBC and SQLJ do not support Java `char` and `Character` types. Instead, use the Java `String` type to represent character data.
- Do not confuse the supported `java.sql.Date` type with `java.util.Date`, which is not directly supported. The `java.sql.Date` class is a wrapper for `java.util.Date` that allows JDBC to identify the data as a SQL `DATE` and adds formatting and parsing operations to support JDBC escape syntax for date values.
- Remember that all numeric types in Oracle9i are stored as `NUMBER`. Although you can specify additional precision when you declare a `NUMBER` during table creation (by declaring 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` instance 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 4-19 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 3-52.
- The SQLJ stream classes are required in using streams as host variables. For information, see "[Support for Streams](#)" on page 5-14.

- Weak types cannot be used for `OUT` or `INOUT` parameters. This applies to the `Struct`, `Ref`, and `Array` standard JDBC 2.0 types, as well as to corresponding Oracle extended types.
- A new set of interfaces, in the `oracle.jdbc` package, is added in Oracle 9i JDBC in place of classes of the `oracle.jdbc.driver` package. These new interfaces provide a more generic way for users to access Oracle-specific features using Oracle JDBC drivers. Specifically, when creating programs for the middle tier, you should use the new API. The Oracle 8i API will continue to be supported for backward compatibility, so no change is required for existing JDBC code to upgrade from Oracle 8i to Oracle 9i. (SQLJ programmers, however, will not typically use these interfaces directly. They are used transparently by the SQLJ runtime or in Oracle-specific generated code.)

For more information, see ["Custom Java Class Interface Specifications"](#) on page 6-6.

The following points relate to Oracle extensions, most of which are covered in ["Support for JDBC 2.0 LOB Types and Oracle Type Extensions"](#) on page 5-29 and in Chapter 6, ["Objects, Collections, and OPAQUE Types"](#):

- Oracle SQLJ requires any class that implements `oracle.sql.ORAData` to set the public static `_SQL_TYPECODE` parameter according to values defined in the `OracleTypes` class. In some cases, an additional parameter must be set as well, such as `_SQL_NAME` for objects and `_SQL_BASETYPE` for object references. This occurs automatically if you use the Oracle JPublisher utility to generate the class.

See ["Oracle Requirements for Classes Implementing ORAData"](#) on page 6-11.

- 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 *Oracle9i JDBC Developer's Guide and Reference*.
- Custom Java classes can map to Oracle objects (implementing `ORAData` or `SQLData`), references (implementing `ORAData` only), collections (implementing `ORAData` only), `OPAQUE` types (implementing `ORAData` only), or other SQL types (for customized handling, implementing `ORAData` only). See ["Custom Java Classes"](#) on page 6-6.

You can use the Oracle JPublisher utility to automatically generate custom Java classes. See ["JPublisher and the Creation of Custom Java Classes"](#) on page 6-28.

- Oracle SQLJ has functionality for automatic blank padding when comparing a string to a `CHAR` column value for a `WHERE` clause. Otherwise the string would

have to be padded to match the number of characters in the database column. This is available as a SQLJ translator option for Oracle-specific code generation, or as an Oracle customizer option for ISO standard code generation. See "[CHAR Comparisons with Blank Padding \(-fixedchar\)](#)" on page 8-58 and "[Oracle Customizer CHAR Comparisons with Blank Padding \(fixedchar\)](#)" on page A-32.

- Weak types cannot be used for `OUT` or `INOUT` parameters. This applies to the `STRUCT`, `REF`, and `ARRAY` Oracle extended types and corresponding standard JDBC 2.0 types, as well as to Oracle `OPAQUE` types.
- Using any of the Oracle extensions requires the following:
 - an Oracle JDBC driver
 - Oracle-specific code generation or Oracle customization during translation
 - the Oracle SQLJ runtime when your application runs

Supported Types and Requirements for JDBC 2.0

As indicated in [Table 5–1](#) above, Oracle JDBC and SQLJ support JDBC 2.0 types in the standard `java.sql` package.

This section lists JDBC 2.0 supported types and related Oracle extensions.

Important: In a Sun Microsystems JDK environment, JDBC 2.0 types require a JDK 1.2.x or higher version. While Oracle JDBC under JDK 1.1.x supports `oracle.jdbc2` extensions to mimic JDBC 2.0 type functionality, Oracle SQLJ has never supported the `oracle.jdbc2` package.

To use JDBC 2.0 types or corresponding Oracle extended types in Oracle SQLJ, use the `SQLJruntime12` or `runtime12ee` library, which support JDK 1.2.x or higher.

[Table 5–2](#) lists the JDBC 2.0 types supported by Oracle SQLJ. You can use them wherever you can use the corresponding Oracle extensions, summarized in the table.

The Oracle extensions have been available in prior releases and are still available as well. These `oracle.sql.*` classes provide functionality to wrap raw SQL data, and are described in the *Oracle9i JDBC Developer's Guide and Reference*.

Table 5–2 Correlation between Oracle Extensions and JDBC 2.0 Types

JDBC 2.0 Type	Oracle Extension
java.sql.Blob	oracle.sql.BLOB
java.sql.Clob	oracle.sql.CLOB
java.sql.Struct	oracle.sql.STRUCT
java.sql.Ref	oracle.sql.REF
java.sql.Array	oracle.sql.ARRAY
java.sql.SQLData	n/a
n/a	oracle.sql.ORAData (<code>_SQL_TYPECODE = OracleTypes.STRUCT</code>)

ORAData functionality is an Oracle-specific alternative to standard SQLData functionality for Java support of user-defined types. For information, see ["Custom Java Classes"](#) on page 6-6.

For information about support for other types in [Table 5–2](#), see ["Support for BLOB, CLOB, and BFILE"](#) on page 5-30 and ["Support for Weakly Typed Objects, References, and Collections"](#) on page 6-80.

The following JDBC 2.0 types are currently *not* supported in Oracle JDBC or SQLJ:

- JAVA_OBJECT—Represents an instance of a Java type in a SQL column.
- DISTINCT—A distinct SQL type represented in or retrievable from a basic SQL type (for example, SHOESIZE --> NUMBER).

Using PL/SQL BOOLEAN, RECORD Types, and TABLE Types

Oracle SQLJ and JDBC do not support calling arguments or return values of the PL/SQL BOOLEAN type or RECORD types. Also, when using the Thin driver, they do not support calling arguments or return values of PL/SQL TABLE types (known as *indexed-by tables*). TABLE types are supported for the OCI driver, however.

Support for TABLE Types (with OCI driver only)

The Oracle JDBC OCI driver has supported scalar PL/SQL indexed-by tables since Oracle8i release 8.1.7. For details about the JDBC support, see the *Oracle9i JDBC Developer's Guide and Reference*.

Oracle9i SQLJ simplifies the process of writing and retrieving data in scalar indexed-by tables. The following array types are supported:

- **numeric types**—`int[]`, `long[]`, `float[]`, `double[]`, `short[]`, `java.math.BigDecimal[]`, `oracle.sql.NUMBER[]`
- **character types**—`java.lang.String[]`, `oracle.sql.CHAR[]`

Here is an example of writing indexed-by table data to the database:

```
int[] vals = {1,2,3};
#sql { call procin(:vals) };
```

Here is an example of retrieving indexed-by table data from the database:

```
oracle.sql.CHAR[] outvals;
#sql { call procout(:OUT outvals/*[111](22)*/) };
```

You must specify the maximum length of the output array being retrieved, using `[xxx]` syntax inside `/* . . . */` syntax as shown. Also, for character-like binds, you can optionally include `(xx)` syntax, as shown, to specify the maximum length (in bytes) of an array element.

Note: The `oracle.sql.Datum` class is not supported directly. You must use an appropriate subclass, such as `oracle.sql.CHAR` or `oracle.sql.NUMBER`.

Workarounds for Non-Supported Types

As a workaround for an unsupported type, you can create wrapper procedures that process the data using supported types. For example, to wrap a stored procedure that uses PL/SQL boolean values, you can create a stored procedure that takes a character or number from JDBC and passes it to the original procedure as `BOOLEAN`, or, for an output parameter, accepts a `BOOLEAN` argument from the original procedure and passes it as a `CHAR` or `NUMBER` to JDBC. Similarly, to wrap a stored procedure that uses PL/SQL records, you can create a stored procedure that handles a record in its individual components (such as `CHAR` and `NUMBER`). To wrap a stored procedure that uses PL/SQL `TABLE` types, you can break the data into components or perhaps use Oracle collection types.

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;
```

Note: When using these non-supported PL/SQL types in method signatures in PL/SQL packages or SQL objects, consider using the Oracle9i JPublisher utility. This facilitates the creation of Java types to call such methods. See "[JPublisher and the Creation of Custom Java Classes](#)" on page 6-28 for an overview of JPublisher, and the *Oracle9i JPublisher User's Guide* for more information.

Backward Compatibility for Previous Oracle JDBC Releases

This section summarizes backward compatibility issues when using Oracle SQLJ with previous Oracle JDBC releases.

Note: Oracle9i release 2 adds support for `OPAQUE` types and `TIMESTAMP` types.

Backward Compatibility for Oracle8i

The following Oracle9i features are not supported, or are supported differently, in the Oracle8i JDBC drivers:

- `oracle.sql.ORADATA` and `ORADATAFACTORY` interfaces for Java mapping of user-defined SQL types

Use the Oracle8i `oracle.sql.CustomDatum` and `CustomDatumFactory` interfaces instead. See "[ORADATA Versus CustomDatum Interfaces](#)" on page 6-8.

- Oracle extensions for character types for globalization support—NCHAR, NCLOB, NString, and NcharCharacterStream (or NcharAsciiStream and NcharUnicodeStream in previous releases)

Backward Compatibility for Oracle 8.0.x and 7.3.x

Some of the Oracle type extensions supported by the Oracle9i JDBC drivers are either not supported or supported differently by the Oracle 8.0.x and 7.3.x JDBC drivers. Following are the key points:

- The Oracle 8.0.x and 7.3.x 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 Oracle 8.0.x and 7.3.x drivers do not support Oracle object and collection types.
- The Oracle 8.0.x and 7.3.x drivers support the Oracle ROWID datatype with the `OracleRowid` class in the `oracle.jdbc` package.
- The Oracle 8.0.x drivers support the Oracle BLOB, CLOB, and BFILE datatypes with the `OracleBlob`, `OracleClob`, and `OracleBfile` classes in the `oracle.jdbc` package. These classes do not include LOB and BFILE manipulation methods such as those discussed in "Support for BLOB, CLOB, and BFILE" on page 5-30. You must, instead, use the PL/SQL DBMS_LOB package, which is discussed in the same section.
- The Oracle 7.3.x drivers do not support BLOB, CLOB, and BFILE.

Table 5-3 summarizes these differences.

Table 5-3 Type Support Differences for Oracle 8.0.x and 7.3.x JDBC Drivers

Java Type (Oracle Extensions)	Oracle Types Definition	Oracle Datatype
<code>oracle.sql.NUMBER</code>	not supported	n/a
<code>oracle.sql.CHAR</code>	not supported	n/a
<code>oracle.sql.RAW</code>	not supported	n/a
<code>oracle.sql.DATE</code>	not supported	n/a
<code>oracle.jdbc.OracleRowid</code>	ROWID	ROWID
<code>oracle.jdbc.OracleBlob</code>	BLOB in 8.0.x not supported in 7.3.x	BLOB in 8.0.x n/a in 7.3.x

Table 5–3 Type Support Differences for Oracle 8.0.x and 7.3.x JDBC Drivers (Cont.)

Java Type (Oracle Extensions)	Oracle Types Definition	Oracle Datatype
oracle.jdbc.OracleClob	CLOB in 8.0.x not supported in 7.3.x	CLOB in 8.0.x n/a in 7.3.x
oracle.jdbc.OracleBfile	BFILE in 8.0.x not supported in 7.3.x	BFILE in 8.0.x n/a in 7.3.x
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 <code>oracle.sql</code> types, including objects, references, and collections)	not supported	n/a

Support for Streams

Standard SQLJ provides two specialized classes, listed below, for convenient processing 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 processing and transferring large data items in manageable chunks.

- `sqlj.runtime.BinaryStream`
- `sqlj.runtime.CharacterStream`

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

Note: As of JDBC 2.0, the `CharacterStream` class replaces the `AsciiStream` and `UnicodeStream` classes. `CharacterStream` shelters users from unnecessary logistics regarding encoding. The `AsciiStream` and `UnicodeStream` classes are still supported for backward compatibility, but are deprecated.

General Use of SQLJ Streams

With respect to Oracle9i, [Table 5-1](#) on page 5-3 lists the datatypes you would typically process using these stream classes. To summarize:

- `BinaryStream` is typically used for datatype `LONG RAW` (`Types.LONGVARBINARY`), but might also be used for datatype `RAW` (`Types.BINARY` or `Types.VARBINARY`).
- `CharacterStream` is typically used for datatype `LONG` (`java.sql.Types.LONGVARCHAR`), but might also be used for datatype `VARCHAR2` (`Types.VARCHAR`).

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). Oracle SQLJ and JDBC support large object types—see "[Support for BLOB, CLOB, and BFILE](#)" on page 5-30.

Both SQLJ stream classes are subclasses of standard Java classes, `java.io.InputStream` for `BinaryStream` and `java.io.Reader` for `CharacterStream`, and act as wrappers to provide the functionality required by SQLJ. This functionality is to communicate to SQLJ the type and length of the underlying data so that it can be processed and formatted properly.

You can use the SQLJ stream types for host variables to either send or retrieve data.

Key Aspects of Stream Support Classes

The following abbreviated code illustrates key aspects of the `BinaryStream` class—what it extends, constructor signatures, and key method signatures:

```
public class sqlj.runtime.BinaryStream extends sqlj.runtime.StreamWrapper
{   public sqlj.runtime.BinaryStream(java.io.InputStream);
    public sqlj.runtime.BinaryStream(java.io.InputStream, int);
    public java.io.InputStream getInputStream();
    public int getLength();
    public void setLength(int);
}
```

And the following abbreviated code illustrates key aspects of the `CharacterStream` class:

```
public class sqlj.runtime.CharacterStream extends java.io.FilterReader
{   public sqlj.runtime.CharacterStream(java.io.Reader);
    public sqlj.runtime.CharacterStream(java.io.Reader, int);
    public int getLength();
    public java.io.Reader getReader();
    public void setLength(int);
}
```

Constructor `int` parameters are for data length, in bytes or characters as applicable.

Notes:

- For any method that takes a `java.io.InputStream` object as input, you can use a `BinaryStream` object instead. Similarly, for any method that takes a `java.io.Reader` object as input, you can use a `CharacterStream` object instead.
 - The deprecated `AsciiStream` and `UnicodeStream` classes have the same key aspects and signatures as `BinaryStream`.
-
-

Using SQLJ Streams to Send Data

Standard SQLJ 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 an appropriate standard Java data object for input. For `BinaryStream` this would be an input stream—an instance of `java.io.InputStream` or some subclass. For `CharacterStream` this would be a reader object—an instance of `java.io.Reader` or some subclass.
3. Create an instance of the appropriate SQLJ stream class (depending on the type of data), passing the data object and length (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.)

The following subsections now go 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 a character file to update a `LONG` column.
- using a byte array to update a `LONG RAW` column

Updating `LONG` or `LONG RAW` from a File

This section shows how to create a `CharacterStream` object or a `BinaryStream` object from a `File` object and use it to update the database. The code example at the end uses a `CharacterStream` for a `LONG` column.

In updating a database column (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.

- 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.

- For character data, create a `java.io.FileReader` object from your `File` object. You can pass the `File` object to the `FileReader` constructor.
For binary data, create a `java.io.FileInputStream` object from your `File` object. You can pass the `File` object to the `FileInputStream` constructor.
- Create an appropriate SQLJ stream object. This would be a `CharacterStream` object for a character file or a `BinaryStream` object for a binary file. Pass the `FileReader` or `FileInputStream` object, as applicable, and the data length (as an `int`) to the SQLJ stream class constructor.
- 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 `chardata` 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.
FileReader filereader = new FileReader(myfile);
CharacterStream charstream = new CharacterStream(filereader, length);
#sql { INSERT INTO filetable (chardata) VALUES (:charstream) };
charstream.close();
...
```

Updating LONG RAW from a Byte Array

This section and code example at the end shows how to create a `BinaryStream` object from a byte array and uses it to update the database.

You must determine the length of the data before updating the database from a byte array. (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 the `ByteArrayInputStream` class.

4. Use the SQLJ stream object as a host variable in an appropriate SQL operation in a SQLJ executable statement.

The following is an example of writing `LONG RAW` data to the database from a byte array. Presume you have a byte array `bytearray[]` and you want to insert its contents into a `LONG RAW` column 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, 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 Oracle9i and an Oracle JDBC driver, you must be careful in how you access and process the stream data.

As the Oracle JDBC drivers access data from an iterator row, they must flush any stream item from the communications pipe before accessing the next data item. Even though the stream data is written to a local stream 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 have only one stream column, and it must be the last column. As soon as you have fetched each row of the iterator, writing the stream item to a local input stream variable in the process, you must read and process the local stream variable before advancing to the next row of the iterator.
- When using a named iterator, you can have multiple stream columns; however, as you process each iterator row, each time you access a stream field, writing the

data to a local stream variable in the process, you must read and process the local stream immediately, before reading anything else from the iterator.

Furthermore, in processing each row of a named iterator, you must call the column accessor methods in the same order in which the database columns were selected in the query that populated the iterator. As mentioned in a 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: Oracle9i and the Oracle JDBC drivers do not support use of streams in `SELECT INTO` statements.

Using SQLJ Streams to Retrieve Data

To retrieve data 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, taking into account the precautions documented in "[Retrieving Data into Streams: Precautions](#)" on page 5-19.

These are general steps. For more information, see "[Stream Class Methods](#)" on page 5-22 and "[Examples of Retrieving and Processing Stream Data](#)" on page 5-24.

Using a SQLJ Stream Column in a Positional Iterator

Use the following steps to retrieve data into a SQLJ stream using a positional iterator:

1. Declare a positional iterator class with the last column being of the appropriate SQLJ stream type.
2. Declare a local variable of your iterator type.
3. Declare a local variable of the appropriate SQLJ stream type. This will be used as a host variable to receive data from each row of the SQLJ stream column of the iterator.
4. Execute a query to populate the iterator you declared in step 2.
5. Process the iterator as usual. (See "[Using Positional Iterators](#)" on page 3-48.) 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 or reader type for each SQLJ stream column in the iterator. These will be used to receive data from the stream-column accessor methods. These local stream variables do not have to be SQLJ stream types; they can be standard `java.io.InputStream` or `java.io.Reader` (as applicable) if desired. They do not have to be SQLJ stream types, because the data was already correctly formatted as a result of the iterator columns being of appropriate SQLJ stream types.
4. Execute a query to populate the iterator you declared in step 2.
5. Process the iterator as usual. (See "[Using Named Iterators](#)" on page 3-43.) 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, the `length` attribute of the stream will not be meaningful. This attribute is meaningful only when you set it explicitly, either using the `setLength()` method that each SQLJ stream class provides, or specifying the length to the constructor (as discussed in ["Using SQLJ Streams to Send Data"](#) on page 5-16).

Stream Class Methods

In processing a SQLJ stream column in a named or positional iterator, the local stream variable used to receive the stream data can be either a SQLJ stream type or the standard `java.io.InputStream` type or `java.io.Reader` type (as applicable). In either case, standard methods of the input data object are supported.

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

Binary Stream Methods

The `BinaryStream` class is a subclass of the `sqlj.runtime.StreamWrapper` class. The `StreamWrapper` class provides the following key methods:

- `InputStream getInputStream()`—You can optionally use this method to get the underlying `java.io.InputStream` object. 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.

The `length` attribute must be set to an appropriate value before you send a SQLJ stream to the database.

- `int getLength()`—This method returns the value of the `length` attribute of a SQLJ stream. This value is meaningful only if you explicitly set it using the stream object constructor or the `setLength()` method. When you retrieve data into a stream, the `length` attribute is not set automatically.

The `sqlj.runtime.StreamWrapper` class is a subclass of the `java.io.FilterInputStream` class, which is a subclass of the `java.io.InputStream` class. The following important methods of the `InputStream` class—the `skip()` method, `close()` method, and three forms of the `read()` method—are supported by the SQLJ `BinaryStream` class as well.

- `int read()`—Reads the next byte of data from the input stream. The byte of data is returned as an `int` value in the range 0 to 255. If the end of the stream has already been reached, then the value -1 is returned. This method blocks program execution until one of the following: 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 program execution 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.

Character Stream Methods

The `CharacterStream` class provides the following key methods:

- `Reader getReader()`—You can optionally use this method to get the underlying `java.io.Reader` object. This is not required, however, as you can also process SQLJ stream objects directly.
- `void setLength(int length)`
- `int getLength()`

Use these to set or get the length of the stream object. This is the same functionality as for binary streams—see "[Binary Stream Methods](#)" immediately above.

The `sqlj.runtime.CharacterStream` class is a subclass of the `java.io.FilterReader` class, which is a subclass of the `java.io.Reader` class. The following important methods of the `Reader` class—the `skip()` method, `close()` method, and three forms of the `read()` method—are supported by the SQLJ `CharacterStream` class as well.

- `int read ()`—Reads the next character of data from the reader. The data is returned as an `int` value in the range 0 to 65535. If the end of the data has already been reached, then the value -1 is returned. This method blocks program execution until one of the following: 1) input data is available; 2) the end of the data is detected; or 3) an exception is thrown.
- `int read (char cbuf[])`—Reads characters into an array, writing the data into the specified `cbuf[]` `char` array. It returns an `int` value indicating how many characters were read or -1 if the end of the data has already been reached. This method blocks program execution until input is available.
- `int read (char cbuf[], int off, int len)`—Reads up to `len` (length) characters of data from the input, starting at the character specified by the offset, `off`, and writing the data into the specified `char[]` `char` array. It returns an `int` value indicating how many characters were read or -1 if the end of the data has already been reached. This method blocks until input is available.
- `long skip (long n)`—Skips over and discards `n` characters of data from the input. In some circumstances, however, this method will actually skip a smaller number of characters. It returns a `long` value indicating the actual number of characters skipped.
- `void close()`—Closes the stream and releases any associated resources.

Examples of Retrieving and Processing Stream Data

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

- using a `SELECT` statement to select data from a `LONG` column and populate a `SQLJ CharacterStream` column in a named iterator
- 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 CharacterStream Column of Named Iterator This example selects data from a `LONG` database column, populating a `SQLJ CharacterStream` column in a named iterator.

Assume there is a table named `FILETABLE` with a `VARCHAR2` column called `FILENAME` that contains file names, and a `LONG` column called `FILECONTENTS` that contains file contents in character format.

Imports and declarations:

```
import sqlj.runtime.*;
import java.io.*;
...
#sql iterator MyNamedIter (String filename, CharacterStream filecontents);
```

Executable code:

```
MyNamedIter namediter = null;
String fname;
CharacterStream charstream;
#sql namediter = { SELECT filename, filecontents FROM filetable };
while (namediter.next()) {
    fname = namediter.filename();
    charstream = namediter.filecontents();
    System.out.println("Contents for file " + fname + ":");
    printStream(charstream);
    charstream.close();
}

namediter.close();
...
public void printStream(Reader in) throws IOException
{
    int character;
    while ((character = in.read()) != -1) {
        System.out.print((char)character);
    }
}
```

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

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

As explained in "[Retrieving Data into Streams: Precautions](#)" on page 5-19, there can be only one stream column in a positional iterator, and it must be the last column.

Assume there is a table named `BINTABLE` with a `NUMBER` column called `IDENTIFIER` and a `LONG RAW` column called `BINDATA` that contains binary data associated with the identifier.

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` and `CharacterStream` 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 use Oracle9i, an Oracle JDBC driver, Oracle-specific code generation or the Oracle customizer, and the Oracle SQLJ runtime:

- 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 `BinaryStream` and `CharacterStream` as the assignment type for a stored procedure or stored function `OUT` or `INOUT` parameter.

Assume the following table definition:

```
CREATE TABLE streamexample (name VARCHAR2 (256), data LONG);
INSERT INTO streamexample (data, name)
VALUES
  ('00000000000111111111122222222223333333333344444444445555555555',
   '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:

```
CharacterStream data;
#sql { CALL out_longdata('StreamExample', :OUT data) };
int c;
while ((c = data.read ()) != -1)
  System.out.print((char)c);
System.out.flush();
data.close();
...
```

Note: Closing the stream is recommended, but not required.

Streams as Stored Function Results

You can use the types `BinaryStream` and `CharacterStream` as the assignment type for a stored function return result.

Assume the same `STREAMEXAMPLE` table definition as in the preceding stored procedure example.

Also assume the following stored function definition, which uses the `STREAMEXAMPLE` table:

```
CREATE OR REPLACE FUNCTION get_longdata (dataname VARCHAR2) RETURN long
  IS longdata LONG;
BEGIN
  SELECT data INTO longdata FROM streamexample WHERE name = dataname;
  RETURN longdata;
END get_longdata;
```

The following sample code uses a call to the `get_longdata` stored function to read the long data.

Imports:

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

Executable code:

```
CharacterStream data;
#sql data = { VALUES(get_longdata('StreamExample')) };
int c;
while ((c = data.read ()) != -1)
  System.out.print((char)c);
System.out.flush();
data.close();
...
```

Note: Closing the stream is recommended, but not required.

Support for JDBC 2.0 LOB Types and Oracle Type Extensions

Oracle SQLJ offers extended functionality for the following JDBC 2.0 and Oracle-specific datatypes:

- JDBC 2.0 LOB datatypes (BLOB and CLOB)
- Oracle BFILE datatype
- Oracle ROWID datatype
- Oracle REF CURSOR datatypes
- other Oracle9i datatypes (such as NUMBER and RAW)

These datatypes are supported by classes in the `oracle.sql` package, discussed below. LOBs and BFILES are handled similarly in many ways, so are discussed together.

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

- `BigDecimal`

JDBC 2.0 functionality for user-defined SQL 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, Collections, and OPAQUE Types"](#).

Note that using Oracle extensions in your code requires the following:

- Use one of the Oracle JDBC drivers.
- Use Oracle-specific code generation (the default `-codegen=oracle` setting) or, for ISO code generation (`-codegen=iso`), customize the profiles appropriately. The default customizer, `oracle.sqlj.runtime.util.OraCustomizer`, is recommended.
- Use the Oracle SQLJ runtime when your application runs.

The Oracle SQLJ runtime and an Oracle JDBC driver are required whenever you use the Oracle customizer, even if you do not actually use Oracle extensions in your code.

For Oracle-specific semantics-checking, you must use an appropriate checker. The default checker, `oracle.sqlj.checker.OracleChecker`, acts as a front end and will run the appropriate checker based on your environment. This will be one of the Oracle-specific checkers if you are using an Oracle JDBC driver.

Oracle-specific types are defined in the `oracle.sql` package, discussed in ["Package oracle.sql"](#) below.

Package oracle.sql

SQLJ users, as well as JDBC users, should be aware of the `oracle.sql` package, which includes classes to support all the Oracle9i 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.

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 8-34 and ["Semantics-Checking and Offline-Parsing Options"](#) on page 8-71.

For more information about the `oracle.sql` classes, see the *Oracle9i JDBC Developer's Guide and Reference*.

Support for BLOB, CLOB, and BFILE

Oracle JDBC and SQLJ support JDBC 2.0 large object (LOB) datatypes—BLOB (binary LOB) and CLOB (character LOB)—and provide similar support for the Oracle-specific BFILE type (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 *Oracle9i 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 the `DBMS_LOB` package.

Using methods of the `BLOB`, `CLOB`, and `BFILE` classes in a Java application is more convenient than using the `DBMS_LOB` package and may also lead to faster execution in some cases.

Note that the type of the chunk being read or written depends on the kind of LOB being manipulated. For example, 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 an Oracle9i 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.

BFILE Class versus DBMS_LOB Functionality for BFILES

The following examples contrast use of the `oracle.sql` methods with use of the `DBMS_LOB` package for BFILES.

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 `BFILE`s 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 `BFILE`s can be manipulated only after being opened with a call to `DBMS_LOB.FILEOPEN` or equivalent method in the `BFILE` class.

BLOB and CLOB Classes versus DBMS_LOB Functionality for LOBs

The following examples contrast use of the `oracle.sql` methods with use of the `DBMS_LOB` package for `BLOB`s and `CLOB`s. 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.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 and BFILE Stored Function Results

Host variables of type BLOB, CLOB, and BFILE can be assigned to the result of a stored function call. The following example is for a CLOB, but code for BLOBs and BFILES would be functionally the same.

First, presume the following function definition:

```
CREATE OR REPLACE function longer_clob (c1 clob, c2 clob) return clob is
    result clob;
BEGIN
    if dbms_lob.getLength(c2) > dbms_lob.getLength(c1) then
        result := c2;
    else
        result := c1;
    end if;
    RETURN result;
```

```
END longer_clob;
```

The following example uses a CLOB as the assignment type for a return value from the 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 and BFILE 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. The following example is for a BLOB and CLOB, but code for BFILES would be functionally the same.

Assume the following table definition:

```
CREATE TABLE basic_lob_table(x varchar2(30), b blob, c clob);
INSERT INTO basic_lob_table
    VALUES('one', '01010101010101010101010101010101', 'onetwothreefour');
INSERT INTO basic_lob_table
    VALUES('two', '02020202020202020202020202020202', '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.

LOBs and BFILES in 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);
#sql iterator NamedFILEIter(BFILE bf);
```

LOB and BFILE 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. Similar code could be written for `BLOBs` and `BFILEs`.

Declaration:

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

Executable code:

```
...
NamedLOBIter iter;
#sql iter = { SELECT c FROM basic_lob_table };
if (iter.next())
    CLOB c1 = iter.c();
if (iter.next())
    CLOB c2 = iter.c();
iter.close();
readFromLongest(c1, c2);
...
```

This example uses an iterator to select two `CLOBs` from the first two rows of the `BASIC_LOB_TABLE`, then prints the larger of the two using the `readFromLongest()` method.

LOB and BFILE Host Variables and Positional Iterator `FETCH INTO` Targets

Host variables of 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 the identical type.

The following example employs table `BASIC_LOB_TABLE` and method `writeToBlob()` defined in previous examples. Similar code could be written for CLOBs and BFILES.

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 Oracle9i 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.

ROWIDs 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("CHUCK 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 Chuck 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 Oracle9i.

```
CREATE OR REPLACE function get_rowid (name varchar2) return rowid is
    rid rowid;
BEGIN
    SELECT rowid INTO rid FROM emp WHERE ename = name;
    RETURN rid;
END get_rowid;
```

Given the preceding stored function, the following example indicates how a ROWID object is used as the assignment type for the function return result.

```
ROWID rowid;
#sql rowid = { values(get_rowid('AMY FEINER')) };
#sql { UPDATE emp SET sal = sal + 500 WHERE rowid = :rowid };
```

This example increases the salary of the employee named Amy Feiner by \$500 according to the ROWID.

ROWID SELECT INTO Targets

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

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

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

ROWID Host Variables and Positional Iterator FETCH INTO Targets

Host variables of type ROWID can appear in the INTO-list of a FETCH INTO statement if the corresponding column attribute in the iterator is of the 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("CHUCK TURNER"))
    {
        #sql { UPDATE emp SET sal = sal + 500 WHERE rowid = :rowid };
    }
}
iter.close();
...
```

This example is similar to the previous named iterator example, but uses a positional iterator with its customary `FETCH INTO` syntax.

Support for Oracle REF CURSOR Types

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

Overview of REF CURSOR Types

Cursor variables are functionally equivalent to JDBC result sets, essentially encapsulating the results of a query. A cursor variable is often referred to as a REF CURSOR, but REF CURSOR itself is a type specifier, not a type name. Instead, named REF CURSOR types must be specified. The following example shows a REF CURSOR type specification:

```
TYPE EmpCurType IS REF CURSOR;
```

Stored procedures and stored functions can return parameters of Oracle REF CURSOR types. You must use PL/SQL to return a REF CURSOR parameter; you cannot accomplish this using SQL alone. A PL/SQL stored procedure or function can declare a variable of some named REF CURSOR type, execute a SELECT statement, and return the results in the REF CURSOR variable.

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 stored function returns
- as output host expressions for stored procedure or function output parameters
- as output host expressions in INTO-lists
- as iterator columns

You can use the Oracle SQL `CURSOR` operator for a nested `SELECT` within an outer `SELECT` statement. This is how you can write a REF CURSOR object to an iterator column or `ResultSet` column in an iterator, or write a REF CURSOR object to an iterator host variable or `ResultSet` host variable in an INTO-list.

["Using Iterators and Result Sets as Host Variables"](#) on page 3-52 has examples showing the use of implicit REF CURSOR variables, including an example of the `CURSOR` operator.

Notes:

- Use the typecode `OracleTypes.CURSOR` for REF CURSOR types.
 - There is no `oracle.sql` class for REF CURSOR types. Use either `java.sql.ResultSet` or an iterator class. Close the result set or iterator to release resources when you are done processing it.
-
-

REF CURSOR Example

The following sample method shows a REF CURSOR type being retrieved from an anonymous block.

```
private static EmpIter refCursInAnonBlock(String name, int no)
    throws java.sql.SQLException {
    EmpIter emps = null;
    System.out.println("Using anonymous block for ref cursor..");
    #sql { begin
        INSERT INTO emp (ename, empno) VALUES (:name, :no);
        OPEN :out emps FOR SELECT ename, empno FROM emp ORDER BY empno;
```

```
        end  
    };  
    return emps;  
}
```

Support for Other Oracle9i 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 the `oracle.sql.NUMBER`, `oracle.sql.CHAR`, and `oracle.sql.RAW` classes.

Because the `oracle.sql.*` classes do not require conversion to Java type format, they offer greater efficiency and precision than equivalent Java types. You would have to convert the data to standard Java types, however, to use it with standard Java programs or to display it to end users.

Extended Support for BigDecimal

SQLJ supports `java.math.BigDecimal` in the following situations:

- as host variables in SQLJ executable statements
- as return values from stored function calls
- as iterator column types

Standard SQLJ has the limitation that a value can be retrieved as `BigDecimal` only if that is the JDBC default mapping, which is the case only for numeric and decimal data. (See [Table 5-1](#) on page 5-3 for more information about JDBC default mappings.)

In Oracle SQLJ, however, you can map to nondefault types as long as the datatype is convertible from numeric and you use Oracle9i, an Oracle JDBC driver, Oracle-specific code generation (or the Oracle customizer), and the Oracle SQLJ runtime. 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: The `BigDecimal` class is in the standard `java.math` package.

Objects, Collections, and OPAQUE Types

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.

There is also a small section at the end regarding Oracle OPAQUE types. These can be similar in functionality to object types, but with a different kind of implementation. Data is represented as an opaque payload of bytes rather than in structured object format.

The following topics are discussed:

- [Oracle Objects and Collections](#)
- [Custom Java Classes](#)
- [User-Defined Types](#)
- [JPublisher and the Creation of Custom Java Classes](#)
- [Strongly Typed Objects and References in SQLJ Executable Statements](#)
- [Strongly Typed Collections in SQLJ Executable Statements](#)
- [Serialized Java Objects](#)
- [Weakly Typed Objects, References, and Collections](#)
- [Oracle OPAQUE Types](#)

Oracle Objects and Collections

This section provides some background conceptual information about Oracle9i objects and collections.

For additional conceptual and reference information about Oracle objects, references, and collections, refer to the *Oracle9i SQL Reference* and the *Oracle9i Application Developer's Guide - Fundamentals*.

For information about how to declare objects and collections, see "[User-Defined Types](#)" on page 6-23.

Introduction to Objects and Collections

Oracle9i 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 either the JDBC 2.0 standard `java.sql.SQLData` interface (for object types only) or the Oracle `oracle.sql.ORAData` interface. Either paradigm is supported by the Oracle9i JPublisher utility, which you can use to automatically generate custom Java classes. Weakly typed representations use the class `oracle.sql.STRUCT` (for objects), `oracle.sql.REF` (for object references), or `oracle.sql.ARRAY` (for collections). Or, alternatively, you can use standard `java.sql.Struct`, `Ref`, or `Array` objects in a weakly typed scenario.

The term "strongly typed" is used where a particular Java type is associated with a particular SQL named (user-defined) type. For example, if there is a `PERSON` type with a corresponding `Person` Java class.

The term "weakly typed" is used where a Java type is used in a generic way and can map to multiple SQL named types. The Java class (or interface) has no special information particular to any SQL type. This is the case for the `oracle.sql.STRUCT`, `REF`, and `ARRAY` types and the `java.sql.Struct`, `Ref`, and `Array` types.

Note that using Oracle extensions in your code requires the following:

- Use one of the Oracle JDBC drivers.
- Use default Oracle-specific code generation or, for ISO code generation, customize the profiles appropriately. (The default customizer, `oracle.sqlj.runtime.util.OraCustomizer`, is recommended.)

For Oracle-specific generated code, produced through the default `-codegen=oracle` translator setting, no profiles are produced so customization is not applicable. Oracle JDBC APIs are called directly through the generated Java code.

- Use the Oracle SQLJ runtime when your application runs.

The Oracle SQLJ runtime and an Oracle JDBC driver are required whenever you use the Oracle customizer, even if you do not actually use Oracle extensions in your code.

For Oracle-specific semantics-checking, you must use an appropriate checker. The default checker, `oracle.sqlj.checker.OracleChecker`, acts as a front end and will run the appropriate checker based on your environment. This will be one of the Oracle-specific checkers if you are using an Oracle JDBC driver.

Oracle-specific types for Oracle objects and collections are included in the `oracle.sql` package.

For information about translator options relating to semantics-checking, see "[Connection Options](#)" on page 8-34 and "[Semantics-Checking and Offline-Parsing Options](#)" on page 8-71.

Custom Java Class Usage Notes

- This chapter primarily discusses the use of custom Java classes with user-defined types; however, classes implementing `ORADATA` can be used for other Oracle SQL types as well. A class implementing `ORADATA` can be employed to perform any kind of desired processing or conversion in the course of transferring data between SQL and Java. See "[Additional Uses for ORADATA Implementations](#)" on page 6-18.
- The `SQLData` interface is intended only for custom object classes. The `ORADATA` interface can be used for any custom Java class.

Terminology Notes

- User-defined SQL object types and user-defined SQL collection types are referred to as *user-defined types* (UDTs).

- Custom Java classes for objects, references, and collections are referred to as *custom object classes*, *custom reference classes*, and *custom collection classes*, respectively.

For general information about Oracle object features and functionality, see the *Oracle9i Application Developer's Guide - Object-Relational Features*.

Oracle Object Fundamentals

Oracle objects (SQL objects) are composite data structures that group related data items, such as facts about each employee, into a single data unit. An object type is functionally similar to a Java class—you can populate and use any number of individual objects of a given object type, just as you can instantiate and use individual objects of a Java type.

For example, you can define an object type `EMPLOYEE` that has the attributes `name` (type `CHAR`), `address` (type `CHAR`), `phonenum` (type `CHAR`), and `employeenumber` (type `NUMBER`).

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

Oracle Collection Fundamentals

There are two categories of Oracle 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; nested table types are used for single-column tables within an outer table. A variable of any `VARRAY` type can be referred to as a `VARRAY`; a variable of any nested table type can be referred to as a *nested table*.

A `VARRAY`, as with any array, is an ordered set of data elements, with each element having an index and all elements being of the same datatype. The *size* of a `VARRAY` refers to the maximum number of elements. Oracle `VARRAY`s, as indicated by their name, are of variable size, but the maximum size of any particular `VARRAY` type must be specified when the `VARRAY` type is declared.

A nested table is an unordered set of elements. Nested table elements within a table can themselves be queried in SQL. A nested table, as with any table, is not created with any particular number of rows—this is determined dynamically.

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. Oracle9i supports nesting of collection types. The elements of a VARRAY or rows of a nested table can be of another VARRAY or nested table type, or these elements can be of a user-defined object type that has VARRAY or nested table attributes.

Object and Collection Datatypes

User-specified object and collection definitions in Oracle9i function as SQL datatype definitions. You can then use these datatypes, as with any other datatype, in defining table columns, SQL object attributes, and stored procedure or function parameters. In addition, once you have defined an object type, the related object reference type can be used as any other SQL reference type.

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 20 characters.

Custom Java Classes

The purpose of custom Java classes is to provide a way to convert data between SQL and Java and make the data accessible, particularly in supporting objects and collections or if you want to perform custom data conversions.

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

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

To be used in SQLJ iterators or host expressions, a custom Java class must implement either the `oracle.sql.ORADData` (and `ORADDataFactory`) interface or the standard `java.sql.SQLData` interface. This section provides an overview of these interfaces and custom Java class functionality, covering the following topics:

- [Custom Java Class Interface Specifications](#)
- [Custom Java Class Support for Object Methods](#)
- [Custom Java Class Requirements](#)
- [Compiling Custom Java Classes](#)
- [Reading and Writing Custom Data](#)
- [Additional Uses for ORADData Implementations](#)

Custom Java Class Interface Specifications

This section discusses specifications of the `ORADData` and `ORADDataFactory` interfaces and the standard `SQLData` interface.

Oracle9i includes a set of new APIs for Oracle-specific custom Java class functionality for user-defined types—`oracle.sql.ORADData` and `oracle.sql.ORADDataFactory`.

The `oracle.sql.CustomDatum` and `oracle.sql.CustomDatumFactory` interfaces used previously for this functionality are deprecated in Oracle9i, but still supported for backward compatibility. You must use the `CustomDatum` interfaces if you are working with an Oracle8i JDBC driver.

ORADData and ORADDataFactory Specifications

Oracle provides the interface `oracle.sql.ORADData` and the related interface `oracle.sql.ORADDataFactory` to use in mapping and converting Oracle object types, reference types, and collection types to custom Java classes.

Data is sent or retrieved in the form of an `oracle.sql.Datum` object, with the underlying data being in the format of the appropriate `oracle.sql.Datum` subclass—`oracle.sql.STRUCT`, for example. This data is still in its SQL 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 *Oracle9i JDBC Developer's Guide and Reference*.)

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

```
interface oracle.sql.ORADData
{
    oracle.sql.Datum toDatum(java.sql.Connection c) throws SQLException;
}
```

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

```
interface oracle.sql.ORADDataFactory
{
    oracle.sql.ORADData create(oracle.sql.Datum d, int sqlType)
        throws SQLException;
}
```

To complete the relationship between the `ORADData` and `ORADDataFactory` interfaces, you must implement a static `getORADDataFactory()` method in any custom Java class that implements the `ORADData` interface. This method returns an object that implements the `ORADDataFactory` interface and that, therefore, can be used to create instances of your custom Java class. This returned object can itself be an instance of your custom Java class, and its `create()` method is used by the

Oracle JDBC driver to produce further instances of your custom Java class, as necessary.

Note: JPublisher output implements the `ORADData` interface and its `toDatum()` method and the `ORADDataFactory` interface and its `create()` method in a single custom Java class; however, `toDatum()` and `create()` are specified in different interfaces to allow the option of implementing them in separate classes. You can have one custom Java class that implements `ORADData`, its `toDatum()` method, and the `getORADDataFactory()` method, and have a separate factory class that implements `ORADDataFactory` and its `create()` method. For purposes of discussion here, however, the assumption is that both interfaces are implemented in a single class.

For information about Oracle SQLJ requirements of a class that implements `ORADData`, see "[Oracle Requirements for Classes Implementing ORADData](#)" on page 6-11.

For more information about the `ORADData` and `ORADDataFactory` interfaces, the `oracle.sql` classes, and the `OracleTypes` class, see the *Oracle9i JDBC Developer's Guide and Reference*.

If you use JPublisher, specifying `-usertypes=oracle` will result in JPublisher generating custom Java classes that implement the `ORADData` and `ORADDataFactory` interfaces and the `getORADDataFactory()` method. Or, for backward compatibility, you have the option of using the JPublisher `-compatible` option in conjunction with `-usertypes=oracle` to use the `CustomDatum` and `CustomDatumFactory` interfaces instead. See the *Oracle9i JPublisher User's Guide* for more information.

ORADData Versus CustomDatum Interfaces

As a result of the `oracle.jdbc` interfaces being introduced in Oracle9i as replacements for the `oracle.jdbc.driver` classes, the `oracle.sql.CustomDatum` and `oracle.sql.CustomDatumFactory` interfaces, formerly used to access customized objects, have been deprecated in favor of new interfaces—`oracle.sql.ORADData` and `oracle.sql.ORADDataFactory`. Like the `CustomDatum` interfaces, these can be used as an Oracle-specific alternative to the standard `SQLData` interface. The `CustomDatum` interfaces are still supported for backward compatibility.

CustomDatum and CustomDatumFactory have the following definitions:

```
public interface CustomDatum
{
    oracle.sql.Datum toDatum(
        oracle.jdbc.driver.OracleConnection conn
    ) throws SQLException;

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

The connection `conn` and typecode `sqlType` are used as described for `ORADData` and `ORADDataFactory` in "[ORADData and ORADDataFactory Specifications](#)" on page 6-7. Note, however, that `CustomDatum` uses the Oracle-specific `OracleConnection` type instead of the standard `Connection` type.

SQLData Specification

Standard JDBC 2.0 supplies the interface `java.sql.SQLData` to use in mapping and converting structured object types to Java classes. This interface is intended for mapping structured object types only, not object references, collections/arrays, or other SQL types.

The `SQLData` interface is a JDBC 2.0 standard, specifying a `readSQL()` method to read data into a Java object, and a `writeSQL()` method to write to the database from a Java object.

For information about functionality that is required of a class that implements `SQLData`, see "[Requirements for Classes Implementing SQLData](#)" on page 6-13.

For additional information about standard `SQLData` functionality, refer to the Sun Microsystems JDBC 2.0 or higher API specification.

If you use `JPublisher`, specifying `-usertypes=jdbc` will result in `JPublisher` generating custom Java classes that implement the `SQLData` interface.

Custom Java Class Support for Object Methods

Methods of Oracle objects can be invoked from custom Java class wrappers. Whether the underlying stored procedure is written in PL/SQL or is written in Java and published to SQL is invisible to the user.

A Java wrapper method used to invoke a server method requires a connection to communicate with the server. The connection object can be provided as an explicit parameter or can be associated in some other way (as an attribute of your custom Java class, for example).

If the connection object used by the wrapper method is a non-static attribute, then the wrapper method must be an instance method of the custom Java class in order to have access to the connection. Custom Java classes generated by JPublisher use this technique.

There are also issues regarding output and input-output parameters in methods of Oracle objects. If a stored procedure (SQL object method) modifies the internal state of one of its arguments, then the actual argument passed to the stored procedure is modified. In Java this is not possible. When a JDBC output parameter is returned from a stored procedure call, it must be stored in a newly created object. The original object identity is lost.

One way to return an output or input-output parameter to the caller is to pass the parameter as an element of an array. If the parameter is input-output, 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.

When you use JPublisher, it implements wrapper methods by default. This is true for generated classes implementing either the `SQLData` interface or the `ORADData` interface. To disable this feature, set the JPublisher `-methods` flag to `false`. See the *Oracle9i JPublisher User's Guide* for more information.

Note: If you are implementing a custom Java class yourself, there are various ways that you can implement wrapper methods. Data processing in the server can be done either through the SQL object method directly, or by forwarding the object value from the client to the server and then executing the method there. To see how JPublisher implements wrapper methods, and whether this may meet your needs, see "[JPublisher Implementation of Wrapper Methods](#)" on page 6-46.

Custom Java Class Requirements

Custom Java classes must satisfy certain requirements to be recognized by the Oracle SQLJ translator as valid host variable types, and to allow type-checking by the translator.

This section discusses Oracle-specific requirements of custom Java classes so they can support this functionality. Requirements for both `ORADData` implementations and `SQLData` implementations are covered.

Note: Custom Java classes for user-defined types are often referred to in this manual as "wrapper classes".

Oracle Requirements for Classes Implementing `ORADData`

Oracle requirements for `ORADData` implementations are primarily the same for any kind of custom Java class but vary slightly depending on whether the class is for mapping to objects, object references, collections, or some other SQL type.

These requirements are as follows:

- The class implements the `oracle.sql.ORADData` interface.
- The class implements a method `getORADDataFactory()` that returns an `oracle.sql.ORADDataFactory` object, as follows:

```
public static oracle.sql.ORADDataFactory getORADDataFactory();
```

If using the deprecated `CustomDatum` interface, the class implements the method `getFactory()` that returns an `oracle.sql.CustomDatumFactory` object as follows:

```
public static oracle.sql.CustomDatumFactory getFactory();
```

- The class has a constant, `_SQL_TYPECODE` (string), initialized to the `oracle.jdbc.OracleTypes` typecode of the `Datum` subclass instance 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 might be appropriate. For example, for using a custom Java class to serialize and deserialize Java objects into or out of RAW fields, a `_SQL_TYPECODE` of `OracleTypes.RAW` is used. See ["Serialized Java Objects"](#) on page 6-71.

(The `OracleTypes` class simply defines a typecode, which is an integer constant, for each Oracle datatype. For standard SQL types, the `OracleTypes` entry is identical to 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 that indicates the relevant user-defined type name.

- Custom object classes and custom collection classes must have a constant, `_SQL_NAME` (string), 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, `_SQL_BASETYPE` (string), 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 `ORAData` uses, specifying a UDT name is not applicable.

Usage Notes

- A collection type name reflects the collection type, not the base type. For example, if you have declared a `VARRAY` or nested table type `PERSON_ARRAY` for `PERSON` objects, then the name of the collection type that you specify for the `_SQL_NAME` entry is `PERSON_ARRAY`, not `PERSON`.
- When specifying the SQL type in a `_SQL_NAME` field, if the SQL type was declared in a case-sensitive way (in quotes), then you must specify the SQL name exactly as it was declared, such as `CaseSensitive` or `SCOTT.CaseSensitive`. (Note that this differs from usage in a `JPublisher` input file, where the case-sensitive name must also appear in quotes.) If you did *not* declare the SQL type in a case-sensitive way (no quotes), then you must specify the SQL name in all uppercase, such as `ADDRESS` or `SCOTT.ADDRESS`.

`JPublisher` automatically generates the value of this field appropriately, according to case-sensitivity and the `JPublisher -omit_schema_names` setting if applicable.

Requirements for Classes Implementing `SQLData`

The ISO SQLJ standard outlines requirements for type map definitions for classes implementing the `SQLData` interface.

Alternatively, `SQLData` wrapper classes can identify associated SQL object types through `public static final` fields. This non-standard functionality was introduced in Oracle SQLJ release 8.1.6 and continues to be supported.

Be aware of the following important points:

- Whether you use a type map or use alternative (non-standard) `public static final` fields to specify mappings, you must be consistent in your approach. Either use a type map that specifies all relevant mappings so that you do not require `public static final` fields, or do not use a type map at all and specify all mappings through `public static final` fields.
- `SQLData`, unlike `ORAData`, is for mapping structured object types only. It is not for object references, collections/arrays, or any other SQL types. If you are not using `ORAData`, then your only choices for mapping object references and collections are the weak types `java.sql.Ref` and `java.sql.Array`, respectively, or `oracle.sql.REF` and `oracle.sql.ARRAY`.

- `SQLData` implementations require a JDK 1.2.x or higher environment. Although Oracle JDBC supports JDBC 2.0 extensions under JDK 1.1.x through the `oracle.jdbc2` package, Oracle SQLJ does not.
- When specifying the mapping from a SQL type to a Java type (described below), if the SQL type was declared in a case-sensitive way (in quotes), then you must specify the SQL name exactly as it was declared, such as `CaseSensitive` or `SCOTT.CaseSensitive`. (Note that this differs from usage in a JPublisher input file, where the case-sensitive name must also appear in quotes.) If you did *not* declare the SQL type in a case-sensitive way (no quotes), then you must specify the SQL name in all uppercase, such as `ADDRESS` or `SCOTT.ADDRESS`.

Mapping Specified in Type Map Resource First, consider the mapping representation according to the ISO SQLJ standard. Assume that `Address`, `pack.Person`, and `pack.Manager.InnerPM` (where `InnerPM` is an inner class of `Manager`) are three wrapper classes that implement `java.sql.SQLData`.

- You must employ these classes only in statements that use explicit connection context instances of a declared connection context type. Assume, for example, that this type is called `SDContext`. Example:

```
Address          a =...;
pack.Person      p =...;
pack.Manager.InnerPM pm =...;
SDContext ctx = new SDContext(url,user,pwd,false);
#sql [ctx] { ... :a ... :p ... :pm ... };
```

- The connection context type must have been declared using the `with` attribute `typeMap` that specifies an associated class implementing a `java.util.PropertyResourceBundle`. In the preceding example, `SDContext` might have been declared as follows:

```
#sql public static context SDContext with (typeMap="SDMap");
```

- The type map resource must provide the mapping from SQL object types to corresponding Java classes that implement the `java.sql.SQLData` interface. This mapping is specified with entries of the following form:

```
class.<java_class_name>=STRUCT <sql_type_name>
```

The keyword `STRUCT` can also be omitted. In the example, the resource file `SDMap.properties` might contain the following entries:

```
class.Address=STRUCT SCOTT.ADDRESS
```



```
class.pack.Person=PERSON
class.pack.Manager$InnerPM=STRUCT PRODUCT_MANAGER
```

Although "." separates package and class name, you *must* use the character "\$" to separate an inner class name.

Important: If you used default Oracle-specific code generation in this example, then any iterator that is used for a statement whose context type is `SDContext` must also have been declared with the same associated type map, `SDMap`, such as in the following example:

```
#sql public static iterator SDIter with (typeMap="SDMap");
...
SDContext sdctx = ...
SDIter sditer;
#sql [sdctx] sditer = { SELECT ...};
```

This is to ensure that proper code is generated for the iterator class.

This mechanism of specifying mappings in a type map resource is more complicated than the non-standard alternative (discussed next). Furthermore, it is not possible to associate a type map resource with the default connection context. The advantage is that all the mapping information is placed in a single location—the type map resource. This means that the type mapping in an already compiled application can be easily adjusted at a later time, for example to accommodate new SQL types and Java wrappers in an expanding SQL-Java type hierarchy.

Be aware of the following:

- You must employ the SQLJ `runtime12` or `runtime12ee` library to use this feature. Type maps are represented as `java.util.Map` objects. These are exposed in the SQLJ runtime API and, therefore, *cannot* be supported by the JDK 1.1 or generic runtime libraries.
- You must use the Oracle SQLJ runtime and Oracle-specific code generation or profile customization if your `SQLData` wrapper classes occur as OUT or INOUT parameters in SQLJ statements. This is because the SQL type of such parameters is required for `registerOutParameter()` by the Oracle JDBC driver. Furthermore, for OUT parameter type registration, the SQL type is "frozen in" by the type map in effect during translation.

- The SQLJ type map is independent of any JDBC type map you may be using on the underlying connection. Thus, you must be careful if you are mixing SQLJ and JDBC code that both use `SQLData` wrappers. However, you can easily extract the type map in effect on a given SQLJ connection context:

```
ctx.getTypeMap();
```

Mapping Specified in Static Field of Wrapper Class Alternatively, a class that implements `SQLData` can satisfy the following non-standard requirement.

- The Java class declares the `public static final String`-valued field `_SQL_NAME`. This field defines the name of the SQL type that is being wrapped by the Java class.

In the example, the `Address` class would have the following field declaration:

```
public static final String _SQL_NAME="SCOTT.ADDRESS";
```

The following declaration would be in `pack.Person`:

```
public static final String _SQL_NAME="PERSON";
```

And the class `pack.Manager.InnerPM` would have the following:

```
public static final String _SQL_NAME="PRODUCT_MANAGER";
```

Note that `JPublisher` always generates `SQLData` wrapper classes with the `_SQL_NAME` field. However, this field is ignored in SQLJ statements that reference a type map.

Notes:

- If a class that implements the `_SQL_NAME` field is used in a SQLJ statement with an explicit connection context type and associated type map, then that type map is used, and the `_SQL_NAME` field is ignored, thereby simplifying migration of existing SQLJ programs to the new ISO SQLJ standard.
 - The static SQL-Java type correspondence specified in the `_SQL_NAME` field is independent from any JDBC type map you may be using on the underlying connection. Thus, you must be careful if you are mixing SQLJ and JDBC code that both use `SQLData` wrappers.
-
-

Compiling Custom Java Classes

You can include any `.java` files for your custom Java classes (whether `ORADData` or `SQLData` implementations) on the SQLJ command line together with the `.sqlj` file(s) for your application. However, this is not necessary if the SQLJ `-checksource` flag is set to `true` (the default) and your classpath includes the directory where the custom Java source is located. (This discussion assumes you are creating `.java` files for your custom objects and collections, not `.sqlj` files. Any `.sqlj` files must be included in the SQLJ command line.)

For example, if `ObjectDemo.sqlj` uses Oracle object types `ADDRESS` and `PERSON` and you have produced custom Java classes for these objects, then you can run SQLJ as follows.

- If `-checksource=true` (default) and the classpath includes the custom Java source location:

```
% sqlj ObjectDemo.sqlj
```

- If `-checksource=false` (this is a single wraparound line):

```
% sqlj ObjectDemo.sqlj Address.java AddressRef.java Person.java  
PersonRef.java
```

You also have the choice of using your Java compiler to compile custom `.java` source files directly. If you do this, you must do it prior to translating `.sqlj` files.

Running the SQLJ translator is discussed in [Chapter 8, "Translator Command Line and Options"](#). For more information about the `-checksource` flag, see ["Source Check for Type Resolution \(-checksource\)"](#) on page 8-69.

Note: Because `ORADData` implementations rely on Oracle-specific features, SQLJ will report numerous portability warnings if you do not use the translator portability setting `-warn=noportable` (the default). For information about the `-warn` flag, see ["Translator Warnings \(-warn\)"](#) on page 8-45.

Reading and Writing Custom Data

Through the use of custom Java class instances, Oracle SQLJ and JDBC allow you to read and write user-defined types as though they are built-in types. Exactly how this is accomplished is transparent to the user.

For the mechanics of how data is read and written, for both `ORADData` implementations and `SQLData` implementations, see the *Oracle9i JDBC Developer's Guide and Reference*.

Additional Uses for `ORADData` Implementations

To this point, discussion of custom Java classes has been for use as one of the following:

- wrappers for SQL objects—custom object classes, for use with `oracle.sql.STRUCT` instances
- wrappers for SQL references—custom reference classes, for use with `oracle.sql.REF` instances
- wrappers for SQL collections—custom collection classes, for use with `oracle.sql.ARRAY` instances

It might be useful, however, to provide custom Java classes to wrap other `oracle.sql.*` types as well, for customized conversions or processing. You can accomplish this with classes that implement `ORADData` (but not `SQLData`), as in the following examples:

- Perform encryption and decryption or validation of data.
- Perform logging of values that have been read or are being written.
- Parse character columns (such as character fields containing URL information) into smaller components.
- Map character strings into numeric constants.
- Map data into more desirable Java formats (such as mapping a `DATE` field to `java.util.Date` format).
- Customize data representation (for example, data in a table column is in feet, but you want it represented in meters after it is selected).
- Serialize and deserialize Java objects—into or out of `RAW` fields, for example

This last use is further discussed in "[Serialized Java Objects](#)" on page 6-71.

The rest of this section provides an example of a class (`BetterDate`) that implements `ORADData` and can be used instead of `java.sql.Date` to represent dates.

Note: This sort of functionality is not possible through the `SQLData` interface, as `SQLData` implementations can wrap only structured object types.

General Use of ORADData—BetterDate.java This example shows a class that implements the `ORADData` interface to provide a customized representation of Java dates.

Note: This is not a complete application—there is no `main()` method.

```
import java.util.Date;
import oracle.sql.ORADData;
import oracle.sql.DATE;
import oracle.sql.ORADDataFactory;
import oracle.jdbc.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 ORADData, ORADDataFactory {
    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 ORADDataFactory getORADDataFactory() { return _BetterDateFactory;}

    // the current time...
    public BetterDate() {
        super();
    }

    public oracle.sql.Datum toDatum(java.sql.Connection conn) {
        return new DATE(toSQLDate());
    }
}
```

```
public oracle.sql.ORAData create(oracle.sql.Datum dat, int intx) {
    if (dat==null) return null;
    DATE DAT = ((DATE)dat);
    java.sql.Date jsd = DAT.dateValue();
    return new BetterDate(jsd);
}

public java.sql.Date toSQLDate() {
    java.sql.Date retval;
    retval = new java.sql.Date(this.getYear()-1900, this.getMonth()-1,
        this.getDate());
    return retval;
}

public BetterDate(java.sql.Date d) {
    this(d.getYear()+1900, d.getMonth()+1, d.getDate());
}

private static int [] deconstructString(String s) {
    int [] retval = new int[3];
    int y,m,d; char temp; int offset;
    StringBuffer sb = new StringBuffer(s);
    temp=sb.charAt(1);
    // figure the day of month
    if (temp < '0' || temp > '9') {
        m = sb.charAt(0)-'0';
        offset=2;
    } else {
        m = (sb.charAt(0)-'0')*10 + (temp-'0');
        offset=3;
    }

    // figure the month
    temp = sb.charAt(offset+1);
    if (temp < '0' || temp > '9') {
        d = sb.charAt(offset)-'0';
        offset+=2;
    } else {
        d = (sb.charAt(offset)-'0')*10 + (temp-'0');
        offset+=3;
    }

    // figure the year, which is either in the format "yy" or "yyyy"
    // (the former assumes the current century)
    if (sb.length() <= (offset+2)) {
        y = (((new BetterDate()).getYear())/100)*100 +
            (sb.charAt(offset)- '0') * 10 +
            (sb.charAt(offset+1)- '0');
```

```

    } else {
        y = (sb.charAt(offset)- '0') * 1000 +
            (sb.charAt(offset+1)- '0') * 100 +
            (sb.charAt(offset+2)- '0') * 10 +
            (sb.charAt(offset+3)- '0');
    }
    retval[0]=y;
    retval[1]=m;
    retval[2]=d;
//    System.out.println("Constructing date from string as: "+d+"/"+m+"/"+y);
    return retval;
}
private BetterDate(int [] stuff) {
    this(stuff[0], stuff[1], stuff[2]);
}
// takes a string in the format: "mm-dd-yyyy" or "mm/dd/yyyy" or
// "mm-dd-yy" or "mm/dd/yy" (which assumes the current century)
public BetterDate(String s) {
    this(BetterDate.deconstructString(s));
}

// years are as '1990', months from 1..12 (unlike java.util.Date!), date
// as '1' to '31'
public BetterDate(int year, int months, int date) {
    super(year-1900,months-1,date);
}
// returns "Date: dd-mon-yyyy"
public String toString() {
    int yr = getYear();
    return getDate()+"-"+monthNames[getMonth()-1]+"-"+
        toDigit[(yr/1000)%10] +
        toDigit[(yr/100)%10] +
        toDigit[(yr/10)%10] +
        toDigit[yr%10];
//    return "Date: " + getDate() + "-" + getMonth() + "-" + (getYear()%100);
}
public BetterDate addDays(int i) {
    if (i==0) return this;
    return new BetterDate(getYear(), getMonth(), getDate()+i);
}
public BetterDate addMonths(int i) {
    if (i==0) return this;
    int yr=getYear();
    int mon=getMonth()+i;
    int dat=getDate();

```

```
        while(mon<1) {
            --yr;mon+=12;
        }
        return new BetterDate(yr, mon,dat);
    }
    // returns year as in 1996, 2007
    public int getYear() {
        return super.getYear()+1900;
    }
    // returns month as 1..12
    public int getMonth() {
        return super.getMonth()+1;
    }
    public boolean equals(BetterDate sd) {
        return (sd.getDate() == this.getDate() &&
            sd.getMonth() == this.getMonth() &&
            sd.getYear() == this.getYear());
    }
    // subtract the two dates; return the answer in whole years
    // uses the average length of a year, which is 365 days plus
    // a leap year every 4, except 100, except 400 years =
    // = 365 97/400 = 365.2425 days = 31,556,952 seconds
    public double minusInYears(BetterDate sd) {
        // the year (as defined 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();
    }
}
```


User-Defined Types

This section contains examples of creating and using user-defined object types and collection types in Oracle9i. For more information about any of the SQL commands used here, refer to the *Oracle9i 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 remainder of this section provides an example of creating user-defined object types in Oracle9i.

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

Here is the script:

```
/** Using user-defined types (UDTs) in SQLJ **/
/
/** Create ADDRESS UDT **/
CREATE TYPE ADDRESS AS OBJECT
(
  street      VARCHAR(60),
  city        VARCHAR(30),
  state       CHAR(2),
  zip_code    CHAR(5)
)
/
```

```
/** Create PERSON UDT containing an embedded ADDRESS UDT */
CREATE TYPE PERSON AS OBJECT
(
  name    VARCHAR(30),
  ssn     NUMBER,
  addr    ADDRESS
)
/
/** Create a typed table for PERSON objects */
CREATE TABLE persons OF PERSON
/
/** Create a relational table with two columns that are REFs
    to PERSON objects, as well as a column which is an Address ADT. */
CREATE TABLE employees
(
  empnumber          INTEGER PRIMARY KEY,
  person_data        REF PERSON,
  manager            REF PERSON,
  office_addr        ADDRESS,
  salary             NUMBER
)
/** Insert some data--2 objects into the persons typed table */
INSERT INTO persons VALUES (
  PERSON('Wolfgang Amadeus Mozart', 123456,
    ADDRESS('Am Berg 100', 'Salzburg', 'AT', '10424'))
)
/
INSERT INTO persons VALUES (
  PERSON('Ludwig van Beethoven', 234567,
    ADDRESS('Rheinallee', 'Bonn', 'DE', '69234'))
)
/
/** Put a row in the employees table */
INSERT INTO employees (empnumber, office_addr, salary) VALUES (
  1001,
  ADDRESS('500 Oracle Parkway', 'Redwood Shores', 'CA', '94065'),
  50000)
/
/** Set the manager and PERSON REFs for the employee */
UPDATE employees
  SET manager =
    (SELECT REF(p) FROM persons p WHERE p.name = 'Wolfgang Amadeus Mozart')
/
UPDATE employees
  SET person_data =
    (SELECT REF(p) FROM persons p WHERE p.name = 'Ludwig van Beethoven')
```

Note: Use of a table alias, such as `p` 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 *Oracle9i SQL Reference* for more information about table aliases.

Creating Collection Types

There are two categories of collections

- variable-length arrays (VARRAYs)
- nested tables

Oracle SQL commands to create VARRAY types are of the following form:

```
CREATE TYPE typename IS VARRAY(n) OF datatype;
```

The *typename* designation is the desired name of your VARRAY type, *n* is the desired maximum number of elements in the array, and *datatype* is the datatype of the array elements. 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;
```

The *typename* designation 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, as with any database table, can have any number of rows. For example:

```
CREATE TYPE person_array AS TABLE OF person;
```

This command creates a nested table where each row consists of a PERSON object.

The rest of this section provides an example of creating a user-defined collection type (as well as object types) in Oracle9i.

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)
- PERSON and ADDRESS objects (repeating the same definitions used earlier in "Creating Object Types" on page 6-23)
- an EMPLOYEES table, which includes a PHONE_ARRAY column

Here is the script:

Rem This is a SQL*Plus script used to create schema to demonstrate collection
Rem manipulation in SQLJ

```
CREATE TYPE PARTICIPANT_T AS OBJECT (  
    empno    NUMBER(4),  
    ename    VARCHAR2(20),  
    job      VARCHAR2(12),  
    mgr      NUMBER(4),  
    hiredate DATE,  
    sal      NUMBER(7,2),  
    deptno   NUMBER(2))  
/  
show errors  
CREATE TYPE MODULE_T AS OBJECT (  
    module_id NUMBER(4),  
    module_name VARCHAR2(20),  
    module_owner REF PARTICIPANT_T,  
    module_start_date DATE,  
    module_duration NUMBER )  
/  
show errors  
create TYPE MODULETBL_T AS TABLE OF MODULE_T;  
/  
show errors  
CREATE TABLE projects (  
    id NUMBER(4),  
    name VARCHAR(30),  
    owner REF PARTICIPANT_T,  
    start_date DATE,  
    duration NUMBER(3),
```

```
modules MODULETBL_T ) NESTED TABLE modules STORE AS modules_tab;

show errors
CREATE TYPE PHONE_ARRAY IS VARRAY (10) OF varchar2(30)
/

/** Create ADDRESS UDT */
CREATE TYPE ADDRESS AS OBJECT
(
  street      VARCHAR(60),
  city        VARCHAR(30),
  state       CHAR(2),
  zip_code    CHAR(5)
)
/
/** Create PERSON UDT containing an embedded ADDRESS UDT */
CREATE TYPE PERSON AS OBJECT
(
  name        VARCHAR(30),
  ssn         NUMBER,
  addr        ADDRESS
)
/
CREATE TABLE employees
( empnumber          INTEGER PRIMARY KEY,
  person_data        REF person,
  manager            REF person,
  office_addr        address,
  salary             NUMBER,
  phone_nums         phone_array
)
/
```

JPublisher and the Creation of Custom Java Classes

Oracle offers flexibility in how users can customize the mapping of Oracle object types, reference types, and collection types to Java classes in a strongly typed paradigm. Developers have the following choices in creating these custom Java classes:

- using Oracle JPublisher to automatically generate custom Java classes and using those classes directly without modification
- using JPublisher to automatically generate custom Java classes and corresponding subclasses, which can subsequently be user-modified for any desired functionality
- manually coding custom Java classes without using JPublisher, if the classes meet the requirements stated in "[Custom Java Class Requirements](#)" on page 6-11

Although you have the option of manually coding your custom Java classes, it is advisable to instead use JPublisher-generated classes directly or modify JPublisher-generated subclasses.

JPublisher can implement either the Oracle `oracle.sql.ORADData` interface or the standard `java.sql.SQLData` interface when it generates a custom object class. If you choose the `ORADData` implementation, then JPublisher will also generate a custom reference class. For compatibility with older JDBC versions, JPublisher can also generate classes that implement the deprecated `oracle.sql.CustomDatum` interface.

The `SQLData` interface is not intended for custom reference or custom collection classes. If you want your code to be portable, you have no choice but to use standard weakly typed `java.sql.Ref` objects to map to references, and `java.sql.Array` objects to map to collections.

This manual provides only minimal information and detail regarding the JPublisher utility. See the *Oracle9i JPublisher User's Guide* for more information.

For detailed discussion of the `ORADData` and `SQLData` interfaces and relative advantages of the `ORADData` interface, see the *Oracle9i JDBC Developer's Guide and Reference*.

What JPublisher Produces

When you use JPublisher to generate custom Java classes, you can use either an `ORADData` implementation (for custom object classes, custom reference classes, or custom collection classes) or a `SQLData` implementation (for custom object classes only). An `ORADData` implementation will also implement the `ORADDataFactory` interface, for creating instances of the custom Java class.

This is controlled by how you set the JPublisher `-usertypes` option. A setting of `-usertypes=oracle` specifies an `ORADData` implementation; a setting of `-usertypes=jdbc` specifies a `SQLData` implementation.

ORADData Implementation

When you run JPublisher for a user-defined object type and use the `ORADData` implementation for your custom object class (through the default `-usertypes=oracle` setting), JPublisher automatically creates the following:

- a custom object class, typically in a `.sqlj` source file, to act as a type definition to correspond to your Oracle object type

This class includes getter and setter methods for each attribute. The method names are of the form `getFoo()` and `setFoo()` for attribute `foo`.

In addition, JPublisher by default will generate wrapper methods in your class that invoke the associated Oracle object methods executing in the server. This can be disabled, however, by setting `-methods=false`. In this case, JPublisher produces no wrapper methods and generates `.java` files instead of `.sqlj` files for custom objects. The `-methods` option is described later in this section.

- a related custom reference class for object references to your Oracle object type

This class includes a `getValue()` method that returns an instance of your custom object class, and a `setValue()` method that updates an object value in the database, taking as input an instance of the custom object class.

A strongly typed reference class is always generated, regardless of whether the SQL object type uses references.

Advantages of using strongly typed instead of weakly typed references are described in ["Strongly Typed Object References for ORADData Implementations"](#) on page 6-30.

- custom classes for any object or collection attributes of the top-level object

This is necessary so that attributes can be materialized in Java whenever an instance of the top-level class is materialized.

When you run JPublisher for a user-defined collection type, choosing the `ORADData` implementation, JPublisher automatically creates the following:

- a custom collection class to act as a type definition to correspond to your Oracle collection type

This class includes overloaded `getArray()` and `setArray()` methods to retrieve or update a collection as a whole, a `getElement()` method and `setElement()` method to retrieve or update individual elements of a collection, and additional utility methods.

- a custom object class for the elements, if the elements of the collection are objects

This is necessary so that object elements can be materialized in Java whenever an instance of the collection is materialized.

JPublisher-generated custom Java classes in any of these categories implement the `ORADData` interface, the `ORADDataFactory` interface, and the `getORADDataFactory()` method.

Notes:

- If you specify the `ORADData` implementation, the generated classes will use Oracle-specific features and therefore will not be portable.
 - JPublisher still supports implementation of the `CustomDatum` interface, replaced by `ORADData` and deprecated in Oracle9i, through the `-compatible` option. This is described in "[Choose the Implementation for Generated Classes](#)" on page 6-32.
-
-

Strongly Typed Object References for `ORADData` Implementations

For Oracle `ORADData` implementations, JPublisher always generates strongly typed object reference classes as opposed to using the weakly typed `oracle.sql.REF` class. This is to provide greater type safety and to mirror the behavior in SQL, where object references are strongly typed. The strongly typed classes (with names such as `PersonRef` for references to `PERSON` objects) are essentially wrappers for the `REF` class.

In these strongly typed `REF` wrappers, there is a `getValue()` method that produces an instance of the SQL object that is referenced, in the form of an instance of the corresponding Java class. (Or, in the case of inheritance, perhaps as an instance of a subclass of the corresponding Java class.) For example, if there is a

PERSON SQL object type, with a corresponding Person Java class, there will also be a PersonRef Java class. The `getValue()` method of the PersonRef class would return a Person instance containing the data for a PERSON object in the database.

Whenever a SQL object type has an attribute that is an object reference, the Java class corresponding to the object type would have an attribute that is an instance of a Java class corresponding to the appropriate reference type. For example, if there is a PERSON object with a MANAGER REF attribute, then the corresponding Person Java class will have a ManagerRef attribute.

SQLData Implementation

When you run JPublisher for a user-defined object type and choose the SQLData implementation for your custom object class (through the `-usertypes=jdbc` setting), JPublisher will produce a custom object class to act as a type definition to correspond to your Oracle object type. This class will include the following:

- getter and setter methods for each attribute
- implementations of the standard SQLData interface `readSQL()` and `writeSQL()` methods
- wrapper methods that invoke the Oracle object methods executing in the server (unless you specify `-methods=false` when you run JPublisher)

Because the SQLData interface is intended only for objects, however, and not for references or collections, JPublisher will not generate a custom reference class for references to the Oracle object type. You will have to use standard weakly typed `java.sql.Ref` instances, or perhaps `oracle.sql.REF` instances if you do not require portability. Note that REF instances, like custom reference class instances, have Oracle extension methods `getValue()` and `setValue()` to read or write instances of the referenced object. Standard Ref instances do not have this functionality.

Similarly, because you cannot use a SQLData implementation for a custom collection class, you must use standard weakly typed `java.sql.Array` instances, or perhaps `oracle.sql.ARRAY` instances if you do not require portability. Array and ARRAY instances, like custom collection class instances, have `getArray()` functionality to read the collection as a whole or in part, but do not have the element-level access and writability offered by the custom collection class `getElement()` and `setElement()` methods.

Note: The `SQLData` interface is defined in the JDBC specification to be portable. However, if you want the `SQLData` implementation produced by JPublisher to be portable, you must avoid using any Oracle-specific features and Oracle type mapping (which uses the Oracle-specific `oracle.sql.*` classes).

Generating Custom Java Classes

This section discusses key JPublisher command-line functionality for specifying the user-defined types that you want to map to Java and for specifying object class names, collection class names, attribute type mappings, and wrapper methods. These key points can be summarized as follows:

- Specify the implementation to use (`ORADData` or `SQLData`), through the JPublisher `-usertypes` option.
- Specify user-defined types to map to Java. You can specify the custom object and custom collection class names for JPublisher to use, or you can accept the default names. Use the JPublisher `-sql`, `-user`, and `-case` options, as appropriate.
- Optionally specify attribute type mappings through the JPublisher `-xxxtypes` options: `-numbertypes`, `-builtintypes`, and `-lobtypes`.
- Choose whether or not JPublisher will create wrapper methods, in particular for Oracle object methods. Use the JPublisher `-methods` flag, which is enabled by default.

Note: Throughout the remainder of this section, we simplify discussion of custom reference classes or custom collection classes by referring only to `ORADData` implementations.

Choose the Implementation for Generated Classes

Before running JPublisher, consider whether you want the generated classes to implement the Oracle `ORADData` interface or the standard `SQLData` interface. Using `SQLData` will likely make your code more portable, but using `ORADData` offers a number of advantages, including no need for type maps.

The preceding section, "[What JPublisher Produces](#)" on page 6-29, discusses some of the implementation details for each scenario.

Remember the following:

- You must use `ORADData` implementations for custom collection classes. The `SQLData` interface does not support collections (arrays).
- Strongly typed reference classes are always generated for `ORADData` custom object class implementations, but not for `SQLData` custom object class implementations. The `SQLData` interface does not support strongly typed object references—use the weak `java.sql.Ref` type or `oracle.sql.REF` type instead.

For detailed discussion of the `ORADData` and `SQLData` interfaces and relative advantages of the `ORADData` interface, see the *Oracle9i JDBC Developer's Guide and Reference*.

Use the `JPublisher -usertypes` option to specify which interface you want your classes to implement. A setting of `-usertypes=oracle` (the default) specifies the `ORADData` interface, while a setting of `-usertypes=jdbc` specifies the `SQLData` interface.

Note: If you have a requirement to implement the `CustomDatum` interface, which is replaced by `ORADData` and deprecated in *Oracle9i*, you can do so with a `JPublisher -compatible` setting of `customdatum`. This, combined with a `-usertypes=oracle` setting, results in generated classes implementing the `CustomDatum` interface. The default is `-compatible=oradata`.

The setting `-compatible=8i` or `-compatible=both8i` also directs `JPublisher` to use `CustomDatum`, as well as resulting in code generation that is backward compatible to *Oracle8i* versions of `JPublisher`. See the *Oracle9i JPublisher User's Guide* for more information.

The following `JPublisher` command-line examples will result in implementation of `ORADData`, `CustomDatum`, and `SQLData`, respectively (assume `%` is a system prompt).

```
% jpub -usertypes=oracle ... <other option settings>
```

```
% jpub -usertypes=oracle -compatible=customdatum ... <other option settings>
```

```
% jpub -usertypes=jdbc ... <other option settings>
```

JPublisher will ignore a `-compatible=customdatum` or `-compatible=oradata` setting if `-usertypes=jdbc`.

Specify User-Defined Types to Map to Java

In using JPublisher to create custom Java classes, use the `-sql` option to specify the user-defined SQL types that you want to map to Java. You can either specify the custom object class names and custom collection class names, or you can accept the defaults.

The default names of your top-level custom classes—the classes that will correspond to the user-defined type names you specify to the `-sql` option—are identical to the user-defined type names as you enter them on the JPublisher command line. Because SQL names in the database are case-insensitive by default, you can capitalize them to ensure that your class names are capitalized according to Java convention. For example, if you want to generate a custom class for `employee` objects, you can run JPublisher as follows:

```
% jpub -sql=Employee ...
```

The default names of other classes, such as for `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 a `HomeAddress.sqlj` or `.java` source file.

Important: Only non-case-sensitive SQL names are supported on the JPublisher command line. If a user-defined type was defined in a case-sensitive way (in quotes) in SQL, then you must specify the name in the JPublisher `INPUT` file instead of on the command line, and in quotes. See ["Using JPublisher INPUT Files"](#) on page 6-42.

Note: For backward compatibility to previous versions of JPublisher, the `-types` option is still accepted as an alternative to `-sql`.

On the JPublisher command line, use the following syntax for the `-sql` option (you can specify multiple actions in a single option setting).

```
-sql=udt1<:mapclass1><,udt2<:mapclass2>>,...,<udtN<:mapclassN>> ...
```

And use the `-user` option to specify the database schema. 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.sqlj` to define a `Myobj` class. For the Oracle collection `MYCOLL`, this command will create source `MyCollClass.java` to define a `MyCollClass` class.

You can optionally specify schema names in the `-sql` option—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 (relevant to `ORADATA` implementations only). For example, if JPublisher produces Java source `Myobj.sqlj` to define custom object class `Myobj`, then it will also produce Java source `MyobjRef.java` to define a `MyobjRef` custom reference class.

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](#)" on page 6-23, 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 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
```

(Each of the preceding two examples is a single wraparound command line.)

The second example will produce Java source files `Address.sqlj`, `AddressRef.java`, `Person.sqlj`, `PersonRef.java`, `PhoneArray.java`, `ParticipantT.sqlj`, `ParticipantTRef.java`, `ModuleT.sqlj`,

`ModuleTRef.java`, and `ModuletblT.java`. Examples of some of these source files are provided in "[JPublisher Custom Java Class Examples](#)" on page 6-47.

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.

Note: As of Oracle9i release 2, as an alternative to specifying multiple mappings in a single `-sql` setting, you can use multiple `-sql` options in the same command line. The effect of multiple `-sql` options is cumulative.

If you want to change how JPublisher uses character case in default names for the methods and attributes that it generates, including lower-level custom Java class names for attributes that are objects or collections, 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.

Note: If you run JPublisher without specifying the user-defined types to map to Java, it will process all user-defined types in the schema. Generated class names, for both your top-level custom classes and any other classes for object attributes or collection elements, will be based on the setting of the `-case` option.

Specify Type Mappings

JPublisher offers several choices for how to map user-defined types and their attribute and element types between SQL and Java. The rest of this section lists categories of SQL types and the mapping options available for each category.

(See "[Supported Types for Host Expressions](#)" on page 5-2 for general information about how Oracle datatypes map to Java types.)

For more information about JPublisher features or options, see the *Oracle9i JPublisher User's Guide*.

Categories of SQL Types JPublisher categorizes SQL types into the following groups, with corresponding JPublisher options as noted:

- **numeric types**—anything stored as SQL type `NUMBER`
Use the JPublisher `-numbertypes` option to specify type-mapping for numeric types.
- **LOB types**—SQL types `BLOB` and `CLOB`
Use the JPublisher `-lobtypes` option to specify type-mapping for LOB types.
- **built-in types**—anything stored as a SQL type not covered by the preceding categories, for example: `CHAR`, `VARCHAR2`, `LONG`, and `RAW`
Use the JPublisher `-builtintypes` option to specify type-mapping for built-in types.

Type-Mapping Modes JPublisher defines the following type-mapping modes:

- **JDBC mapping** (setting `jdbc`)—Uses standard default mappings between SQL types and Java native types. This setting is valid for the `-numbertypes`, `-lobtypes`, and `-builtintypes` options.
- **Oracle mapping** (setting `oracle`)—Uses corresponding `oracle.sql` types to map to SQL types. This setting is valid for the `-numbertypes`, `-lobtypes`, and `-builtintypes` options.
- **object-JDBC mapping** (setting `objectjdbc`)—This is an extension of JDBC mapping. Where relevant, object-JDBC mapping uses numeric object types from the standard `java.lang` package (such as `java.lang.Integer`, `Float`, and `Double`) instead of primitive Java types (such as `int`, `float`, and `double`). The `java.lang` types are nullable; the primitive types are not. This setting is valid for the `-numbertypes` option only.

- **BigDecimal mapping (setting `bigdecimal`)**—Uses `java.math.BigDecimal` to map to all numeric attributes; appropriate if you are dealing with large numbers but do not want to map to the `oracle.sql.NUMBER` type. This setting is valid for the `-numbertypes` option only.

Note: Using `BigDecimal` mapping can significantly degrade performance.

The next section discusses type mapping options that you can use for object attributes and collection elements.

Mapping Attribute or Element Types to Java If you do not specify mappings for the attribute types of a SQL object type or the element types of a SQL collection type, then JPublisher uses the following defaults:

- For numeric types, object-JDBC mapping is the default mapping.
- For LOB types, Oracle mapping is the default mapping.
- For built-in type types, JDBC mapping is the default mapping.

If you want alternate mappings, use the `-numbertypes`, `-lobtypes`, and `-builtintypes` options as necessary, depending on the attribute types you have and the mappings you desire.

If an attribute type is itself a SQL object type, it will be mapped according to the `-usertypes` setting.

Important: Be especially aware that if you specify a `SQLData` implementation for the custom object class and want the code to be portable, you must use portable mappings for the attribute types. The defaults for numeric types and built-in types are portable, but for LOB types you must specify `-lobtypes=jdbc`.

Summary of SQL Type Categories and Mapping Settings [Table 6-1](#) summarizes JPublisher categories for SQL types, the mapping settings relevant for each category, and the default settings.

Table 6–1 *JPublisher SQL Type Categories, Supported Settings, and Defaults*

SQL Type Category	JPublisher Mapping Option	Mapping Settings	Default
UDT types	-usertypes	oracle, jdbc	oracle
numeric types	-numbertypes	oracle, jdbc, objectjdbc, bigdecimal	objectjdbc
LOB types	-lobtypes	oracle, jdbc	oracle
built-in types	-builtintypes	oracle, jdbc	jdbc

Note: The JPublisher `-mapping` option used in previous releases is deprecated but still supported. For information about how JPublisher converts `-mapping` option settings to settings for the new mapping options, see the *Oracle9i JPublisher User's Guide*.

Generate Wrapper Methods

In creating custom object classes to map Oracle objects to Java, the `-methods` option instructs JPublisher whether to include Java wrappers for Oracle object methods (member functions). The default `-methods=true` setting generates wrappers, and also results in JPublisher generating a `.sqlj` file instead of a `.java` file for a custom object class (unless the underlying SQL object actually has no methods).

Wrapper methods generated by JPublisher are always instance methods, even when the original object methods are static. See "[Custom Java Class Support for Object Methods](#)" on page 6-10 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 derived in the same fashion as custom Java class names (as described in "[Specify User-Defined Types to Map to Java](#)" on page 6-34), except that the initial character will be lowercase. For example, by default an object method name of `CALC_SAL` results in a Java wrapper method of `calcSal()`.

Alternatively, you can specify desired Java method names, but this requires use of a JPublisher `INPUT` file and is discussed in "[Creating Custom Java Classes and Specifying Member Names](#)" on page 6-45.

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 *Oracle9i JPublisher User's Guide* for information.

Regarding Overloaded Methods If you run JPublisher for an Oracle object that has an overloaded method where multiple signatures have the same corresponding Java signature, then JPublisher will generate a uniquely named method for each signature. It accomplishes this by appending `_n` to function names, where `n` is a number. This is to ensure that no two methods in the generated custom Java class have the same name and signature. Consider, for example, the SQL functions defined in creating a `MY_TYPE` object type:

```
CREATE OR REPLACE TYPE my_type AS OBJECT
(
  ...

  MEMBER FUNCTION myfunc(x INTEGER)
    RETURN my_return IS
    BEGIN
      ...
    END;

  MEMBER FUNCTION myfunc(y SMALLINT)
    RETURN my_return IS
    BEGIN
      ...
    END;

  ...
);
```

Without precaution, both definitions of `myfunc` result in the following name and signature in Java:

```
myfunc(Integer)
```

This is because both `INTEGER` and `SMALLINT` in SQL map to the Java `Integer` type.

Instead, JPublisher might call one `myfunc_1` and the other `myfunc_2`. (The `_n` is unique for each. In simple cases it will likely be `_1`, `_2`, and so on, but it might sometimes be arbitrary, other than being unique for each.)

Note: How JPublisher handles overloaded wrapper methods applies to SQL functions created within an object or within a package, but not to top-level functions—overloading is not allowed at the top level.

Generate Custom Java Classes and Map Alternate Classes

You can use JPublisher to generate a custom Java class but instruct it to map the object type (or collection type) to an alternative class instead of to the generated class.

A typical scenario is to treat JPublisher-generated classes as superclasses, extend them to add functionality, and map the object types to the subclasses. For example, presume you have an Oracle object type `ADDRESS` and want to produce a custom Java class for it that has functionality beyond what is produced by JPublisher. You can use JPublisher to generate a 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`.

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.sqlj` (or possibly `.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, you will retrieve it into a `MyAddress` instance.
- Creates a `MyAddressRef` class in `MyAddressRef.java`, instead of creating a `JAddressRef` class.

- Creates an initial version of the `MyAddress` class in a `MyAddress.sqlj` source file (or possibly `MyAddress.java`), unless the file already exists (in which case it is not changed).

`MyAddress` subclasses `JAddress`. In order to implement the extended functionality for `MyAddress`, you can start with the JPublisher-generated `MyAddress` source file, editing it as desired.

For further discussion about subclassing JPublisher-generated classes (continuing the preceding example), see ["Extending Classes Generated by JPublisher"](#) on page 6-51.

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`.

Important: If a user-defined type was defined in a case-sensitive way (in quotes) in SQL, then you must specify the name in quotes. For example:

```
SQL "CaseSensitiveType" AS CaseSensitiveType
```

or, if also specifying a non-case-sensitive schema name:

```
SQL SCOTT."CaseSensitiveType" AS CaseSensitiveType
```

or, if also specifying a case-sensitive schema name:

```
SQL "Scott"."CaseSensitiveType" AS CaseSensitiveType
```

The AS clauses are optional.

Avoid using a dot (".") as part of the schema name or type name itself.

INPUT File Example In the following example, JPublisher will pick up the `-user` option from the command line and go to INPUT file `myinput.in` for type mappings.

Command line:

```
% jpub -input=myinput.in -user=scott/tiger
```

Contents of INPUT file `myinput.in`:

```
SQL Myobj
SQL mycoll AS MyCollClass
SQL employee GENERATE Employee AS MyEmployee
```

This accomplishes the following:

- User-defined type `MYOBJ` gets the custom object class name `Myobj` because that is how you typed it—JPublisher creates source `Myobj.sqlj` (or possibly `Myobj.java`, if `Myobj` has no methods) and `MyobjRef.java`.
- User-defined type `MYCOLL` is mapped to `MyCollClass`. JPublisher creates a `MyCollClass.java` source file.
- User-defined type `EMPLOYEE` is mapped to the `MyEmployee` class. JPublisher creates source `Employee.sqlj` (or possibly `Employee.java`) and `MyEmployeeRef.java`, as well as an initial version of `MyEmployee.sqlj` (or `.java`) unless the file already exists. If you retrieve an object from the database that has an `EMPLOYEE` attribute, this attribute would be created as an instance

of `MyEmployee` in Java. Or, if you retrieve an `EMPLOYEE` object directly, presumably you will retrieve it into a `MyEmployee` instance. You are responsible for the `MyEmployee` code, but for convenience you can start with the JPublisher-generated `MyEmployee` source file and edit it to implement your specialized functionality for `EMPLOYEE` objects in Java. `MyEmployee` subclasses the `Employee` class.

Using JPublisher Properties Files

You can use the JPublisher `-props` command-line option to specify a properties file for JPublisher to use for additional type mappings and other option settings.

In a properties file, `"jpub."` (including the period) is equivalent to the command-line `"-"` (single-dash), and other syntax remains the same. Specify only one option per line.

For type mappings, for example, `"jpub.sql"` is equivalent to `"-sql"`. You can specify multiple mappings in a single `jpub.sql` setting. Alternatively, as of Oracle9i release 2, you can use multiple `jpub.sql` options—the effect would be cumulative (as for multiple `-sql` options on the command line).

Note: As of Oracle9i release 2, the behavior of properties files is to ignore any line that does not begin with `"jpub."` or `"--jpub."` (two dashes followed by `"jpub."`). This allows you to use the same file as both a SQL script to create the types, and a properties file for JPublisher. If you start each JPublisher statement with `"--"`, which indicates a SQL comment, it will be ignored by SQL*Plus. And SQL statements will be ignored by JPublisher.

Properties File Example In the following example, JPublisher will pick up the `-user` option from the command line and go to 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.usertypes=oracle
```

This produces the same results as the input-file example above, explicitly specifying the `oracle` mapping setting.

Note: Unlike SQLJ, JPublisher has no default properties file. To use a properties file, you must use the `-props` option.

Creating Custom Java Classes and Specifying Member Names

In generating custom Java classes, you can specify the names of any attributes or methods of the custom class. This cannot be specified on the JPublisher command line, however—only 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
```

Notes:

- 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; `gethomeAddress()` does not.
-
-

JPublisher Implementation of Wrapper Methods

This section describes how JPublisher generates wrapper methods and how wrapper method calls are processed at runtime.

Generation of Wrapper Methods

The following points describe how JPublisher generates wrapper methods:

- JPublisher-generated wrapper methods are implemented in SQLJ; therefore, whenever `-methods=true`, the custom object class will be defined in a `.sqlj` file instead of in a `.java` file, assuming the object type defines methods. Run SQLJ to translate the `.sqlj` file.

Note: Even if the object type does not define methods, you can ensure that a `.sqlj` file is generated by setting `-methods=always`. See the *Oracle9i JPublisher User's Guide* for more information.

- All wrapper methods generated by JPublisher are implemented as instance methods. This is because a database connection is required for you to invoke the corresponding server method. Each instance of a JPublisher-generated custom Java class has a connection associated with it.

Runtime Execution of Wrapper Method Calls

The following points describe what JPublisher-generated Java wrapper methods execute at runtime. In this discussion, "Java wrapper method" refers to a method in the custom Java object, while "wrapped SQL method" refers to the SQL object method that is wrapped by the Java wrapper method.

- The custom Java object is converted to a SQL object and passed to the database, where the wrapped SQL method is invoked. After this method invocation, the new value of the SQL object is returned to Java in a new custom Java object, either as a function return from the wrapped SQL method (if the SQL method is a stored procedure), or, if there already is a function return, as an array element in an additional output parameter (if the SQL method is a stored function).
- 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 6-10.) If the parameter is input-output, then the wrapper

method takes the array element as input; after processing, the wrapper assigns the output to the array element.

JPublisher Custom Java Class Examples

This section provides examples of JPublisher-generated `ORADData` implementations for the following user-defined types (created in "User-Defined Types" on page 6-23):

- a custom object class (`Address`, corresponding to the Oracle object type `ADDRESS`) and related custom reference class (`AddressRef`)
- a custom collection class (`ModuletblT`, corresponding to the Oracle collection type `MODULETBL_T`)

Assume that the `-methods` option has its default `true` setting and that the `ADDRESS` type has methods, so that a `.sqlj` file is generated for the `Address` class.

Note: For examples of JPublisher-generated `SQLData` implementations, as well as further examples of JPublisher-generated `ORADData` implementations, see the *Oracle9i JPublisher User's Guide*.

Custom Object Class—Address.sqlj

Following is an example of the source code that JPublisher generates for a custom object class. Implementation details have been omitted.

In this example, unlike in "Creating Object Types" on page 6-23, assume the Oracle object `ADDRESS` has only the `street` and `zip_code` attributes.

```
package bar;

import java.sql.SQLException;
import java.sql.Connection;
import oracle.jdbc.OracleTypes;
import oracle.sql.ORADData;
import oracle.sql.ORADDataFactory;
import oracle.sql.Datum;
import oracle.sql.STRUCT;
import oracle.jpub.MutableStruct;

public class Address implements ORADData, ORADDataFactory
{
    public static final String _SQL_NAME = "SCOTT.ADDRESS";
```

```
public static final int _SQL_TYPECODE = OracleTypes.STRUCT;

public static ORADDataFactory getORADDataFactory()
{ ... }

/* constructors */
public Address()
{ ... }

public Address(String street, java.math.BigDecimal zip_code)
    throws SQLException
{ ... }

/* ORADData interface */
public Datum toDatum(Connection c) throws SQLException
{ ... }

/* ORADDataFactory interface */
public ORADData create(Datum d, int sqlType) throws SQLException
{ ... }

/* accessor methods */
public String getStreet() throws SQLException
{ ... }

public void setStreet(String street) throws SQLException
{ ... }

public java.math.BigDecimal getZipCode() throws SQLException
{ ... }

public void setZipCode(java.math.BigDecimal zip_code) throws SQLException
{ ... }
}
```

Custom Reference Class—AddressRef.java

Following is an example of the source code that JPublisher generates for a custom reference class to be used for references to ADDRESS objects. Implementation details have been omitted.

```
package bar;

import java.sql.SQLException;
import java.sql.Connection;
import oracle.jdbc.OracleTypes;
import oracle.sql.ORAData;
import oracle.sql.ORADataFactory;
import oracle.sql.Datum;
import oracle.sql.REF;
import oracle.sql.STRUCT;

public class AddressRef implements ORAData, ORADataFactory
{
    public static final String _SQL_BASETYPE = "SCOTT.ADDRESS";
    public static final int _SQL_TYPECODE = OracleTypes.REF;

    public static ORADataFactory getORADataFactory()
    { ... }

    /* constructors */
    public AddressRef()
    { ... }

    public static AddressRef(ORAData o) throws SQLException
    { ... }

    /* ORAData interface */
    public Datum toDatum(Connection c) throws SQLException
    { ... }

    /* ORADataFactory interface */
    public ORAData create(Datum d, int sqlType) throws SQLException
    { ... }

    public static AddressRef cast(ORAData o) throws SQLException
    { ... }

    public Address getValue() throws SQLException
    { ... }

    public void setValue(Address c) throws SQLException
    { ... }
}
```

Custom Collection Class—ModuletblT.java

Following is an example of the source code that JPublisher generates for a custom collection class. Implementation details have been omitted.

```
import java.sql.SQLException;
import java.sql.Connection;
import oracle.jdbc.OracleTypes;
import oracle.sql.ORAData;
import oracle.sql.ORADataFactory;
import oracle.sql.Datum;
import oracle.sql.ARRAY;
import oracle.sql.ArrayDescriptor;
import oracle.jpub.runtime.MutableArray;

public class ModuletblT implements ORAData, ORADataFactory
{
    public static final String _SQL_NAME = "SCOTT.MODULETBL_T";
    public static final int _SQL_TYPECODE = OracleTypes.ARRAY;

    public static ORADataFactory getORADataFactory()
    { ... }

    /* constructors */
    public ModuletblT()
    { ... }

    public ModuletblT(ModuleT[] a)
    { ... }

    /* ORAData interface */
    public Datum toDatum(Connection c) throws SQLException
    { ... }

    /* ORADataFactory interface */
    public ORAData create(Datum d, int sqlType) throws SQLException
    { ... }

    public String getBaseTypeName() throws SQLException
    { ... }

    public int getBaseType() throws SQLException
    { ... }

    public ArrayDescriptor getDescriptor() throws SQLException
    { ... }
```

```
/* array accessor methods */
public ModuleT[] getArray() throws SQLException
{ ... }

public void setArray(ModuleT[] a) throws SQLException
{ ... }

public ModuleT[] getArray(long index, int count) throws SQLException
{ ... }

public void setArray(ModuleT[] a, long index) throws SQLException
{ ... }

public ModuleT getObjectElement(long index) throws SQLException
{ ... }

public void setElement(ModuleT a, long index) throws SQLException
{ ... }
}
```

Extending Classes Generated by JPublisher

You might want to enhance the functionality of a custom Java class generated by JPublisher by adding methods and transient fields. You can accomplish this by extending the JPublisher-generated class.

For example, suppose you want JPublisher to generate the class `JAddress` from the SQL object type `ADDRESS`. You also want to use a class `MyAddress` to represent `ADDRESS` objects and implement special functionality. The `MyAddress` class must extend `JAddress`.

Another way to enhance the functionality of a JPublisher-generated class is to simply add methods to it. However, adding methods to the generated class is not recommended if you anticipate running JPublisher at some future time to regenerate the class. If you run JPublisher to regenerate a class that you have modified in this way, you would have to save a copy and then manually merge your changes back in.

JPublisher Functionality for Extending Generated Classes

As discussed in "[Generate Custom Java Classes and Map Alternate Classes](#)" on page 6-41, the syntax to have JPublisher 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 reference class `MyAddressRef` (in `MyAddressRef.java`) rather than `JAddressRef`.

In addition, JPublisher alters the code it generates to implement the following functionality:

- The `MyAddress` class, instead of the `JAddress` class, is used to represent attributes whose SQL type is `ADDRESS`.
- The `MyAddress` class, instead of the `JAddress` class, is used to represent method arguments and function results whose type is `ADDRESS`.
- The `MyAddress` factory, instead of the `JAddress` factory, is used to construct Java objects whose SQL type is `ADDRESS`.

You would presumably use `MyAddress` similarly in any additional code that you write.

At runtime, the Oracle JDBC driver will map any occurrences of `ADDRESS` data in the database to `MyAddress` instances, instead of to `JAddress` instances.

Requirements of Extended Classes

By default, JPublisher will create an initial version of the user subclass `MyAddress` in a file `MyAddress.sqlj` or `MyAddress.java` (`MyAddress.sqlj` if the original class uses methods and you are publishing these methods), unless the file to be created already exists, in which case it will not be changed. You can edit this file as necessary to add your desired functionality.

`MyAddress` must have a no-argument constructor. The easiest way to construct a properly initialized object is to invoke the constructor of the superclass, either explicitly or implicitly.

As a result of subclassing the JPublisher-generated class, the subclass will inherit definitions of the `_SQL_NAME` field, which it requires, and the `_SQL_TYPECODE` field.

In addition, one of the following will be true.

- If the JPublisher-generated class implements the `ORADData` and `ORADDataFactory` interfaces, then the subclass will inherit this implementation and the necessary `toDatum()` and `create()` functionality of the generated class. The subclass implements a `getORADDataFactory()` method that returns an instance of your map class (such as a `MyAddress` object).

or:

- If the JPublisher-generated class implements the `SQLData` interface, then the subclass will inherit this implementation and the necessary `readSQL()` and `writeSQL()` functionality of the generated class.

JPublisher-Generated Custom Object Class—`JAddress.sqlj`

The code for the JPublisher-generated `JAddress` class, implementing `ORADData` and `ORADDataFactory`, is mostly identical to the code shown previously for the `Address` class, with the exception that mentions of `Address` are replaced by mentions of `JAddress`.

JPublisher-Generated Alternate Reference Class—`MyAddressRef.java`

Continuing the example in the preceding sections, consider code for the JPublisher-generated reference class, `MyAddressRef` (as opposed to `JAddressRef`, because `MyAddress` is the class that `ADDRESS` objects map to). This class also implements `ORADData` and `ORADDataFactory`. The implementation is nearly identical to that of `AddressRef.java`, except for the change in class name and the fact that setter and getter methods use `MyAddress` instances instead of `Address` instances.

Extended Custom Object Class—`MyAddress.sqlj`

Again continuing the example, here is sample code for a `MyAddress` class that subclasses the JPublisher-generated `JAddress` class. The comments in the code show what is inherited from `JAddress`. Implementation details have been omitted.

```
import java.sql.SQLException;
import oracle.sql.ORADData;
import oracle.sql.ORADDataFactory;
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 ORADDataFactory getORADDataFactory()
    {
        return _myAddressFactory;
    }

    /* constructor */
    public MyAddress()
    { super(); }

    /* ORADData interface */
    /* toDatum() inherited from JAddress */

    /* ORADDataFactory interface */
    public ORADData create(oracle.sql.Datum d, int sqlType) throws SQLException
    { ... }

    /* accessor methods inherited from JAddress */

    /* Additional methods go here. These additional methods (not shown)
       are the reason that JAddress was extended.
    */
}
```


Strongly Typed Objects and References in SQLJ Executable Statements

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 SQL datatypes of the attributes.

For host expressions, you can use host variables of your custom object class type or custom reference class type. Alternatively, you can use host variables that correspond to object attributes, using variable types that appropriately map to the SQL datatypes of the attributes.

The remainder of this section provides examples of how to manipulate Oracle objects using custom object classes, custom object class attributes, and custom reference classes for host variables and iterator columns in SQLJ executable statements.

The first two examples operate at the object level:

1. [Selecting Objects and Object References into Iterator Columns](#)
2. [Updating an Object](#)

The third example operates at the scalar-attribute level:

3. [Inserting an Object Created from Individual Object Attributes](#)

The fourth example operates through a reference:

4. [Updating an Object Reference](#)

Refer back to the Oracle object types ADDRESS and PERSON in "[Creating Object Types](#)" on page 6-23.

Selecting Objects and Object References into Iterator Columns

This example uses a custom Java class and a custom reference class (ORADaTa implementations) 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 `EMPADDRS`, which includes an `ADDRESS` column and an `ADDRESS` reference column:

```
CREATE TABLE empaddr  
( 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 empaddr };  
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:

```
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 6-23.

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 empnum = 1001;

    try {
        #sql {
            SELECT office_addr
            INTO :addr
            FROM employees
            WHERE empnumber = :empnum };
        System.out.println("Current office address of employee 1001:");

        printAddressDetails(addr);

        /* Now update the street of address */

        String street = "100 Oracle Parkway";
        addr.setStreet(street);

        /* Put updated object back into the database */

        try {
            #sql {
                UPDATE employees
                SET office_addr = :addr
                WHERE empnumber = :empnum };
        }
    }
}
```

```
System.out.println
    ("Updated employee 1001 to new address at Oracle Parkway.");

/* Select new address to verify update */

try {
    #sql {
        SELECT office_addr
        INTO :new_addr
        FROM employees
        WHERE empnumber = :empnum };

    System.out.println("New office address of employee 1001:");
    printAddressDetails(new_addr);

    } catch (SQLException exn) {
    System.out.println("Verification SELECT failed with "+exn); }

    } catch (SQLException exn) {
    System.out.println("UPDATE failed with "+exn); }

    } catch (SQLException exn) {
    System.out.println("SELECT failed with "+exn); }
}
...
```

Note the use of the `setStreet()` accessor method of the `Address` object. Remember that `JPublisher` provides such accessor methods for all attributes in any custom Java class that it produces.

This example uses the `printAddressDetails()` utility. Here is the source code for this method:

```
static void printAddressDetails(Address a) throws SQLException
{
    if (a == null) {
        System.out.println("No Address available.");
        return;
    }

    String street = ((a.getStreet()==null) ? "NULL street" : a.getStreet());
    String city = (a.getCity()==null) ? "NULL city" : a.getCity();
    String state = (a.getState()==null) ? "NULL state" : a.getState();
    String zip_code = (a.getZipCode()==null) ? "NULL zip" : a.getZipCode();
```

```

System.out.println("Street: '" + street + "'");
System.out.println("City:   '" + city   + "'");
System.out.println("State:  '" + state  + "'");
System.out.println("Zip:    '" + zip_code + "'");
}

```

Inserting an Object Created from Individual Object Attributes

This example declares and sets input host variables corresponding to attributes of `PERSON` and nested `ADDRESS` objects, then uses these values to insert a new `PERSON` object into the `persons` table in the database.

```

...
// Inserting an object

static void insertObject()
{
    String new_name  = "NEW PERSON";
    int    new_ssn   = 987654;
    String new_street = "NEW STREET";
    String new_city  = "NEW CITY";
    String new_state = "NS";
    String new_zip   = "NZIP";
    /*
    * Insert a new PERSON object into the persons table
    */
    try {
        #sql {
            INSERT INTO persons
            VALUES (PERSON(:new_name, :new_ssn,
                ADDRESS(:new_street, :new_city, :new_state, :new_zip))) };

        System.out.println("Inserted PERSON object NEW PERSON.");

    } catch (SQLException exn) { System.out.println("INSERT failed with "+exn); }
}
...

```

Updating an Object Reference

This example selects a `PERSON` reference from the `persons` table and uses it to update a `PERSON` reference in the `employees` table. It uses simple (`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 empnum = 1001;
    String new_manager = "NEW PERSON";

    System.out.println("Updating manager REF.");
    try {
        #sql {
            UPDATE employees
            SET manager =
                (SELECT REF(p) FROM persons p WHERE p.name = :new_manager)
            WHERE empnumber = :empnum };

        System.out.println("Updated manager of employee 1001. Selecting back");

    } catch (SQLException exn) {
        System.out.println("UPDATE REF failed with "+exn); }

    /* Select manager back to verify the update */
    Person manager;
    try {
        #sql {
            SELECT deref(manager)
            INTO :manager
            FROM employees e
            WHERE empnumber = :empnum };

        System.out.println("Current manager of "+empnum+":");
        printPersonDetails(manager);
    } catch (SQLException exn) {
        System.out.println("SELECT REF failed with "+exn); }
    }
    ...
}
```

Note: This example uses table alias syntax (`p`) as discussed previously. Also, the `REF` syntax is required in selecting a reference through the object to which it refers, and the `DEREF` syntax is required in selecting an object through a reference. See the *Oracle9i SQL Reference* for more information about table aliases, `REF`, and `DEREF`.

Strongly Typed Collections in SQLJ Executable Statements

As with strongly typed objects and references, Oracle SQLJ supports different scenarios for reading and writing data through strongly typed collections, using either iterators or host expressions.

From the perspective of a SQLJ developer, both categories of collections—VARRAY and nested table—are treated essentially the same, but there are some differences in implementation and performance.

Oracle SQLJ, and Oracle SQL in general, support syntax choices so that nested tables can be accessed and manipulated either apart from or together with their outer tables. In this section, manipulation of a nested table by itself will be referred to as *detail-level* manipulation; 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 defined in "[Creating Collection Types](#)" on page 6-25.

Following the nested table discussion are some brief VARRAY examples.

Notes: In Oracle SQLJ, VARRAY types and nested table types can be retrieved only 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 data in nested tables. Use the `CURSOR` keyword in the outer `SELECT` statement to encapsulate the inner `SELECT` statement. This is shown in "[Selecting Data from a Nested Table Using a Nested Iterator](#)" on page 6-67.

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 *Oracle9i 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 (ORADa implementation) for `MODULETBL_T` nested tables, `ModuleT` is a JPublisher-generated custom object class for `MODULE_T` objects, and `MODULETBL_T` nested tables contain `MODULE_T` objects.

A nested table of `MODULE_T` objects is selected from the `modules` column of the `projects` table into a `ModuletblT` host variable.

Following that, the `ModuletblT` variable (containing the nested table) is passed to a method that accesses its elements through its `getArray()` method, writing the data to a `ModuleT[]` array. All custom collection classes generated by JPublisher include a `getArray()` method. Then each element is copied from the `ModuleT[]` array into a `ModuleT` object, and individual attributes are retrieved through accessor methods (`getModuleName()`, for example) and then printed. All JPublisher-generated custom object classes include such accessor methods.

```
static ModuletblT mymodules=null;  
...  
  
public static void getModules2(int projId)  
throws Exception  
{  
    System.out.println("Display modules for project " + projId );  
  
    try {  
        #sql {SELECT modules INTO :mymodules  
            FROM projects WHERE id=:projId };  
    }  
}
```

```

        showArray(mymodules);
    } catch(Exception e) {
        System.out.println("Error:getModules2");
        e.printStackTrace();
    }
}

public static void showArray(ModuleTblT a)
{
    try {
        if ( a == null )
            System.out.println( "The array is null" );
        else {
            System.out.println( "printing ModuleTable array object of size "
                               +a.length());
            ModuleT[] modules = a.getArray();

            for (int i=0;i<modules.length; i++) {
                ModuleT module = modules[i];
                System.out.println("module "+module.getModuleId()+
                                   ", "+module.getModuleName()+
                                   ", "+module.getModuleStartDate()+
                                   ", "+module.getModuleDuration());
            }
        }
    }
    catch( Exception e ) {
        System.out.println("Show Array");
        e.printStackTrace();
    }
}

```

Manipulating a Nested Table Using TABLE Syntax

This example uses TABLE syntax to work at the detail level to access and update nested table elements directly, based on master-level criteria.

The `assignModule()` method selects a nested table of `MODULE_T` objects from the `MODULES` column of the `PROJECTS` table, then updates `MODULE_NAME` for a particular row of the nested table.

Similarly, the `deleteUnownedModules()` method selects a nested table of `MODULE_T` objects, then deletes any unowned modules in the nested table (where `MODULE_OWNER` is null).

These methods use table alias syntax, as discussed previously—in this case, *m* for the nested table and *p* for the `participants` table. See the *Oracle9i 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 example below.

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 *Oracle9i 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 a host expression. Presume the following SQL definitions:

```
CREATE TYPE PHONE_ARRAY IS VARRAY (10) OF varchar2(30)
/
/** Create ADDRESS UDT */
CREATE TYPE ADDRESS AS OBJECT
(
  street      VARCHAR(60),
  city        VARCHAR(30),
  state       CHAR(2),
  zip_code    CHAR(5)
)
/
/** Create PERSON UDT containing an embedded ADDRESS UDT */
CREATE TYPE PERSON AS OBJECT
(
  name        VARCHAR(30),
  ssn         NUMBER,
  addr        ADDRESS
)
/

CREATE TABLE employees
( empnumber      INTEGER PRIMARY KEY,
  person_data    REF person,
  manager        REF person,
  office_addr    address,
  salary         NUMBER,
  phone_nums     phone_array
)
/
```

And presume that JPublisher is used to create a custom collection class `PhoneArray` to map from the `PHONE_ARRAY` SQL type.

The following method selects a row from this table, placing the data into a host variable of the `PhoneArray` type.

```
private static void selectVarray() throws SQLException
{
    PhoneArray ph;
    #sql {select phone_nums into :ph from employees where empnumber=2001};
    System.out.println(
        "there are "+ph.length()+" phone numbers in the PhoneArray.  They are:");

    String [] pharr = ph.getArray();
    for (int i=0;i<pharr.length;++i)
        System.out.println(pharr[i]);
}
```

Inserting a Row that Includes a VARRAY

This section provides an example of inserting data from a host expression into a VARRAY, using the same SQL definitions and custom collection class (PhoneArray) as in the previous section.

The following methods populate a PhoneArray instance and use it as a host variable, inserting its data into a VARRAY in the database.

```
// creates a varray object of PhoneArray and inserts it into a new row
private static void insertVarray() throws SQLException
{
    PhoneArray phForInsert = consUpPhoneArray();
    // clean up from previous demo runs
    #sql {delete from employees where empnumber=2001};
    // insert the PhoneArray object
    #sql {insert into employees (empnumber, phone_nums)
        values(2001, :phForInsert)};
}

private static PhoneArray consUpPhoneArray()
{
    String [] strarr = new String[3];
    strarr[0] = "(510) 555.1111";
    strarr[1] = "(617) 555.2222";
    strarr[2] = "(650) 555.3333";
    return new PhoneArray(strarr);
}
```


Serialized Java Objects

When writing and reading instances of Java objects to or from the database, it is sometimes advantageous to define a SQL object type that corresponds to your Java class, and use the mechanisms of mapping custom Java classes described previously. This fully permits SQL queries on your Java objects.

In some cases, however, you may want to store Java objects "as-is" and retrieve them later, using database columns of type RAW or BLOB. There are different ways to accomplish this:

- You can map a serializable Java class to RAW or BLOB columns by using a non-standard extension to the type map facility, or by adding a typecode field to the serializable class, so that instances of the serializable class can be stored as RAW or BLOB.
- You can use the ORAData facility to define a serializable wrapper class whose instances can be stored in RAW or BLOB columns.

Serializing in any of these ways works for any Oracle SQLJ runtime library except `runtime-nonoracle`.

Serializing Java Classes to RAW and BLOB Columns

If you want to store instances of Java classes directly in RAW or BLOB columns, then you must meet certain non-standard requirements to specify the desired SQL-Java mapping. (Note that in SQLJ statements the serializable Java objects can be transparently read and written as if they were built-in types.)

You have two options in specifying the SQL-Java type mapping:

- Declare a type map in the connection context declaration and use this type map to specify mappings.
- Use the public static final field `_SQL_TYPECODE` to specify the mapping.

The rest of this section describes each of these options.

Defining a Type Map for Serializable Classes

Consider an example where `SAddress`, `pack.SPPerson`, and `pack.Manager.InnerSPM` (where `InnerSPM` is an inner class of `Manager`) are serializable Java classes. In other words, these classes implement the `java.io.Serializable` interface.

You must employ the classes only in statements that use explicit connection context instances of a declared connection context type, such as `ServletContext` in the following example:

```
SAddress          a =...;
pack.SPerson      p =...;
pack.Manager.InnerSPM pm =...;
ServletContext ctx = new ServletContext(url,user,pwd,false);
#sql [ctx] { ... :a ... :OUT p ... :INOUT pm ... };
```

The following is required:

- The connection context type must have been declared using the `typeMap` attribute of a `with` clause to specify an associated class implementing a `java.util.PropertyResourceBundle`. In the example, `ServletContext` might have been declared as follows.

```
#sql public static context ServletContext with (typeMap="SerMap");
```

- The type map resource must provide non-standard mappings from RAW or BLOB columns to the serializable Java classes. This mapping is specified with entries of the following form, depending on whether the Java class is mapped to a RAW or a BLOB column:

```
oracle-class.<java_class_name>=JAVA_OBJECT RAW
oracle-class.<java_class_name>=JAVA_OBJECT BLOB
```

The keyword `oracle-class` marks this as an Oracle-specific extension. In the example, the resource file `SerMap.properties` might contain the following entries:

```
oracle-class.SAddress=JAVA_OBJECT RAW
oracle-class.pack.SPerson=JAVA_OBJECT BLOB
oracle-class.packManager$InnerSPM=JAVA_OBJECT RAW
```

(Although "." separates package and class names, you *must* use the character "\$" to separate an inner class name.)

Note that this Oracle-specific extension can be placed in the same type map resource as standard `SQLData` type map entries.

Using Fields to Determine Mapping for Serializable Classes

As an alternative to using a type map for a serializable class, you can use static fields in the serializable class to determine type mapping.

You can add either of the following fields to a class that implements the `java.io.Serializable` interface, such as the `SAddress` and `SPerson` classes from the example in "Defining a Type Map for Serializable Classes" on page 6-71.

```
public final static int _SQL_TYPECODE = oracle.jdbc.OracleTypes.RAW;
```

or:

```
public final static int _SQL_TYPECODE = oracle.jdbc.OracleTypes.BLOB;
```

Note: Using the type map facility supersedes manually adding the `_SQL_TYPECODE` field to the class.

Limitations on Serializing Java Objects

You should be aware of the effect of serialization. If two objects, A and B, share the same object, C, then upon serialization and subsequent deserialization of A and B, each will point to its own clone of the object C. Sharing is broken.

In addition, note that for a given Java class, you can declare only one kind of serialization: either into `RAW` or into `BLOB`. The SQLJ translator can check only that the actual usage conforms to either `RAW` or `BLOB`.

`RAW` columns are limited in size—you may experience runtime errors if the actual size of the serialized Java object exceeds the size of the column.

Column size is much less restrictive for `BLOB` columns. As of Oracle9i release 2, writing a serialized Java object to a `BLOB` column is supported by the Oracle JDBC OCI and Thin drivers. (In Oracle9i release 1, this was supported by only the OCI driver.) Retrieving a serialized object from a `BLOB` column is supported by all Oracle JDBC drivers, for both release 1 and release 2.

Finally, treating serialized Java objects this way is an Oracle-specific extension and requires the Oracle SQLJ runtime as well as either the default Oracle-specific code generation (`-codegen=oracle` during translation) or, for ISO standard code generation (`-codegen=iso`), Oracle-specific profile customization. Note that future versions of Oracle may support SQL types that directly encapsulate Java serialized objects — these are described as `JAVA_OBJECT` SQL types in JDBC 2.0. At that point, you can replace each of the `BLOB` and `RAW` designations by the names of their corresponding `JAVA_OBJECT` SQL types, and you can drop the `oracle-` prefix on the entries.

Note: The implementation of this particular serialization mechanism does not use JDBC type maps. The map (to BLOB or to RAW) is hardcoded in the Oracle profile customization at translation time, or is generated directly into Java code.

SerializableDatum: an ORADData Implementation

"[Additional Uses for ORADData Implementations](#)" on page 6-18 includes examples of situations where you might want to define a custom Java class that maps to some `oracle.sql.*` type other than `oracle.sql.STRUCT`, `oracle.sql.REF`, or `oracle.sql.ARRAY`.

An example of such a situation is if you want to serialize and deserialize Java objects into and out of RAW fields, with a custom Java class that maps to the `oracle.sql.RAW` type. (This could apply equally to BLOB fields, with a custom Java class that maps to the `oracle.sql.BLOB` type.)

This section presents an example of such an application, creating a class `SerializableDatum` that implements the `ORADData` interface and follows the general form of custom Java classes, as described in "[Custom Java Classes](#)" on page 6-6.

The example starts with a step-by-step approach to the development of `SerializableDatum`, followed by the complete sample code.

Note: This application uses classes from the `java.io`, `java.sql`, `oracle.sql`, and `oracle.jdbc` packages. The import statements are not shown here.

1. Begin with a skeleton of the class.

```
public class SerializableDatum implements ORADData
{
    // <Client methods for constructing and accessing the Java object>

    public Datum toDatum(java.sql.Connection c) throws SQLException
    {
        // <Implementation of toDatum()>
    }

    public static ORADDataFactory getORADDataFactory()
```

```

    {
        return FACTORY;
    }

    private static final ORADDataFactory FACTORY =
        // <Implementation of an ORADDataFactory for SerializableDatum>

        // <Construction of SerializableDatum from oracle.sql.RAW>

    public static final int _SQL_TYPECODE = OracleTypes.RAW;
}

```

`SerializableDatum` does not implement the `ORADDataFactory` interface, but its `getORADDataFactory()` 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.

2. Define client methods that perform the following:

- Create a `SerializableDatum` object.
- Populate a `SerializableDatum` object.
- Retrieve data from a `SerializableDatum` object.

```

// Client methods for constructing and accessing a SerializableDatum

private Object m_data;
public SerializableDatum()
{
    m_data = null;
}
public void setData(Object data)
{
    m_data = data;
}
public Object getData()
{
    return m_data;
}

```

3. Implement a `toDatum()` method that serializes data from a `SerializableDatum` object to an `oracle.sql.RAW` object. The implementation of `toDatum()` must return a serialized representation of the object in the `m_data` field as an `oracle.sql.RAW` instance.

```
// Implementation of toDatum()
try {
    ByteArrayOutputStream os = new ByteArrayOutputStream();
    ObjectOutputStream oos = new ObjectOutputStream(os);
    oos.writeObject(m_data);
    oos.close();
    return new RAW(os.toByteArray());
} catch (Exception e) {
    throw new SQLException("SerializableDatum.toDatum: "+e.toString()); }
}
```

4. Implement data conversion from an `oracle.sql.RAW` object to a `SerializableDatum` object. This step deserializes the data.

```
// Constructing SerializableDatum from oracle.sql.RAW
private SerializableDatum(RAW raw) throws SQLException
{
    try {
        InputStream rawStream = new ByteArrayInputStream(raw.getBytes());
        ObjectInputStream is = new ObjectInputStream(rawStream);
        m_data = is.readObject();
        is.close();
    } catch (Exception e) {
        throw new SQLException("SerializableDatum.create: "+e.toString()); }
}
```

5. Implement an `ORADDataFactory`. In this case, it is implemented as an anonymous class.

```
// Implementation of an ORADDataFactory for SerializableDatum
new ORADDataFactory()
{
    public ORADData create(Datum d, int sqlCode) throws SQLException
    {
        if (sqlCode != _SQL_TYPECODE)
        {
            throw new SQLException
                ("SerializableDatum: invalid SQL type "+sqlCode);
        }
        return (d==null) ? null : new SerializableDatum((RAW)d);
    }
};
```

SerializableDatum in SQLJ Applications

Given the `SerializableDatum` class created in the preceding section, this section shows how to use an instance of it in a SQLJ application, both as a host variable and as an iterator column.

Presume the following table definition:

```
CREATE TABLE PERSONDATA (NAME VARCHAR2(20) NOT NULL, INFO RAW(2000));
```

SerializableDatum as Host Variable

The following uses a `SerializableDatum` instance as a host variable.

```
...
SerializableDatum pinfo = new SerializableDatum();
pinfo.setData (
    new Object[] { "Some objects", new Integer(51), new Double(1234.27) } );
String pname = "MILLER";
#sql { INSERT INTO persondata VALUES (:pname, :pinfo) };
...
```

SerializableDatum in Iterator Column

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.*;

public class SerializableDatum implements ORADData
{
    // Client methods for constructing and accessing a SerializableDatum

    private Object m_data;
    public SerializableDatum()
    {
        m_data = null;
    }
    public void setData(Object data)
    {
        m_data = data;
    }
    public Object getData()
    {
        return m_data;
    }

    // Implementation of toDatum()

    public Datum toDatum(Connection c) throws SQLException
    {
        try {
            ByteArrayOutputStream os = new ByteArrayOutputStream();
            ObjectOutputStream oos = new ObjectOutputStream(os);
            oos.writeObject(m_data);
            oos.close();
            return new RAW(os.toByteArray());
        } catch (Exception e) {
            throw new SQLException("SerializableDatum.toDatum: "+e.toString()); }
    }
}
```



```
public static ORADDataFactory getORADDataFactory()
{
    return FACTORY;
}

// Implementation of an ORADDataFactory for SerializableDatum

private static final ORADDataFactory FACTORY =

    new ORADDataFactory()
    {
        public ORADData create(Datum d, int sqlCode) throws SQLException
        {
            if (sqlCode != _SQL_TYPECODE)
            {
                throw new SQLException(
                    "SerializableDatum: invalid SQL type "+sqlCode);
            }
            return (d==null) ? null : new SerializableDatum((RAW)d);
        }
    };

// Constructing SerializableDatum from oracle.sql.RAW

private SerializableDatum(RAW raw) throws SQLException
{
    try {
        InputStream rawStream = new ByteArrayInputStream(raw.getBytes());
        ObjectInputStream is = new ObjectInputStream(rawStream);
        m_data = is.readObject();
        is.close();
    } catch (Exception e) {
        throw new SQLException("SerializableDatum.create: "+e.toString()); }
}

public static final int _SQL_TYPECODE = OracleTypes.RAW;
}
```

Weakly Typed Objects, References, and Collections

Weakly typed objects, references, and collections are supported by SQLJ. Their use is not generally recommended, and there are some specific restrictions, but in some circumstances they can be useful. For example, you might have generic code that can use "any STRUCT" or "any REF".

Support for Weakly Typed Objects, References, and Collections

In using Oracle objects, references, or collections in a SQLJ application, you have the option of using generic and weakly typed `java.sql` or `oracle.sql` instances instead of the strongly typed custom object, reference, and collection classes that implement the `ORADData` interface or the strongly typed custom object classes that implement the `SQLData` interface. (Note that if you use `SQLData` implementations for your custom object classes, you will have no choice but to use weakly typed custom reference instances.)

The following weak types can be used for iterator columns or host expressions in Oracle SQLJ:

- `java.sql.Struct` or `oracle.sql.STRUCT` for objects
- `java.sql.Ref` or `oracle.sql.REF` for object references
- `java.sql.Array` or `oracle.sql.ARRAY` for collections

In host expressions, they are supported as follows:

- as input host expressions
- as output host expressions in an INTO-list

Using these weak types is not generally recommended, however, as you would lose all the advantages of the strongly typed paradigm that SQLJ offers.

Each attribute in a STRUCT object or each element in an ARRAY object is stored in an `oracle.sql.Datum` object, with the underlying data being in the form of the appropriate `oracle.sql.*` subtype of `Datum` (such as `oracle.sql.NUMBER` or `oracle.sql.CHAR`). Attributes in a STRUCT object are nameless.

Because of the generic nature of the STRUCT and ARRAY classes, SQLJ cannot perform type checking where objects or collections are written to or read from instances of these classes.

It is generally recommended that you use custom Java classes for objects, references, and collections, preferably classes generated by JPublisher.

Restrictions on Weakly Typed Objects, References, and Collections

A weakly typed object (`Struct` or `STRUCT` instance), reference (`Ref` or `REF` instance), or collection (`Array` or `ARRAY` instance) *cannot* be used in host expressions in the following circumstances:

- `IN` parameter if null
- `OUT` or `INOUT` parameter in a stored procedure or function call
- `OUT` parameter in a stored function result expression

They cannot be used in these ways because there is no way to know the underlying SQL type name (such as `Person`), which is required by the Oracle JDBC driver to materialize an instance of a user-defined type in Java.

Oracle OPAQUE Types

Oracle OPAQUE types are abstract data types. With data implemented as simply a series of bytes, the internal representation is not exposed. Typically an OPAQUE type will be provided by Oracle, not implemented by a customer.

OPAQUE types are similar in some basic ways to object types, with similar concepts of static methods, instances, and instance methods. Typically, only the methods supplied with an OPAQUE type allow you to manipulate the state and internal byte representation. In Java, an OPAQUE type can be represented as

`oracle.sql.OPAQUE` or as a custom class implementing the

`oracle.sql.ORAData` interface. On the client side, Java code can be implemented to manipulate the bytes, assuming the byte pattern is known. The Oracle9i *JPublisher* utility can be useful in this way, creating a custom class implementing `ORAData` to allow you to manipulate data without having to make repeated round trips to the database. See the *Oracle9i JPublisher User's Guide* for more information.

A key example of an OPAQUE type is `XMLType`, provided with Oracle9i. This Oracle-supplied type facilitates handling XML data natively in the database.

`SYS.XMLType` offers the following features, exposed through the Java `oracle.xdb.XMLType` class:

- It can be used as the datatype of a column in a table or view. `XMLType` can store any content but is designed to optimally store XML content. An instance of it can represent an XML document in SQL.
- It has a SQL API with built-in member functions that operate on XML content. For example, you can use `XMLType` functions to create, query, extract, and index XML data stored in an Oracle9i database.
- It can be used in stored procedures for parameters, return values, and variables.
- Its functionality is also available through APIs provided in PL/SQL, Java, and C (OCI).

`XMLType` is discussed in detail in the *Oracle9i XML Database Developer's Guide - Oracle XML DB*.

Advanced Language Features

This chapter discusses advanced SQLJ language features for use 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](#)
- [Support for Dynamic SQL](#)

Connection Contexts

SQLJ supports the concept of *connection contexts*, allowing strongly typed connections for use with different sets of SQL entities. You can think of a connection context as being associated with a particular set of SQL entities such as tables, views, and stored procedures. SQLJ lets you declare additional connection context classes so that you can use each class for connections that use a particular set of SQL entities. Different instances of a single connection context class are not required to use the same physical entities or connect to the same schema, but will at least use sets of entities with the same names and datatypes.

Note: For an overview of connection basics, focusing on situations where you are using just a single set of SQL entities and a single connection context class, see "[Connection Considerations](#)" on page 4-6.

Connection Context Concepts

If your application uses different sets of SQL entities, then you will typically want to declare and use one or more additional connection context classes, as discussed in "[Overview of SQLJ Declarations](#)" on page 3-2. Each connection context class can be used for a particular set of interrelated SQL entities, meaning that all the connections you define using a particular connection context class will use tables, views, stored procedures, and so on, that have the same names and use the same datatypes.

An example of a set of SQL entities 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 set of SQL entities 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 the stored procedures `GIVE_RAISE` and `CHANGE_WITHHOLDING`.

The advantage in tailoring connection context classes to sets of SQL entities is in the degree of online semantics-checking that this allows. Online checking verifies that all the SQL entities appearing in SQLJ statements that use a given connection context class match SQL entities found in the *exemplar schema* used during translation. An exemplar schema is a database account that SQLJ connects to for online checking of all the SQLJ statements that use a particular connection context class. You provide exemplar schemas to the translator through the SQLJ command-line `-user`, `-password`, and `-url` options. (See "[Connection Options](#)")

on page 8-34 for information about these options.) An exemplar schema might or might not be the same account your application will use at runtime.

If you have SQLJ statements that use a broad and perhaps unrelated group of SQL entities, but you use only a single connection context class for these statements, then the exemplar schema you provide must be very general. It must contain all the tables, views, and stored procedures used throughout all the statements. Alternatively, if all the SQLJ statements using a given connection context class use a tight, presumably interrelated, set of SQL entities, then you can provide a more specific exemplar schema that allows more thorough and meaningful semantics-checking.

Notes:

- Be aware that a connection context class declaration does not define a set of SQL entities to be used with the declared connection context class, and it is permissible to use the same connection context class for connections that use disparate and unrelated sets of entities. How you use your connection context classes is at your discretion. All that limits the SQL entities you can use with a particular connection context class are the set of entities available in the exemplar schema (if you use online semantics-checking during translation) and the set of entities available in the schema you connect to at runtime, using instances of the connection context class.
 - If you use qualified SQL names in your application—names such as `SCOTT.EMP`, which specifies the schema where the entity resides—then the exemplar schema (if you use online checking) and runtime schema must have permission to access resources by these fully qualified names.
 - It is possible to use a single connection context class, even for connections to databases from different vendors, as long as each schema you connect to has entities that are accessible by the same names and that use compatible datatypes.
-
-

Connection Context Logistics

Declaring a connection context class results in the SQLJ translator defining a class for you in the translator-generated code. In addition to any connection context classes that you declare, there is always the default connection context class:

```
sqlj.runtime.ref.DefaultContext
```

When you construct a connection context instance, specify a particular schema (user name, password, and URL) and a particular session and transaction in which SQL operations will execute. You typically accomplish this by specifying a user name, password, and database URL as input to the constructor of the connection context class. The connection context instance manages the set of SQL operations performed during the session.

In each SQLJ statement, you can specify a connection context instance to use, as discussed in ["Specifying a Connection Context Instance for a SQLJ Clause"](#) on page 7-7.

The following example shows basic declaration and use of a connection context class, `MyContext`, to connect to two different schemas. For typical usage, assume these schemas include a set of SQL entities with common names and datatypes.

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 ["More About Declaring and Using a Connection Context Class"](#) on page 7-5.

In addition, note that you can connect to the same schema with different connection context instances. In the example above, both `mctx1` and `mctx2` could specify `scott/tiger` if desired. During runtime, however, one connection context instance would not see changes to the database made from the other until the changes are committed. The only exception to this would be if both connection context instances were created from the same underlying JDBC connection instance. (One of the constructors of any connection context class takes a JDBC connection instance as input.)

More About Declaring and Using a Connection Context Class

This section gives a detailed example of how to declare a connection context class, then define a database connection using an instance of the class.

A connection context class has constructors for opening a connection to a database schema, given any of the following (as with the `DefaultContext` class):

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

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 4-26.
 - If a connection context class is declared with a data source `with` clause, then it incorporates a different set of constructors. See "[Standard Data Source Support](#)" on page 7-13 for more information.
-
-

Declaring the Connection Context Class

The following declaration creates a connection context class:

```
#sql context OrderEntryCtx <implements_clause> <with_clause>;
```

This results in the SQLJ translator generating a class that implements the `sqlj.runtime.ConnectionContext` interface and extends some base class (probably an abstract class) that also implements the `ConnectionContext` interface. This base class would be a feature of the particular SQLJ implementation you are using.

The `implements` clause and `with` clause are optional, specifying additional interfaces to implement and variables to define and initialize, respectively. See ["Declaration IMPLEMENTS Clause"](#) on page 3-5 and ["Declaration WITH Clause"](#) on page 3-6. For information about data source `with` clauses in particular, see ["Standard Data Source Support"](#) on page 7-13.

The following is an example of what the SQLJ translator generates (with method implementations omitted):

```
class OrderEntryCtx implements sqlj.runtime.ConnectionContext
    extends ...
{
    public OrderEntryCtx(String url, Properties info, boolean autocommit)
        throws SQLException {...}
    public OrderEntryCtx(String url, boolean autocommit)
        throws SQLException {...}
    public OrderEntryCtx(String url, String user, String password,
        boolean autocommit) throws SQLException {...}
    public OrderEntryCtx(Connection conn) throws SQLException {...}
    public OrderEntryCtx(ConnectionContext other) throws SQLException {...}

    public static OrderEntryCtx getDefaultContext() {...}
    public static void setDefaultContext(OrderEntryCtx ctx) {...}
}
```

Creating a Connection Context Instance

Continuing the preceding example, instantiate the `OrderEntryCtx` class with the following syntax:

```
OrderEntryCtx myOrderConn = new OrderEntryCtx
    (url, username, password, autocommit);
```

For example:

```
OrderEntryCtx myOrderConn = new OrderEntryCtx
    ("jdbc:oracle:thin:@localhost:1521:orcl", "scott", "tiger", true);
```

This is accomplished in the same way as instantiating the `DefaultContext` class. All connection context classes, including `DefaultContext`, have the same constructor signatures.

Notes:

- You typically must register your JDBC driver prior to constructing a connection context instance. See ["Driver Selection and Registration for Runtime"](#) on page 4-5.
 - If a connection context class is declared with a data source `with` clause, then it incorporates a different set of constructors. See ["Standard Data Source Support"](#) on page 7-13 for more information.
-
-

Specifying a Connection Context Instance for a SQLJ Clause

Recall that the basic SQLJ statement syntax is as follows:

```
#sql <[<conn><, ><exec>]> { SQL operation };
```

Specify the connection context instance inside square brackets following the `#sql` token. For example, in the following SQLJ statement, the connection context instance is `myOrderConn` from the previous example:

```
#sql [myOrderConn] { UPDATE TAB2 SET COL1 = :w WHERE :v < COL2 };
```

In this way, you can specify an instance of either the `DefaultContext` class or any declared connection context class.

Closing a Connection Context Instance

It is advisable to close all connection context instances when you are done. Each connection context class includes a `close()` method, as discussed for the `DefaultContext` class in ["Closing Connections"](#) on page 4-11.

In closing a connection context instance that shares the underlying connection with another connection instance, you might want to keep the underlying connection open. See ["Closing Shared Connections"](#) on page 7-57.

Example of Multiple Connection Contexts

The following is an example of a SQLJ application using multiple connection contexts. It implicitly uses an instance of the `DefaultContext` class for one set of SQL entities, and uses an instance of the declared connection context class `DeptContext` for another set of SQL entities.

This example uses the static `Oracle.connect()` method to establish a default connection, then constructs an additional connection by using the static `Oracle.getConnection()` method to pass another `DefaultContext` instance to the `DeptContext` constructor. As previously mentioned, this is just one of several ways you can construct a SQLJ connection context instance.

```
import java.sql.SQLException;
import oracle.sqlj.runtime.Oracle;

// declare a new context class for obtaining departments
#sql context DeptContext;

#sql iterator Employees (String ename, int deptno);

class MultiSchemaDemo
{
    public static void main(String[] args) throws SQLException
    {
        /* 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 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 this interface include the following:

- `close(boolean CLOSE_CONNECTION/KEEP_CONNECTION)`—Releases all resources used in maintaining this connection and closes any open connected profiles. It might or might not close the underlying JDBC connection, depending on whether `CLOSE_CONNECTION` or `KEEP_CONNECTION` is specified. These are static boolean constants of the `ConnectionContext` interface.

For further discussion, see "[Closing Shared Connections](#)" on page 7-57.

- `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 7-24.

Additional Connection Context Class Methods

In addition to the methods specified and defined in the `ConnectionContext` interface, each connection context class defines the following methods:

- `YourCtxClass getDefaultContext()`—This is a static method that returns the default connection context instance for a given connection context class.
- `setDefaultContext(YourCtxClass connctxinstance)`—This is a static method that defines the given connection context instance as the default connection context instance for its class.

Although it is true that you can use an instance of only the `DefaultContext` class as your default connection, it might still be useful to designate an instance of a declared connection context class as the default context for that class, using the `setDefaultContext()` method. Then you could conveniently retrieve it using the `getDefaultContext()` method of the particular class. This would 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, autocommit);  
...  
MyContext.setDefaultContext(myctx1);  
...  
#sql [MyContext.getDefaultContext()] { SQL operations };  
...
```

Additionally, each connection context class defines methods for control of SQLJ statement caching. The following are static methods:

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

And the following are instance methods:

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

By default, statement caching is enabled. See "[Connection Context Methods for Statement Caching \(Oracle-Specific Code\)](#)" on page 10-5 for more information. (This is a subsection under "[Statement Caching](#)" on page 10-4, which provides an overview of statement caching.)

Using 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 (presuming `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 strong typing of connections, with each connection context class typically used for operations on a particular set of interrelated SQL entities. This doesn't mean that all the connection instances of a single class use the same physical entities, but that they use entities that have the same properties, such as names and privileges associated with tables and views, datatypes of their rows, and names and definitions of stored procedures. This strong typing allows SQLJ semantics-checking to verify during translation that you are using your SQL operations correctly, with respect to your database connections.

To use online semantics-checking during translation, provide a sample schema (that includes an appropriate set of SQL entities) for each connection context class. These sample schemas are referred to as *exemplar schemas*. Provide exemplar schemas through an appropriate combination of the SQLJ `-user`, `-password`, and `-url` options. Following are two examples, one for the `DefaultContext` class and one for a declared connection context class, where the user, password, and URL are all specified through the `-user` option:

```
-user=scott/tiger@jdbc:oracle:oci:@  
-user@MyContext=scott/tiger@jdbc:oracle:oci:@
```

(For information about these SQLJ options, see "[Connection Options](#)" on page 8-34.)

During semantics-checking, the translator connects to the specified exemplar schema for a particular connection context class and accomplishes the following:

- It examines each SQLJ statement in your code that specifies an instance of the connection context class and checks its SQL operations (such as what tables you access and what stored procedures you use).
- It verifies that entities in the SQL operations match the set of entities existing in the exemplar schema.

It is your responsibility to pick an exemplar schema that represents the runtime schema in appropriate ways. For example, it must have tables, views, stored functions, and stored procedures with names and datatypes that match what are used in your SQL operations, and with privileges set appropriately.

If no appropriate exemplar schema is available during translation for one of your connection context classes, then it is not necessary to specify SQLJ translator options (`-user`, `-password`, `-url`) for that particular connection context class. In that case, SQLJ statements specifying connection objects of that connection context class are semantically checked only to the extent possible.

Note: Remember that the exemplar schema you specify in your translator option settings does not specify the schema to be used at runtime. The exemplar schema furnishes the translator only with a set of SQL entities to compare against the entities you use in your SQLJ executable statements.

Standard Data Source Support

The JDBC 2.0 extended API specifies the use of *data sources* and JNDI as a portable alternative to the `DriverManager` mechanism for obtaining JDBC connections. It permits database connections to be established through a JNDI name lookup. This name is bound to a particular database and schema prior to program runtime through a `javax.sql.DataSource` object, typically installed through a GUI JavaBeans deployment tool. The name can be bound to different physical connections without any source code changes simply by rebinding the name in the directory service.

SQLJ uses the same mechanism to create connection context instances in a flexible and portable way. Data sources can also be implemented using a connection pool or distributed transaction service, as defined by the JDBC 2.0 extended API.

For more information about data sources, see the *Oracle9i JDBC Developer's Guide and Reference*.

Associating a Connection Context with a Data Source

In SQLJ it is natural to associate a connection context class with a logical schema, in much the same way that a data source name serves as a symbolic name for a JDBC connection. Combine both concepts by adding the data source name to the connection context declaration.

```
#sql context EmpCtx with (dataSource="jdbc/EmpDB");
```

Any connection context class that you declare with a `dataSource` property provides additional constructors. To continue the `EmpCtx` example, the following constructors are provided:

- `public EmpCtx()`—Looks up the data source for `jdbc/EmpDB` and then calls the `getConnection()` method on the data source to obtain a connection.
- `public EmpCtx(String user, String password)`—Looks up the data source for `jdbc/EmpDB` and calls the `getConnection(user, password)` method on the data source to obtain a connection.

- `public EmpCtx(ConnectionContext ctx)`—Delegates to `ctx` to obtain a connection.

Any connection context class declared with a `dataSource` property also omits a number of `DriverManager`-based constructors. Continuing the `EmpCtx` example, the following constructors are omitted:

- `public EmpCtx(Connection conn)`
- `public EmpCtx(String url, String user, String password, boolean autoCommit)`
- `public EmpCtx(String url, boolean autoCommit)`
- `public EmpCtx(String url, java.util.Properties info, boolean autoCommit)`
- `public EmpCtx(String url, boolean autoCommit)`

Auto-Commit Mode for Data Source Connections

Unlike the `DriverManager`-based constructors they replace, the new data-source-based constructors do not include an explicit auto-commit parameter. They always use the auto-commit mode defined by the data source.

Data sources are configured to have a default auto-commit mode depending on the deployment scenario. For example, data sources in the server and middle tier typically have auto-commit off; those on the client may have it on. However, it is also possible to configure data sources with a specific auto-commit setting. This permits data sources to be configured for a particular application and deployment scenario. Contrast this with JDBC URLs that may specify only a single database/driver configuration.

Programs can verify and possibly override the current auto-commit setting with the JDBC connection that underlies their connection context instance.

Note: Be aware of the auto-commit status of the connections you establish.

- If you use the `Oracle` class, auto-commit is off unless you turn it on explicitly.
 - If you use `DefaultContext` or a connection context class with `DriverManager`-style constructors, then the auto-commit setting must always be specified explicitly.
 - If you use the data source mechanism, then the auto-commit setting is inherited from the underlying data source. In most environments, the data source object originates from JDBC, and the auto-commit option is on. To avoid unexpected behavior, always check the auto-commit setting.
-
-

Associating a Data Source with the Default Context

If a SQLJ program accesses the default connection context, and the default context has not yet been set, then the SQLJ runtime will use the SQLJ default data source to establish its connection. The SQLJ default data source is bound to the JNDI name "jdbc/defaultDataSource".

This mechanism provides a portable means to define and install a default JDBC connection for the default SQLJ connection context.

Data Source Support Requirements

For your program to use data sources, you must supply the packages `javax.sql.*` and `javax.naming.*`, and an `InitialContext` provider in your Java environment. The latter is required to obtain the JNDI context in which the SQLJ runtime can look up the data source object.

Typically, you would use data sources in a JDK 1.2.x environment with the Java Extension classes, or in a J2EE environment. However, you can also use data sources under JDK 1.1.x with the Java Extension classes.

All SQLJ runtime libraries provided by Oracle support data sources. However, if you use the `runtime12ee` library you must have `javax.sql.*` and `javax.naming.*` in your classpath in order for the runtime to load. By contrast, the other runtime libraries use reflection to retrieve `DataSource` objects.

SQLJ-Specific Data Sources

As of Oracle9i release 2, Oracle SQLJ provides SQLJ-specific data source support in the `runtime12ee` library. Currently, SQLJ-specific data sources can be used in client-side or middle-tier applications, but not inside the server.

SQLJ-specific data sources extend JDBC data source functionality with methods that return SQLJ connection context instances. This enables a SQLJ developer to manage connection contexts just as a JDBC developer manages connections. In general, each SQLJ-specific data source interface or class is based on a corresponding standard JDBC data source interface or Oracle JDBC data source class.

The rest of this section describes the SQLJ-specific data source interfaces and classes, then concludes with examples of their use.

SQLJ Data Source Interfaces

The `sqlj.runtime.ConnectionContextFactory` interface acts as a base interface for SQLJ data source functionality. It is implemented by a set of more specialized Oracle data source interfaces that add support for features such as connection pooling, connection caching, or distributed transactions.

The `ConnectionContextFactory` interface specifies methods, listed below, to return SQLJ connection context instances. The `getDefaultContext()` methods return a `sqlj.runtime.ref.DefaultContext` instance for the SQLJ default context. The `getContext()` methods return a `sqlj.runtime.ConnectionContext` instance—specifically, an instance of a user-declared connection context class that is specified in the method call.

For both `getDefaultContext()` and `getContext()` there are signatures that allow you to specify connection parameters for the JDBC connection that underlies the connection context instance—the auto-commit setting, user and password settings, or all three. If you do not specify the user and password, they are obtained from the underlying data source that generates the connection. If you do not specify an auto-commit setting, the default is `false` unless it was explicitly set to `true` for the underlying data source.

- `DefaultContext getDefaultContext()`
- `DefaultContext getDefaultContext(boolean autoCommit)`
- `DefaultContext getDefaultContext(String user,
String password)`
- `DefaultContext getDefaultContext(String user,
String password,boolean autoCommit)`

- `ConnectionContext getContext(Class aContextClass)`
- `ConnectionContext getContext(Class aContextClass, boolean autoCommit)`
- `ConnectionContext getContext(Class aContextClass, String user, String password)`
- `ConnectionContext getContext(Class aContextClass, String user, String password, boolean autoCommit)`

Each Oracle data source interface that implements `ConnectionContextFactory` also implements a standard JDBC data source interface to specify methods for the appropriate functionality, such as for basic data sources, connection pooling data sources, or distributed transaction (XA) data sources. Oracle has implemented the `SqljDataSource`, `SqljConnectionPoolDataSource`, and `SqljXADataSource` interfaces, located in the `sqlj.runtime` package and specified as follows:

- ```
public interface SqljDataSource extends
 javax.sql.DataSource, ConnectionContextFactory { }
```
- ```
public interface SqljDataSource extends
    javax.sql.ConnectionPoolDataSource,
    ConnectionContextFactory { }
```
- ```
public interface SqljXADataSource extends
 javax.sql.XADataSource, ConnectionContextFactory { }
```

### SQLJ Data Source Classes

Oracle provides SQLJ-specific counterparts for the following Oracle JDBC data source classes: `OracleDataSource`, `OracleConnectionPoolDataSource`, `OracleXADataSource`, `OracleConnectionCacheImpl`, `OracleXAConnectionCacheImpl`, and `OracleOCIConnectionPool`. See the *Oracle9i JDBC Developer's Guide and Reference* for information about these classes.

Oracle SQLJ-specific data source classes are located in two packages: `oracle.sqlj.runtime` and `oracle.sqlj.runtime.client`.

The `oracle.sqlj.runtime` package includes the following:

- ```
public class OracleSqljDataSource
    extends oracle.jdbc.pool.OracleDataSource
    implements ConnectionContextFactory
```

Note: The `OracleSqljDataSource` class implements the `java.io.Serializable` interface. It is therefore serializable and can be used in clustered environments such as Oracle9iAS Containers for J2EE (OC4J).

- `public class OracleSqljConnectionPoolDataSource`
extends `oracle.jdbc.pool.OracleConnectionPoolDataSource`
implements `ConnectionFactory`
- `public abstract class OracleSqljXADataSource`
extends `oracle.jdbc.xa.OracleXADataSource`
implements `ConnectionFactory`;
- `public class OracleSqljConnectionCacheImpl`
extends `oracle.jdbc.pool.OracleConnectonCacheImpl`
implements `ConnectionFactory`
- `public class OracleSqljXAConnectionCacheImpl`
extends `oracle.jdbc.pool.OracleXAConnectonCacheImpl`
implements `ConnectionFactory`
- `public class OracleSqljOCIConnectionPool`
extends `oracle.jdbc.pool.OracleOCIConnectonPool`
implements `ConnectionFactory`

The `oracle.sqlj.runtime.client` package includes the following:

- `public class OracleSqljXADataSource`
extends `oracle.jdbc.xa.client.OracleXADataSource`
implements `ConnectionFactory`

You can use these classes in place of the corresponding JDBC classes that they extend. They include `getDefaultContext()` and `getContext()` methods as described in "[SQLJ Data Source Interfaces](#)" on page 7-16. When you call these methods, the following steps take place for you:

1. A new logical JDBC connection is acquired from the present data source.
2. A connection context instance is created from the logical connection, and returned.

Examples: Using SQLJ Data Sources

When used in middle-tier environments, SQLJ-specific data sources, like JDBC data sources, are bound to JNDI locations. You can do the binding explicitly, as in the following example:

```
//Initialize datasource
SqljXADatasource sqljDS = new OracleSqljXADatasource();
sqljDS.setUser("scott");
sqljDS.setPassword("tiger");
sqljDS.setServerName("myserver");
sqljDS.setDatabaseName("ORCL");
sqljDS.setDataSourceName("jdbc/OracleSqljXADS");

//Bind the datasource to JNDI
Context ctx = new InitialContext();
ctx.bind("jdbc/OracleSqljXADS");
```

In a middle-tier Oracle9iAS Containers for J2EE (OC4J) environment, another alternative is to instantiate data sources and bind them to JNDI through settings in the `j2ee/home/config/data-sources.xml` file. For example, the following `<data-source>` element in that file creates an `OracleSqljXADatasource` instance and binds it to the JNDI location `jdbc/OracleSqljXADS`:

```
<data-source
  class="oracle.sqlj.runtime.OracleSqljXADatasource"
  name="jdbc/OracleSqljXADS"
  location="jdbc/OracleSqljXADS"
  xa-location="jdbc/OracleSqljXADS/xa"
  username="scott"
  password="tiger"
  url="jdbc:oracle:thin:@dlsun960:1521:ORCL"
/>
```

See the *Oracle9iAS Containers for J2EE Services Guide* for information about data sources in OC4J.

A SQLJ-specific data source bound to a JNDI location can be looked up and used in creating connection context instances. The following code segment uses information from the preceding `<data-source>` element to create connection context instances—a `DefaultContext` instance and an instance of a user-declared class `MyCtx`, respectively:

```
sqlj.runtime.SqljDataSource sqljDS;
InitialContext initCtx = new InitialContext();
sqljDS = (sqlj.runtime.SqljDataSource)initCtx.lookup("jdbc/OracleSqljXADS");
```

```
// getDefaultContext
DefaultContext ctx = sqljDS.getDefaultContext();
// getContext
/* Declare MyCtx connection context class. You could optionally use a "with"
   clause to specify any desired connection parameters not available
   through the underlying data source.
*/
#sql public static context MyCtx;
MyCtx ctx = (MyCtx) sqljDS.getContext(MyCtx.class);
```

SQLJ-Specific Connection JavaBeans for JavaServer Pages

Oracle has implemented a set of JavaBeans for database connections from within JSP pages. The original beans, `ConnBean` and `ConnCacheBean` in package `oracle.jsp.dbutil`, are documented in the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*.

As of Oracle9i release 2, Oracle SQLJ provides the following extensions of these JavaBeans in the `runtime12ee` library for use in SQLJ JSP pages:

- `oracle.sqlj.runtime.SqljConnBean`
- `oracle.sqlj.runtime.SqljConnCacheBean`

`ConnBean` and `ConnCacheBean` include methods that return JDBC connection objects. `SqljConnBean` and `SqljConnCacheBean` extend this functionality to support a bean property called `ContextClass` of type `String` and to return SQLJ connection context instances.

Note: The `SqljConnBean` class implements the `java.io.Serializable` interface. It is therefore serializable and can be used in clustered environments such as Oracle9iAS Containers for J2EE (OC4J).

`SqljConnBean` and `SqljConnCacheBean` provide the following methods:

- `void setContextClass(String contextClassName)`
- `String getContextClass()`
- `DefaultContext getDefaultContext()`
- `ConnectionContext getContext()`

The `ContextClass` property specifies the name of a user-declared connection context class, if you are not using `DefaultContext`. You can set this property through the `setContextClass()` method.

To retrieve a connection context instance, use `getDefaultContext()` or `getContext()`, as appropriate. The former returns a `sqlj.runtime.ref.DefaultContext` instance; the latter returns a `sqlj.runtime.ConnectionContext` instance—specifically, an instance of the class specified in the `ContextClass` property (by default, `DefaultContext`).

Note, however, that the `getDefaultContext()` and `getContext()` methods are implemented differently between `SqljConnBean` and `SqljConnCacheBean`, as described in the following subsections. The discussion concludes with a sample SQLJ JSP page using `SqljConnCacheBean`.

Behavior of `SqljConnBean` (Simple Connections)

A `SqljConnBean` instance can wrap only one logical JDBC connection and one SQLJ connection context instance at any given time.

The first `getDefaultContext()` or `getContext()` method call will create and return a connection context instance based on the underlying JDBC connection. This connection context instance will also be stored in the `SqljConnBean` instance.

Once a connection context instance has been created and stored, the behavior of subsequent `getDefaultContext()` or `getContext()` calls will depend on the type of the stored connection context and, for `getContext()`, on the connection context type specified in the `ContextClass` property, as follows:

- For subsequent `getDefaultContext()` calls if the stored connection context instance is a `DefaultContext` instance: The method will keep returning that instance.
- For subsequent `getDefaultContext()` calls if the stored connection context instance is *not* a `DefaultContext` instance: The method will close the stored connection context instance and reuse the underlying JDBC connection to create and return a new connection context as a `DefaultContext` instance (regardless of the previous connection context type). This becomes the new connection context instance stored in the `SqljConnBean` instance.
- For subsequent `getContext()` calls if the stored connection context instance is of the same type as that specified by the `ContextClass` property: The method will keep returning that instance.

- For subsequent `getContext()` calls if the stored connection context instance is *not* of the same type as that specified by `ContextClass`: The method will close the stored connection context instance and reuse the underlying JDBC connection to create and return a new connection context instance—an instance of what is specified in `ContextClass`. This becomes the new connection context instance stored in the `SqljConnBean` instance.

Note: When `SqljConnBean` closes a connection context instance, it does so with the `KEEP_CONNECTION` setting, leaving the underlying JDBC connection intact. See "[Closing Shared Connections](#)" on page 7-57 for related information.

Behavior of `SqljConnCacheBean` (Connection Caching)

Unlike with `SqljConnBean`, the `SqljConnCacheBean` JavaBean creates and returns a new connection context instance, based on a new logical JDBC connection, for each invocation of `getDefaultContext()` or `getContext()`. The connection context type will be `DefaultContext` for a `getDefaultContext()` call, or the type specified in the `ContextClass` property for a `getContext()` call.

`SqljConnCacheBean` does not store the connection context instances it creates.

Example: SQLJ JSP Page Using `SqljConnCacheBean`

The following program, `SQLJSelectInto.sqljsp`, uses `SqljConnCacheBean`, its `ContextClass` bean property, and its `getContext()` method.

Note: This example uses the `ContextClass` property for illustrative purposes. Be aware, however, that `DefaultContext` is the default value anyway, and that if you want to use `DefaultContext`, then the value of `ContextClass` is irrelevant if you use `getDefaultContext()` instead of `getContext()`.

```
<%@ page language="sqlj"
    import="java.sql.*, oracle.sqlj.runtime.SqljConnCacheBean" %>
<jsp:useBean id="cbean" class="oracle.sqlj.runtime.SqljConnCacheBean"
    scope="session">
    <jsp:setProperty name="cbean" property="User" value="scott"/>
    <jsp:setProperty name="cbean" property="Password" value="tiger"/>
    <jsp:setProperty name="cbean" property="URL"
        value="jdbc:oracle:thin:@pdcsun-dev3:1521:view13"/>
```

```

        <jsp:setProperty name="cbean" property="ContextClass"
                        value="sqlj.runtime.ref.DefaultContext"/>
    </jsp:useBean>
    <HTML>
    <HEAD> <TITLE> The SQLJSelectInto JSP </TITLE> </HEAD>
    <BODY BGCOLOR=white>
    <% String empno = request.getParameter("empno");
       if (empno != null) { %>
        <H3> Employee # <%=empno %> Details: </H3>
        <% String ename = null; double sal = 0.0; String hireDate = null;
           StringBuffer sb = new StringBuffer();
           sqlj.runtime.ref.DefaultContext ctx=null;
           try {
               // Make the Connection
               ctx = (sqlj.runtime.ref.DefaultContext) cbean.getContext();
           } catch (SQLException e) {
           }
           try {
               #sql [ctx] { SELECT ename, sal, TO_CHAR(hiredate, 'DD-MON-YYYY')
                           INTO :ename, :sal, :hireDate
                           FROM scott.emp WHERE UPPER(empno) = UPPER(:empno)
                           };
               sb.append("<BLOCKQUOTE><BIG><B><PRE>\n");
               sb.append("Name          : " + ename + "\n");
               sb.append("Salary       : " + sal + "\n");
               sb.append("Date hired  : " + hireDate);
               sb.append("</PRE></B></BIG></BLOCKQUOTE>");
           } catch (java.sql.SQLException e) {
               sb.append("<P> SQL error: <PRE> " + e + " </PRE> </P>\n");
           } finally {
               if (ctx!= null) ctx.close();
           }
           %>
        <H3><%=sb.toString()%></H3>
    <%}
    %>
    <B>Enter an employee number:</B>
    <FORM METHOD=get>
    <INPUT TYPE="text" NAME="empno" SIZE=10>
    <INPUT TYPE="submit" VALUE="Ask Oracle">
    </FORM>
    </BODY>
    </HTML>

```

Execution Contexts

An *execution context* is an instance of the `sqlj.runtime.ExecutionContext` class and provides a context in which SQL operations are executed. An execution context instance is associated either implicitly or explicitly with each SQL operation in your SQLJ application.

The `ExecutionContext` class contains methods for the following features:

- Execution control operations modify the semantics of subsequent SQL operations.
- Execution status operations describe the results of the most recent SQL operation.
- Execution cancellation operations terminate the SQL operation that is currently executing.
- Update-batching operations enable and disable update batching, set the batch limit, and get update counts. (For information about update batching, see ["Update Batching"](#) on page 10-12.)
- Savepoint operations set a savepoint, roll back to a savepoint, and release a savepoint.
- Closure operations close the execution context instance to avoid resource leakage.

Note: There is only one execution context class, unlike connection context classes where you declare additional classes as desired. Every execution context is an instance of the `ExecutionContext` class. So while the term *connection context* usually refers to a *class* that you have declared, the term *execution context* always refers to an *instance* of the `ExecutionContext` class. This document specifies *connection context class*, *connection context instance*, and *execution context instance* to avoid confusion.

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 might appear to be associated with connection context instances (given that each connection context instance has a default execution context instance, and you can specify a connection context instance and an execution context instance together for a particular SQLJ statement), they actually operate independently. You can employ different execution context instances in statements that employ the same connection context instance, and vice versa.

For example, it is useful to use multiple execution context instances with a single connection context instance if you use multithreading, with a separate execution context instance for each thread. And you can use multiple connection context instances with a single explicit execution context instance if your program is single-threaded and you want the same set of SQL control parameters to apply to all the connection context instances. (See ["Execution Context Methods"](#) on page 7-28 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.

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. There are no input parameters for the `ExecutionContext` 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 also 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 the default connection and its default execution context instance.
-
-

Execution Context Synchronization

`ExecutionContext` methods (discussed in ["Execution Context Methods"](#) on page 7-28) are all `synchronized` methods. Therefore, for ISO standard code generation, anytime a statement tries to use an execution context instance (in essence, tries to use a method of an execution context instance) already in use, the second statement will be blocked until the first statement completes.

In a client application, this typically involves multithreading situations. A thread that tries to use an execution context instance currently in use by another thread will be blocked.

To avoid such blockage, you must specify a separate execution context instance for each thread that you use, as discussed in ["Multithreading in SQLJ"](#) on page 7-35.

The preceding discussion does not apply for default Oracle-specific code generation, however (`-codegen=oracle`). For performance reasons, SQLJ performs no additional synchronization against `ExecutionContext` instances for Oracle-specific generated code. Therefore, you are responsible for ensuring that the same execution context instance will not be used by more than one thread. If multiple threads use the same execution context, then your application, rather than blocking, will experience errors such as incorrect results or `NullPointerException` exceptions.

Another exception to the discussion is for recursion, which is encountered only in the server. Multiple SQLJ statements in the same thread are allowed to simultaneously use the same execution context instance if this situation results from recursive calls. An example of this is where a SQLJ stored procedure or function has a call to another SQLJ stored procedure or function. If both use the default execution context instance, as is typical, then the SQLJ statements in the second procedure will use this execution context while the SQLJ call statement from the first procedure is also still using it. This is allowed, and is further discussed in ["Recursive SQLJ Calls in the Server"](#) on page 11-26.

Execution Context Methods

This section lists the methods of the `ExecutionContext` class, categorized as status methods, control methods, cancellation method, update batching methods, savepoint methods, and closure method.

Status Methods

Use the following methods of an execution context instance to obtain status information about the most recent SQL operation that completed using that instance:

- `SQLWarning getWarnings()`—Returns a `java.sql.SQLWarning` object containing the first warning reported by the most recent SQL operation that completed using this execution context instance. Warnings are returned in a chain—use the `getWarnings()` method of the execution context instance to get the first warning, then use the `getNextWarning()` method of each `SQLWarning` object to get the next warning. The chain contains all warnings generated during the execution of the SQL operation.
- `int getUpdateCount()`—Except when update batching is enabled, this returns an `int` value specifying the number of rows updated by the last SQL operation that completed using this execution context instance. Zero (0) is returned if the last SQL operation was not a 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.

For batch-enabled applications, the value returned by `getUpdateCount()` would be one of several batch-related constant values—`NEW_BATCH_COUNT`, `ADD_BATCH_COUNT`, or `EXEC_BATCH_COUNT`. See ["Execution Context Update Counts"](#) on page 10-18 for more information.

Control Methods

Use the following methods of an execution context instance to control the operation of future SQL operations executed using that instance (operations that have not yet started):

- `int getMaxFieldSize()`—Returns an `int` value specifying the maximum amount of data (in bytes) that would be returned from a SQL operation subsequently, using this execution context instance. This applies only to columns of type `BINARY`, `VARBINARY`, `LONGVARBINARY`, `CHAR`, `VARCHAR`, or `LONGVARCHAR`.

By default this parameter is set to 0, meaning there is no size limit.

- `setMaxFieldSize(int)`—Takes an `int` value as input to modify the field-size maximum.
- `int getMaxRows()`—Returns an `int` value specifying the maximum number of rows that can be contained by any SQLJ iterator or JDBC result set created using this execution context instance. If the limit is exceeded, the excess rows are silently dropped without any error report or warning.

By default, this parameter is set to 0, meaning there is no row limit.

- `setMaxRows(int)`—Takes an `int` value as input to modify the row maximum.
- `int getQueryTimeout()`—Returns an `int` value specifying the timeout limit, in seconds, for any SQL operation that uses this execution context instance. 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(int)`—Takes an `int` value as input to modify the query timeout limit.
- `int getFetchSize()`—Retrieves the number of rows that is the current fetch size for iterator objects generated from this `ExecutionContext` object. If this `ExecutionContext` object has not set a fetch size by calling `setFetchSize()`, then the value returned is zero. If this `ExecutionContext` object has set a non negative fetch size by calling the method `setFetchSize()`, then the return value is the fetch size specified on `setFetchSize()`.
- `setFetchSize(int)`—Gives the SQLJ runtime a hint as to the number of rows that should be fetched when more rows are needed. The number of rows specified affects only iterator objects created using this `ExecutionContext` object. Specifying zero means that an implementation-dependent default value will be used for the fetch size.
- `int getFetchDirection()`—Retrieves the default direction for fetching data, for scrollable iterator objects that are generated from this `ExecutionContext` object. If this `ExecutionContext` object has not set a fetch direction by calling the method `setFetchDirection()`, the return value is `FETCH_FORWARD`.
- `setFetchDirection(int)`—Gives the SQLJ runtime a hint as to the direction in which rows of scrollable iterator objects are processed. The hint

applies only to scrollable iterator objects that are created using this `ExecutionContext` object. The default value is:

```
sqlj.runtime.ResultSetIterator.FETCH_FORWARD.
```

This method throws a `SQLException` if the given direction is not one of `FETCH_FORWARD`, `FETCH_REVERSE`, or `FETCH_UNKNOWN` (int constants).

Cancellation Method

Use the following method to cancel SQL operations in a multithreading environment or to cancel a pending statement batch if update batching is enabled:

- `cancel()`—In a multithreading environment, use this method in one thread to cancel a SQL operation currently executing in another thread. It cancels the most recent operation that has started, but not completed, using this execution context instance. This method has no effect if no statement is currently being executed using this execution context instance.

In a batch-enabled environment, use this to cancel a pending statement batch. The batch is emptied, and none of the statements in the batch are executed. After you cancel a batch, the next batchable statement encountered will be added to a new batch. (["Canceling a Batch"](#) on page 10-17 discusses this.)

Update Batching Methods

Use the following methods to control update batching if you want your application to use that performance enhancement feature. These methods, and update batching in general, are further discussed in ["Update Batching"](#) on page 10-12:

- `setBatching(boolean)`—Takes a boolean value to enable update batching. See ["Enabling and Disabling Update Batching"](#) on page 10-14 for more information.

Update batching is disabled by default.

- `boolean isBatching()`—Returns a boolean value indicating whether update batching is enabled.

This does not indicate whether there is currently a pending batch, but you can use the `getUpdateCount()` method described in ["Status Methods"](#) on page 7-28 to see whether a batch has been newly created, added to, or executed.

- `int getBatchLimit()`—Returns an int value indicating the current batch limit. If there is a batch limit, a pending batch is implicitly executed once it

contains that number of statements. See ["Setting a Batch Limit"](#) on page 10-19 for more information.

By default, the batch limit is set to the `ExecutionContext` static constant value `UNLIMITED_BATCH`, meaning there is no batch limit.

- `setBatchLimit(int)`—Takes a positive, nonzero `int` value as input to set the current batch limit. Two special values you can input are `UNLIMITED_BATCH`, which means there is no limit, and `AUTO_BATCH`, which lets the SQLJ runtime dynamically determine a batch limit.
- `int[] executeBatch()`—Executes the pending statement batch, returning an array of `int` update counts that have meanings as described in ["Execution Context Update Counts"](#) on page 10-18. See ["Explicit and Implicit Batch Execution"](#) on page 10-14 for more information. Regarding error conditions, see ["Error Conditions During Batch Execution"](#) on page 10-22.
- `int[] getBatchUpdateCounts()`—Returns an array of `int` update counts for the last batch executed, with meanings as described in ["Execution Context Update Counts"](#) on page 10-18. This method is useful in situations where the batch was executed implicitly.

Savepoint Methods

Starting with Oracle9i release 2, Oracle SQLJ supports JDBC 3.0 savepoints. SQLJ savepoint statements are described in ["Using Savepoints"](#) on page 4-30. Savepoints are stored in the `ExecutionContext` instance, which provides the following support methods:

- `setSavepoint(String)`
Specify a desired savepoint name for the current location in the transaction.
- `rollbackToSavepoint(String)`
Roll back changes to the specified savepoint.
- `releaseSavepoint(String)`
Release the specified savepoint (removes it from the execution context).

In your code, however, you would typically use savepoint statements rather than these methods.

Note: As of Oracle9i release 2, Oracle9i and Oracle9i JDBC do not support release-savepoint functionality.

Closure Method

Oracle SQLJ provides extended functionality with a `close()` method for the `ExecutionContext` class:

- `close()`—To avoid resource leakage, use this in the following set of circumstances:
 - You are using the Oracle-specific code generation (the default `-codegen=oracle` setting).
- and:
 - You explicitly created and used the `ExecutionContext` instance, instead of using the default instance available through the connection context instance.
- and:
 - You are *not* issuing SQLJ rollback or commit statements explicitly using the `ExecutionContext` instance:

```
#sql [ec] { COMMIT };  
#sql [ec] { ROLLBACK };
```
- and:
 - You are *not* calling `executeBatch()` on the `ExecutionContext` instance.

Under this set of circumstances, a batchable statement might remain open on the `ExecutionContext` instance, and over time you may run out of database cursors. To avoid this, use the `close()` method as in the following example:

```
ExecutionContext ec = new ExecutionContext();  
...  
try {  
    ...  
    #sql [ec] { SQL operation };  
    ...  
} finally { ec.close(); }
```

Note: When an execution context instance is associated with a connection context instance (instead of being declared explicitly as above), then closing the connection context instance (with or without closing the underlying JDBC connection) will automatically close any statement remaining on the execution context instance.

Example: Using ExecutionContext Methods

The following code demonstrates the use of some `ExecutionContext` methods:

```
ExecutionContext execCtx =
    DefaultContext.getDefaultContext().getExecutionContext();

// Wait only 3 seconds for operations to complete
execCtx.setQueryTimeout(3);

// delete using execution context of default connection context
#sql { DELETE FROM emp WHERE sal > 10000 };

System.out.println
    ("removed " + execCtx.getUpdateCount() + " employees");
```

Relation of Execution Contexts to Multithreading

Do not use multiple threads with a single execution context. If you do, and two SQL statements try to use the same execution context simultaneously, then the second statement will be blocked until the first statement completes. Furthermore, status information from the first operation will likely be overwritten before it can be retrieved.

Therefore, if you are using multiple threads with a single connection context instance, you should take the following steps:

1. Instantiate a unique execution context instance for use with each thread.
2. Specify execution contexts with your `#sql` statements so that each thread uses its own execution context. (See "[Creating and Specifying Execution Context Instances](#)" on page 7-26.)

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 7-35 for more information about multithreading.

Note: For performance reasons, SQLJ performs no additional synchronization against `ExecutionContext` instances for Oracle-specific generated code. Therefore, you are responsible for ensuring that the same execution context instance will not be used by more than one thread. If multiple threads use the same execution context, then your application, rather than blocking, will experience errors such as incorrect results or `NullPointerException` exceptions.

Multithreading in SQLJ

This section discusses SQLJ support and requirements for multithreading and the relation between multithreading and execution context instances.

You can use SQLJ in writing multithreaded applications; however, any use of multithreading in your SQLJ application is subject to the limitations of your JDBC driver or proprietary database access vehicle. This includes any synchronization limitations.

You are required to use a different execution context instance for each thread. You can accomplish this in one of two ways:

- Specify connection context instances for your SQLJ statements such that a different connection context instance is used for each thread. Each connection context instance automatically has its own default execution context instance.
- If you are using the same connection context instance with multiple threads, then declare additional execution context instances and specify execution context instances for your SQLJ statements such that a different execution context instance is used for each thread.

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 data access is sequential—only one thread is accessing data at any given time. Synchronization refers to the control flow of the various stages of the SQL operations executing through your threads. Each statement, for example, can bind input parameters, then execute, then bind output parameters. With some JDBC drivers, special care must be taken not to intermingle these stages.

For ISO standard code generation, if a thread attempts to execute a SQL operation that uses an execution context that is in use by another operation, then the thread is blocked until the current operation completes. If an execution context were shared between threads, 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.

Important: The preceding paragraph does not apply if you use default Oracle-specific code generation (`-codegen=oracle`). For performance reasons, SQLJ performs no additional synchronization against `ExecutionContext` instances for Oracle-specific generated code. Therefore, you are responsible for ensuring that the same execution context instance will not be used by more than one thread. If multiple threads use the same execution context, then your application, rather than blocking, will experience errors such as incorrect results or `NullPointerException` exceptions.

Multithreading—MultiThreadDemo.sqlj The following is an example of a SQLJ application using multithreading.

A `ROLLBACK` operation is executed before closing the connection, so the data is not permanently altered.

```
import java.sql.SQLException;
import java.util.Random;
import sqlj.runtime.ExecutionContext;
import oracle.sqlj.runtime.Oracle;
/**
 * Each instance of MultiThreadDemo is a thread that gives all employees
 * a raise of some amount 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);
    }
}
try { #sql { ROLLBACK }; Oracle.close(); } catch (SQLException e) { }
}
static double calcAvgSal() throws SQLException
{
    double avg;
    #sql { SELECT AVG(sal) INTO :avg FROM emp };
    return avg;
}
MultiThreadDemo(double raise)
{
    this.raise = raise;
}
public void run()
{
    // Since all threads will be using the same default connection
    // context, each run uses an explicit execution context instance to
    // avoid conflict during execution
    try {
        delay();
        ExecutionContext execCtx = new ExecutionContext();
        #sql [execCtx] { UPDATE EMP SET sal = sal + :raise };
        int updateCount = execCtx.getUpdateCount();
        System.out.println("Gave raise of " + raise + " to " +
            updateCount + " employees");
    } catch (SQLException e) {
        System.err.println("error updating employees: " + e);
    }
}
// delay is used to introduce some randomness into the execution order
private void delay()
{
    try {
        sleep((long)Math.abs(randomizer.nextInt()/1000000));
    } catch (InterruptedException e) {}
}
}
}

```

Iterator Class Implementation and Advanced Functionality

This section discusses how iterator classes are implemented and what additional functionality is available beyond the essential methods discussed in "Using Named Iterators" on page 3-43 and "Using Positional Iterators" on page 3-48. The following topics are covered:

- [Implementation and Functionality of Iterator Classes](#)
- [Using the IMPLEMENTS Clause in Iterator Declarations](#)
- [Support for Subclassing of Iterator Classes](#)
- [Result Set Iterators](#)
- [Scrollable Iterators](#)

Implementation and Functionality of Iterator Classes

Any named iterator class you declare will be generated by the SQLJ translator to implement the `sqlj.runtime.NamedIterator` interface. Classes implementing the `NamedIterator` interface have functionality that maps iterator columns to database columns by name (not by position).

Any positional iterator class you declare will be generated by the SQLJ translator to implement the `sqlj.runtime.PositionedIterator` interface. Classes implementing the `PositionedIterator` interface have functionality that maps iterator columns to database columns by position (not by name).

Both the `NamedIterator` interface and the `PositionedIterator` interface, and therefore all generated SQLJ iterator classes as well, implement or extend the `sqlj.runtime.ResultSetIterator` interface.

The `ResultSetIterator` interface specifies the following methods for all SQLJ iterators (both named and positional):

- `close()`—Closes the iterator.
- `ResultSet getResultSet()`—Extracts the underlying JDBC result set from the iterator.
- `boolean isClosed()`—Determines if the iterator has been closed.
- `boolean next()`—Moves to the next row of the iterator (returning `true` if there is a valid next row to go to).

The `PositionedIterator` interface adds the following method specification for positional iterators:

- `boolean endFetch()`—Determines if you have reached the last row of a positional iterator.

As discussed in ["Using Named Iterators"](#) on page 3-43, 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 3-48, use a `FETCH INTO` statement together with the `endFetch()` method to advance through the rows of a positional iterator and retrieve the data. A `FETCH INTO` statement implicitly calls the `next()` method. Do not explicitly use the `next()` method in a positional iterator unless you are using the special `FETCH CURRENT` syntax (described in ["FETCH CURRENT Syntax: from JDBC Result Sets to SQLJ Iterators"](#) on page 7-46). The `FETCH INTO` statement also implicitly calls accessor methods that are named according to iterator column numbers. The SQLJ generation of a positional iterator class defines an accessor method for each iterator column, where each method name corresponds to the column position.

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 7-58.

Note: Alternatively, you can use a `ResultSetIterator` instance (or a `ScrollableResultSetIterator` instance) directly as a weakly typed iterator. (`ScrollableResultSetIterator` extends `ResultSetIterator`.) This is convenient if you are interested only in converting it to a JDBC result set and you do not need named or positional iterator functionality. You can also access it through SQLJ `FETCH CURRENT` syntax introduced in Oracle SQLJ release 8.1.7. See ["Result Set Iterators"](#) on page 7-41.

Using the IMPLEMENTS Clause in Iterator Declarations

There might 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 might, 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 3-43, a named iterator class generated by SQLJ has an accessor method for each column in the iterator. If you want to restrict access to certain columns, you can create an interface with only a subset of the accessor methods, then expose instances of the interface type to the user instead of exposing instances of the iterator class type.

For example, assume you are creating a named iterator of employee data, with columns `ENAME` (employee name), `EMPNO` (employee number), and `SAL` (salary). Accomplish this as follows:

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

This generates a class `EmpIter` with `ename()`, `empno()`, and `sal()` accessor methods.

Assume, though, that you want to prevent access to the `SAL` column. You can create an interface `EmpIterIntfc` that has `ename()` and `empno()` methods, but no `sal()` method. Then you can use the following iterator declaration instead of the declaration above (presuming `EmpIterIntfc` is in package `mypackage`):

```
#sql iterator EmpIter implements mypackage.EmpIterIntfc
    (String ename, int empno, float sal);
```

Then if you code your application so that users can access data only through `EmpIterIntfc` instances, they will not have access to the `SAL` column.

Support for Subclassing of Iterator Classes

SQLJ supports the ability to subclass iterator classes. This feature can be very useful in allowing you to add functionality to your queries and query results.

The one key requirement of an iterator subclass is that you must supply a public constructor that takes an instance of `sqlj.runtime.RTRResultSet` as input. The SQLJ runtime will call this constructor in assigning query results to an instance of your subclass. Beyond that, you provide functionality as you choose.

You can continue to use functionality of the original iterator class (the superclass of your subclass). For example, you can advance through query results by calling the `super.next()` method.

Result Set Iterators

You may have situations where you do not require the strongly typed functionality of a SQLJ iterator.

For such circumstances, you can directly use instances of the type `sqlj.runtime.ResultSetIterator` to receive query data, so that you are not required to declare a named or positional iterator class. Alternatively, you can use the `sqlj.runtime.ScrollableResultSetIterator` type, which extends `ResultSetIterator`. This allows you to use SQLJ scrollable iterator functionality, as described in "[Scrollable Result Set Iterators](#)" on page 7-47.

In using a result set iterator instead of a strongly typed iterator, you are trading the strong type-checking of the SQLJ `SELECT` operation for the convenience of not having to declare an iterator class.

As discussed in "[Iterator Class Implementation and Advanced Functionality](#)" on page 7-38, the `ResultSetIterator` interface underlies all named and positional iterator classes and specifies the `getResultSet()` and `close()` methods.

If you want to use SQLJ to process a result set iterator instance, then use a `ScrollableResultSetIterator` instance, and use `FETCH CURRENT` syntax as described in "[FETCH CURRENT Syntax: from JDBC Result Sets to SQLJ Iterators](#)" on page 7-46.

If you want to use JDBC to process a result set iterator instance, you can use its `getResultSet()` method, as described in "[Using and Converting Weakly Typed Iterators \(ResultSetIterator\)](#)" on page 7-61, then process the underlying result set that you retrieve.

If you process a result set iterator through its underlying result set, you should close the result set iterator, not the result set, when you are finished. Closing the result set iterator will also close the result set, but closing the result set will not close the result set iterator.

Note: Oracle9i SQLJ supports result set iterators for use as host expressions and to represent cursors in `FETCH` statements. This functionality was not supported in Oracle8i releases.

Scrollable Iterators

The ISO standard for SQLJ supports scrollable iterators, with functionality being patterned after the JDBC 2.0 specification for scrollable JDBC result sets. Oracle9i SQLJ adds support for this functionality.

For general information about scrollable result sets, see the *Oracle9i JDBC Developer's Guide and Reference*.

Declaring Scrollable Iterators

To characterize an iterator as scrollable, add the following clause to the iterator declaration:

```
implements sqlj.runtime.Scrollable
```

This instructs the SQLJ translator to generate an iterator that implements the `Scrollable` interface. Here is an example of a declaration of a named, scrollable iterator:

```
#sql public static MyScrIter implements sqlj.runtime.Scrollable
    (String ename, int empno);
```

The code that the SQLJ translator generates for the `MyScrIter` class will automatically support all the methods of the `Scrollable` interface, described in "[The Scrollable Interface](#)" below.

Scrollable Iterator Sensitivity

You can declare scrollable iterators, like scrollable result sets, to have sensitivity to changes to the underlying data. By default, scrollable iterators in Oracle SQLJ have a `sensitivity` setting of `INSENSITIVE`, meaning they do not detect any such changes in the underlying data. You can, however, use a declaration `with` clause to alter this setting. The following example expands an earlier example to specify sensitivity:

```
#sql public static MyScrIter implements sqlj.runtime.Scrollable
    with (sensitivity=SENSITIVE)
    (String ename, int empno);
```

Important: The `implements` clause must precede the `with` clause.

The SQLJ standard also allows a setting of `ASENSITIVE`, but in Oracle SQLJ this is undefined. Setting `sensitivity` to `ASENSITIVE` results instead in the default setting, `INSENSITIVE`, being used.

Given the preceding declaration, `MyScriter` instances will be sensitive to data changes, subject to factors such as the fetch size window. For general information about the behavior of sensitive scrollable JDBC result sets, which underlie sensitive scrollable iterators, see the *Oracle9i JDBC Developer's Guide and Reference*.

The Scrollable Interface

This section documents some key methods of the `sqlj.runtime.Scrollable` interface.

You can provide hints about the fetch direction to scrollable iterators. The following methods are defined on scrollable iterators as well as on execution contexts. Use an `ExecutionContext` instance to provide the default direction to be used in creation of scrollable iterators.

- `setFetchDirection(int)` — Gives the SQLJ runtime a hint as to the direction in which rows are processed. The direction should be one of `sqlj.runtime.ResultSetIterator.FETCH_FORWARD`, `FETCH_REVERSE`, or `FETCH_UNKNOWN`.

If you do not specify a value for the direction on the `ExecutionContext`, then `FETCH_FORWARD` will be used as a default.

- `int getFetchDirection()` — Retrieves the current direction for fetching rows of data (one of the integer constants described immediately above).

There are also a number of scrollable iterator methods that will return information about the current position of the iterator object in the underlying result set. All these methods will return `false` whenever the result set underlying the iterator contains no rows:

- `boolean isBeforeFirst()` — Indicates whether the iterator object is before the first row in the result set.
- `boolean isFirst()` — Indicates whether the iterator object is on the first row of the result set.
- `boolean isLast()` — Indicates whether the iterator object is on the last row of the result set. Note that calling the method `isLast()` may be expensive, because the JDBC driver might have to fetch ahead one row to determine whether the current row is the last row in the result set.

- `boolean isAfterLast()`—Indicates whether the iterator object is after the last row in the result set.

Note: Additional methods for navigation, also defined in the `Scrollable` interface, are available as well. These are described in "[Scrollable Named Iterators](#)" below.

Scrollable Named Iterators

Named iterators use navigation methods, defined in the `Scrollable` interface, to move through the rows of a result set. As described earlier in this manual, non-scrollable iterators have only the following method for navigation:

- `boolean next()`—Moves the iterator object to the next row in the result set.

(See "[Using Named Iterators](#)" on page 3-43 for more information.)

Additional navigation methods are available for scrollable named iterators. These methods function similarly to the `next()` method, in that they try to position the iterator on an actual row of the result set. They return `true` if the iterator ends up on a valid row and `false` if it does not. Additionally, if you attempt to position the iterator object before the first row or after the last row in the result set, this leaves the iterator object in the "before first" or "after last" position, respectively.

The following methods are supported:

- `boolean previous()`—Moves the iterator object to the previous row in the result set.
- `boolean first()`—Moves the iterator object to the first row in the result set.
- `boolean last()`—Moves the iterator object to the last row in the result set.
- `boolean absolute(int)` — Moves the iterator object to the given row number in the result set. The first row is row 1, the second is row 2, and so on. If the given row number is negative, the iterator object moves to a row position relative to the end of the result set. For example, calling `absolute(-1)` positions the iterator object on the last row, `absolute(-2)` indicates the next-to-last row, and so on.
- `boolean relative(int)`—Moves the iterator object a relative number of rows, either positive or negative from the current position. Calling `relative(0)` is valid, but does not change the iterator position.

The methods `beforeFirst()` and `afterLast()` return `void`, because they never place the iterator object on an actual row of the result set.

- `void beforeFirst()`—Moves the iterator object to the front of the result set, before the first row. This has no effect if the result set contains no rows.
- `void afterLast()`—Moves the iterator object to the end of the result set, after the last row. This has no effect if the result set contains no rows.

Scrollable Positional Iterators

General `FETCH` syntax for positional iterators was described earlier, in ["Using Positional Iterators"](#) on page 3-48. For example:

```
#sql { FETCH :iter INTO :x, :y, :z };
```

This is actually an abbreviated version of the following syntax.

```
#sql { FETCH NEXT FROM :iter INTO :x, :y, :z };
```

This suggests the pattern for alternatively moving to the previous, first, or last row in the result set. (Unfortunately, JDBC 2.0—after which the movement methods were modeled—uses `previous()`, whereas the `FETCH` syntax, which is patterned after SQL, employs `PRIOR`. In case you should forget this inconsistency, the Oracle9i SQLJ translator will also accept `FETCH PREVIOUS`.)

```
#sql { FETCH PRIOR FROM :iter INTO :x, :y, :z };  
#sql { FETCH FIRST FROM :iter INTO :x, :y, :z };  
#sql { FETCH LAST FROM :iter INTO :x, :y, :z };
```

There is also syntax to pass a numeric value for absolute or relative movements, to move to a particular (absolute) row, or to move forward or backward from the current position:

```
#sql { FETCH ABSOLUTE :n FROM :iter INTO :x, :y, :z };  
#sql { FETCH RELATIVE :n FROM :iter INTO :x, :y, :z };
```

Note: In all of the preceding cases, the iterator `endFetch()` method returns `true` whenever the `FETCH` fails to move to a valid row and retrieve values.

Note that you *must* use a host expression to specify the movement. You cannot simply use a constant for the numeric value. Thus, instead of:

```
#sql { FETCH RELATIVE 0 FROM :iter INTO :x, :y, :z };
```

you must write the following:

```
#sql { FETCH RELATIVE :(0) FROM :iter INTO :x, :y, :z };
```

Incidentally, this command leaves the position of the iterator unchanged and—if the iterator is on a valid row—just populates the variables.

Note: Alternatively, you can navigate through a scrollable positional iterator through a combination of the navigation methods described in ["Scrollable Named Iterators"](#) on page 7-44, and `FETCH CURRENT` syntax described in ["FETCH CURRENT Syntax: from JDBC Result Sets to SQLJ Iterators"](#) below.

FETCH CURRENT Syntax: from JDBC Result Sets to SQLJ Iterators

Consider a situation where you have an existing JDBC program that you want to rewrite in SQLJ with as little modification as possible.

Your JDBC result set will use only movement methods, such as `next()`, `previous()`, `absolute()`, and so on. You can immediately model this in SQLJ through a named iterator. However, this also implies that all columns of the SQL result set must have a proper name. In practice many (if not all) columns of the result set will require introduction of alias names. This is unacceptable if the query text is to remain untouched.

The alternative, to avoid change to the query source, is to define a positional iterator type for the result set. However, this approach forces changes to the control-flow logic of the program. Consider the following JDBC code sample:

```
ResultSet rs = ... // execute ...query...;
while (rs.next()) {
    x := rs.getXxx(1); y:=rs.getXxx(2);
    ...process...
}
```

This translates along the following lines to SQLJ:

```
MyIter iter;
#sql iter = { ...query... };
while(true) {
    #sql { FETCH :iter INTO :x, :y };
    if (iter.endFetch()) break;
    ...process...
}
```

The transformations to the program logic will become even more difficult when considering arbitrary movements on scrollable iterators. Because positional iterators implement all the movement commands of named iterators, it is possible to exploit this and use `RELATIVE : (0)` to populate variables from the iterator:

```
MyIter iter;
#sql iter = { ...query... };
while (iter.next()) {
    #sql { FETCH RELATIVE : ( 0 ) FROM :iter INTO :x, :y };
    ...process...
}
```

Now, you can preserve both the original query and the original program logic. Unfortunately, there still is one drawback to this approach—the iterator type `MyIter` must implement the `Scrollable` interface, even if this property is not really needed. To address this, Oracle9i SQLJ introduces the following syntax extension:

```
#sql { FETCH CURRENT FROM :iter INTO :x, :y, :z };
```

Given this syntax, you can rewrite the JDBC example in SQLJ for scrollable as well as non-scrollable iterators:

```
AnyIterator ai;
#sql ai = { ...query... };
while (ai.next()) {
    #sql { FETCH CURRENT FROM :ai INTO :x, :y };
    ...process...
}
```

Scrollable Result Set Iterators

In Oracle9i SQLJ, support for weakly typed result set iterators is extended to add a scrollable result set iterator type. The definition follows.

```
package sqlj.runtime;
public interface ScrollableResultSetIterator
        extends ResultSetIterator
        implements Scrollable
{ }
```

Because this type extends `sqlj.runtime.ResultSetIterator`, it supports the methods described in ["Result Set Iterators"](#) on page 7-41.

Because it also implements the `sqlj.runtime.Scrollable` interface, it supports the methods described in ["The Scrollable Interface"](#) on page 7-43 and ["Scrollable Named Iterators"](#) on page 7-44.

Furthermore, scrollable result set iterators support the `FETCH CURRENT` syntax described in ["FETCH CURRENT Syntax: from JDBC Result Sets to SQLJ Iterators"](#) on page 7-46.

Consider the following JDBC code:

```
Statement st = conn.createStatement("SELECT ename, empid FROM emp");
ResultSet rs = st.executeQuery();
while (rs.next()) {
    x = rs.getString(1);
    y = rs.getInt(2);
}
rs.close();
```

You can use a SQLJ result set iterator in writing equivalent code, as follows:

```
sqlj.runtime.ResultSetIterator rsi;
#sql rsi = { SELECT ename, empid FROM emp };
while (rsi.next()) {
    #sql { FETCH CURRENT FROM :rsi INTO :x, :y };
}
rsi.close();
```

To take advantage of scrollability features, you could also write the following code:

```
sqlj.runtime.ScrollableResultSetIterator srsi;
#sql srsi = { SELECT ename, empid FROM emp };
srsi.afterLast();
while (srsi.previous()) {
    #sql { FETCH CURRENT FROM :srsi INTO :x, :y };
}
srsi.close();
```

Advanced Transaction Control

SQLJ supports the SQL `SET TRANSACTION` statement to specify the access mode and isolation level of any given transaction. Standard SQLJ supports `READ ONLY` and `READ WRITE` access mode settings, but Oracle JDBC does not support `READ ONLY`. (You can set permissions to have the same effect, however.) 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 *Oracle9i SQL Reference*. You might also consult any guide to standard SQL for additional conceptual information.

For an overview of transactions, including SQLJ support for the basic transaction control operations `COMMIT` and `ROLLBACK`, see "[Basic Transaction Control](#)" on page 4-26.

SET TRANSACTION Syntax

In SQLJ, the `SET TRANSACTION` statement has the following syntax:

```
#sql { SET TRANSACTION <access_mode>, <ISOLATION LEVEL isolation_level> };
```

If you do not specify a connection context instance, then the statement applies to the default connection.

If you use `SET TRANSACTION`, it must be the first statement in a 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 standard SQLJ, any access mode or isolation level you set will remain in effect across transactions until you explicitly reset it at the beginning of a subsequent transaction.

In a standard SQLJ `SET TRANSACTION` statement, you can optionally specify the isolation level first, or specify only the access mode, or only the isolation level. Following are some examples:

```
#sql { SET TRANSACTION READ WRITE };
```

```
#sql { SET TRANSACTION ISOLATION LEVEL SERIALIZABLE };
```

```
#sql { SET TRANSACTION READ WRITE, ISOLATION LEVEL SERIALIZABLE };
```

```
#sql { SET TRANSACTION ISOLATION LEVEL READ COMMITTED, READ WRITE };
```

You can also specify a particular connection context instance for a `SET TRANSACTION` statement, as opposed to having it apply to the default connection:

```
#sql [myCtxt] { SET TRANSACTION ISOLATION LEVEL SERIALIZABLE };
```

Note that in SQLJ, both the access mode and the isolation level can be set in a single `SET TRANSACTION` statement. This is not true in other Oracle SQL tools such as Server Manager or SQL*Plus, where a single statement can set one or the other, but not both.

Access Mode Settings

The `READ WRITE` and `READ ONLY` access mode settings (where supported) have the following functionality:

- `READ WRITE` (default)—In a `READ WRITE` transaction, the user is allowed to update the database. `SELECT`, `INSERT`, `UPDATE`, and `DELETE` are all legal.
- `READ ONLY` (supported by Oracle JDBC as of Oracle9i release 2)—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 Oracle9i)—*Dirty reads, non-repeatable reads, and phantom reads* are all allowed. See below for definitions of the italicized terms.
- `READ COMMITTED` (default for Oracle9i)—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.
- `REPEATABLE READ` (not supported by Oracle9i)—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 might have had changes committed after the transaction began. Such DML statements will fail.

A *dirty read* occurs when transaction B accesses a row that was updated by transaction A, but transaction A later rolls back the updates. As a result, transaction B sees data that was never actually committed to the database.

A *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 Oracle9i—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 JDBC connection instance of your connection context instance. SQLJ code using these JDBC methods is not portable, however.

Following are the `Connection` class methods for access mode and isolation level settings:

- `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, SQLJ statements are typically used for static SQL operations. Oracle9i has extensions to support dynamic SQL as well, but another alternative is to use JDBC code within your SQLJ application for dynamic operations (which would be more portable). And there might be additional scenarios where using JDBC code in your SQLJ application might be useful or even required.

Because of this, SQLJ allows you to use SQLJ and JDBC statements concurrently and provides interoperability between SQLJ and JDBC constructs.

Two kinds of interactions between SQLJ and JDBC are particularly useful:

- between SQLJ connection contexts and JDBC connections
- between SQLJ iterators and JDBC result sets

For general information about JDBC functionality, see the *Oracle9i 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 underlying physical connection. See ["About Shared Connections"](#) on page 7-56.

Converting from Connection Contexts to JDBC Connections

If you want to perform a JDBC operation through a database connection that you have established in SQLJ (for example, if your application calls a library routine that returns a JDBC connection object), then you must convert the SQLJ connection context instance to a JDBC connection instance.

Any connection context instance in a SQLJ application, whether an instance of the `sqlj.runtime.ref.DefaultContext` class or of a declared connection context class, contains an underlying JDBC connection instance and a `getConnection()` method that returns that JDBC connection instance. Use the JDBC connection instance to create JDBC statement objects if you want to use JDBC operations.

Following is an example of how to use the `getConnection()` method.

Imports:

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

Executable code:

```
DefaultContext ctx = new DefaultContext
    ("jdbc:oracle:thin:@localhost:1521:orcl", "scott", "tiger", true);
...
(SQLJ operations through SQLJ ctx connection context instance)
...
Connection conn = ctx.getConnection();
...
(JDBC operations through JDBC conn connection instance)
...
```

The connection context instance can be an instance of the `DefaultContext` class or of any connection context class that you have declared.

To retrieve the underlying JDBC connection of your default SQLJ connection, you can use `getConnection()` directly from a `DefaultContext.getDefaultContext()` call, where `getDefaultContext()` returns a `DefaultContext` instance that you had previously initialized as your default connection, and `getConnection()` returns its underlying JDBC connection instance. In this case, because you do not have to use the `DefaultContext` instance explicitly, you can also use the `Oracle.connect()` method. This method implicitly creates the instance and makes it the default connection.

(See ["Connection Considerations"](#) on page 4-6 for an introduction to connection context instances and default connections. See ["More About the Oracle Class"](#) on page 4-12 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();
...
(JDBC operations through JDBC conn connection instance)
...
```

Example: JDBC and SQLJ Connection Interoperability for Dynamic SQL Following is a sample method that uses the underlying JDBC connection instance of the default SQLJ connection context instance to perform dynamic SQL operations in JDBC. The dynamic operations are performed using JDBC `java.sql.Connection`, `java.sql.PreparedStatement`, and `java.sql.ResultSet` objects. (For information about such basic features of JDBC programming, see the *Oracle9i JDBC Developer's Guide and Reference*.)

Alternatively, you can use Oracle SQLJ extensions for dynamic SQL operations. See ["Support for Dynamic SQL"](#) on page 7-63 for general information. For a rework of this example using SQLJ dynamic SQL functionality with `FETCH` functionality from a result set iterator, see [Example 5: Dynamic SQL with FETCH from Result Set Iterator](#) on page 7-68.

```
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 Connection` instance but later want to use it as a SQLJ connection context instance (for example, if you want to use it in a context expression to specify the connection to use for a SQLJ executable statement), you can convert the JDBC connection instance to a SQLJ connection context instance.

The `DefaultContext` class and all declared connection context classes have a constructor that takes a JDBC connection instance as input and constructs a SQLJ connection context instance.

For example, presume you instantiated and defined the JDBC connection instance `conn` and want to use the same connection for an instance of a declared SQLJ connection context class `MyContext`. You can do this as follows:

```
...
#sql context MyContext;
...
MyContext myctx = new MyContext(conn);
...
```

About Shared Connections

A SQLJ connection context instance and the associated JDBC connection instance share the same underlying physical connection. As a result, the following is true:

- When you get a JDBC connection instance from a SQLJ connection context instance (using the connection context `getConnection()` method), the `Connection` instance inherits the state of the connection context instance. Among other things, the `Connection` instance will retain the auto-commit setting of the connection context instance.
- When you construct a SQLJ connection context instance from a JDBC connection instance (using the connection context constructor that takes a connection instance as input), the connection context instance inherits the state of the `Connection` instance. Among other things, the connection context instance will retain the auto-commit setting of the `Connection` instance. By default, a JDBC connection instance has an auto-commit setting of `true`, but you can alter this through the `setAutoCommit()` method of the `Connection` instance.
- Given a SQLJ connection context instance and associated JDBC connection instance, calls to methods that alter session state in one instance will also affect the other instance, because it is actually the underlying shared session that is being altered.

- Because there is just a single underlying physical connection, there is also a single underlying set of transactions. A `COMMIT` or `ROLLBACK` operation in one connection instance will affect any other connection instances that share the same underlying connection.

Note: It is also possible for multiple SQLJ connection context instances to be created from the same JDBC connection instance and, therefore, to share the same underlying physical connection. This might be useful, for example, if you want to share the same set of transactions between program modules. The preceding notes apply to this situation as well.

Closing Shared Connections

When you get a JDBC connection instance from a SQLJ connection context instance (using the `getConnection()` method) or you create a SQLJ connection context instance from a JDBC connection instance (using the connection context constructor), you must close only the connection context instance. By default, calling the `close()` method of a connection context instance closes the associated JDBC connection instance and the underlying physical connection, thereby freeing all resources associated with the connection.

If you want to close a SQLJ connection context instance *without* closing the associated JDBC connection instance (if, for example, the `Connection` instance is being used elsewhere, either directly or by another connection context instance), then you can specify the boolean constant `KEEP_CONNECTION` to the `close()` method, as follows (assume a connection context instance `ctx`):

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

If you do not specify `KEEP_CONNECTION`, then the associated JDBC connection instance is closed by default. You can also specify this explicitly:

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

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

If you 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.

If you close only the JDBC connection instance, this will *not* close the associated SQLJ connection context instance. The underlying physical connection would be closed, but the resources of the connection context instance would not be freed until garbage collection.

Notes:

- If the same underlying JDBC connection is shared by multiple connection context instances, then use `KEEP_CONNECTION` when closing all but the last remaining open connection context instance.
 - An error message will be issued if you try to close a connection context instance whose underlying JDBC connection has already been closed, or if you try to close the underlying connection when it has already been closed. If you encounter this, verify that the JDBC connection is not being closed independently by JDBC code, and that all preceding `close()` calls on SQLJ connection context instances that use the underlying connection use the `KEEP_CONNECTION` parameter.
-
-

SQLJ Iterator and JDBC Result Set Interoperability

SQLJ 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 might find yourself manipulating JDBC result sets. For example, another package might be implemented in JDBC and provide access to data only through result sets, or might require `ResultSetMetaData` information because it is a routine written generically for any type of result set. Or your SQLJ application might invoke a stored procedure that returns a JDBC result set.

If the dynamic result set has a known structure, it is typically desirable to manipulate it as an iterator to use the strongly typed paradigm that iterators offer.

In SQLJ, you can populate a named or positional iterator object by converting an existing JDBC result set object. This can be thought of as casting a result set to an iterator, and the syntax reflects this, as follows.

```
#sql iter = { CAST :rs };
```

This binds the result set object `rs` into the SQLJ executable statement, converts the result set, and populates the iterator `iter` with the result set data.

Following is an example. Assume `myEmpQuery()` is a static Java function in a class called `RSClass`, with a predefined query that returns a JDBC result set object.

Imports and declarations:

```
import java.sql.*;
...
#sql public iterator MyIterator (String ename, float sal);
...
```

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, then the SQLJ translator will generate a warning unless you use the `-warn=nostrict` option setting.
- The result set being cast must implement the `java.sql.ResultSet` interface. The class `oracle.jdbc.OracleResultSet` implements this interface, as does any standard result set class.

- The iterator receiving the cast must be an instance of an iterator class that was declared as `public`.
- Do not access data from the result set, either before or after the conversion. Access data from the iterator only.
- When you are finished, close the iterator, not the result set. Closing the iterator will also close the result set, but closing the result set will not close the iterator. When interoperating with JDBC, always close the SQLJ entity.

Converting from Named or Positional Iterators to Result Sets

You might also encounter situations where you want to define a query using SQLJ but ultimately need a result set. (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 ename, float sal);
...
```

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 might have a situation similar to what is discussed in "[Converting from Named or Positional Iterators to Result Sets](#)" on page 7-60, but where you do not require the strongly typed functionality of the iterator. All you might care about is being able to use SQLJ syntax for the query and then processing the data dynamically from a result set.

For such circumstances, you can directly use the type `sqlj.runtime.ResultSetIterator` to receive query data. See "[Result Set Iterators](#)" on page 7-41 for general information about the result set iterator types.

In using SQLJ statements and `ResultSetIterator` functionality instead of using JDBC statements and standard result set functionality, you enable yourself to use the more concise `SELECT` syntax of SQLJ.

Following is an example of how to use and convert a weakly typed result set iterator.

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();
...
```

Note: Oracle SQLJ permits navigation through a result set iterator using the `next()` method and `FETCH CURRENT` syntax, as described in ["FETCH CURRENT Syntax: from JDBC Result Sets to SQLJ Iterators"](#) on page 7-46. Furthermore, for scrollable result set iterators, additional navigation methods are supported. These methods are described in ["Scrollable Named Iterators"](#) on page 7-44.

Support for Dynamic SQL

Oracle9i SQLJ includes extensions to support *dynamic SQL*—operations that are not predefined and can change in real-time. Dynamic SQL expressions embedded in SQLJ statements are referred to as *meta bind expressions* and are described immediately below.

Note: Using JDBC code is still an option for dynamic SQL in Oracle9i, and might be preferable if code portability is a concern, but Oracle9i SQLJ support for dynamic SQL permits use of SQLJ as a single, simplified API for data access. (SQLJ-JDBC interaction is discussed under "[SQLJ and JDBC Interoperability](#)" on page 7-53.)

Meta Bind Expressions

Meta bind expressions are used for dynamic SQL in SQLJ statements, where otherwise static SQL clauses would appear. A meta bind expression contains a Java identifier of type `String` or a string-valued Java expression that is interpreted at runtime. In addition, so that SQLJ can perform online semantics-checking, a meta bind expression can optionally include static SQL replacement code to be used for checking during translation.

This section describes usage, restrictions, syntax, and behavior for meta bind expressions.

Meta Bind Expressions—General Usage and Restrictions

You can use a meta bind expression in place of any of the following:

- a table name
- a column name in a SELECT-list (without the column alias, if one was specified)
- all or part of a WHERE clause condition
- a role, schema, catalog, or package name in a DDL or DML statement
- a SQL literal value or SQL expression

Be aware of the following restrictions on meta bind expressions, enforced to ensure that the SQLJ translator can properly determine the nature of the SQL operation and can perform syntactic analysis of the SQLJ statement as a whole:

- A meta bind expression cannot be the first non-comment of the SQL operation within a SQLJ statement.

- A meta bind expression cannot contain the `INTO` token of a `SQLJ SELECT INTO` statement and cannot expand to become the INTO-list of a `SELECT INTO` statement.
- A meta bind expression cannot appear in any of the following kinds of `SQL/SQLJ` instructions or clauses: `CALL`, `VALUES`, `PSM SET`, `COMMIT`, `ROLLBACK`, `FETCH INTO`, or `CAST`.

Meta Bind Expressions—Syntax and Behavior

Following is the general syntax for meta bind expressions:

```
:{ Java_bind_expression }
```

or:

```
:{ Java_bind_expression :: SQL_replacement_code }
```

Spaces are optional.

There can be multiple meta bind expressions within the `SQL` instructions of a `SQLJ` statement.

Java Bind Expression A Java bind expression can be either of the following:

- a Java identifier of type `String`
- a Java expression that evaluates to a character string

Java bind expressions within meta bind expressions are subject to standard Java lexing rules, and have syntax similar to that of `SQLJ` host expressions (described in "[Java Host Expressions, Context Expressions, and Result Expressions](#)" on page 3-16). However, unlike host expressions, Java bind expressions within meta bind expressions are not enclosed within parentheses. This is because if there is `SQL` replacement code, the `::` token acts as a separator between the Java bind expression and the `SQL` code; if there is no `SQL` replacement code, the closing `}` acts as a terminator. In either case, there is no ambiguity.

Note: There can be no mode specifiers (`IN`, `OUT`, or `INOUT`) within a Java bind expression, or between `:` and `{` of the meta bind expression.

SQL Replacement Code A SQL replacement code clause consists of a sequence of zero or more SQL tokens, with the following requirements and restrictions:

- It is subject to SQL lexing rules.
- Braces—"{" and "}"—must occur in matching pairs (with the exception of those that are part of a SQL comment, constant, or identifier).
- There can be no SQLJ host expressions or nested meta bind expressions within the SQL instructions.

Note: It is permissible for the SQL replacement code to be empty.

Translation-Time Behavior Whenever there is SQL replacement code (even if only an empty string) in a meta bind expression, then the meta bind expression is replaced by the SQL code during translation. The purpose of SQL replacement code is to enable the SQLJ translator to perform online semantics-checking.

If any meta bind expression within a SQLJ statement has no SQL replacement code clause, then the SQLJ translator cannot perform online semantics-checking on the statement—it is only checked syntactically.

Runtime Behavior At runtime, each meta bind expression is replaced by the evaluation of its Java bind expression.

If a Java bind expression evaluates to `null`, then the dynamic SQL statement as a whole becomes undefined.

SQLJ Dynamic SQL Examples

This section provides examples of dynamic SQL usage in SQLJ code.

Example 1

```
...
int x = 10;
int y = x + 10;
int z = y + 10;
String table = "new_Emp";
#sql { INSERT INTO :{table}::emp} VALUES (:x, :y, :z) };
...
```

During translation, the SQL operation becomes:

```
INSERT INTO emp VALUES (10, 20, 30);
```

SQLJ can perform online semantics-checking against a schema that has an `emp` table. (Perhaps `new_Emp` only exists in the runtime schema, and is not created until the application executes.)

During runtime, the SQL operation becomes:

```
INSERT INTO new_Emp VALUES (10, 20, 30);
```

Example 2

```
...
String table = "new_Emp";
String query = "ename LIKE 'S%' AND sal>1000";
#sql myIter = { SELECT * FROM :{table :: emp2}
                WHERE :{query :: ename='SCOTT'} };
...
```

During translation, the SQL operation becomes:

```
SELECT * FROM emp2 WHERE ename='SCOTT';
```

SQLJ can perform online semantics-checking against a schema that has an `emp2` table.

During runtime, the SQL operation becomes:

```
SELECT * FROM new_Emp WHERE ename LIKE 'S%' AND sal>1000;
```

Example 3

```
...
double raise = 1.12;
String col = "comm";
String whereQuery = "WHERE "+col+" IS NOT null";
for (int i=0; i<5; i++)
{
    #sql { UPDATE :{"emp"+i :: emp}
          SET :{col :: sal} = :{col :: sal} * :raise :{whereQuery ::} };
}
...
```

During translation, the SQL operation becomes:

```
UPDATE emp SET sal = sal * 1.12;
```

SQLJ can perform online semantics-checking against a schema that has an emp table. There is no WHERE clause during translation, because the SQL replacement code is empty.

During runtime, the SQL operation is executed five times, becoming:

```
UPDATE emp0 SET comm = comm * 1.12 WHERE comm IS NOT null;
UPDATE emp1 SET comm = comm * 1.12 WHERE comm IS NOT null;
UPDATE emp2 SET comm = comm * 1.12 WHERE comm IS NOT null;
UPDATE emp3 SET comm = comm * 1.12 WHERE comm IS NOT null;
UPDATE emp4 SET comm = comm * 1.12 WHERE comm IS NOT null;
```

Example 4

```
...
double raise = 1.12;
String col = "comm";
String whereQuery = "WHERE "+col+" IS NOT null";
for (int i=0; i<10; i++)
{
    #sql { UPDATE :{"emp"+i}
          SET :{col :: sal} = :{col :: sal} * :raise :{whereQuery ::} };
}
...
```

The runtime behaviors of [Example 4](#) and [Example 3](#) are identical. A difference occurs during translation, however, where SQLJ cannot perform online semantics-checking for [Example 4](#) because there is no SQL replacement code for the first meta bind expression, `:{ "emp "+i }`.

Example 5: Dynamic SQL with FETCH from Result Set Iterator This example is a rework of ["Example: JDBC and SQLJ Connection Interoperability for Dynamic SQL"](#) on page 7-55, using SQLJ statements instead of JDBC statements. This example also uses `FETCH CURRENT` functionality, as described in ["FETCH CURRENT Syntax: from JDBC Result Sets to SQLJ Iterators"](#) on page 7-46, from a result set iterator.

```
import java.sql.*;

public static void projectsDue(boolean dueThisMonth) throws SQLException {

    ResultSetIterator rsi;
    String andClause = (dueThisMonth) ?
        " AND to_char(start_date + duration, 'fmMonth' ) "
        + " = to_char(sysdate, 'fmMonth') "
        : "";
    #sql rsi = { SELECT name, start_date + duration FROM projects
                WHERE start_date + duration >= sysdate :{andClause :: } };
    while (rsi.next())
    {
        String name = null;
        java.sql.Date deadline = null;
        #sql { FETCH CURRENT FROM :rsi INTO :name, :deadline };
        System.out.println("Project: " + name + "Deadline: " + deadline);
    }
    rsi.close();
}
```

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](#)
- [Basic Translator Options](#)
- [Advanced Translator Options](#)
- [Translator Support and Options for Alternative Environments](#)

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 the options available. It then discusses SQLJ properties files, which you can use instead of the command line to set most options, and the `SQLJ_OPTIONS` environment variable, which you can use in addition to or instead of the command line for setting options.

For detailed information about settings for basic options, see "[Basic Translator Options](#)" on page 8-22.

For information about more advanced options, see "[Advanced Translator Options](#)" on page 8-60 and "[Translator Support and Options for Alternative Environments](#)" on page 8-80.

The `sqlj` script invokes a Java virtual machine (JVM) and passes the class name of the SQLJ translator (`sqlj.tools.Sqlj`) to the JVM. The JVM invokes the translator and performs operations such as parsing the command line and properties files. For simplicity, running the script is referred to as "running SQLJ", and its command line is referred to as the "SQLJ command line".

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 pass to the Java interpreter, compiler, and customizer (if applicable).

The *file list* is the list of files, delimited by spaces, to be processed by the SQLJ translator. The files can be `.sqlj`, `.java`, `.ser`, or `.jar` files, as explained in "[Command-Line Syntax and Operations](#)" on page 8-12. The `*` wildcard entry can be used in file names. For example, `Foo*.sqlj` would find `Foo1.sqlj`, `Foo2.sqlj`, and `Foobar.sqlj`.

Notes:

- It is not required that all the options precede the file list. Options may appear anywhere in the command line and are processed in order.
 - All command-line options apply to all files being translated. It is not possible to have file-specific option settings.
-
-

Do not include `.class` files in the file list, but do be sure that your classpath is set so that the SQLJ translator can find any classes it must have for type resolution of variables in your SQLJ source files.

If the `-checksource` flag is enabled (its default setting), the SQLJ translator can also find classes it needs in uncompiled `.java` files in the classpath. See "[Source Check for Type Resolution \(-checksource\)](#)" on page 8-69.

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, and only a small subset of options are supported. For information, see "[Option Support in the Server Embedded Translator](#)" on page 11-18.
 - 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 JVM, which the SQLJ script invokes, and to the Java compiler and SQLJ profile customizer, which the JVM invokes.

Summary of SQLJ Options

[Table 8-1](#) below lists options supported by the SQLJ translator, categorized as follows:

- Flags, options, and prefixes listed as "command-line only" cannot be set in a properties file.
- Flags and options listed as "Basic" are discussed in "[Basic Translator Options](#)" on page 8-22.
- Flags, options, and prefixes listed as "Advanced" are discussed in "[Advanced Translator Options](#)" on page 8-60.

- Flags and options listed as "Environment" are discussed in ["Translator Support and Options for Alternative Environments"](#) on page 8-80. These flags and options are for use of a non-standard JVM, compiler, or customizer.
- Options with a category of "javac" are `javac` options that SQLJ recognizes directly, without the compiler prefix. They are passed to the Java compiler, typically `javac`, and some also affect SQLJ translator settings. These options are discussed in ["Option Support for javac"](#) on page 8-10.

Notes Regarding Options, Flags, and Prefixes:

- 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.
- Most SQLJ options can also be set in a *properties file*. See ["Properties Files for Option Settings"](#) on page 8-16.
- 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 8-19.
- 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`, or `1/0`.

For discussion and an example of command-line syntax and operations, see ["Command-Line Syntax and Operations"](#) on page 8-12.

Table 8-1 SQLJ Translator Options

Option	Description	Default	Category
<code>-bind-by-identifier</code>	Flag to treat multiple appearances of the same host variable in a given SQLJ statement as a single bind occurrence.	<code>false</code>	Advanced
<code>-C</code>	Prefix that marks options to pass to the Java compiler.	<code>n/a</code>	Advanced
<code>-cache</code>	Enables caching of online semantics-checking results (to reduce trips to database).	<code>false</code>	Advanced

Table 8–1 SQLJ Translator Options (Cont.)

Option	Description	Default	Category
-checkfilename	Specifies whether a warning is issued during translation if a source file name does not correspond to the name of the public class (if any) defined there.	true	Environment
-checksource	Instructs SQLJ type resolution to examine source files in addition to class files in certain circumstances.	true	Advanced
-classpath (command-line only)	Specifies the classpath to the JVM and Java compiler; also passed to <code>javac</code> .	none	Basic
-codegen	Specifies mode of code generation: <code>oracle</code> for Oracle-specific code generation with direct Oracle JDBC calls; <code>iso</code> for ISO standard SQLJ code generation.	oracle	Basic
-compile	Enables/disables the Java compilation step—for <code>.java</code> files generated during the current SQLJ run, or for previously generated or other <code>.java</code> files specified on the command line.	true	Advanced
-compiler-executable	Specifies the Java compiler to use.	javac	Environment
-compiler-encoding-flag	Instructs SQLJ whether to pass the <code>-encoding</code> setting (if that option is set) to the Java compiler.	true	Environment
-compiler-output-file	Specifies a file to which the Java compiler output should be written. If this option is not set, then SQLJ assumes that compiler output goes to standard output.	none	Environment
-compiler-pipe-output-flag	Instructs SQLJ whether to set the <code>javac.pipe.output</code> system property, which determines whether the Java compiler outputs errors and messages to <code>STDOUT</code> instead of <code>STDERR</code> .	true	Environment
-d	Specifies the output directory for <code>.ser</code> profile files (if applicable) generated by SQLJ and <code>.class</code> files generated by the compiler; also passed to <code>javac</code> .	empty (use directory of <code>.java</code> files for <code>.class</code> files; use directory of <code>.sqlj</code> files for <code>.ser</code> files)	Basic
-default-customizer	Determines the profile customizer to use. Specify a class name.	oracle.sqlj.runtime.util. .OraCustomizer	Environment

Table 8–1 SQLJ Translator Options (Cont.)

Option	Description	Default	Category
-default-url-prefix	Sets the default prefix for URL settings.	jdbc:oracle:thin:	Basic
-depend (requires compiler prefix if set in properties file)	Passed to <code>javac</code> ; enables <code>-checksource</code> .	n/a	javac
-deprecation (requires compiler prefix if set in properties file)	Passed to <code>javac</code> only.	n/a	javac
-dir	Sets the output directory for SQLJ-generated <code>.java</code> files.	empty (use directory of <code>.sqlj</code> input file)	Basic
-driver	Determines the JDBC driver class to register. Specify a class name or comma-delimited list of class names.	oracle.jdbc. OracleDriver	Basic
-encoding (also recognized as <code>-e</code> if on command line)	Specifies the encoding that SQLJ and the compiler will use in globalization support; also passed to <code>javac</code> .	JVM file.encoding setting	Basic
-explain	Flag to request "cause" and "action" information to be displayed with translator error messages.	false	Basic
-fixedchar	Flag to account for blank padding when binding a string into a <code>WHERE</code> clause for comparison with <code>CHAR</code> data.	false	Basic
-g (requires compiler prefix if set in properties file)	Passed to <code>javac</code> ; enables <code>-linemap</code> .	n/a	javac
-help (also recognized as <code>-h</code>) -help-long -help-alias (all command-line only)	Flags to display different levels of information about SQLJ option names, descriptions, and current values.	disabled	Basic
-jdbblinemap	Variant of <code>-linemap</code> option for use with the Sun Microsystems <code>jdb</code> debugger.	false	Basic
-J (command-line only)	Prefix that marks options to pass to the JVM.	n/a	Advanced
-linemap	Enables mapping of line numbers between the generated Java class file and the original SQLJ code.	false	Basic

Table 8–1 SQLJ Translator Options (Cont.)

Option	Description	Default	Category
-n (command-line only; alternatively -vm=echo)	Instructs the <code>sqlj</code> script to echo the full command line as it would be passed to the SQLJ translator (including settings in <code>SQLJ_OPTIONS</code>) without having the translator execute it.	disabled	Basic
-nowarn (requires compiler prefix if set in properties file)	Passed to <code>javac</code> ; sets <code>-warn=none</code> .	n/a	javac
-O (requires compiler prefix if set in properties file)	Passed to <code>javac</code> ; disables <code>-linemap</code> .	n/a	javac
-offline	Determines the offline checker to use for semantics-checking. Specify a list of fully qualified class names.	oracle.sqlj.checker. OracleChecker	Advanced
-online	Determines the online checker to use for semantics-checking. Specify a fully qualified class name. (You must also set <code>-user</code> to enable online checking.)	oracle.sqlj.checker. OracleChecker	Advanced
-optcols	Enables iterator column type and size definitions to optimize performance. Used directly by the translator for Oracle-specific code generation, or forwarded to the Oracle customizer along with <code>user/password/URL</code> settings for ISO code generation.	false	Basic
-optparams	Enables parameter size definitions to optimize JDBC resource allocation (used with <code>-optparamdefaults</code>). This is used directly by the translator for Oracle-specific code generation, or forwarded to the Oracle customizer for ISO code generation.	false	Basic
-optparamdefaults	Sets parameter size defaults for particular datatypes (used with <code>-optparams</code>). This is used directly by the translator for Oracle-specific code generation, or forwarded to the Oracle customizer for ISO code generation.	false	Basic
-P	Prefix that marks options to pass to the SQLJ profile customizer.	n/a	Advanced

Table 8–1 SQLJ Translator Options (Cont.)

Option	Description	Default	Category
-parse	Option to enable the offline SQL parser. Possible settings: <code>both</code> , <code>online-only</code> , <code>offline-only</code> , <code>none</code> , or the name of a Java class that implements an alternative parser. Note: Some settings for this option will also disable online semantics-checking, overriding the effect of the <code>-user</code> option.	<code>both</code>	Advanced
-passes (command-line only)	Instructs the <code>sqlj</code> script to run SQLJ in two separate passes, with compilation in between.	<code>false</code>	Environment
-password (also recognized as <code>-p</code> if on command line)	Sets the user password for the database connection for online semantics-checking.	<code>none</code>	Basic
-profile	For ISO code generation, enables/disables the profile customization step for profile files generated during the current SQLJ run.	<code>true</code>	Advanced
-props (command-line only)	Specifies a properties file—an alternative to the command line for setting options. (The <code>sqlj.properties</code> is also still read.)	<code>none</code>	Basic
-ser2class	For ISO code generation, instructs SQLJ to translate generated <code>.ser</code> profiles to <code>.class</code> files.	<code>false</code>	Advanced
-status (also recognized as <code>-v</code> if on command line)	Requests SQLJ to display status messages as it runs.	<code>false</code>	Basic
-url	Sets the URL for the database connection for online semantics-checking.	<code>jdbc:oracle:oci:@</code>	Basic
-user (also recognized as <code>-u</code> if on command line)	Enables online semantics-checking and sets the user name (and optionally password and URL) for the database connection.	<code>none</code> (no online semantics-checking)	Basic
-verbose (requires compiler prefix if set in properties file)	Passed to <code>javac</code> ; enables <code>-status</code> .	<code>n/a</code>	<code>javac</code>

Table 8–1 SQLJ Translator Options (Cont.)

Option	Description	Default	Category
-version -version-long (both command-line only)	Flag to display different levels of SQLJ and JDBC driver version information.	disabled	Basic
-vm (command-line only)	Specifies the JVM to use for running the SQLJ translator.	java	Environment
-warn	Comma-delimited list of flags to enable or disable different SQLJ warnings. Individual flags are <code>cast/nocast</code> , <code>precision/noprecision</code> , <code>nulls/nonulls</code> , <code>portable/noportable</code> , <code>strict/nostRICT</code> , and <code>verbose/noverbose</code> . The global flag is <code>all/none</code> .	<code>cast</code> <code>precision</code> <code>nulls</code> <code>noportable</code> <code>strict</code> <code>noverbose</code>	Basic

Options for loadjava Compatibility

For compatibility with the `loadjava` utility used to load Java and SQLJ applications into Oracle9i, the following alternative syntax is recognized for the indicated options when specified on the command line. (This is also noted in [Table 8–1](#) above.)

- `-e` (for `-encoding`)
- `-h` (for `-help`)
- `-p` (for `-password`)
- `-u` (for `-user`)
- `-v` (for verbose message output; equivalent to `-status`)

To maintain full consistency with `loadjava` syntax, you can use a space instead of "=" in setting these options, as in the following example:

```
-u scott/tiger -v -e SJIS
```

For general information about the `loadjava` utility, see the *Oracle9i Java Developer's Guide*.

Note: This alternative option syntax is recognized only on the command line or in the `SQLJ_OPTIONS` environment variable, not in properties files.

Option Support for javac

SQLJ supports option settings for `javac`—the Java compiler supplied with the Sun Microsystems JDK—in several ways:

- Some `javac` options that take values are combined into SQLJ options (`-classpath`, `-d`, `-encoding`).
- For other `javac` options that take values, special processing has been implemented to correctly pass the value to the compiler (`-bootclasspath`, `-extdirs`, `-target`). These require a compiler prefix. They have no effect on SQLJ operation.
- Flags for `javac` are recognized on the command line without a compiler prefix (`-depend`, `-deprecation`, `-g`, `-nowarn`, `-O`, `-verbose`). Some of these flags affect SQLJ translator flag settings as well.

This is summarized in [Table 8–2](#). All of these options can be set in the SQLJ command line or in a properties file, though sometimes a compiler prefix is required, as noted in the table.

Notes:

- By default, `javac` compiles classes against the bootstrap and extension classes of the platform with which it was shipped. But `javac` also supports cross-compiling classes against bootstrap and extension classes of a different Java platform. The `javac -bootclasspath` and `-extdirs` options are for use in cross-compiling (JDK 1.2 or higher).
 - By default, `javac` generates `.class` files that are compatible with both JDK 1.1 and JDK 1.2 or higher. Use the `-target` option to alter this.
-
-

Table 8–2 *SQLJ Support for javac Options*

Command-Line Option (with -C Prefix if Noted)	Description	Relationship to SQLJ
<code>-C-bootclasspath</code>	Instructs <code>javac</code> to cross-compile against the specified set of bootstrap classes.	none
<code>-classpath</code>	Sets the classpath for <code>javac</code> and the JVM.	Also a SQLJ option.

Table 8–2 SQLJ Support for javac Options (Cont.)

Command-Line Option (with -C Prefix if Noted)	Description	Relationship to SQLJ
-d	Sets the output directory for <code>.class</code> files and SQLJ profile files.	Also a SQLJ option.
-depend	Instructs <code>javac</code> to compile out-of-date files recursively.	Enables the SQLJ <code>-checksource</code> option.
-deprecation	Instructs <code>javac</code> to output source locations where deprecated APIs are used.	none
-encoding	Sets the encoding for both SQLJ and <code>javac</code> .	Also a SQLJ option.
-C-extdirs	Instructs <code>javac</code> to cross-compile against the specified extension directories.	none
-g	Generates <code>javac</code> debugging information.	Enables the SQLJ <code>-linemap</code> option.
-nowarn	Instructs <code>javac</code> to generate no warnings.	Sets the SQLJ option <code>-warn=none</code> .
-O	Instructs <code>javac</code> to optimize.	Disables the SQLJ <code>-linemap</code> option.
-C-target	Instructs <code>javac</code> to generate <code>.class</code> files to work only on JVMs of the specified JDK version level or higher (1.1, 1.2, or 1.3).	none
-verbose	Instructs <code>javac</code> to output real-time status messages.	Enables the SQLJ <code>-status</code> option.

Refer to `javac` documentation for additional information about `javac` option settings and functionality.

Syntax Notes for `javac` Options:

- If you want to set different classpath values for the Java compiler and for the JVM that runs SQLJ, you must use separate settings, one with a `-C` prefix and one with a `-J` prefix. Otherwise, no prefix is required.
- Do not use the `-C` prefix to specify the `-d` or `-encoding` compiler options. Note that this also means that SQLJ and the compiler use the same settings for `-d` and `-encoding`.

- You can optionally use the `-C` prefix for `-depend`, `-deprecation`, `-g`, `-nowarn`, `-O`, and `-verbose`.
- All `javac` options, aside from those that are also SQLJ options (`-classpath`, `-d`, and `-encoding`) require the `"compile."` prefix if you set them in a properties file.
- For consistency, it is advisable to use an equals sign (=) for options that take values, but a space also works when using a compiler prefix (`-C` on the command line or `"compile."` in a properties file).

Example The following example (a single wraparound command line) uses the `-C-bootclasspath`, `-C-extdirs`, and `-C-target` options.

```
% sqlj -vm=/usr/local/packages/jdk1.2.2/bin/java
      -compiler-executable=/usr/local/packages/jdk1.2.2/bin/javac
      -C-bootclasspath=/usr/local/packages/jdk1.3.1/jre/lib/rt.jar
      -C-extdirs="" -C-target=1.1.8 Demo.sqlj
```

Profile Customizer Options

Profile customizer options—options for the customizer harness front end, the default Oracle customizer, and special customizers for debugging and deployment-time semantics-checking—are documented in ["Customization Options and Choosing a Customizer"](#) on page A-11. This is relevant for ISO standard code generation only (`-codegen=iso`).

Command-Line Syntax and Operations

The general sequence of events triggered by running the script `sqlj` was discussed in ["SQLJ Translation Steps"](#) on page 1-10. This section will add some operational details to that discussion, as part of this overview of the command line.

Use of Command-Line Arguments

Recall the typical general syntax for the command line:

```
sqlj <optionlist> filelist
```

When the `sqlj` script invokes a JVM, it passes all of its command-line arguments to the JVM, which later passes them elsewhere (such as to the Java compiler or profile customizer) as appropriate.

Use an equals sign (=) to specify option and flag settings, although for simplicity you do not have to specify `=true` to turn on a flag—typing the flag name alone will

suffice. 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

Note: If the same option appears more than once on the command line (or in the properties file), then the last value is used.

Arguments from the Option List Option list arguments are used in the following ways:

- Options not designated by the `-J`, `-C`, or `-P` prefixes are SQLJ options (except for directly supported compiler options) and are passed to the SQLJ translator as the JVM invokes it.
- Options designated by the `-J` prefix are JVM options and are used by the JVM directly. Such options must be specified on the command line or in the `SQLJ_OPTIONS` environment variable. As with translator options, use an equals sign (=) in setting the option, such as:

```
-J-Djavac.pipe.output=true
```

If you want to set different classpath values for the Java compiler and for the JVM that runs SQLJ, you must use separate settings, one with a `-C` prefix and one with a `-J` prefix.

- Options designated by the `-C` prefix are Java compiler options and are passed to the compiler as the JVM invokes it. Compiler options taking values require special support, which has been implemented for `javac` options, as described in "[Option Support for javac](#)" on page 8-10. You can use an equals sign for these, as follows (though a space also works):

```
-C-bootclasspath=/usr/local/packages/jdk1.3.1/jre/lib/rt.jar
```

- Options designated by the `-P` prefix are SQLJ profile customizer options and are passed to the customizer as the JVM invokes it (relevant only for ISO standard code generation, `-codegen=iso`). As with translator options, use an equals sign (=) in setting the option, such as:

```
-P-user=scott/tiger
```

Any profile customization other than what SQLJ performs automatically is considered an advanced feature and is covered in [Appendix A, "Customization and Specialized Customizers"](#).

Arguments from the File List The SQLJ front end parses the file list, processes wildcard characters, and expands file names. By default, files are processed as follows:

- The `.sqlj` files are processed by the SQLJ translator, Java compiler, and SQLJ profile customizer (profile customizer for `-codegen=iso` only).
- The `.java` files are processed by the Java compiler and are also used by the SQLJ translator for type resolution.
- The `.ser` profiles and `.jar` files are processed only by the profile customizer (relevant only for `-codegen=iso`).

Note that you can specify `.sqlj` files together with `.java` files on the command line, or you can specify `.ser` files together with `.jar` files, but you cannot mix the two categories. (See ["JAR Files for Profiles"](#) on page A-38 for details about how `.jar` files are processed.)

If you have `.sqlj` files and `.java` files with interdependencies (each requiring access to code in the others), then enter them all on the command line for a single execution of SQLJ. You cannot specify them for separate executions of SQLJ, because then SQLJ would be unable to resolve all the types.

Note: As an alternative to entering `.java` file names on the command line, you can enable the `-checksource` option and then just be sure that the `.java` files are in the classpath. See ["Source Check for Type Resolution \(-checksource\)"](#) on page 8-69.

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. In addition, 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`
 The translator discards `Foo.java` from the command line. Otherwise, this command line would result in the translator both writing to and reading from `Foo.java` in the same execution.
- `sqlj Foo.*`
 Again, the translator discards `Foo.java` from the command line. Otherwise, the translator would find both `Foo.sqlj` and `Foo.java`, which again would cause it to both write to and read from `Foo.java` in the same execution.
- `sqlj -dir=outdir -d=outclasses 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. Presuming that `Foo.java` and `Foo.sqlj` define classes in different packages, the `.class` files created by Java compilation will be placed in different subdirectories under the `outclasses` directory hierarchy.

This processing of the command line means that you can, for example, type the following command and have it execute without difficulty (with file references being automatically discarded as necessary):

```
sqlj *.sqlj *.java
```

This is convenient in many situations.

Command-Line Example and Results

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 JVM, passes it the class name of the SQLJ translator, then passes it the command-line arguments. The JVM passes the SQLJ options to the translator and compiler. If there are any options for the JVM, as designated by `-J`, the script passes them to the JVM ahead of the translator class file name (just as you would type Java options prior to typing the class file name if you were invoking

Java by hand). There is no customization in this example, because it uses the default Oracle-specific code generation.

After these steps are completed, the results are equivalent to the user having typed the following (presuming `SushiBar.sqlj`, `DiveBar.sqlj`, `FooBar.java`, and `FooBaz.java` were all in the current directory):

```
java -Duser.language=ja -prof sqlj.tools.Sqlj -warn=none -encoding=SJIS
SushiBar.sqlj DiveBar.sqlj FooBar.java FooBaz.java
```

(This is one wraparound command line.)

For more information about how JVM options are handled, see "[Options to Pass to the Java Virtual Machine \(-J\)](#)" on page 8-60.

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 "[Command Line Echo without Execution \(-n\)](#)" on page 8-27.

Properties Files for Option Settings

You can use *properties files*, instead of the command line, to set options for the SQLJ translator, Java compiler, and SQLJ profile customizer (for ISO code generation).

In addition, if your Java compiler will be running in a separate JVM and you want to specify options to this JVM regarding operation of the compiler, then you can use properties files to supply such options. Such options are passed to the JVM at the time the compiler is run, after the SQLJ translation step. (It is more typical, however, to pass options to the JVM of the compiler by using the command-line `-C-J` prefix.)

You *cannot* use properties files to set the following SQLJ options, flags, and prefixes:

- `-classpath`
- `-help`, `-help-long`, `-help-alias`, `-C-help`, `-P-help`
- `-J`
- `-n`
- `-passes`
- `-props`

- `-version, -version-long`
- `-vm`

It is not possible to use properties files to specify options to the JVM, for example, because properties files are read after the JVM is invoked.

Also note that in properties files you cannot use option abbreviations recognized on the command line for compatibility with `loadjava` (`-e, -h, -p, -u, -v`).

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, and only a small subset of options are supported. For information, see ["Option Support in the Server Embedded Translator"](#) on page 11-18.

Properties File Syntax

Option settings in a properties file are placed one per line. Lines with SQLJ options, compiler options, and customizer options can be interspersed. They are parsed by the SQLJ front end and processed appropriately.

Syntax for the different kinds of options is as follows:

- Each SQLJ option is prefixed by `"sqlj."` (including the period) instead of an initial hyphen; only options that start with this prefix are passed to the SQLJ translator. For example:

```
sqlj.warn=none
sqlj.linemap=true
```

- Each Java compiler option is prefixed by `"compile."` (including the period) instead of `"-C-"`; options that start with this prefix are passed to the Java compiler. For example:

```
compile.verbose
compile.bootclasspath=/usr/local/packages/jdk1.3.1/jre/lib/rt.jar
```

- 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 this prefix are passed to the profile customizer. For example:

```
profile.backup
profile.user=scott/tiger
```

You can also specify options to a particular customizer by using "profile.C" as follows:

```
profile.Csummary
profile.Coptparamdefaults=VAR%(50),LONG%(500),RAW_TYPE()
```

Any profile customization other than the default Oracle customization is considered an advanced feature and is covered in [Appendix A, "Customization and Specialized Customizers"](#).

- Comment lines start with a pound sign (#). For example:
Comment line.
- Blank lines are also permitted.

As on the command line, a flag can be enabled/disabled in a properties file with =true/=false, =on/=off, =1/=0, or =yes/=no. A flag can also be enabled simply by entering it without a setting, such as the following:

```
sqlj.linemap
```

Note: For consistency, it is best to always use the equals sign (=) in a properties file for options that take values, even though there are some circumstances where a space also works.

Properties File: Simple Example The following are sample properties file entries:

```
# Set user and JDBC driver
sqlj.user=scott
sqlj.driver=oracle.jdbc.OracleDriver

# Turn on the compiler verbose option
compile.verbose
```

These entries are equivalent to having the following on the SQLJ command line:

```
sqlj -user=scott -driver=oracle.jdbc.OracleDriver -C-verbose
```

Properties File: Nondefault Connection Context Classes Following is a sample properties file that specifies settings for a connection context class `SourceContext` that you declared:

```
# JDBC driver
sqlj.driver=oracle.jdbc.OracleDriver

# Oracle 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 8-20.

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 JVM. And it is particularly useful for command-line-only options, such as `-classpath`, that you use repeatedly with the same setting.

Following is an example of a `SQLJ_OPTIONS` setting:

```
-vm=jview -J-verbose
```

When you use `SQLJ_OPTIONS`, SQLJ effectively inserts the `SQLJ_OPTIONS` settings, in order, at the beginning of the SQLJ command line, prior to any other command-line option settings.

Note: Generally, syntax in `SQLJ_OPTIONS` is the same as on the command line, but this may depend on your operating system. There can be 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. It sets options to default settings (where applicable).
2. It looks for a `sqlj.properties` file in the Java home directory; if it finds one, it sets options as specified there.
3. It looks for a `sqlj.properties` file in the user home directory; if it finds one, it sets options as specified there.
4. It looks for a `sqlj.properties` file in the current directory; if it finds one, it sets options as specified there.
5. It looks for option settings in the `SQLJ_OPTIONS` environment variable and effectively prepends them to the beginning of the command line. It sets options as specified in `SQLJ_OPTIONS`.
6. It looks for option settings on the command line and sets options as specified there. As SQLJ processes the command line, it looks in any file specified by the `-props` option and sets options as specified there.

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 effectively inserts the option settings of the file into the position on the command line where the `-props` option was specified.
 - SQLJ reads options on the command line, with options from a `-props` file inserted, in order from left to right. Any later (right-hand) setting takes precedence over earlier (left-hand) settings.
-

Example Presume SQLJ is run as follows:

```
sqlj -user=scott -props=myprops.properties -dir=/home/java
```

And presume the file `myprops.properties` is in the current directory and contains the following entries:

```
sqlj.user=tony  
sqlj.dir=/home/myjava
```

These settings are processed as if they were inserted into the command line where the `-props` option was specified. Therefore, the `tony` entry takes precedence over the `scott` entry for the `user` option, but the `/home/java` entry takes precedence over the `/home/myjava` entry for the `dir` option.

Basic Translator Options

This section documents the syntax and functionality of the basic flags and options you can specify in running SQLJ. These options 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 8-16.)

This discussion includes the following categories:

- [Basic Options for the Command Line Only](#)
- [Options for Output Files and Directories](#)
- [Connection Options](#)
- [Options for Reporting and Line-Mapping](#)
- [Options for Code Generation, Optimizations, and CHAR Comparisons](#)

More advanced command-line flags and options are discussed in ["Advanced Translator Options"](#) on page 8-60 and ["Translator Support and Options for Alternative Environments"](#) on page 8-80.

Basic Options for the Command Line Only

The following basic options can be specified only on the SQLJ command line or, equivalently, in the `SQLJ_OPTIONS` environment variable. 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: Additionally, there are advanced options, flags, and prefixes that can be set only on the command line or in `SQLJ_OPTIONS`: `-J`, `-passes`, and `-vm`.

Input Properties File (-props)

The `-props` option specifies a properties file from which SQLJ can read option settings (an alternative to specifying option settings on the command line).

See ["Properties Files for Option Settings"](#) on page 8-16 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 Virtual Machine and Compiler (-classpath)

For compatibility with the syntax of most JVMs and compilers, SQLJ recognizes the `-classpath` option if it is specified on the command line. In setting this option, you can use either a space, as with most JVMs or compilers, or "=", as with other SQLJ options. The following examples (both for a UNIX environment) demonstrate this:

```
-classpath . : ${ORACLE_HOME}/jdbc/lib/classes12.jar:${ORACLE_HOME}/jdbc/lib/nls_charset12.jar
```

or:

```
-classpath . : ${ORACLE_HOME}/jdbc/lib/classes12.jar:${ORACLE_HOME}/jdbc/lib/nls_charset12.jar
```

The `-classpath` option sets the Java classpath for both the JVM and the Java compiler. If you do not want to use the same classpath for both, set them separately using the SQLJ `-J` and `-C` prefixes, described in ["Prefixes that Pass Option Settings to Other Executables"](#) on page 8-60.

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 JVM and compiler. This is because JVMs 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 performed when you use the `-help` flag in any of these forms, even if you include file names and other options on the command line as well. SQLJ assumes that you either want to run the translator or you want help, but not both.

You can also receive information about the profile customizer or Java compiler, requesting help through the `-P` and `-C` prefixes as in the following examples. These prefixes are discussed in ["Prefixes that Pass Option Settings to Other Executables"](#) on page 8-60. As with the `-help` flag, no translation is performed if you request customizer or compiler help.

```
sqlj -P-help
```

```
sqlj -C-help
```

As with other command-line-only flags, `-help` (as well as `-P-help` and `-C-help`) does *not* support `=true` syntax. Enable it by typing only the desired flag setting.

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 8-9.
 - You can use multiple `-help` flag settings on the same command line, including `-P-help` and `-C-help`.
 - Although `-P` and `-C` settings can generally be set in properties files, `-P-help` and `-C-help` are for only the command line.
 - Help is also provided if you run SQLJ without specifying any files to process. This is equivalent to using the `-help` setting.
-
-

The `-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 `int` or `String`)
- description
- current value

- how the current value was set (from the command line, from a properties file, or by default)

Note: It is often useful to include other option settings on the command line with a `-help-long` option, especially with complex options (such as `-warn`) or combinations of options, so that you can see what option settings resulted from your actions. (The `-help-long` mode displays current settings of all options.)

The `-help-alias` Setting This setting provides a synopsis of the command-line abbreviations 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 disabled

SQLJ Version Number (`-version`)

The following settings of the `-version` flag, specified on the command-line, instruct SQLJ to display varying levels of information about SQLJ and JDBC driver versions:

- `-version`
- `-version-long`

You can enable this option by typing the desired setting on the command line as in the following examples:

```
sqlj -version
```

or:

```
sqlj -version-long
```

No input-file translation is performed when you use the `-version` option, even if you include file names and other options on the command line. SQLJ assumes that you either want to run the translator or you want version information, but not both. Properties files and anything else you type on the command line are ignored.

As with other command-line-only flags, `-version` does *not* support `=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 9.2".

The `-version-long` Setting The `-version-long` setting displays information about the SQLJ and SQLJ runtime library release and build versions, the JDBC driver release number if one can be found, and the Java environment. For example, if an Oracle JDBC driver is used, this option would display something such as "Oracle JDBC version 9.2 (9.2.0.0)".

This flag offers a good way to check your SQLJ installation and the JDBC and JDK versions you are using.

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 disabled

Command Line Echo without Execution (-n)

The `-n` flag, specified on the command line, instructs the `sqlj` script to construct the full command line that would be passed to the SQLJ translator, including any `SQLJ_OPTIONS` settings, and echo it to the user without having the SQLJ translator execute it. This includes capturing and echoing the name of the JVM that would be

launched to execute the SQLJ translator and echoing the full class name of the translator. This does *not* include settings from properties files.

This is useful in 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.

As with 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 wraparound line.)

Note:

- As an alternative to `-n`, you can use the `-vm=echo` setting.
 - Another effective way to check option settings is to use the `-help-long` flag. This displays current settings for all options, including other options you set on the command line as well as settings in properties files and in `SQLJ_OPTIONS`. See "[SQLJ Option Information \(-help\)](#)" on page 8-24.
-
-

Command-line syntax `-n`

Command-line example `-n`

Properties file syntax `n/a`

Properties file example `n/a`

Default value `disabled`

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 the encoding to be applied to `.sqlj` and `.java` input files and `.java` generated files for globalization support. For compatibility with `javac`, you can use either a space or "=" in setting this option on the command line, as in the following examples:

```
-encoding=SJIS
```

```
-encoding SJIS
```

If setting `sqlj.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.

Note the following:

- 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 JVM and compiler. This is because JVMs and compilers do not support the "=" syntax in their option settings.

- For compatibility with the `loadjava` utility, `-e` is recognized as equivalent to `-encoding` when specified on the command line. See ["Options for loadjava Compatibility"](#) on page 8-9.
- The `-encoding` option does not apply to Java properties files—`sqlj.properties` and `connect.properties`, for example. Properties files always use the encoding `8859_1`. This is a feature of Java in general, not SQLJ in particular. You can, however, use Unicode escape sequences in a properties file. (You can use the `native2ascii` utility to create escape sequences for a natively encoded file—see ["Using native2ascii for Source File Encoding"](#) on page 9-30.)

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 JVM system property `file.encoding`

Output Directory for Generated `.ser` and `.class` Files (-d)

The `-d` option specifies the root output directory for profiles generated by the SQLJ translator (relevant for ISO standard code generation, `-codegen=iso`), and is also passed to the Java compiler to specify the root output directory for `.class` files generated by the compiler. Whether profiles are generated as `.ser` files (default) or `.class` files (if the `-ser2class` option is enabled) is irrelevant for placement through the `-d` option.

Whenever a directory is specified, the output files are generated under this directory according to the package name, if applicable. For example, if you have source files in package `a.b.c` and specify directory `/mydir`, output files will be placed in the `/mydir/a/b/c` directory.

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 empty or not specified, then `.class` files and `.ser` files generated by the translation process will be placed as follows:

- A `.class` file corresponding to a `.java` file that was generated by the translator is placed in the same directory as the generated `.java` file, which is according to the `-dir` option.
- A `.class` file corresponding to a `.java` file that you specified on the command line will be placed in the same directory as the `.java` file.
- A `.ser` file is placed in the same directory as the `.sqlj` source file from which it resulted.

Notes:

- You can specifically set `-d` to be empty (to override settings in a properties file, for example) as follows:
`-d=`
 - 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 JVM.
 - 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 JVM and compiler. This is because JVMs 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 (`.class` files go with `.java` files; `.ser` files go with `.sqlj` files)

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 package `a.b.c` and specify directory `/mydir`, then output files will be placed in the `/mydir/a/b/c` directory.

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:

```
-dir=mysubdir/myothersubdir
```

then `/root/home/mydir/mysubdir/myothersubdir` will be the root directory for generated `.java` files.

You can also use standard syntax such as a period for the current directory or two periods to go up a level (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=
```

Notes:

- If you specify the `-dir` option but not the `-d` option, then generated `.class` files will also be placed in the directory specified by `-dir`, but generated `.ser` files will be placed in the directory of the `.sqlj` file.
 - 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 JVM.
-
-

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

You can use the following options for the database connection for online semantics-checking:

- `-user`
- `-password`
- `-url`
- `-default-url-prefix`
- `-driver`

There is no requirement for the SQLJ translator to connect to the same database or schema as the application does at runtime. The connection information in application source code can be independent of the connection information in the SQLJ options. In fact, the deployment environment might be unavailable during development and testing.

Online Semantics-Checking and User Name (-user)

Simple semantics-checking not involving a database connection is referred to as *offline checking*. The more thorough semantics-checking requiring a connection is referred to as *online checking*. Online checking offers one of the prime advantages of the SQLJ strong-typing paradigm—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 user name (schema name) for the exemplar schema, which is the sample database schema that you provide to the translator for it to use in performing the checking. You can also use the `-user` option to specify the password and URL, as opposed to using the `-password` and `-url` options separately.

Note that there is no other flag to enable or disable online semantics-checking; SQLJ enables 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 employing the default connection context class only; and 2) effect of `-user` when you are employing nondefault or multiple connection context classes. Nondefault 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 4-6.

Notes:

- Some settings of the SQLJ `-parse` option will disable online semantics-checking, overriding the effect of the `-user` option. See "[Offline Parser \(-parse\)](#)" on page 8-79.
- 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 8-9.
- User names cannot contain the characters `"/` or `"@`.
- You are allowed to use a space instead of `"=` in a user name setting on the command line, as in the following examples:

```
-user scott/tiger
-user@CtxClass scott/tiger
-u scott/tiger
-u@CtxClass scott/tiger
```

- If a password contains the character `"@`", then you cannot set the password through the `-user` option. You must use separate `-user` and `-password` settings.
 - If your login name is a member of the DBA group, you may have special privilege to connect as `SYSDBA` to the `SYS` schema. In this case, you can specify the user name `SYS` or `INTERNAL`.
 - For ISO code generation, the translator `-user` setting is forwarded to the profile customizer, but can be overridden by the customizer `user` setting.
-
-

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 your SQLJ executable statements. This example results in online checking against the `scott` schema.

You can also specify the password, URL, or both along with the user name, using syntax as in the following examples (with `"/"` preceding the password and `"@"` preceding the URL):

```
-user=scott/tiger
```

or:

```
-user=scott@jdbc:oracle:oci:@
```

or:

```
-user=scott/tiger@jdbc:oracle:oci:@
```

Otherwise the URL can be specified through the `-url` option, and the password can be specified interactively or through the `-password` option.

You can disable online semantics-checking by setting the `-user` option to an empty string:

```
-user=
```

Again, when you are using only the default connection or other instances of the `DefaultContext` class, this will apply to all your SQLJ executable statements.

Disabling online semantics-checking is useful, for example, if you have online checking enabled in a properties file but want to override that on the command line, or have it enabled in the default properties file but want to override that in a user-specified properties file (specified using the `-props` option).

There is also a special user name, `URL.CONNECT`, which you can use when the URL specifies the user and password as well as the other details of the connection. To see what the URL would look like in such a case, see "[Connection URL for Online Semantics-Checking \(-url\)](#)" on page 8-41.

Effect of -user When Using Nondefault 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 user name for online checking against a particular connection context class (`CtxClass`, for example) as follows:

```
-user@CtxClass=scott
```

This results in online checking against the `scott` schema for any of your SQLJ executable statements that specify a connection context instance of the class `CtxClass`.

As with the default connection context class, you can also specify the password or URL in your `-user` setting for a particular connection context class, as in the following example:

```
-user@CtxClass=scott/tiger@jdbc:oracle:oci:@
```

The `CtxClass` connection context class must be declared in your source code or previously compiled into a `.class` file. (See "[Connection Contexts](#)" on page 7-2 for more information.)

Employ the `-user` option separately for each connection context class for which you want to enable online checking and set a user name; these settings have no influence on each other:

```
-user@CtxClass1=user1 -user@CtxClass2=user2 -user@CtxClass3=user3
```

When you are using multiple connection context classes in your application, a `-user` setting that does not specify a class will apply to the `DefaultContext` class as well as to all classes for which you do not otherwise specify a `-user` setting. Presumably, though, you will specify a `-user` setting for each connection context class, given that different connection context classes are typically intended for use with different sets of SQL objects.

Consider a situation where you have declared connection context classes `CtxClass1`, `CtxClass2`, and `CtxClass3` and you set `-user` as follows:

```
-user@CtxClass2=scott/tiger -user=bill/lion
```

Any statement in your application that uses an instance of `CtxClass2` will be checked against the `scott` schema. Any statement that uses an instance of `DefaultContext`, `CtxClass1`, or `CtxClass3` will be checked against the `bill` schema.

In addition, once you enable online checking by setting the `-user` option, you can disable online checking for a particular connection context by setting the `-user` option again with an empty user name for that connection context. For example, consider the following setting:

```
-user@CtxClass2=
```

This disables online semantics-checking for any SQLJ executable statements that specify a connection object that is an instance of `CtxClass2`.

To disable online semantics-checking for the default connection context class and any other connection context classes for which you do not specify a user name:

```
-user=
```

Command-line syntax `-user<@conn_context_class>=username</password><@url>`

Command-line examples

```
-user=scott
-user=scott/tiger
-user=scott@jdbc:oracle:oci:@
-user=scott/tiger@jdbc:oracle:oci:@
-user=
-user=URL.CONNECT
-user@CtxClass=scott/tiger
-user@CtxClass=
```

Properties file syntax `sqlj.user<@conn_context_class>=username</password><@url>`

Properties file examples

```
sqlj.user=scott
sqlj.user=scott/tiger
sqlj.user=scott@jdbc:oracle:oci:@
sqlj.user=scott/tiger@jdbc:oracle:oci:@
sqlj.user=
sqlj.user=URL.CONNECT
sqlj.user@CtxClass=scott/tiger
sqlj.user@CtxClass=
```

Default value none (no online semantics-checking)

Note: Be aware of the difference in format between specifying user, password, and URL in the `-user` option and specifying them in the `-url` option. In the `-url` option, the user name and password are included in the URL, immediately following the JDBC driver type; in the `-user` option they precede the URL. Also see "[Connection URL for Online Semantics-Checking \(-url\)](#)" on page 8-41.

User Password for Online Semantics-Checking (`-password`)

The `-password` option specifies the user password for the database connection for online semantics-checking. For the `-password` setting to be meaningful, the `-user` option must also be set.

You can also specify the password as part of the `-user` option setting. See "[Online Semantics-Checking and User Name \(-user\)](#)" on page 8-34. 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 exemplar schemas to use in testing statements that use those connection context classes. Similarly, use the `-password` option to specify passwords for those schemas, as in the following example:

```
-password@CtxClass1=tiger
```

A connection context class without a password setting, either through the `-password` setting or the `-user` setting, uses the password setting for the default connection context class. If you set no password for the default connection context class, then SQLJ prompts you interactively for that password. If you also set no password for a user-defined connection context class, then SQLJ prompts you interactively for that password as well. An exception to this discussion is where user name `URL.CONNECT` is used, as discussed in "[Connection URL for Online Semantics-Checking \(-url\)](#)" on page 8-41. In this case, user name and password are

determined from the string specified in the `-url` setting, and any setting of the `-password` option is ignored.

You can specifically set an empty password to override other settings of the `-password` option, such as in a properties file, and be prompted interactively. You can do this for the `DefaultContext` class or any particular connection context class, as in the following examples:

```
-password=
```

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
```

The Oracle9i database, however, does not permit an empty password.

Notes:

- When specified on the command line, `-p` is recognized as equivalent to `-password`.
 - You are allowed to use a space instead of "=" in a password setting on the command line, as in the following examples:

```
-password tiger  
-password@CtxClass tiger  
-p tiger  
-p@CtxClass tiger
```
 - For ISO code generation, the translator `-password` setting is forwarded to the profile customizer, but can be overridden by the customizer `password` setting.
-
-

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 (password for `DefaultContext` is used, or user is prompted)

Connection URL for Online Semantics-Checking (-url)

The `-url` option specifies a URL for establishing a database connection for online semantics-checking. As necessary, the URL can include a host name, port number, and Oracle SID.

You can also specify the URL as part of the `-user` option setting. See "[Online Semantics-Checking and User Name \(-user\)](#)" on page 8-34. 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 connection for checking all your SQLJ statements:

```
-url=jdbc:oracle:oci:@
```

or, to include the 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:@
```

A `-url` setting of `localhost:1521:orcl` would result in the following URL:

```
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 8-43.

You can specify the user and password in the `-url` setting, instead of in the `-user` and `-password` settings. In such a case, set `-user` to `URL.CONNECT`, as follows:

```
-url=jdbc:oracle:oci:scott/tiger@ -user=URL.CONNECT
```

If you declare and use additional connection context classes, `CtxClass1` for example, you will presumably specify additional exemplar schemas to use in testing statements that use those connection context classes. You can use the `-url` option to specify URLs for those schemas, as in the following example:

```
-url@CtxClass1=jdbc:oracle:oci:@
```

Any connection context class without a URL setting, either through the `-url` setting or the `-user` setting, uses the URL setting for the default connection context class, presuming a URL has been set for the default context class.

Notes:

- Remember that any connection context class with a URL setting must also have a user name setting for online checking to occur.
- You are allowed to use a space instead of "=" in a URL setting on the command line, as in the following examples:

```
-url jdbc:oracle:oci:@  
-url@CtxClass jdbc:oracle:oci:@
```

- For ISO code generation, the translator `-url` setting is forwarded to the profile customizer, but can be overridden by the customizer `url` setting.
-
-

Command-line syntax `-url<@conn_context_class>=URL`

Command-line examples

```
-url=jdbc:oracle:oci:@  
-url=jdbc:oracle:thin:@hostname:1521:orcl  
-url=jdbc:oracle:oci:scott/tiger@
```

```
-url=hostname:1521:orcl
-url@CtxClass=jdbc:oracle:oci:@
```

Properties file syntax `sqlj.url<@conn_context_class>=URL`

Properties file examples

```
sqlj.url=jdbc:oracle:oci:@
sqlj.url=jdbc:oracle:thin:@hostname:1521:orcl
sqlj.url=jdbc:oracle:oci:scott/tiger@
sqlj.url=hostname:1521:orcl
sqlj.url@CtxClass=jdbc:oracle:oci:@
```

Default value `jdbc:oracle:oci:@`

Note: Be aware of the difference in format between specifying user, password, and URL in the `-user` option and specifying them in the `-url` option. In the `-url` option, the user name and password are included in the URL, immediately following the JDBC driver type. In the `-user` option, they precede the URL. Also see "[Online Semantics-Checking and User Name \(-user\)](#)" on page 8-34.

Default URL Prefix (-default-url-prefix)

Use the `-default-url-prefix` option to alter or remove the default prefix.

The following is the default prefix for any URL setting you specify that does not already start with `jdbc`:

```
jdbc:oracle:thin:@
```

This allows you to use a shorthand in specifying a URL setting, either in the `-user` option or the `-url` option—it is permissible to specify only the host, port, and 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:oci:@orcl
```

However, if you want your URL settings to default to the OCI driver, for example, instead of the Thin driver, then set the default prefix as follows:

```
-default-url-prefix=jdbc:oracle:oci:@
```

With this prefix, a setting of `-url=orcl` is equivalent to the `-url=jdbc:oracle:oci:@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:oci:@  
-default-url-prefix=
```

Properties file syntax `sqlj.default-url-prefix=url_prefix`

Properties file examples

```
sqlj.default-url-prefix=jdbc:oracle:oci:@  
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-delimited list of classes.

The default, `OracleDriver`, supports the Oracle OCI, Thin, and server-side JDBC drivers for use with Oracle9i.

Command-line syntax `-driver=driver1<,driver2,driver3,...>`

Command-line examples

```
-driver=oracle.jdbc.OracleDriver
-driver=oracle.jdbc.OracleDriver,sun.jdbc.odbc.JdbcOdbcDriver
```

Properties file syntax `sqlj.driver=driver1<,driver2,driver3,...>`

Properties file examples

```
sqlj.driver=oracle.jdbc.OracleDriver
sqlj.driver=oracle.jdbc.OracleDriver,sun.jdbc.odbc.JdbcOdbcDriver
```

Default value `oracle.jdbc.OracleDriver`

Options for Reporting and Line-Mapping

The following options specify what types of conditions SQLJ should monitor, whether to generate real-time error and status messages, and whether to include "cause" and "action" information with translator error messages:

- `-warn`
- `-status`
- `-explain`

The following options enable line-mapping from the generated Java `.class` file back to the `.sqlj` source file, so that you can trace runtime errors back to the appropriate location in your original source code. Use `-jdblinemap` in conjunction with the Sun Microsystems `jdb` debugger; otherwise use `-linemap`.

- `-linemap`
- `-jdblinemap`

Translator Warnings (-warn)

There are various warnings and informational messages that the SQLJ translator can display as dictated by conditions it encounters during the translation. The `-warn` option consists of a set of flags that specify which of those warnings and messages should be displayed (in other words, which conditions should be monitored and which should be ignored).

All the flags for this option must be combined into a single, comma-delimited string.

Table 8-3 documents the conditions that can be tested, what the `true` and `false` flag values are for each condition, what a `true` flag value means, and which value is the default.

Table 8-3 Tests and Flags for SQLJ Warnings

Tests and Flag Functions	TRUE/FALSE Values
Test for requirement of subtypes of declared object type in an inheritance hierarchy—Enable <code>cast</code> to receive warnings when usage of SQL object types in a SQL inheritance hierarchy requires that subtypes of a declared type must be passed at runtime.	<code>cast</code> (default) <code>nocast</code>
Data precision test—Enable <code>precision</code> to receive warnings if there was a possible loss of precision when moving values from database columns to Java host variables.	<code>precision</code> (default) <code>noprecision</code>
Conversion loss test for nullable data—Enable <code>nulls</code> to receive warnings if there was possible conversion loss when moving nullable columns or nullable Java types from database columns to Java host variables.	<code>nulls</code> (default) <code>nonnulls</code>
Portability test—Enable <code>portable</code> to check SQLJ clauses for portability and receive warnings if there are non-portable clauses. (Where <i>non-portable</i> refers to the use of extensions to the SQLJ standard, such as vendor-specific types or features.)	<code>portable</code> <code>noportable</code> (default)
Strict matching test for named iterators—Enable <code>strict</code> to instruct SQLJ to require that the number of columns selected from the database must equal the number of columns in the named iterator being populated. A warning is issued for any column in the database cursor for which there is no corresponding column in the iterator. The <code>nostrict</code> setting allows more (but not fewer) columns in the database cursor; unmatched columns are ignored.	<code>strict</code> (default) <code>nostrict</code>
Translation-time informational messages—Enable <code>verbose</code> to provide additional informational messages about the translation process (such as what database connections were made for online checking).	<code>verbose</code> <code>noverbose</code> (default)
Global enabling/disabling of warnings—Use <code>all</code> or <code>none</code> to enable or disable all warnings.	<code>all</code> <code>none</code>

The `verbose/noverbose` flag works differently from the others. It does not enable a particular test but enables output of general informational messages about the semantics-checking.

Note: Do not confuse `-warn=verbose` with the `-status` flag. The `-status` flag provides real-time informational messages about all aspects of SQLJ translation—translation, semantics-checking, compilation, and profile customization (if applicable). The `-warn=verbose` flag results in additional reporting about the translation phase only.

The global `all/none` flag takes priority over default settings. You can use it to enable or disable all flags, or to serve as an initialization to make sure all flags are off before you turn selected flags on, or vice versa.

The `all` setting is equivalent to the following:

```
cast,precision,nulls,portable,strict,verbose
```

And the `none` setting is equivalent to the following:

```
nocast,noprecision,nonulls,noportable,nostrict,noverbose
```

There is no default for `all/none`; there are only defaults for individual flags.

Here are some examples:

- Use the following sequence to make sure only the `nulls` flag is on:

```
-warn=none,nulls
```
- This sequence will have the same result, because the `verbose` setting will be overridden:

```
-warn=verbose,none,nulls
```
- Use the following to make sure everything except the portability flag is on:

```
-warn=all,noportable
```
- This sequence will have the same result, because the `nonulls` setting will be overridden:

```
-warn=nonulls,all,noportable
```

Other than placement of the `all/none` flag, the order in which flags appear in a `-warn` setting is unimportant, except in the case of conflicting settings. If there are conflicts—such as in `-warn=portable,noportable`—then the last (right-most) setting is used.

Separate settings of the `-warn` option in properties files and on the command line are *not* cumulative. Only the last setting is processed. In the following example, the `-warn=portable` setting is ignored—that flag and all other flags besides `nulls/nonulls` are set according to their defaults:

```
-warn=portable -warn=nulls
```

Note: The cast, precision, nullability, and strictness tests are part of online semantics-checking and require a database connection.

Command-line syntax `-warn=comma-delimited_list_of_flags`

Command-line example `-warn=none,nulls,precision`

Properties file syntax `sqlj.warn=comma-delimited_list_of_flags`

Properties file example `sqlj.warn=none,nulls,precision`

Default values `cast,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 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 8-9.
-
-

Command-line syntax `-status<=true|false>`

Command-line example `-status`

Properties file syntax `sqlj.status<=true|false>`

Properties file example `sqlj.status`

Default value `false`

Cause and Action for Translator Errors (-explain)

The `-explain` flag instructs the SQLJ translator to include "cause" and "action" information (as available) with translator error message output for the first occurrence of each error.

This is the same information provided in ["SQLJ Translation Time Messages"](#), starting on page B-2.

Command-line syntax `-explain<=true|false>`

Command-line example `-explain`

Properties file syntax `sqlj.explain<=true|false>`

Properties file example `sqlj.explain`

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, when Java runtime errors occur, the line number reported by the JVM 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 performing this, SQLJ takes the following into account:

- the `-d` option setting, which determines the root directory for `.class` files
- the `-dir` option setting, which determines the root directory for generated `.java` files

Notes:

- If you are processing a `.sqlj` file and the compilation step is skipped due to error, then no line-mapping can be performed either, because no `.class` file is available for mapping.
 - When the Java compiler is invoked from SQLJ (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.
 - If you are using the Sun Microsystems `jdb` debugger, then use the `-jdblinemap` option (discussed next) instead of the `-linemap` option.
-
-

Command-line syntax `-linemap<=true|false>`

Command-line example `-linemap`

Properties file syntax `sqlj.linemap<=true|false>`

Properties file example `sqlj.linemap`

Default value `false`

Line-Mapping to SQLJ Source File for `jdb` Debugger (`-jdblinemap`)

This option is equivalent to the `-linemap` option (discussed in the preceding section), but you should use it instead of `-linemap` if you are using the `jdb` debugger provided with the Sun Microsystems JDK. This is because `jdb` can access only source files with a `.java` file name extension.

With the `-jdblinemap` setting, SQLJ does the following:

- It overwrites the contents of the `.java` file generated by the translator with the contents of the original `.sqlj` file.
- It preserves the `.java` file name, instead of the `.sqlj` file name, in the generated `.class` file.

In this way, the SQLJ source code is accessible to `jdb`.

Command-line syntax `-jdblinemap<=true|false>`

Command-line example `-jdblinemap`

Properties file syntax `sqlj.jdblinemap<=true|false>`

Properties file example `sqlj.jdblinemap`

Default value `false`

Options for Code Generation, Optimizations, and CHAR Comparisons

By default, as of Oracle9i release 2, Oracle SQLJ uses Oracle-specific code generation, which generates Oracle JDBC code directly, as an alternative to ISO standard SQLJ code generation. With Oracle-specific code generation, no profiles are generated, and the SQLJ runtime is largely bypassed during code execution.

Because profile customization is not applicable with Oracle-specific code generation, some generally useful optimization options, formerly available only through the Oracle customizer, are now available directly through the SQLJ translator.

There is also an option for CHAR comparisons in a WHERE clause, accounting for any blank padding in the column. This option is also available as either a translator option (for Oracle-specific code generation) or an Oracle customizer option (for ISO standard code generation).

This section describes these code generation, optimization, and CHAR comparison options:

- `-codegen`
- `-optcols`
- `-optparams`

- `-optparamdefaults`
- `-fixedchar`

Code Generation (-codegen)

Oracle SQLJ can either generate Oracle-specific JDBC code directly or generate ISO standard code that calls the SQLJ runtime, which in turn calls JDBC. With Oracle-specific code generation, there are no profile files and the SQLJ runtime is largely bypassed during program execution.

Use the SQLJ translator `-codegen` option if you want to specify code generation according to the ISO standard (the default in previous releases), as follows:

```
-codegen=iso
```

As of Oracle9i release 2, the default is Oracle-specific SQLJ code generation, but you can also explicitly specify this as follows:

```
-codegen=oracle
```

See "[Oracle-Specific Code Generation \(No Profiles\)](#)" on page 4-39 for information about advantages, disadvantages, limitations, and special considerations regarding Oracle-specific code generation.

Notes:

- When `codegen=iso`, translator settings for `-user`, `-password`, `-url`, `-optparams`, `-optparamdefaults`, and `-fixedchar` are forwarded to the profile customizer as well. If you want to override these settings for customization, however (particularly for `-user`, `-password`, and `-url`), you can do so by setting the customizer options directly.
 - If an application compiles under both JDK 1.1.x and JDK 1.2.x (or higher) and is translated with the `-codegen=oracle` setting, and if JDK 1.2.x (or higher) is used to compile it, then the application will likely not run under JDK 1.1.x. You should use JDK 1.1.x to compile an application that will run under 1.1.x.
-
-

Command-line syntax `-codegen=iso|oracle`

Command-line example `-codegen=iso`

Properties file syntax `sqlj.codegen=iso|oracle`

Properties file example `sqlj.codegen=iso`

Default value `oracle`

Column Definitions (-optcols)

Use the SQLJ translator `-optcols` flag to instruct the translator to determine types and sizes of iterator or result set columns. This enables registration of the columns with the Oracle JDBC driver when your application runs, saving round trips to Oracle9i depending on the particular driver implementation. Specifically, this is effective for the Thin driver and positional iterators.

For an overview of column definitions, see "[Column Definitions](#)" on page 10-23.

Note: This translator option is equivalent to the `optcols` Oracle customizer option, and was created for the default Oracle-specific code generation scenario (`-codegen=oracle`), where there are no profiles. But it is also applicable for ISO standard code generation (`-codegen=iso`)—in this case, setting the translator option will automatically set the customizer option as well. See "[Oracle Customizer Column Definition Option \(optcols\)](#)" on page A-27.

You can enable or disable this flag on the SQLJ command line or in a properties file.

Enable it on the command line as follows:

```
-optcols
```

or:

```
-optcols=true
```

This flag is disabled by default, but you can also disable it explicitly. Disable it on the command line as follows:

```
-optcols=false
```

Column definitions require a database connection for examination of the columns of tables being queried, so the SQLJ translator `-user`, `-password`, and `-url` options must also be set appropriately. For example:

```
sqlj -user=scott/tiger@jdbc:oracle:oci:@ -optcols MyApp.sqlj
```

Notes:

- Because definitions are created for all columns that you select, it is advisable in your SQL operations to explicitly select the columns you will use, rather than using "SELECT *" syntax, if you might not actually use all the columns selected. A situation where you select more than you need exposes you to a greater risk of runtime errors if any changes were made to the table between customization and runtime, especially when you have customized with column definitions. You might want to translate with the SQLJ `-warn=strict` flag set, which will warn you if additional (unwanted) columns will be selected by your query.
 - Column definitions are not possible for any iterator or result set that includes one or more object or collection columns.
 - An error will be generated if you enable the `-optcols` option without setting the user name, password, and URL for a database connection.
 - The translator does not have to connect to the same schema or even the same database that your application will connect to at runtime, but the relevant columns will have to be in the same order and of identical types and sizes to avoid runtime errors.
-
-

Command-line syntax `-optcols<=true|false>`

Command-line example `-optcols`

Properties file syntax `sqlj.optcols<=true|false>`

Properties file example `sqlj.optcols`

Default value `false`

Parameter Definitions (-optparams)

Use the SQLJ translator `-optparams` flag to enable parameter size definitions. If this flag is enabled, SQLJ will register your input and output parameters (host variables) to optimize JDBC resource allocations according to sizes you specify, with the following precedence:

1. size specified in a source code hint, if any
2. default size, if any, specified for the corresponding datatype in the `-optparamdefaults` option setting

If there is no source code hint or default datatype size for a given host variable, then resource allocation is left to JDBC.

For an overview of parameter size definitions and a discussion of source code hints, see "[Parameter Size Definitions](#)" on page 10-25.

Note: This translator option is equivalent to the `optparams` Oracle customizer option. It was created for the default Oracle-specific code generation scenario (`-codegen=oracle`), where there are no profiles. But it is also applicable for ISO standard code generation (`-codegen=iso`)—in this case, setting the translator option will automatically set the customizer option as well. See "[Oracle Customizer Parameter Definition Option \(optparams\)](#)" on page A-29.

You can enable or disable the `-optparams` flag on the command line or in a SQLJ properties file.

Enable it on the command line as follows:

```
-optparams
```

or:

```
-optparams=true
```

This flag is disabled by default, but you can also disable it explicitly. Disable it on the command line as follows:

```
-optparams=false
```

Note: Unlike the `-optcols` option, the `-optparams` option does not require a database connection, because you are providing the size specifications yourself.

Following is a command-line example (omitting a setting for the `-optparamdefaults` option, which is discussed in the next section):

```
sqlj -optparams -optparamdefaults=defaults_string MyApp.sqlj
```

Command-line syntax `-optparams<=true|false>`

Command-line example `-optparams`

Properties file syntax `sqlj.optparams<=true|false>`

Properties file example `sqlj.optparams`

Default value `false`

Parameter Default Size (-optparamdefaults)

If you enable the `-optparams` option to set parameter sizes, use the `-optparamdefaults` option as desired to set default sizes for specified datatypes. If `-optparams` is *not* enabled, then any `-optparamdefaults` setting is ignored.

If a host variable has a source code hint to specify its size, that takes precedence over the corresponding datatype default size set with this option. If there is no source code hint or corresponding datatype default size for a particular host variable, then resource allocation for that variable is determined by the JDBC driver, just as it would be if `-optparams` were not enabled.

There is no requirement to use the `-optparamdefaults` option, although it is typically used whenever `-optparams` is enabled. If `-optparams` is enabled and there are no default size settings, then resources are allocated either according to source code hints (if any) or according to the JDBC driver.

For an overview of parameter size definitions and a discussion of source code hints, see "[Parameter Size Definitions](#)" on page 10-25.

Note: This translator option is equivalent to the `optparamdefaults` Oracle customizer option. It was created for the default Oracle-specific code generation scenario (`-codegen=oracle`), where there are no profiles. But it is also applicable for ISO standard code generation (`-codegen=iso`)—in this case, setting the translator option will automatically set the customizer option as well. See ["Oracle Customizer Parameter Default Size Option \(`optparamdefaults`\)"](#) on page A-30.

You can set the `-optparamdefaults` flag on the command line or in a SQLJ properties file.

Set it on the command line as follows:

```
-optparamdefaults=datatype1(size1),datatype2(size2),...
```

All sizes are in bytes. Do not include any white space. Use empty parentheses for a null setting.

For example, the following will set sizes of 30 bytes for `VARCHAR2` and 1000 bytes for `RAW`, and will specify a null size setting for `CHAR`. So for any host variable corresponding to the `CHAR` datatype, if there is no source code hint, then the JDBC driver is left to allocate the resources.

```
-optparamdefaults=VARCHAR2(30),RAW(1000),CHAR( )
```

The `-optparamdefaults` option recognizes the following datatype names:

- `CHAR`
- `VARCHAR`, `VARCHAR2` (synonymous)
- `LONG`, `LONGVARCHAR` (synonymous)
- `BINARY`, `RAW` (synonymous)
- `VARBINARY`
- `LONGVARBINARY`, `LONGRAW` (synonymous)

The `-optparamdefaults` option also recognizes group names and wildcards, as follows:

- `CHAR_TYPE` covers `CHAR`, `VARCHAR/VARCHAR2`, and `LONG/LONGVARCHAR`.
- `RAW_TYPE` covers `BINARY/RAW`, `VARBINARY`, and `LONGVARBINARY/LONGRAW`.

- A percent sign ("%") by itself covers all recognized datatypes, or, appended to a partial name, covers a subset of datatypes. For example, VAR% includes all datatypes that start with "VAR".

The `-optparamdefaults` setting is processed from left to right. When using group names or wildcards, you can override a group setting for particular datatypes.

The following example sets a general default size of 50 bytes, overrides that with a setting of 500 bytes for raw types, then overrides the raw type group setting with a null setting for `VARBINARY` (leaving that to JDBC for corresponding host variables with no source code hints):

```
-optparamdefaults=%(50),RAW_TYPE(500),VARBINARY()
```

Following is a command-line example, including the `-optparams` setting as well:

```
sqlj -optparams -optparamdefaults=CHAR_TYPE(50),RAW_TYPE(500),CHAR(10) MyApp.sqlj
```

Note: If at runtime the actual size exceeds the registered size of any parameter, runtime errors will occur.

Command-line syntax `-optparamdefaults=defaults_string`

Command-line example `-optparamdefaults=VAR%(50),LONG%(500),RAW_TYPE()`

Properties file syntax `sqlj.optparamdefaults=defaults_string`

Properties file example `sqlj.optparamdefaults=VAR%(50),LONG%(500),RAW_TYPE()`

Default value `null`

CHAR Comparisons with Blank Padding (-fixedchar)

Set this flag to `true` to account for blank padding in `CHAR` database columns when binding character strings for `WHERE` clause comparisons. This way, for example, "mystring" would compare positively against "mystring ".

This functionality uses the Oracle JDBC `setFixedCHAR()` method, an Oracle extension to take padding into account. The standard JDBC `setString()` method does not account for blank padding.

Here is an example of `-fixedchar` usage:

```
% sqlj -fixedchar MyProgram.sqlj AnotherProg.java ...
```

Notes:

- This translator option is equivalent to the `fixedchar` Oracle customizer option. It was created for the default Oracle-specific code generation scenario (`-codegen=oracle`), where there are no profiles. But it is also applicable for ISO standard code generation (`-codegen=iso`)—in this case, setting the translator option will automatically set the customizer option as well. See ["Oracle Customizer CHAR Comparisons with Blank Padding \(fixedchar\)"](#) on page A-32.
 - In `CHAR` or `VARCHAR2` columns, Oracle SQL treats the values `NULL` and `""` (empty string) synonymously. Unfortunately, however, while you can insert the string `""`, you cannot successfully compare against it without using `IS NULL` syntax. Using `-fixedchar` functionality does not resolve this issue.
-
-

Command-line syntax `-fixedchar<=true|false>`

Command-line example `-fixedchar`

Properties file syntax `sqlj.fixedchar<=true|false>`

Properties file example `sqlj.fixedchar`

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 JVM, 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 8-16 for more information.)

This discussion includes the following categories:

- [Prefixes that Pass Option Settings to Other Executables](#)
- [Flags for Special Processing](#)
- [Semantics-Checking and Offline-Parsing Options](#)

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 8-80. More basic command line flags and options are discussed in ["Basic Translator Options"](#) on page 8-22.

Prefixes that Pass Option Settings to Other Executables

The following flags mark options to be passed to the Java interpreter, Java compiler, and SQLJ profile customizer:

- `-J` (mark options for the Java interpreter)
- `-C` (mark options for the Java compiler)
- `-P` (mark options for the profile customizer, for ISO code generation only)

Options to Pass to the Java Virtual Machine (-J)

The `-J` prefix, specified on the command line, marks options to be passed to the JVM from which SQLJ was invoked. This prefix immediately precedes a JVM option, with no spaces in between. After stripping off the `-J` prefix, the `sqlj` script passes the Java option to the JVM.

For example:

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

After stripping the `-J` prefix, the `sqlj` script passes the `-Duser.language=ja` argument as is to the JVM. In the Sun Microsystems JDK, the 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 JVM from a properties file, because properties files are read after the JVM is invoked.

Notes:

- While it is not possible to use a properties file to pass options directly to the JVM in which the SQLJ translator runs, it *is* possible to use the `SQLJ_OPTIONS` environment variable for this purpose. See "[SQLJ_OPTIONS Environment Variable for Option Settings](#)" on page 8-19. It is also possible (if applicable) to use a properties file to pass options to the JVM in which the Java compiler runs. See "[Options to Pass to the Java Compiler \(-C\)](#)" on page 8-62 for information.
- The JVM `file.encoding` setting does not apply to Java properties files—`sqlj.properties` and `connect.properties`, for example. Properties files always use the encoding `8859_1`. This is a feature of Java in general, not SQLJ in particular. You can, however, use Unicode escape sequences in a properties file. (You can use the `native2ascii` utility to determine escape sequences—see "[Using native2ascii for Source File Encoding](#)" on page 9-30.)

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 the Java Compiler (-C)

The `-C` prefix marks options to pass to the Java compiler invoked from the `sqlj` script. This prefix immediately precedes a Java compiler option, with no spaces in between. After stripping off the `-C` prefix, the `sqlj` script passes the compiler option to the Java compiler (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.)

Generally, compiler options are passed without change, but when you use an equals sign (=) to set a compiler option that takes a value—such as for `-bootclasspath`, `-extdirs`, or `-target`—the equals sign is stripped out when the option is passed to the compiler. Consider the following example:

```
sqlj -C-bootclasspath=/usr/local/packages/jdk1.3.1/jre/lib/rt.jar myfile.sqlj
```

Also note that if the Java compiler runs in its own JVM, then you can pass options to that JVM through the compiler. Accomplish this by prefixing the JVM option with `-C-J` with no spaces between this prefix combination and the option.

For example:

```
-C-J-Duser.language=de
```

Observe the following restrictions in using the `-C` prefix:

- Do not use `-C-encoding` to specify encoding of `.java` files processed by the Java compiler. Instead, use the SQLJ `-encoding` option, which specifies encoding of `.sqlj` files processed by SQLJ and `.java` files generated by SQLJ, and is also passed to the compiler. This ensures that `.sqlj` files and `.java` files receive the same encoding. For information about the `-encoding` option, see ["Encoding for Input and Output Source Files \(-encoding\)"](#) on page 8-29.
- 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 8-30.)

Notes:

- If you specify compiler options but disable compilation (`-compile=false`), then the compiler options are silently ignored.
 - The compiler help option (`-C-help`, presuming your compiler supports `-help`) can be specified only on the command line or in the `SQLJ_OPTIONS` variable, not in a properties file. As with the `SQLJ -help` option, no translation will be done. This is true even if you also specify files to process. (SQLJ assumes that you want help or you want translation, but not both.)
-
-

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 the Profile Customizer (-P)

During the customization phase (relevant only for ISO standard code generation, `-codegen=iso`), 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 a generic customizer 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 assumes that you want help or you want translation, but not both.)
 - For ISO code generation, if you specify customization options but turn off customization for `.sqlj` files (and have no `.ser` files on the command line), then the customization options are silently ignored.
 - The `-P` prefix is not applicable for the default Oracle-specific code generation (`-codegen=oracle`), where no profiles are produced and so no customization is performed.
-
-

For information about available generic and Oracle-specific customizer options, see "[Customization Options and Choosing a Customizer](#)" on page A-11.

Command-line syntax `-P-<C>profile_customizer_option`

Command-line examples

```
-P-driver=oracle.jdbc.OracleDriver
-P-Csummary
```

Properties file syntax `profile.<C>profile_customizer_option`

Properties file example

```
profile.driver=oracle.jdbc.OracleDriver
profile.Csummary
```

Default value n/a

Flags for Special Processing

As mentioned above, `.sqlj` files are typically processed by the SQLJ translator, the Java compiler, and, for ISO code generation, the SQLJ profile customizer. The following flags limit this processing, directing the SQLJ startup script to skip the indicated process:

- `-compile`
- `-profile`

The following flag, for ISO code generation, directs SQLJ to convert profiles from serialized resource (`.ser`) files to class files after customization:

- `-ser2class`

The following flag instructs SQLJ type resolution, in certain circumstances, to examine source files as well as class files or files specified on the SQLJ command line:

- `-checksource`

The following flag specifies that SQLJ treat multiple appearances of the same host variable in a given SQLJ statement as a single bind occurrence.

- `-bind-by-identifier`

Compilation Flag (-compile)

The `-compile` flag enables or disables processing of `.java` files by the compiler. This applies both to generated `.java` files and to `.java` files specified on the command line. This flag is useful, for example, if you want to compile `.java` files

later using a compiler other than `javac`. The flag is `true` by default; setting it to `false` disables compilation.

When you process a `.sqlj` file with `-compile=false`, you are responsible for compiling and customizing it later as necessary.

Setting `-compile=false` also implicitly sets `-profile=false`. In other words, whenever `-compile` is `false`, both compilation and customization are skipped. If you set `-compile=false` and `-profile=true`, then your `-profile` setting is ignored.

Notes: There are situations where it is sensible for `-compile` to be set to `false` even when `.java` files must be accessed for type resolution. You might do this, for example, if you are translating a `.sqlj` file and want to specify one or more `.java` files on the command line for type resolution during translation, but want to compile all your `.java` files later using a particular compiler.

Note, however, that the `-checksource` option can simplify the type resolution process by eliminating the need to enter `.java` files for resolution on the SQLJ command line. See "[Source Check for Type Resolution \(-checksource\)](#)" on page 8-69.

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` (compile)

Profile Customization Flag (-profile)

For ISO code generation, the `-profile` flag enables or disables processing of generated profile (`.ser`) files by the SQLJ profile customizer. However, this applies only to `.ser` files generated by the SQLJ translator from `.sqlj` files that you specify on the current command line; it does not apply to previously generated `.ser` files (or to `.jar` files) that you specify on the command line. The flag is `true` by default; setting it to `false` disables customization.

This option 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 you might want other operations, such as line mapping, to be performed on a `.java` file. There are, however, no other operations that can be performed on a `.ser` or `.jar` file specified on the command line.

When you process a `.sqlj` file with `-profile=false`, you are responsible for customizing it later, as necessary.

Notes:

- Set this option to `false` if you do not want your application to require the Oracle SQLJ runtime and an Oracle JDBC driver when it runs. (Or accomplish this by specifying a nondefault customizer, using the `-default-customizer` option.) If no customization is performed, then the generic SQLJ runtime will be used when your application runs.
 - Setting `-compile=false` also implicitly sets `-profile=false`. In other words, whenever `-compile` is `false`, both compilation and customization are skipped. If you set `-compile=false` and `-profile=true`, then your `-profile` setting is ignored.
 - This option is not applicable for the default Oracle-specific code generation (`-codegen=oracle`), where no profiles are produced and so no customization is performed.
-
-

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` (customize)

Conversion of .ser File to .class File (-ser2class)

With ISO standard SQLJ code generation (`-codegen=iso`), 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. Loaded class schema objects have a simpler naming convention than loaded resource schema objects. (This is discussed in ["Naming of Loaded Class and Resource Schema Objects"](#) on page 11-11.)

The conversion is performed after profile customization so that it includes your customizations.

The base names of converted files are identical to those of the original files; the only difference in the file name is `.ser` being replaced by `.class`. For example:

```
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 delete the `.class` file and rerun SQLJ to re-create the profile.
 - Where encoding is necessary, the `-ser2class` option always uses `8859_1` encoding, ignoring the SQLJ `-encoding` setting.
 - If you use the default Oracle-specific code generation (`-codegen=oracle`), then no profiles are produced and the `-ser2class` option does not apply.
-
-

Command-line syntax `-ser2class<=true|false>`

Command-line example `-ser2class`

Properties file syntax `sqlj.ser2class<=true|false>`

Properties file example `sqlj.ser2class`

Default value `false`

Source Check for Type Resolution (-checksource)

It may not be sufficient for the SQLJ type resolution process to examine only class files in the classpath and class or source files specified on the SQLJ command line. The `-checksource` flag instructs SQLJ to also examine source files in the classpath under the following circumstances:

- if a class file cannot be found for a required class, but a source file can be found
- if a source file has a more recent modification date than its corresponding class file

Note: This applies only to Java types that appear in `#sql` statements, not elsewhere in your Java code. Therefore, you should always explicitly provide the names of any required `.sqlj` files on the SQLJ command line.

Command-line syntax `-checksource<=true|false>`

Command-line example `-checksource=false`

Properties file syntax `sqlj.checksource=<=true|false>`

Properties file example `sqlj.checksource=false`

Default value `true`

Binding Host Expressions by Identifier (-bind-by-identifier)

In keeping with the SQLJ standard, Oracle SQLJ by default creates a unique name for each host-variable bind reference in a statement, even if there are multiple occurrences of the same host variable. The SQLJ standard is based on JDBC, and JDBC does not make provisions for binding the same variable into different positions—instead, each bind position (identified by "?") is bound to an individual value.

In some situations this causes errors, such as in the following example:

```
#sql emps = { SELECT substr(ename, 1, :bind_var), sum(sal) FROM emp
              GROUP BY substr(ename, 1, :bind_var) };
```

Because separate bind reference names are created for the two occurrences of `bind_var`, this results in a SQL exception at runtime. When the differing bind names are detected, the SQL engine concludes that the `GROUP BY` clause is not part of the `SELECT`-list.

To avoid such problems, Oracle SQLJ extends standard functionality with the `-bind-by-identifier` flag. A setting of `true` results in all bind occurrences of the same identifier in a given SQLJ statement or PL/SQL block being treated as a single bind occurrence. A SQLJ statement with four bind operations—`:x`, `:x`, `:y`, `:x`—would be bound as `:1`, `:1`, `:2`, `:1` instead of `:1`, `:2`, `:3`, `:4`.

In the example above, both bindings would be as `substr(ename, 1, :1)` instead of as `substr(ename, 1, :1)` and `substr(ename, 1, :2)`.

Note: The `-bind-by-identifier` flag applies only to host expressions that are simple host variables.

Command-line syntax `-bind-by-identifier[=<true|false>]`

Command-line example `-bind-by-identifier`

Properties file syntax `sqlj.bind-by-identifier[=<true|false>]`

Properties file example `sqlj.bind-by-identifier`

Default value `false`

Semantics-Checking and Offline-Parsing Options

The following options specify characteristics of online and offline semantics-checking and offline parsing:

- `-offline`
- `-online`
- `-cache`
- `-parse`

Description of these options is preceded by two introductory discussions:

- a discussion of `OracleChecker`—the default front-end class for semantics-checking—and an introduction to the Oracle semantics-checkers
- a comparison of online semantics-checking versus offline parsing

Note: As described in "[Online Semantics-Checking and User Name \(-user\)](#)" on page 8-34, online semantics-checking is enabled by setting the translator `-user` option. However, the setting of the `-parse` option, which is used to enable or disable offline parsing, can override this. See "[Offline Parser \(-parse\)](#)" on page 8-79.

Semantics-Checkers and the OracleChecker Front End (default checker)

Oracle supplies Oracle-specific offline checkers, a generic offline checker, Oracle-specific online checkers, and a generic online checker. The generic checkers assume you use only standard SQL92 and standard JDBC features. Oracle recommends that you use the Oracle-specific checkers when using an Oracle database.

The default checker, which is satisfactory in the great majority of circumstances, is `oracle.sqlj.checker.OracleChecker` for both online and offline checking. This class acts as a front end and runs the appropriate semantics-checker, depending on your environment and whether you choose offline or online checking.

For Oracle, there are the following categories of checkers (for both online and offline checking):

- Oracle8 checkers for Oracle9i and Oracle8i types (as used in Oracle9i and Oracle8i JDBC)

- Oracle80 checkers for Oracle 8.0.x types (as used in Oracle 8.0.x JDBC)
- Oracle7 checkers for Oracle 7.3.x types (as used in Oracle 7.3.x or 8.0.x JDBC)
- Oracle8To7 checkers for using an Oracle8*i* or Oracle9*i* JDBC driver, but only with the subset of types that are compatible with an Oracle 7.3.x database

The Oracle80 and Oracle7 checkers are incompatible with the Oracle8*i* and Oracle9*i* JDBC drivers, and the Oracle8 and Oracle8To7 checkers are incompatible with the Oracle 8.0.x and Oracle 7.3.x JDBC drivers. The Oracle8To7 checkers were created so there is a way to use an Oracle8*i* or Oracle9*i* JDBC driver and check against an Oracle 7.3.x 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 8-4](#) summarizes the choices for the possible combinations of database version and driver version, and also notes any other Oracle checkers that would be legal.

Table 8-4 Oracle Online Semantics-Checkers Chosen by OracleChecker

Database Version	JDBC Version	Chosen Online Checker	Other Legal Online Checkers
Oracle9 <i>i</i> , 8 <i>i</i> , or 8.0.x	Oracle9 <i>i</i> or 8 <i>i</i>	Oracle8JdbcChecker	Oracle8To7JdbcChecker
Oracle9 <i>i</i> , 8 <i>i</i> , or 8.0.x	Oracle 8.0.x	Oracle80JdbcChecker	Oracle7JdbcChecker
Oracle9 <i>i</i> , 8 <i>i</i> , or 8.0.x	Oracle 7.3.x	Oracle7JdbcChecker	none
Oracle 7.3.x	Oracle9 <i>i</i> or 8 <i>i</i>	Oracle8To7JdbcChecker	none
Oracle 7.3.x	Oracle 8.0.x	Oracle7JdbcChecker	none
Oracle 7.3.x	Oracle 7.3.x	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 8-5](#) summarizes the possible choices. (Note that there is an `Oracle8To7OfflineChecker`, but it can be used only by selecting it manually.)

Table 8-5 Oracle Offline Semantics-Checkers Chosen by OracleChecker

JDBC Version	Chosen Offline Checker	Other Legal Offline Checkers
Oracle9 <i>i</i> or 8 <i>i</i>	Oracle8OfflineChecker	Oracle8To7OfflineChecker

Table 8–5 Oracle Offline Semantics-Checkers Chosen by OracleChecker (Cont.)

JDBC Version	Chosen Offline Checker	Other Legal Offline Checkers
Oracle 8.0.x	Oracle80OfflineChecker	Oracle7OfflineChecker
Oracle 7.3.x	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

Online Semantics-Checking Versus Offline Parsing

In earlier SQLJ releases, any SQL verification during translation required a database connection for online semantics-checking. With Oracle9i release 2, there is a feature known as *offline parsing* that offers a limited alternative. Offline parsing does not use a database connection, so cannot perform verification of operations against the database schema, but it does offer syntax-checking of all SQL and PL/SQL statements. This was previously unavailable in an offline mode.

[Table 8–6](#) provides a comparative summary of what offline parsing and online semantics-checking offer.

Table 8–6 Feature Comparison: Offline Parsing Versus Online Semantics-Checking

Feature	By Offline Parsing?	By Online Checking?
Verify DML, SELECT, and PL/SQL syntax	Yes	Yes
Verify DDL syntax	Yes	No
Verify DML, SELECT, and PL/SQL semantics (comparison against database schema)	No	Yes
Verify DDL semantics (comparison against database schema)	No	No

Note: "DDL" in the table refers to SQL statements such as CREATE, ALTER, DROP, and SET.

Online checking offers the primary advantage of verifying SQL and PL/SQL operations against the database schema. This includes verifying that column types match SQL operations and verifying the existence of called stored procedures. It requires a database connection during translation, however, which may be problematic in some circumstances. It also performs no verification of DDL operations.

Offline parsing offers the advantage of SQL syntax-checking without a database connection during translation, and also includes DDL operations in its syntax verifications.

Note that neither mode performs DDL semantics-checking against the database schema.

Notes:

- If both offline parsing and online checking are enabled, some types of errors will be reported twice.
 - Problems detected by either the offline parser or the online checker are reported at a warning or advisory level, not a fatal level.
 - Do not confuse offline parsing with offline semantics-checking. Offline checking consists of basic semantics-checking steps that always occur, regardless of whether online checking is enabled and regardless of whether offline parsing is enabled: analysis of the types of Java expressions in your SQLJ executable statements, and categorization of embedded SQL operations according to keyword (such as `SELECT`).
 - Compatibility of data corresponding to weakly typed host expressions is never checked (host expressions using the `oracle.sql` package `STRUCT`, `REF`, and `ARRAY` classes, which are discussed in ["Weakly Typed Objects, References, and Collections"](#) on page 6-80).
 - Mode compatibility (`IN`, `OUT`, or `IN OUT`) of expressions in PL/SQL anonymous blocks is never checked.
-
-

Offline Semantics-Checker (-offline)

The `-offline` option specifies a Java class that implements the semantics-checking component of SQLJ for offline checking. With offline checking, there is no database connection—only SQL syntax and usage of Java types is checked. (For information about what offline and online semantics-checkers accomplish and how they function, see "[SQL Semantics-Checking and Offline Parsing](#)" 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 8-71, will serve your needs unless you want to specify a particular checker that would not be chosen by `OracleChecker`. For example, you might 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 a `CtxClass` instance.

The `CtxClass` connection context class must be declared in your source code or previously compiled into a `.class` file. (See "[Connection Contexts](#)" on page 7-2 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.Oracle800OfflineChecker
-offline@CtxClass=oracle.sqlj.checker.Oracle800OfflineChecker
```

Properties file syntax `sqlj.offline<@conn_context_class>=checker_class`

Properties file examples

```
sqlj.offline=oracle.sqlj.checker.Oracle800OfflineChecker
sqlj.offline@CtxClass=oracle.sqlj.checker.Oracle800OfflineChecker
```

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.

Note that online checking is not enabled by the `-online` option—you must enable it through the `-user` option. The `-password`, `-url`, and `-driver` options must be set appropriately as well. (For information about what offline and online semantics-checkers accomplish and how they function, see "[SQL Semantics-Checking and Offline Parsing](#)" on page 9-2.)

Note: Some settings of the SQLJ `-parse` option will disable online semantics-checking, overriding the effect of the `-user` option. See "[Offline Parser \(-parse\)](#)" on page 8-79.

You can specify different online checkers for different connection contexts, and you can list multiple checkers (separated by commas) for any given context. In cases where multiple checkers are listed for a single context, SQLJ uses the first checker (reading from left to right in the list) that accepts the database connection established for online checking. At analysis time, a connection is passed to each online checker, and the checker decides whether it recognizes the database.

The default `OracleChecker`, a front-end class discussed in "[Semantics-Checkers and the OracleChecker Front End \(default checker\)](#)" on page 8-71, will serve your needs unless you want to specify a particular checker that would not be chosen by `OracleChecker`. For example, you might 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 the `oracle.sqlj.checker.Oracle7JdbcChecker` semantics-checker. If connection is made to any other kind of database, then SQLJ uses the generic `sqlj.semantics.JdbcChecker` semantics-checker. This is similar functionally to what the default `OracleChecker` 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.

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@CtxClass=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@CtxClass=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. Delete the `SQLChecker.cache` file to empty the cache.

Command-line syntax `-cache<=true|false>`

Command-line example `-cache`

Properties file syntax `sqlj.cache<=true|false>`

Properties file example `sqlj.cache`

Default value `false`

Offline Parser (-parse)

Use the `-parse` option to enable offline parsing—a complement to online semantics-checking that offers SQL and PL/SQL syntax-checking without a database connection during translation. Offline parsing also checks syntax for DDL statements, which online checking does not. For a comparison of features, see ["Online Semantics-Checking Versus Offline Parsing"](#) on page 8-73.

Also be aware that the setting of the `-parse` option can override the enabling of online checking by the `-user` option. Possible `-parse` settings are as follows:

- `both` (default)—Enable the offline parser and allow online checking. In this case, online checking is determined by the `-user` option.
- `online-only`—Disable the offline parser and allow online checking. Again, online checking is determined by the `-user` option.
- `offline-only`—Enable the offline parser and disallow online checking. This overrides any `-user` option setting that otherwise enables online checking.
- `none`—Disable the offline parser and disallow online checking. This overrides any `-user` option setting that would otherwise enable online checking.
- `parserclassname`—Specify the name of a Java class that implements an alternative SQL parser. The class must implement the `sqlj.framework.checker.SimpleChecker` interface. This setting enables the specified parser, and only that parser is used for SQL-checking. The standard offline parser and online checking are both disabled.

The `offline-only` and `none` settings are not typical modes of operation, nor is specifying your own parser. Let the `-user` option determine online checking.

Note: In modes where both offline parsing and online checking are enabled, there may be duplicate reporting of some problems.

Command-line syntax `-parse=both|online-only|offline-only|none|parserclassname`

Command-line example `-parse=online-only`

Properties file syntax `sqlj.parse=both|online-only|offline-only|none|parserclassname`

Properties file example `sqlj.parse=online-only`

Default value `both`

Translator Support and Options for Alternative Environments

By default, Oracle9i SQLJ is configured to run under the Sun Microsystems JDK 1.2.x (or higher) or 1.1.x and to use the Sun Microsystems compiler `javac`. These are not requirements, however. You can configure SQLJ to work with alternative JVMs or compilers. To do so, you must supply SQLJ with the following information:

- the name of the JVM to use (`-vm` option)
- the name of the Java compiler to use (`-compiler-executable` option)
- any settings the compiler requires

A set of SQLJ options allows you to provide this information. These options are described in "[Java and Compiler Options](#)" below.

SQLJ also defaults to the Oracle profile customizer, but can work with alternative customizers as well. See "[Customization Options](#)" on page 8-87 for how to instruct SQLJ to use a different customizer.

Note: Be aware of the limitations of any operating system and environment you use. In particular, the complete, expanded SQLJ command line must not exceed the maximum command-line size (for example, 250 characters for Windows 95 and 4000 characters for Windows NT). Consult your operating system documentation.

Java and Compiler Options

The following options relate to the operation of the JVM and Java compiler:

- `-vm` (specify the JVM; command-line only)
- `-compiler-executable` (specify the Java compiler)
- `-compiler-encoding-flag`
- `-compiler-output-file`
- `-compiler-pipe-output-flag`

Some compilers, such as the standard `javac`, require a Java source file name to match the name of the public class (if any) defined there. Therefore, by default the SQLJ translator verifies that this is true. You can use the following option, however, to instruct SQLJ not to verify this:

- `-checkfilename`

For some JVM and compiler configurations, there might be problems with the way SQLJ normally invokes the compiler. You can use the following option to alleviate this by breaking SQLJ processing into a two-pass process:

- `-passes`

You can also pass options directly to the particular JVM or compiler you use, through the `-J` and `-C` prefixes discussed in ["Prefixes that Pass Option Settings to Other Executables"](#) on page 8-60.

Note: The `-vm` option, `-passes` option, and `-J` prefix cannot be used in a properties file. You can set them on the command line or, more conveniently, in the `SQLJ_OPTIONS` environment variable. See ["SQLJ_OPTIONS Environment Variable for Option Settings"](#) on page 8-19.

Name of the Java Virtual Machine (-vm)

Use the `-vm` option if you want to specify a particular JVM for SQLJ to use. Otherwise SQLJ uses the standard `java` from the Sun Microsystems JDK.

You cannot set this option in a properties file, because properties files are read after the JVM is invoked.

If you do not specify a directory path along with the name of the JVM executable file, then SQLJ looks for the executable according to the setting of your operating system `PATH` variable.

Note: Special functionality of this option, `-vm=echo`, is supported. This is equivalent to the `-n` option, instructing the `sqlj` script to construct the full command line that would be passed to the SQLJ translator, and echo it to the user without having the translator execute it. For more information, see ["Command Line Echo without Execution \(-n\)"](#) on page 8-27.

Command-line syntax `-vm=JVM_path+name`

Command-line example `-vm=/myjavadir/myjavavm`

Properties file syntax `n/a`

Properties file example n/a

Default value java

Name of the Java Compiler (-compiler-executable)

Use the `-compiler-executable` option if you want to specify a particular Java compiler for SQLJ to use. Otherwise SQLJ uses the standard `javac` from the Sun Microsystems JDK.

If you do not specify a directory path along with the name of the compiler executable file, then SQLJ looks for the executable according to the setting of your operating system `PATH` variable.

The following is required of any Java compiler that you use:

- It can 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.
- It must return a nonzero exit code to the operating system whenever a compilation error occurs.
- The line information that it provides in any errors or messages must be in one of the following two formats (items in `<>` brackets being optional):

- Sun Microsystems `javac` format

filename.java:line<.column><-line<.column>>

Example: `myfile.java:15: Illegal character: '\u01234'`

- Microsoft `javc` format

filename.java(line,column)

Example: `myfile.java(15,7) Illegal character: '\u01234'`

As always, SQLJ processes compiler line information so that it refers to line numbers in the original `.sqlj` file, not in the produced `.java` file.

Note: For a compiler that does not support an `-encoding` option, disable the `-compiler-encoding-flag`, described in "[Compiler Encoding Support \(-compiler-encoding-flag\)](#)" on page 8-83.

Command-line syntax `-compiler-executable=Java_compiler_path+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 8-29, 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 (if, for example, 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). As appropriate, specify an absolute path, or a relative path from the current directory.

Note: You cannot use this option if you enable `-passes`, which requires output to `STDOUT`.

Command-line syntax `-compiler-output-file=output_file_path+name`

Command-line example `-compiler-output-file=/myjavadir/mycmloutput`

Properties file syntax `sqlj.compiler-output-file=output_file_path+name`

Properties file example `sqlj.compiler-output-file=/myjavadir/mycmloutput`

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 it can be captured reliably.

If SQLJ sets the `javac.pipe.output` system property to `true`, which is the SQLJ 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 to *not* 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.

Notes:

- For a Java compiler that originates from Sun Microsystems and writes its output to `STDERR` by default, you must leave `-compiler-pipe-output-flag` enabled if you enable `-passes`, which requires output to `STDOUT`.
 - Sun Microsystems JDK 1.3.x versions appear to no longer support `javac.pipe.output` functionality.
-
-

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`

Source File Name Check (-checkfilename)

It is generally advisable for the source file name to always match the name of the public class defined, or, if there is no public class, the name of the first class defined. For example, public class `MyPublicClass` should be defined in a `MyPublicClass.sqlj` source file.

The `-checkfilename` flag instructs SQLJ whether to verify that the SQLJ source file name matches the name of the public class (if any) defined there. Some compilers, such as the standard `javac`, require this to be the case; others do not.

To maximize portability of your code, this flag should be enabled, which it is by default.

Note: If you are translating in the server, where there is no equivalent naming requirement, there is no `-checkfilename` option, and the translator executes no such check.

Command-line syntax `-checkfilename<=true|false>`

Command-line example `-checkfilename=false`

Properties file syntax `sqlj.checkfilename<=true|false>`

Properties file example `sqlj.checkfilename=false`

Default value `true`

SQLJ Two-Pass Execution (-passes)

By default, the following sequence occurs when you invoke the `sqlj` script:

1. The `sqlj` script invokes your JVM, which runs the SQLJ translator.
2. The translator completes the semantics-checking and translation of your `.sqlj` files, generating translated `.java` files.
3. The translator invokes your Java compiler, which compiles the generated `.java` files.
4. The translator processes the compiler output.
5. If any profile files were generated, the translator invokes a profile customizer to customize them.

For some JVM and compiler configurations, however, the compiler invocation in step 3 might not return, in which case your translation will suspend.

If you encounter this situation, the solution is to instruct SQLJ to run in two passes, with the compilation step in between. To accomplish this, you must enable the two-pass execution flag as follows:

```
-passes
```

The `-passes` option must be specified on the command line or, equivalently, in the `SQLJ_OPTIONS` environment variable. It cannot be specified in a properties file.

Notes:

- If you enable `-passes`, then compiler output must go to `STDOUT`, so leave `-compiler-pipe-output-flag` enabled (which is its default). In addition, 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 JVM, which runs the SQLJ translator for its first pass.

2. The translator completes the semantics-checking and translation of your `.sqlj` files, generating translated `.java` files.
3. The JVM is terminated.
4. The `sqlj` script invokes the Java compiler, which compiles the generated `.java` files.
5. The `sqlj` script invokes your JVM again, which runs the SQLJ translator for its second pass.
6. The translator processes compiler output.
7. If any profile files were generated, the JVM runs your profile customizer to customize them.

With this sequence, you circumvent any problems the JVM might have in invoking the Java compiler.

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, if applicable:

- `-default-customizer`
- options passed directly to the customizer

Note: If you use the default Oracle-specific code generation (`-codegen=oracle`), then SQLJ generates no profiles and so performs no customization. In that case, the options discussed here do not apply.

Default Profile Customizer (-default-customizer)

Use the `-default-customizer` option to instruct SQLJ to use a profile customizer other than the default, which is:

```
oracle.sqlj.runtime.util.OraCustomizer
```

In particular, use this option if you are not using an Oracle database.

This option takes a fully qualified Java class name as its argument.

Notes: You can override this option with the `-P-customizer` option in your SQLJ command line (or properties file). For more information, see "[Options to Pass to the Profile Customizer \(-P\)](#)" on page 8-63.

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 and ISO code generation, Oracle recommends that you use the default `OraCustomizer` for your profile customization.

Options Passed Directly to the Customizer

As with the JVM and compiler, you can pass options directly to the profile customizer harness using a prefix, in this case `-P`. This is discussed in "[Options to Pass to the Profile Customizer \(-P\)](#)" on page 8-63.

Details about these options, both general customization options and Oracle-specific customizer options, are covered in "[Customization Options and Choosing a Customizer](#)" on page A-11.

Translator and Runtime Functionality

This chapter discusses internal operations and functionality of the Oracle SQLJ translator and runtime.

The following topics are covered:

- [Internal Translator Operations](#)
- [Functionality of Translator Errors, Messages, and Exit Codes](#)
- [SQLJ Runtime](#)
- [Globalization Support in the Translator and Runtime](#)

Internal Translator Operations

The following topics summarize the operations executed by the SQLJ translator during a translation:

- [Java and SQLJ Code-Parsing and Syntax-Checking](#)
- [SQL Semantics-Checking and Offline Parsing](#)
- [Code Generation](#)
- [Java Compilation](#)
- [Profile Customization \(ISO Code Generation\)](#)

Java and SQLJ Code-Parsing and Syntax-Checking

In this first phase of SQLJ translation, a SQLJ parser and a Java parser are used to process all the source code and check syntax.

As the SQLJ translator parses the `.sqlj` file, it invokes a Java parser to check the syntax of Java statements and a SQLJ parser to check the syntax of SQLJ constructs (anything preceded by `#sql`). The SQLJ parser also invokes the Java parser to check the syntax of Java host variables and expressions within SQLJ executable statements.

The SQLJ parser checks the grammar of SQLJ constructs according to the SQLJ language specification. It does not check the grammar of the embedded SQL operations, however. SQL syntax is not checked until the semantics-checking or offline parsing 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, then the translation is aborted and the errors are reported to the user.

SQL Semantics-Checking and Offline Parsing

Once the SQLJ and Java application source code is verified as syntactically correct, the translator enters into the semantics-checking phase and invokes a SQL semantics-checker or a SQL offline parser or both, according to SQLJ option settings.

Setting the `-user` option enables online checking, and 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, typically sufficient, is a checker front end called `OracleChecker` that chooses the most appropriate checker, according to whether you have enabled online checking and which JDBC driver you are using.

The `-parse` option, true by default, is for enabling the offline parser, which offers a way to verify SQL and PL/SQL syntax (but not data types against database columns) without necessitating a database connection during translation. Note that some settings of the `-parse` option will override the `-user` option and disable online checking.

For more information about online checking, offline parsing, and comparative features, see "[Connection Options](#)" on page 8-34 and "[Semantics-Checking and Offline-Parsing Options](#)" on page 8-71.

Note: For ISO code generation, semantics-checking can also be performed on a profile that was produced during a previous execution of the SQLJ translator. See "[SQLJCheckerCustomizer for Profile Semantics-Checking](#)" on page A-40.

The following two tasks are always performed during semantics-checking, regardless of the status of online checking or offline parsing:

1. SQLJ analyzes the types of Java expressions in your SQLJ executable statements.

This includes examining the SQLJ source files being translated, any `.java` files entered on the command-line, and any imported Java classes whose `.class` files or `.java` files can be found through the classpath. SQLJ examines whether and how stream types are used in `SELECT` or `CAST` statements, what Java types are used in iterator columns or `INTO`-lists, what Java types are used as input host variables, and what Java types are used as output host variables.

SQLJ also processes `FETCH`, `CAST`, `CALL`, `SET TRANSACTION`, `VALUES`, and `SET` statements syntactically.

Any Java expression in a SQLJ executable statement must have a Java type valid for the given situation and usage. For example, consider the following statement:

```
#sql [myCtx] { UPDATE ... };
```

The `myCtx` variable, which might be used to specify a connection context instance or execution context instance for this statement, must actually resolve to a SQLJ connection context type or execution context type.

Now consider the following example:

```
#sql { UPDATE emp SET sal = :newSal };
```

If `newSal` is a variable (as opposed to a field), then an error is generated if `newSal` was not previously declared. In any case, an error is generated if it cannot be assigned to a valid Java type, or its Java type cannot be used in a SQL statement (a `java.util.Vector`, for example).

Note: Be aware 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 by the Java compiler.

2. SQLJ tries to categorize your embedded SQL operations—each operation must have a recognizable keyword, such as `SELECT` or `INSERT`, so that SQLJ knows what kind of operation it is. For example, the following statement will generate an error:

```
#sql { foo };
```

The following task is performed if either online checking or offline parsing (or both) is enabled:

3. SQLJ analyzes and verifies the syntax of embedded SQL and PL/SQL operations.

The following task is performed only if online checking is enabled:

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 SQL entities used in your SQLJ executable statements (such as tables, views, and stored procedures) actually exist in the database. SQLJ also checks nullability of database columns whose data is being selected into iterator columns of Java primitive types, which cannot process null data. Nullability is not checked for stored procedure and function output parameters and return values, however.

Code Generation

For the `.sqlj` application source file, the SQLJ translator generates a `.java` file and, for ISO standard SQLJ code generation, at least one profile (either in `.ser` or `.class` files). The `.java` contains your translated application source code, class definitions for any private iterators and connection contexts you declared, and, for ISO code, a profile-keys class definition generated and used internally by SQLJ.

Note: No profiles or profile-keys class are generated if you use the default Oracle-specific code generation mode (`-codegen=oracle`). See "[Oracle-Specific Code Generation \(No Profiles\)](#)" on page 4-39.

With ISO code generation, there are no profiles or profile-keys class if you do not use any SQLJ executable statements in your code.

Generated Application Code in `.java` File

For the default Oracle-specific code generation, the generated `.java` file for your application contains direct calls to Oracle JDBC in place of the original SQLJ executable statements. There are also calls to an Oracle-specific SQLJ runtime.

For ISO standard SQLJ code generation, SQLJ executable statements are replaced by calls to the SQLJ runtime, which in turn contains calls to the JDBC driver.

For convenience, generated `.java` files also include a comment for each of your `#sql` statements, repeating the statement in its entirety for reference.

The generated `.java` file will have the same base name as the input `.sqlj` file, which would be the name of the public class defined in the `.sqlj` file or the first class defined if there are no public classes. For example, `FOO.sqlj` defines class `FOO`, and source file `FOO.java` will be generated by the translator.

The location of the generated `.java` file depends on whether and how you set the SQLJ `-dir` option. By default, the `.java` file will be placed in the directory of the `.sqlj` input file. See "[Output Directory for Generated `.java` Files \(`-dir`\)](#)" on page 8-32 for more information.

Generated Profile-Keys Class in `.java` File (ISO Code Generation)

Is you use ISO standard SQLJ code generation, 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.

Note: If you use the default Oracle-specific code generation, no profiles or profile-keys classes are generated.

The profile-keys class is defined in the same `.java` output file that has your translated application source code, with a class name based on the base name of your `.sqlj` source file as follows:

```
Basename_SJProfileKeys
```

For example, translating `Foo.sqlj` defines the following profile-keys class in the generated `.java` file:

```
Foo_SJProfileKeys
```

If your application is in a package, this is reflected appropriately. For example, translating `Foo.sqlj` in the package `a.b` defines the following class:

```
a.b.Foo_SJProfileKeys
```

Generated Profiles in `.ser` or `.class` Files (ISO Code Generation)

If you use ISO standard SQLJ code generation, SQLJ generates profiles that it uses to store information about the SQL operations found in the input file. A profile is generated for each connection context class that you use in your application. It describes the operations to be performed using instances of the associated connection context class, such as SQL operations to execute, tables to access, and stored procedures and functions to call.

Note: If you use the default Oracle-specific code generation, then information about the SQL operations is embedded in the generated code, which calls Oracle JDBC directly. In this case, SQLJ does not generate profiles.

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. In this case, no further customization of the profile is possible.

(You would have to delete the `.class` file and rerun the SQLJ translator to regenerate the profile.)

Profile base names are generated similarly to the profile-keys class name. They are fully qualified with the package name, followed by the `.sqlj` file base name, followed by the string:

```
_SJProfilen
```

Where *n* is a unique number, starting with 0, for each profile generated for a particular `.sqlj` input file.

Again using the example of the 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. See "[Conversion of .ser File to .class File \(-ser2class\)](#)" on page 8-68 for more information.

The location of these files depends on how the SQLJ `-d` option is set, which determines where all generated `.ser` and `.class` files are placed. See "[Output Directory for Generated .ser and .class Files \(-d\)](#)" on page 8-30 for more information.

In a later step in the SQLJ process, your profiles are customized for use with your particular database. See "[Profile Customization \(ISO Code Generation\)](#)" on page 9-10.

More About Generated Calls to SQLJ Runtime

When `#sql` statements are replaced by calls to Oracle JDBC (for Oracle-specific code generation) or to the SQLJ runtime (for ISO standard SQLJ code generation), these calls implement the following steps:

Table 9–1 Steps for Generated Calls, ISO Standard Versus Oracle-Specific

Steps for ISO Standard Code Generation	Steps for Oracle Code Generation
Get a SQLJ statement object, using information stored in the associated profile entry.	Get an Oracle JDBC statement object.
Bind inputs into the statement, using <code>setXXX()</code> methods of the statement object.	Bind inputs using Oracle JDBC statement methods and, if necessary, register output parameters.
Execute the statement, using the <code>executeUpdate()</code> or <code>executeQuery()</code> method of the statement object.	Execute the Oracle statement.
Create iterator instances, if applicable.	Create iterator instances, if applicable.
Retrieve outputs from the statement, using <code>getXXX()</code> methods of the statement object.	Retrieve outputs from the statement using appropriate Oracle JDBC getter methods.
Close the SQLJ statement object (by default, recycling it through the SQLJ statement cache).	Close the Oracle JDBC statement object (by default, recycling it through the JDBC statement cache).

A SQLJ runtime uses SQLJ statement objects that are similar to JDBC statement objects, although a particular implementation of SQLJ might or might not employ JDBC statement classes directly. SQLJ statement classes add functionality particular to SQLJ. For example:

- Standard SQLJ statement objects raise a SQL exception if a null value from the database is to be output to a primitive Java type such as `int` or `float`, which cannot take null values.
- Oracle SQLJ statement objects allow user-defined object and collection types to be passed to or retrieved from 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 if you use ISO code generation (and presuming your application uses SQLJ executable statements). Any `.java` files you specified directly on the SQLJ command line (for type-resolution, for example) are compiled at this time as well.

In the example used in "[Code Generation](#)" on page 9-5, the following `.class` files would be produced in the appropriate directory (given package information in the source code):

- `Foo.class`
- `Foo_SJProfileKeys.class` (ISO code generation only)
- a `.class` file for each additional class you defined in `Foo.sqlj`
- a `.class` file for each iterator and connection context class you declared in `Foo.sqlj` (whether public or private)

To ensure that `.class` files and profiles (if any, whether `.ser` or `.class`) will be located in the same directory, SQLJ passes its `-d` option to the Java compiler. If the `-d` option is not set, then `.class` files and profiles are placed in the same directory as the generated `.java` file (which is placed according to the `-dir` option setting).

In addition, so that SQLJ and the Java compiler will use the same encoding, SQLJ passes its `-encoding` option to the Java compiler unless the SQLJ `-compiler-encoding-flag` is turned off. If the `-encoding` option is not set, SQLJ and the compiler will use the setting in the JVM `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 SQLJ support for compiler options and compiler-related SQLJ options, see the following:

- ["Option Support for javac" on page 8-10](#)
- ["Output Directory for Generated .ser and .class Files \(-d\)" on page 8-30](#)
- ["Encoding for Input and Output Source Files \(-encoding\)" on page 8-29](#)
- ["Options to Pass to the Java Compiler \(-C\)" on page 8-62](#)
- ["Compilation Flag \(-compile\)" on page 8-65](#)
- ["Compiler Encoding Support \(-compiler-encoding-flag\)" on page 8-83](#)
- ["Name of the Java Compiler \(-compiler-executable\)" on page 8-82](#)
- ["Compiler Output File \(-compiler-output-file\)" on page 8-83](#)
- ["Compiler Message Output Pipe \(-compiler-pipe-output-flag\)" on page 8-84](#)

Profile Customization (ISO Code Generation)

After Java compilation, if you are using ISO standard code generation (`-codegen=iso`), the generated profiles containing information about your embedded SQL instructions are customized so that your application can work efficiently with your database and use vendor-specific extensions.

Note: If you use the default Oracle-specific code generation (`-codegen=oracle`), SQLJ produces no profiles and skips the customization step. Your code will support Oracle-specific features through direct calls to Oracle JDBC APIs. See ["Oracle-Specific Code Generation \(No Profiles\)" on page 4-39](#) for more information.

To accomplish customization, SQLJ invokes a front end called the *customizer harness*, which is a Java class that functions as a command-line utility. The harness, in turn, invokes a particular customizer, either the default Oracle customizer or a customizer that you specify through SQLJ option settings.

During customization, profiles are updated in 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 Oracle9i `PERSON` type that you had defined. You could then use `PERSON` as you would any other supported datatype.

You also must customize with the Oracle customizer to use any of the `oracle.sql` type extensions.

Notes: Be aware of the following regarding profile customization:

- The Oracle SQLJ runtime and an Oracle JDBC driver will be required by your application whenever you use the Oracle customizer during translation, even if you do not use Oracle extensions in your code.
 - The generic SQLJ runtime will be used if your application has no customizations, or none suitable for the connection.
 - You can customize previously created profiles by specifying `.ser` files, or `.jar` files containing `.ser` files, on the command line. But you cannot do this in the same running of SQLJ where translations are taking place. You can specify `.ser/.jar` files to be customized, or `.sqlj/.java` files to be translated, compiled, and customized, but not both categories. For more information about how `.jar` files are used, see "[JAR Files for Profiles](#)" on page A-38.
-
-

For more information about profile customization, see [Appendix A, "Customization and Specialized Customizers"](#).

Also see the following for information about SQLJ options related to profile customization:

- "[Default Profile Customizer \(-default-customizer\)](#)" on page 8-88
- "[Options to Pass to the Profile Customizer \(-P\)](#)" on page 8-63
- "[Profile Customization Flag \(-profile\)](#)" on page 8-66
- "[Customization Options and Choosing a Customizer](#)" on page A-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 might encounter during the translation phase: *error*, *warning*, and *information*. Warning messages can be further broken down into two levels: *non-suppressible* and *suppressible*. Therefore, there are four message categories (in order of seriousness):

1. errors
2. non-suppressible warnings
3. suppressible warnings
4. information

You can control suppressible 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 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-suppressible warning messages, prefixed by `Warning:`, indicate that one of the following has been encountered:

- a condition that would probably, but not necessarily, result in a runtime error if the code were executed (for example, a `SELECT` statement whose output is not assigned to anything)
- a condition that compromises the ability of SQLJ to verify 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-suppressible 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 SQL parser that will be used at runtime. Therefore, errors might occasionally be detected during translation that will not actually cause problems when your application runs. Accordingly, such errors are reported as non-suppressible warnings, rather than fatal errors.

Suppressible warning messages, also prefixed by `Warning:`, indicate that there is a problem with a particular aspect of your application, such as portability. An example of this is using an Oracle-specific type such as `oracle.sql.NUMBER` to read from or write to Oracle9i.

Informational or status messages prefixed by `Info:` do not indicate an error condition. They merely provide additional information about what occurred during the translation phase.

Suppressible warning and status messages can be suppressed by using the various `-warn` option flags:

- `cast/nocast`—The `nocast` setting suppresses warnings about possible runtime errors when trying to cast an object type instance to an instance of a subtype.
- `precision/noprecision`—The `noprecision` setting suppresses warnings regarding possible loss of data precision during conversion.
- `nulls/nonulls`—The `nonulls` setting suppresses warnings about possible runtime errors due to nullable columns or types.
- `portable/noportable`—The `noportable` setting suppresses warnings regarding SQLJ code that uses Oracle-specific features or might otherwise be 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.

- `verbose/noverbose`—The `noverbose` setting suppresses status messages that are merely informational and do not indicate error or warning conditions.

See "[Translator Warnings \(-warn\)](#)" on page 8-45 for more information about the `-warn` option and how to set the flags.

If you receive warnings during your SQLJ translation, you can try running the translator again with `-warn=none` to see if any of the warnings are of the more serious (non-suppressible) variety.

Notes: For information about particular error, warning, and information messages, see "[SQLJ Translation Time Messages](#)" on page B-2 and "[SQLJ Runtime Messages](#)" on page B-47.

[Table 9-2](#) summarizes the categories of error and status messages generated by the SQLJ translator.

Table 9-2 SQLJ Translator Error Message Categories

Message Category	Prefix	Indicates	Suppressed By
Error	Error:	fatal error that will cause compilation failure or runtime failure (translation is aborted)	n/a
Non-suppressible warning	Warning:	condition that prevents proper translation or might cause runtime failure (translation is completed)	n/a
Suppressible warning	Warning:	problem regarding a particular aspect of your application (translation is completed)	-warn option flags: nocast noprecision nonnulls noportable nostrict
Informational/status message	Info:	information regarding the translation process	-warn option flag: noverbose

Translator Status Messages

In addition to the error, warning, and information messages, SQLJ can produce status messages throughout all phases of SQLJ operation—translation, compilation, and customization. Status messages are 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 described in "[Real-Time Status Messages \(-status\)](#)" on page 8-48.

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 presents information about the Oracle SQLJ runtime, which is a thin layer of pure Java code that runs above the JDBC driver.

If you use the default Oracle-specific code generation (`-codegen=oracle`), the SQLJ runtime layer becomes even thinner, with a runtime subset being used in conjunction with an Oracle JDBC driver. Most of the runtime functionality is compiled directly into Oracle JDBC calls. You cannot use a non-Oracle JDBC driver. See "[Oracle-Specific Code Generation \(No Profiles\)](#)" on page 4-39.

When Oracle SQLJ translates SQLJ source code using ISO standard code generation (`-codegen=iso`), 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.

Generally speaking, however, 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 datatypes 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.

Note: For ISO standard SQLJ code generation, the Oracle SQLJ runtime and an Oracle JDBC driver will be required by your application whenever you use the Oracle customizer during translation, even if you do not use Oracle extensions in your code. The generic SQLJ runtime will be used if your application has no customizations, or none suitable for the connection.

SQLJ Runtime Packages

The Oracle SQLJ runtime includes packages you will likely import and use directly, and others that are used only indirectly.

Note: These packages are included in the runtime libraries `runtime12`, `runtime12ee`, `runtime11`, `runtime`, and `runtime-nonoracle`.

Packages Used Directly

This section lists packages containing classes you can import and use directly in your application. Packages whose names begin with `oracle` are for Oracle-specific SQLJ features.

- `sqlj.runtime`

This package includes the `ExecutionContext` class, `ExecutionContext` interface, `ConnectionFactory` interface, `ResultSetIterator` interface, `ScrollableResultSetIterator` interface, and wrapper classes for streams (`BinaryStream` and `CharacterStream`, as well as the deprecated `AsciiStream` and `UnicodeStream`).

Interfaces and abstract classes in this package are implemented by classes in the `sqlj.runtime.ref` package or the `oracle.sqlj.runtime` package or by classes generated by the SQLJ translator.

- `sqlj.runtime.ref`

The classes in this package implement interfaces and abstract classes in the `sqlj.runtime` package. You will likely use the `sqlj.runtime.ref.DefaultContext` class, which is used to specify your default connection and create default connection context instances. The other classes in this package are used internally by SQLJ in defining classes during code generation, such as iterator classes and connection context classes that you declare in your SQLJ code.

- `oracle.sqlj.runtime`

This package contains the `Oracle` class that you can use to instantiate the `DefaultContext` class and establish your default connection. It also contains Oracle-specific runtime classes used by the Oracle implementation of SQLJ, including functionality to convert to and from Oracle type extensions.

Packages Used Indirectly

This section lists packages containing classes that are for internal use by SQLJ.

- `sqlj.runtime.profile`

This package contains interfaces and abstract classes that define what SQLJ profiles look like (applicable only for ISO standard code generation). This includes the `EntryInfo` 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 (for ISO standard code generation only). It also provides the default JDBC-based runtime implementation.

- `sqlj.runtime.error`

This package, used internally by SQLJ, contains resource files for all generic (not Oracle-specific) error messages that can be generated by the SQLJ translator.

- `oracle.sqlj.runtime.error`

This package, used internally by SQLJ, contains resource files for all Oracle-specific error messages that can be generated by the SQLJ translator.

Categories of Runtime Errors

Runtime errors can 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 as a subclass such as `sqlj.runtime.SQLNullException`.

Depending on where the error came from, there might be meaningful information you can retrieve from an exception using the `getSQLState()`, `getErrorCode()`, and `getMessage()` methods. SQLJ errors, for example, include meaningful SQL states and messages. For information, see ["Retrieving SQL States and Error Codes"](#) on page 4-24.

If errors are generated by the Oracle JDBC driver or RDBMS at runtime, look at the prefix and consult the appropriate documentation:

- *Oracle9i JDBC Developer's Guide and Reference* for JDBC errors
- *Oracle9i Database Error Messages* reference for RDBMS errors

For a list of SQLJ runtime errors, see ["SQLJ Runtime Messages"](#) on page B-47.

Globalization Support in the Translator and Runtime

Oracle SQLJ uses the Java built-in capabilities for globalization support (otherwise known as National Language Support, or NLS). This section discusses the following:

- basics of SQLJ support for globalization and native character encoding, starting with background information covering some of the implementation details of character encoding and language support in Oracle SQLJ
- options available through the Oracle SQLJ command line that allow you to adjust your Oracle Globalization Support configuration
- extended Oracle SQLJ globalization support
- relevant manipulation outside of Oracle SQLJ for globalization support

Some prior knowledge of Oracle Globalization Support is assumed, particularly regarding character encoding and locales. For information, see the *Oracle9i Database Globalization Support Guide*.

Character Encoding and Language Support

There are two main areas of SQLJ globalization support:

- character encoding

There are three parts to this:

- character encoding for reading and generating source files during SQLJ translation
- character encoding for generating error and status messages during SQLJ translation
- character encoding for generating error and status messages when the application runs

- language support

This determines which translations of error and status message lists are used when SQLJ outputs messages to the user, either during SQLJ translation or SQLJ runtime.

Globalization support at runtime is transparent to the user, presuming your SQLJ source code and SQL character data use only characters that are within the database character set. SQL character data is transparently mapped into and out of Unicode.

Note that for multi-language applications, it is advisable to use one of the following options:

- Use a database whose character set supports Unicode.

or:

- Even if your database character set does not support Unicode, specify that the national language character set supports Unicode. (See the *Oracle9i Database Globalization Support Guide*.) In this case, you will typically use the SQLJ Unicode character types described in "[Oracle SQLJ Extended Globalization Support](#)" on page 9-25.

Notes:

- The SQLJ translator fully supports Unicode 2.0 and Java Unicode escape sequences. The SQLJ command-line utility, however, does not support Unicode escape sequences—you can use only native characters supported by the operating system. Command-line options requiring Unicode escape sequences can be entered in a SQLJ properties file instead, because properties files do support Unicode escape sequences.
 - Encoding and conversion of characters in your embedded SQL operations, and characters read from or written to the database, are handled by JDBC directly. SQLJ does not play a role in this. If online semantics-checking is enabled during translation, however, you will be warned if there are characters within the text of your SQL DML operations that might not be convertible to the database character set.
 - For information about JDBC globalization support functionality, see the *Oracle9i 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 JVM `file.encoding` property. If your source files use other encodings, then you must indicate this to SQLJ so that appropriate conversion can be performed.

Use the SQLJ `-encoding` option to accomplish this. SQLJ also passes the `-encoding` setting to the compiler for it to use in reading `.java` files, unless the SQLJ `-compiler-encoding-flag` is off.

Important: Do not alter the `file.encoding` system property to specify encodings for source files. This might impact other aspects of your Java operation and might offer only a limited number of encodings, depending on platform or operating system considerations.

The system character-encoding setting also determines how SQLJ error and status messages are represented when output to the user, either during translation or during 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 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 (or replace "1.2" in the URL with the appropriate JDK level, such as "1.1" or "1.3"):

<http://www.javasoft.com/products/jdk/1.2/docs/tooldocs/solaris/native2ascii.html>

Dozens of encodings are supported by the Sun Microsystems JDK. These include 8859_1 through 8859_9 (ISO Latin-1 through ISO Latin-9), JIS (Japanese), SJIS (shift-JIS, Japanese), and UTF8.

Character Encoding Notes Be aware of the following:

- A character that is not representable in the encoding used, for either messages or source files, can always be represented as a Java Unicode escape sequence. This is of the form `\uHHHH`, where each H is a hexadecimal digit.
- As a `.sqlj` source file is read and processed during translation, error messages quote source locations based on character position (not byte position) in the input encoding.
- Encoding settings, either set through the SQLJ `-encoding` option or the Java `file.encoding` setting, do not apply to Java properties

files—`sqlj.properties` and `connect.properties`, for example. Properties files always use the encoding `8859_1`. This is a feature of Java in general, not SQLJ in particular. You can, however, use Unicode escape sequences in a properties file. You can use the `native2ascii` utility to determine escape sequences—see ["Using native2ascii for Source File Encoding"](#) on page 9-30.

Overview of Language Support

SQLJ error and status reporting, either during translation or during runtime, uses the Java locale setting in the JVM `user.language` property. Users typically do not have to alter this setting.

Language support is implemented through message resources that use key/value pairs. For example, where an English-language resource has a key/value pair of `"OkKey", "Okay"`, a German-language resource has a key/value pair of `"OkKey", "Gut"`. The locale setting determines the message resources used.

SQLJ supports locale settings of `en` (English), `de` (German), `fr` (French), and `ja` (Japanese).

Note: Java locale settings can support country and variant extensions in addition to language extensions. For example, consider `ErrorMessages_de_CH_var1`, where `CH` is the Swiss country extension of German, and `var1` is an additional variant. SQLJ, however, currently supports only language extensions (`de` in this example), ignoring country and variant extensions.

SQLJ and Java Settings for Character Encoding and Language Support

Oracle SQLJ provides syntax that allows you to set the following:

- character encoding used by the SQLJ translator and Java compiler in representing source code
Use the SQLJ `-encoding` option.
- 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.
- 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 generated by the translator, and `.java` files read by the compiler. The option setting is passed by `SQLJ` to the compiler, unless the `SQLJ -compiler-encoding-flag` is off.

This option can be set on the command line or `SQLJ_OPTIONS` environment variable, as in the following example:

```
-encoding=SJIS
```

or in a `SQLJ` properties file, as follows:

```
sqlj.encoding=SJIS
```

If the encoding option is not set, then both the translator and compiler will use the encoding specified in the JVM `file.encoding` property. This can also be set through the `SQLJ` command line, as discussed in "[Setting Character Encoding and Locale for SQLJ Messages](#)" on page 9-23.

For more information, see "[Encoding for Input and Output Source Files \(-encoding\)](#)" on page 8-29 and "[Compiler Encoding Support \(-compiler-encoding-flag\)](#)" on page 8-83.

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 8-16.

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 JVM properties in the `SQLJ` command line by using the `SQLJ -J` prefix. Options marked by this prefix are passed to the JVM.

Set the character encoding as in the following example, which specifies shift-JIS Japanese character encoding:

```
-J-Dfile.encoding=SJIS
```

Note: Only a limited number of encodings might be available, depending on platform or operating system considerations.

Set the locale as in the following example (which specifies Japanese locale):

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

The `-J` prefix can be used on the command line or `SQLJ_OPTIONS` environment variable only. It cannot be used in a properties file, because properties files are read after the JVM is invoked.

Notes:

- 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 8-19.
 - Remember that if you do not set the `SQLJ -encoding` option, then setting `file.encoding` will affect encoding for source files as well as error and status messages.
 - Be aware that altering the `file.encoding` property might have unforeseen consequences on other aspects of your Java operations. Also, any new setting must be compatible with your operating system.
-
-

For additional information about the `SQLJ -J` prefix, see "[Command-Line Syntax and Operations](#)" on page 8-12 and "[Options to Pass to the Java Virtual Machine \(-J\)](#)" on page 8-60.

SQLJ Command-Line Example: Setting Encoding and Locale

Following is a complete SQLJ command line, including JVM `file.encoding` and `user.language` settings:

```
sqlj -encoding=8859_1 -J-Dfile.encoding=SJIS -J-Duser.language=ja Foo.sqlj
```

This example uses the `SQLJ -encoding` option to specify `8859_1` (Latin-1) for source code representation during SQLJ translation. This encoding is used by the translator in reading the `.sqlj` input file and in generating the `.java` output file. The encoding is then passed to the Java compiler to be used in reading the generated `.java` file. The `-encoding` option, when specified, is always passed to the Java compiler unless the `SQLJ -compiler-encoding-flag` is disabled.

For error and status messages output during translation of `Foo.sqlj`, the SQLJ translator uses the `SJIS` encoding and the `ja` locale.

Oracle SQLJ Extended Globalization Support

Oracle9i SQLJ adds support for Java types (Unicode character types) derived from existing character and stream types that convey expected usage for globalization support in the server. (In SQLJ it is not possible to use JDBC statement or result set methods directly that otherwise serve the purpose of globalization support, but if you are interested in information about those methods, refer to the *Oracle9i JDBC Developer's Guide and Reference*.)

If the database natively supports Unicode, then the types described in "[Java Types for Globalization Support](#)" below are unnecessary—globalization support will be handled transparently. It is when the database does not natively support Unicode, but has a national language character set that does support Unicode, that you will typically use these types (for columns that employ the national language character set).

Java Types for Globalization Support

Oracle9i SQLJ provides a number of Java types for globalization support. [Table 9-3](#) notes the correspondence between these globalization support types and general-use JDBC and SQLJ character and stream types. Each globalization support type, except for `NString`, is a subclass of its corresponding JDBC or SQLJ type.

Table 9–3 JDBC and SQLJ Types and Corresponding Globalization Types

JDBC and SQLJ Types	Globalization Support Types
JDBC types:	
oracle.sql.CHAR	oracle.sql.NCHAR
java.lang.String	oracle.sql.NString
oracle.sql.CLOB	oracle.sql.NCLOB
SQLJ types:	
sqlj.runtime.CharacterStream	oracle.sqlj.runtime.NcharCharacterStream
sqlj.runtime.AsciiStream (deprecated ; use CharacterStream)	oracle.sqlj.runtime.NcharAsciiStream (deprecated ; use NcharCharacterStream)
sqlj.runtime.UnicodeStream (deprecated ; use CharacterStream)	oracle.sqlj.runtime.NcharUnicodeStream (deprecated ; use NcharCharacterStream)

In situations where your application must handle national language character strings, either inserting them into or selecting them from national language character set columns, use the globalization support types instead of the corresponding general-use types.

Notes:

- All globalization support types add automatic registration of intended usage for IN and OUT parameters, but are otherwise identical in usage to the corresponding JDBC or SQLJ type (including constructors).
 - Use of globalization support types is unnecessary in iterator columns, because the underlying network protocol supports national language characters implicitly for the underlying result sets.
 - In Oracle9i there is no direct support for these globalization support types in Oracle JDBC.
-
-

NString Class Usage and Notes

The `oracle.sql.CHAR` class, and therefore its `NCHAR` subclass, provides only constructors that require explicit knowledge of the database character set. Therefore, the `oracle.sql.NString` class, a wrapper for `java.lang.String`, is preferable in most circumstances.

The `NString` class provides simpler constructors and ensures that the national language character form of use is registered with the JDBC driver.

Following are the key `NString` methods:

- `NString(String)`—This constructor creates an `NString` instance from an existing `String` instance.
- `String toString()`—This method returns the underlying `String` instance.
- `String getString()`—This method also returns the underlying `String` instance.

The `toString()` method allows you to employ the `NString` instance in string concatenation expressions (such as `"a"+b`, where `b` is a string). The `getString()` method, provided in the `CHAR` superclass, is supported as well for uniformity. In addition, the member methods of the `String` class are carried over to the `NString` wrapper class to allow you to write more concise code.

Globalization Support Examples

The following examples show use of the `NString` class.

NString as IN Argument This example uses an `NString` instance as an input parameter to the database.

```
import oracle.sql.NString;
...
NString nc_name = new NString("Name with strange characters");
#sql { update PEOPLE
        set city = :(new NString("\uffff2")), name = :nc_name
        where num= :n };
...
```

NString as OUT Argument This example uses an `NString` instance as an output parameter from the database.

```
import oracle.sql.NString;
...
```

```
NString nstr;  
#sql { call foo(:out nstr) };  
System.out.println("Result is: "+nstr);  
// or, explicitly: System.out.println("Result is: "+nstr.toString());  
...
```

NString as Result Set Column This example uses the `NString` type for an iterator column. Such usage is superfluous, given that the underlying network protocol supports national language characters implicitly, but harmless. This example also shows use of one of the `String` methods, `substring()`, that is carried over to `NString`.

```
import oracle.sql.NString;  
import oracle.sql.NCLOB;  
...  
#sql iterator NIter(NString title, NCLOB article);  
  
NIter nit;  
#sql nit = { SELECT article, title FROM page_table };  
while (nit.next())  
{  
    System.out.println("<TITLE>"+nit.title()+"</TITLE>");  
    ...  
    nit.article().substring(0, 1000); ...  
}
```

Note: Using the `NCHAR` type instead of the `NString` type for the preceding examples requires the following changes:

- Use the appropriate `NCHAR` constructor. `NCHAR` constructors mirror `CHAR` constructors, such as the following:

```
NCHAR(String str, oracle.sql.CharacterSet charset)
```
 - Although you have the option of using either `toString()` or `getString()` to retrieve the underlying `String` instance from an `NString` instance, for an `NCHAR` instance you must use the `getString()` method. When using the `NString` type, the `toString()` method is used automatically for string concatenation, such as in "[NString as OUT Argument](#)" above.
-
-

Manipulation Outside of SQLJ for Globalization Support

This section discusses ways to manipulate your Oracle Globalization Support configuration outside of SQLJ.

Setting Encoding and Locale at Application Runtime

As with any end user running any Java application, those running your SQLJ application can specify JVM properties such as `file.encoding` and `user.language` directly, as they invoke the JVM to run your application. This determines the encoding and locale used for message output as your application executes.

They can accomplish this as in the following example:

```
java -Dfile.encoding=SJIS -Duser.language=ja Foo
```

This will use `SJIS` encoding and Japanese locale.

Using API to Determine Java Properties

In Java code, you can determine values of Java properties by using the `java.lang.System.getProperty()` method, specifying the appropriate property. For example:

```
public class Settings
{
    public static void main (String[] args)
    {
        System.out.println("Encoding: " + System.getProperty("file.encoding")
            + ", Language: " + System.getProperty("user.language"));
    }
}
```

You can compile this and run it as a standalone utility.

There is also a `getProperties()` method that returns the values of all properties, but this will raise a security exception if you try to use it in code that runs in the server.

You can get information about the `java.lang.System` class at the following Web site (or replace "1.2" in the URL with the appropriate JDK level, such as "1.1" or "1.3"):

<http://www.javasoft.com/products/jdk/1.2/docs/api/java.lang.System.html>

Using native2ascii for Source File Encoding

If you are using a Sun Microsystems 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: To use SQLJ to translate source created by `native2ascii`, ensure that the JVM that invokes SQLJ has a `file.encoding` setting that supports some superset of 7-bit ASCII. This is not the case with settings for EBCDIC or Unicode encoding.

Run `native2ascii` as follows:

```
% native2ascii <options> <inputfile> <outputfile>
```

Standard input or standard output are used if you omit the input file or output file. Two options are supported:

- `-reverse` (reverse the conversion; convert from Latin-1 or Unicode to native encoding)
- `-encoding <encoding>`

For example:

```
% native2ascii -encoding SJIS Foo.sqlj Temp.sqlj
```

For more information, see the following Web site (or replace "1.2" in the URL with the appropriate JDK level, such as "1.1" or "1.3"):

<http://www.javasoft.com/products/jdk/1.2/docs/tooldocs/solaris/native2ascii.html>

10

Performance and Debugging

This chapter discusses features, utilities, and tips to enhance performance of your SQLJ application and to debug your SQLJ source code at runtime. The following topics are discussed:

- [Performance Enhancement Features](#)
- [SQLJ Debugging Features](#)

Performance Enhancement Features

Oracle SQLJ includes features to enhance your performance by making data access more efficient. These include the following:

- row prefetching—Query results are sent to your application in groups of rows, instead of one at a time.
- update batching—Database updates, inserts, and deletes are sent to the database in batches, instead of one at a time.
- statement caching—Prepared statements are saved in memory and reused, to avoid repeated processing in the server.
- column definitions—Column types and sizes are predefined, possibly saving round trips to the database.
- parameter size definitions—Sizes of host variables are predefined for more efficient memory usage.

For information about underlying Oracle JDBC support for these features, see the *Oracle9i JDBC Developer's Guide and Reference*.

Your application will likely benefit from the default Oracle-specific code generation (`-codegen=oracle`). The generated code will be optimized with direct calls to Oracle JDBC, eliminating the overhead of intermediate calls to the SQLJ runtime (which in turn would call JDBC). For information, see "[Oracle-Specific Code Generation \(No Profiles\)](#)" on page 4-39.

Note: Neither Oracle SQLJ nor Oracle JDBC supports batch fetches, which is the fetching of sets of rows into arrays of values. You may, however, be able to use Oracle row prefetching to obtain some of the benefits of batch fetching.

In addition to the preceding Oracle SQLJ (and JDBC) performance enhancements, you can employ optimizer hints in the SQL operations within a SQLJ program, as you can in any Oracle SQL operations.

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 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 Oracle9i. 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 *Oracle9i SQL Reference*.

Note that using Oracle performance extensions in your code requires the following:

- Use one of the Oracle JDBC drivers.
- Use the default Oracle-specific code generation, or customize profiles appropriately.

For ISO standard code generation, the default customizer, `oracle.sqlj.runtime.util.OraCustomizer`, is recommended.

- Use the Oracle SQLJ runtime when your application runs.

The Oracle SQLJ runtime and an Oracle JDBC driver are required by your application whenever you customize profiles with the Oracle customizer, even if you do not actually use Oracle extensions in your code.

Row Prefetching

Standard JDBC receives the results of a query one row at a time, with each row requiring a separate round trip to the database. *Row prefetching* allows you to receive the results more efficiently, in groups of multiple rows each.

Use the `setFetchSize()` method of an `ExecutionContext` instance to set the number of rows to be prefetched whenever you execute a `SELECT` statement (for SQLJ statements using the particular `ExecutionContext` instance).

The `getFetchSize()` method of an `ExecutionContext` instance returns the current prefetch size, as an `int` value.

Here is an example of setting the prefetch size to 20 by getting the default execution context instance of the default connection context instance and calling the `setFetchSize()` method:

```
DefaultContext.getDefaultContext().getExecutionContext().setFetchSize(20);
```

(It is also possible to set the prefetch size directly on the underlying `OracleConnection` object, using the JDBC API, but in SQLJ this is discouraged.)

To specify the number of rows to prefetch for queries that use a given connection context instance, use the underlying JDBC connection, cast to a `Connection`

instance. Following is an example that sets the prefetch value to 20 for your default connection:

```
((Connection)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:

```
((Connection)ctx.getConnection()).setDefaultRowPrefetch(20);
```

There is no maximum row-prefetch value. The default is 10 in JDBC, and this is inherited by SQLJ. This value is effective in typical circumstances, although you might want to increase it if you receive a large number of rows.

Statement Caching

SQLJ offers a *statement caching* feature that improves performance by saving executable statements that are used repeatedly, such as in a loop or in a method that is called repeatedly. The statement cache is located on the client. When a statement is cached before it is re-executed, the code does not have to be reparsed (either on the client or in the server), the statement object does not have to be re-created, and the parameter size definitions do not have to be recalculated. Without this feature, repeated statements would have to be reparsed on the client, and perhaps in the server as well (depending on whether a statement is still available in the general server-side SQL cache when it is encountered again).

For Oracle-specific code generation (the default `-codegen=oracle` setting), Oracle SQLJ statement caching relies on the Oracle JDBC driver, using the Oracle JDBC *explicit caching* mechanism. This is distinct from the Oracle JDBC *implicit caching* mechanism, although there are interdependencies as described below. For more information about these mechanisms, see the *Oracle9i JDBC Developer's Guide and Reference*. With Oracle-specific code, statement caching is controlled through connection methods.

For ISO code generation (`-codegen=iso`), SQLJ has its own statement caching mechanism through functionality of the SQLJ runtime. With ISO code, statement caching is controlled through the Oracle customizer `stmtcache` option.

This section covers the following topics:

- [Connection Context Methods for Statement Caching \(Oracle-Specific Code\)](#)
- [Enabling and Disabling Statement Caching \(Oracle-Specific Code\)](#)
- [Key Interactions Between SQLJ/Explicit Caching and JDBC Implicit Caching](#)

- [JDBC Support for Statement Caching \(ISO Code\)](#)
- [Oracle Customizer Option for Statement Cache Size \(ISO Code\)](#)
- [Additional Statement Caching Behavior](#)
- [Statement Caching Limitations and Notes](#)

Notes:

- For Oracle-specific code generation, explicit caching is the only statement caching mechanism that can be manipulated through SQLJ APIs. For the discussion in this document, it will be referred to as "SQLJ/explicit" statement caching.
 - For those who have used statement caching in the past, note that much of the functionality of SQLJ and JDBC statement caching changes significantly as of Oracle9i release 2. Backward compatibility is assured, however.
-
-

Connection Context Methods for Statement Caching (Oracle-Specific Code)

If you use Oracle-specific code generation, which is the case with the SQLJ translator default `-codegen=oracle` setting, use connection context methods for statement caching functionality.

Note that any statement cache size greater than 0 results in SQLJ/explicit statement caching being enabled. By default, it is enabled with a cache size of 5 (five statements).

The following Oracle-specific (non-standard) static methods have been added to the `sqlj.runtime.ref.DefaultContext` class, and are also included in any connection context classes you declare:

- `public static void setDefaultStmtCacheSize(int)`

This sets the default statement cache size for *all* connection contexts. This becomes the initial statement cache size for any subsequently created instance of *any* connection context class, not just the class upon which you call the method. The method call does not affect connection context instances that already exist.

- `public static int getDefaultStmtCacheSize()`

This retrieves the current default statement cache size for connection contexts.

And the following Oracle-specific instance methods have also been added to the `DefaultContext` class and are included in any other connection context classes:

- `public void setStmtCacheSize(int)`
throws `java.sql.SQLException`

This sets the statement cache size for the underlying connection of the particular connection context instance (overrides the default).

Note: If SQLJ/explicit caching is already disabled, setting the size to 0 leaves it disabled. If it is already enabled, setting the size to 0 leaves it "enabled", but renders it non-functional.

- `public int getStmtCacheSize()`

This verifies whether SQLJ/explicit statement caching is enabled for the underlying connection of the connection context. If so, it returns the current statement cache size. It can also return either of the following integer constants:

```
public static int STMT_CACHE_NOT_ENABLED
public static int STMT_CACHE_EXCEPTION
```

It is possible for a `getStmtCacheSize()` call to cause a SQL exception; however, for backward compatibility, this method does not throw the exception directly. When an exception occurs, the method returns the constant `STMT_CACHE_EXCEPTION`. In this case, you can call the `getStmtCacheException()` method (described immediately below) to find out what exception occurred.

If you call `getStmtCacheSize()` when SQLJ/explicit caching is disabled, then the method returns the constant `STMT_CACHE_NOT_ENABLED`. As of Oracle9i release 2, this is distinguished from a cache size of 0. Technically, it is possible for SQLJ/explicit caching to be "enabled" (though useless) with a cache size of 0.

- `public java.sql.Exception getStmtCacheException()`

See if there is a statement caching exception. There are two scenarios for using this method:

- Call it if a `getStmtCacheSize()` call returns `STMT_CACHE_EXCEPTION`.
- Call it whenever you create a connection context instance with which you want to use statement caching. This is because of automatic manipulation

that occurs with respect to statement cache size whenever you create a connection context instance. (This is described in "[Additional Statement Caching Behavior](#)" on page 10-9.) If you care about statement caching for the connection context instance, call `getStmtCacheException()` after creating the instance, to verify there were no problems.

Enabling and Disabling Statement Caching (Oracle-Specific Code)

With Oracle-specific code, to reiterate what was stated earlier, any nonzero statement cache size results in SQLJ/explicit caching being enabled. Because the default size is 5, statement caching is enabled by default.

You cannot explicitly disable SQLJ/explicit statement caching through SQLJ APIs, although you can effectively disable it (render it non-functional) by setting the statement cache size to 0. In this case, the connection context `getStmtCacheSize()` method might return 0, *not* `STMT_CACHE_NOT_ENABLED`.

You *can* explicitly disable SQLJ/explicit statement caching, or JDBC implicit caching, through JDBC connection APIs. Because SQLJ/explicit caching and JDBC implicit caching use the same cache size, as described in "[Key Interactions Between SQLJ/Explicit Caching and JDBC Implicit Caching](#)" on page 10-8, there might sometimes be reason to do so. The following methods are available through the `OracleConnection` class:

- `public void setExplicitCachingEnabled(boolean)`
- `public boolean getExplicitCachingEnabled()`
- `public void setImplicitCachingEnabled(boolean)`
- `public boolean getImplicitCachingEnabled()`

You have access to these methods if you retrieve the `OracleConnection` instance from within a SQLJ connection context instance. (See "[SQLJ Connection Context and JDBC Connection Interoperability](#)" on page 7-53.)

Note: In SQLJ, JDBC implicit caching is disabled by default, and remains disabled unless you explicitly enable it through the `setImplicitCachingEnabled()` method.

Key Interactions Between SQLJ/Explicit Caching and JDBC Implicit Caching

With regard to statement caching in Oracle-specific code, this document naturally emphasizes SQLJ/explicit caching rather than JDBC implicit caching. If you do not use JDBC code in your application, SQLJ/explicit caching is the only statement caching that is relevant. However, there are situations where you might want to use both SQLJ and JDBC code in your application, and in these circumstances you might also want to use implicit caching.

As described earlier, SQLJ/explicit caching and JDBC implicit caching are enabled independently of each other. Furthermore, you do not have access to the implicit cache through SQLJ.

There is, however, a key interaction between the two—they share the same cache size. If, for example, the statement cache size is 5, then you can have a maximum total of five statements cached for SQLJ/explicit caching and implicit caching combined.

An important point related to this is that if you choose to effectively disable SQLJ/explicit statement caching by setting the cache size to 0, then you have also effectively disabled implicit caching.

Also be aware that if SQLJ/explicit caching is disabled, changing the cache size to a value greater than 0 will enable it, but this does not affect whether implicit caching is enabled.

JDBC Support for Statement Caching (ISO Code)

With ISO standard code generation, specified through the SQLJ translator `-codegen=iso` setting, statement caching is a standard SQLJ feature that does not require any particular JDBC driver. However, using a driver that implements the interface `sqlj.runtime.profile.ref.ClientDataSupport` allows more robust caching. Oracle9i JDBC drivers implement this interface, providing the following features:

- a separate cache for each database connection, instead of a single static cache for the entire application
- the ability to share cached statements between multiple instances of a connection context class that share the same underlying connection

When a single cache is used, as is the case with a generic JDBC driver that does not implement `ClientDataSupport`, a statement executed in one connection can cause a cached statement from another connection to be flushed (if the statement cache size—the maximum number of statements that can be cached—is exceeded).

Oracle Customizer Option for Statement Cache Size (ISO Code)

With ISO standard code generation, statement caching is enabled in your application by default with a cache size of 5 (the same default size as with Oracle-specific code) when you use the Oracle customizer, which is typically executed as part of Oracle SQLJ translation.

You can alter the statement cache size as desired, or effectively disable statement caching with a cache size of 0, through the Oracle customizer `stmtcache` option. This is set as `-P-Cstmtcache=n`, where `n` is an integer. For information, see ["Oracle Customizer Statement Cache Size Option \(stmtcache\)"](#) on page A-34.

If you use multiple connection context classes and, therefore, have multiple profiles, you can set their statement cache sizes individually by running SQLJ (actually, the customizer) separately for each profile.

At runtime, the appropriate SQLJ profile determines the statement cache size for a connection. This would be the profile that corresponds to the first connection context class instantiated for this connection. Its statement cache size setting, if any, is determined according to how you set the Oracle customizer `stmtcache` option when you customized the profile. The runtime statement cache size for a connection is set when the first statement on that connection is executed.

Additional Statement Caching Behavior

This section describes additional behaviors and interactions of statement caching, applicable with either Oracle-specific or ISO code generation.

Automatic Manipulation of JDBC Statement Cache Size When a SQLJ connection context object is instantiated, if the statement cache size on the underlying JDBC connection is smaller than the default size for the connection context class, then the SQLJ runtime will attempt to increase the JDBC statement cache size to the connection context default value. This manipulation occurs even with ISO code generation, enabling Oracle JDBC explicit statement caching in the process, although this is actually of no relevance in the ISO code case.

If, on the other hand, the actual JDBC statement cache size is larger, then the SQLJ runtime will not attempt to perform a change in the cache size. The SQLJ runtime checks the actual JDBC cache size against the default size set whenever it creates a SQLJ connection context instance.

It is important to note that these methods have the same effect regardless of the context class on which they are issued, since they modify or report the same underlying static field.

As an example, assume the following connection context class declarations:

```
#sql context CtxtA;  
#sql context CtxtB;
```

In this case, each of the following three code instructions has the same effect—that whenever a new SQLJ connection context instance is subsequently created, it will *not* try to enable SQLJ/explicit statement caching:

```
sqlj.runtime.ref.DefaultContext.setDefaultStmtCacheSize(0);
```

or:

```
CtxtA.setDefaultStmtCacheSize(0);
```

or:

```
CtxtB.setDefaultStmtCacheSize(0);
```

Important: If a SQLJ connection context instance is created on an underlying JDBC *pooled connection*, then SQLJ will not be able to change the JDBC statement cache size. (For Oracle-specific code, you can retrieve the resulting exception through the connection context `getStmtCacheException()` method.) In this case, the desired JDBC statement cache size must be set explicitly on the underlying physical connection(s). For data sources, the cache size is set through vendor-specific data source attributes.

Statement Caching Semantics SQLJ/explicit caching and JDBC implicit caching functionality have different semantics and behaviors. As noted earlier, SQLJ statement caching applies only to single statements used repeatedly, such as in a loop or through repeated calls to the same method. Consider the following example:

```
#sql { same SQL operaton }; // occurrence #1  
...  
Java code  
...  
#sql { same SQL operaton }; // occurrence #2  
...  
Java code  
...  
#sql { same SQL operaton }; // occurrence #3  
...
```


Assume the three SQL operations are identical, including white space.

SQLJ caching would consider these three occurrences of the same SQL operation to be three different statements. They will occupy three separate slots in the cache. JDBC implicit caching, however, would recognize these as identical statements, using only a single cache slot for all three. The statement would be reused for occurrence #2 and occurrence #3.

Statement Caching Limitations and Notes

Using a statement cache, even of size 1, will improve the performance of almost any SQLJ application. Be aware of the following, however:

- There is no benefit if each statement is executed only once.
- Try to avoid interleaving statements executed once with statements executed multiple times. The statements being executed only once would needlessly take up space in the statement cache, which becomes an issue when you reach the statement cache size limit. As an alternative, if you use ISO code generation you can use a separate connection context class for statements that are executed only once, and disable statement caching for that connection context class.
- Distinct statements with identical SQL operations are treated the same way as any distinct statements—each is processed and cached separately. As an alternative, put the SQL operation in a method, and call the method repeatedly, instead of using distinct statements.
- Be careful in choosing an appropriate statement cache size. If it is too small, then the cache might fill up, resulting in statements being flushed before they are re-executed. If it is too large, then database resources or program resources may be exhausted.

Also be aware of the following general notes regarding statement caching.

- With Oracle-specific code generation, using separate SQLJ connection context instances to have separate statement caching behavior will not work if the connection contexts share the same underlying JDBC connection instance. This is because under Oracle-specific code generation, SQLJ uses the JDBC statement cache.
- For Oracle applications, the statement cache size plus the maximum number of open JDBC statements in your application (both directly and through SQLJ) should total less than the maximum number of cursors available for a session, because the maximum number of cursors defines the maximum number of statements that can be open simultaneously.

- Using a statement cache generally does not change the execution semantics of an operation itself, although there are some scenarios where it does. For example, if you have a statement that throws an exception when its resources are released, then using a cache would mean that the exception would not be thrown until the connection is closed or the statement is flushed from the cache (which happens when the cache size is exceeded).

Update Batching

Update batching (referred to as *batch updates* in the Sun Microsystems JDBC 2.0 specification) allows `UPDATE`, `DELETE`, and `INSERT` statements that are batchable and compatible (as discussed below) to be collected into a batch and sent to the database for execution at once, saving round trips to the database. This feature is included in the JDBC 2.0 and SQLJ specifications, and is therefore supported by both Oracle9i JDBC and Oracle9i SQLJ. Update batching is typically used for an operation that is executed repeatedly within a loop.

In SQLJ, update batching is tied to execution context usage. This feature is enabled or disabled in each execution context, independently of any other execution context, and each execution context instance maintains its own batch.

This section covers the following topics:

- [Batchable and Compatible Statements](#)
- [Enabling and Disabling Update Batching](#)
- [Explicit and Implicit Batch Execution](#)
- [Canceling a Batch](#)
- [Execution Context Update Counts](#)
- [Setting a Batch Limit](#)
- [Batching Incompatible Statements](#)
- [Using Implicit Execution Contexts for Update Batching](#)
- [General Cautions Regarding Update Batching](#)
- [Error Conditions During Batch Execution](#)
- [Recursive Call-ins and Update Batching](#)

Important: Be aware of the following for update batching:

- You must use the default Oracle-specific code generation or, for ISO code generation, customize your application with the Oracle customizer.
 - It is highly advisable to disable auto-commit mode. This gives you control of what to commit and what to roll back in case of an error during batch execution.
-
-

Batchable and Compatible Statements

Two criteria determine whether a statement can be added to an existing batch of statements:

- Is it *batchable*? You cannot batch some kinds of statements under any circumstances.
- Is it *compatible* with statements in the existing batch?

Batchable For Oracle9i SQLJ, the following kinds of statements are batchable:

- UPDATE
- INSERT
- DELETE

Note the following restriction, however:

- UPDATE and INSERT statements with one or more stream host expressions are *not* batchable.

In future releases or other implementations of SQLJ, additional kinds of statements might be batchable (such as stored procedure calls or DDL statements).

Compatible In Oracle9i SQLJ, only multiple instances of the same statement are compatible. This can occur in one of two circumstances:

- A statement is executed repeatedly in a loop.
- A statement is executed in a method, and the method is called repeatedly.

In future releases or other implementations of SQLJ, additional kinds of statements might be compatible (such as instances of different statements that have no host expressions).

Enabling and Disabling Update Batching

SQLJ performs update batching separately for each execution context instance. Each one can have update batching enabled independently of your other execution context instances, and each maintains its own batch.

To enable or disable update batching for a particular execution context instance, use the `setBatching()` method of that execution context instance. This method takes boolean input, as follows:

```
...
ExecutionContext ec = new ExecutionContext();
ec.setBatching(true);
...
```

or:

```
...
ExecutionContext ec = new ExecutionContext();
ec.setBatching(false);
...
```

Update batching is disabled by default.

Note: The `setBatching()` method does not affect an existing statement batch. Neither enabling nor disabling update batching causes an existing batch to be executed or canceled.

Use the `isBatching()` method of an execution context instance to determine if update batching is enabled for that execution context:

```
ExecutionContext ec = new ExecutionContext();
...
boolean batchingOn = ec.isBatching();
```

This does not, however, indicate whether a batch is currently pending.

Explicit and Implicit Batch Execution

You can explicitly execute a pending update batch as desired, but it might also be implicitly executed under certain circumstances.

Note: It is important to be aware of what happens when an exception occurs in the middle of a batch execution. See "[Error Conditions During Batch Execution](#)" on page 10-22.

Explicit Use the `executeBatch()` method of the execution context instance to explicitly execute an update batch. This method returns an `int` array of update counts with meanings as described in "[Execution Context Update Counts](#)" on page 10-18.

Following is an example of explicitly executing a batch:

```
...
ExecutionContext ec = new ExecutionContext();
ec.setBatching(true);
...
double[] sals = ...;
String[] empnos = ...;
for (int i = 0; i < empnos.length; i++)
{
    #sql [ec] { UPDATE emp SET sal = :(sals[i]) WHERE empno = :(empnos[i]) };
}
int[] updateCounts = ec.executeBatch();
...
```

Notes: If you invoke `executeBatch()` when the execution context instance has no pending batch, then the method returns `null`.

Implicit When a pending update batch exists, it is implicitly executed in the following circumstances:

- An executable statement is encountered that is not batchable. In this case the existing batch is executed first, then the non-batchable statement is executed.
- An update statement is encountered that is batchable, but is not compatible with the statements in the existing batch (in other words, is not an instance of the same statement). In this case the batch is executed, then a new batch is created, starting with the incompatible statement.
- A predefined batch limit—that is, a specified number of statements—is reached. This is discussed in "[Setting a Batch Limit](#)" on page 10-19.

Following is an example. First one batch is created and executed implicitly when an unbatchable statement is encountered, then a new batch is created and executed implicitly when a batchable, but incompatible, statement is encountered:

```
ExecutionContext ec = new ExecutionContext();
ec.setBatching(true);
...
/* Statements in the following loop will be placed in a batch */
double[] sals = ...;
String[] empnos = ...;
for (int i = 0; i < empnos.length; i++)
{
    #sql [ec] { UPDATE emp SET sal = :(sals[i]) WHERE empno = :(empnos[i]) };
}

/* a SELECT is unbatchable so causes the batch to be executed */
double avg;
#sql [ec] { SELECT avg(sal) INTO :avg FROM emp };

/* Statements in the following loop will be placed in a new batch */
double[] comms = ...;
for (int i = 0; i < empnos.length; i++)
{
    #sql [ec] { UPDATE emp SET comm = :(comms[i]) WHERE empno = :(empnos[i]) };
}

/* the following update is incompatible with the second batch, so causes it to
be executed */
int smithdeptno = ...;
#sql [ec] { UPDATE emp SET deptno = :smithdeptno WHERE ename = 'Smith' };
```

To obtain the update count array for a batch executed implicitly, invoke the `getBatchUpdateCounts()` method of the execution context instance. This returns the update counts for the last batch to be executed successfully in this execution context instance. The following code statement could be inserted after the `SELECT` and after the last `UPDATE`:

```
int[] updateCounts = ec.getBatchUpdateCounts();
```

The meanings of these update counts are described in ["Execution Context Update Counts"](#) on page 10-18.

Note: If no update batch has been executed successfully for the execution context instance, then `getBatchUpdateCounts()` returns `null`.

Canceling a Batch

To cancel the batch that is pending in an execution context, use the `cancel()` method of the execution context instance. You can, for example, cancel a batch that has been executed, but not yet committed, in the event that an exception occurred during batch execution. Following is an example:

```
...
ExecutionContext ec = new ExecutionContext();
ec.setBatching(true);
...
double[] sals = ...;
String[] empnos = ...;
for (int i = 0; i < empnos.length; i++)
{
    #sql [ec] { UPDATE emp SET sal = :(sals[i]) WHERE empno = :(empnos[i]) };
    if (!check(sals[i], empnos[i])) //assume "check" is a user-supplied function
    {
        ec.cancel();
        throw new SQLException("Process canceled.");
    }
}

try
{
    int[] updateCounts = ec.executeBatch();
} catch ( SQLException exception) { ec.cancel(); }
...
```

When you cancel a batch, the next batchable statement will start a new batch.

Notes:

- Calling `cancel()` will also cancel any statement currently executing.
 - Canceling a batch does *not* disable update batching.
-
-

Execution Context Update Counts

In Oracle9i SQLJ, the array of update counts returned by the `executeBatch()` method or the `getBatchUpdateCounts()` method of an execution context instance does *not* contain counts of the number of rows updated by the batched statements, but simply values indicating whether each statement was successful. So its functionality differs from that of the single update count returned by the `getUpdateCount()` method of the execution context instance when batching is not enabled, which is described in ["Status Methods"](#) on page 7-28.

As statements are batched, and after batch execution, the single update count returned by `getUpdateCount()` is also affected.

Value Available from `getUpdateCount()` In a batch-enabled environment, the value available from the `getUpdateCount()` method of the execution context instance is modified after each statement is encountered. It will be updated with one of several `ExecutionContext` class static `int` constant values, as follows:

- `NEW_BATCH_COUNT`—Indicates that a new batch was created for the last statement encountered.
- `ADD_BATCH_COUNT`—Indicates that the last statement encountered was added to an existing batch.
- `EXEC_BATCH_COUNT`—Indicates that the pending batch was executed, either explicitly or implicitly, after the last statement was encountered.

If you refer to these constants, use the qualified names:

```
ExecutionContext.NEW_BATCH_COUNT  
ExecutionContext.ADD_BATCH_COUNT  
ExecutionContext.EXEC_BATCH_COUNT
```

Values Available from `executeBatch()` or `getBatchUpdateCounts()` After a batch has been executed, either explicitly or implicitly, the array of values returned by `executeBatch()` or `getBatchUpdateCounts()` indicates only whether the statements executed successfully. There is an array element for each batched statement. In accordance with the JDBC 2.0 specification, a value of -2 for an array element indicates that the corresponding statement completed successfully, but that the number of rows it affected is unknown.

Checking all the array values after execution of a batch would not be meaningful. As currently implemented, the only useful test of this array would be to verify the number of statements that were in the batch prior to execution, by checking the

number of elements in the array after a successful execution (essentially, after a batch execution that does not produce an exception).

Note that the update counts array is not modified as statements are batched, only as the batch is executed.

Setting a Batch Limit

You can specify that each update batch be executed after a predefined number of statements have been batched, before the next statement would be added. Use the `setBatchLimit()` method of the execution context instance, inputting a positive, nonzero integer as follows:

```
...
ExecutionContext ec = new ExecutionContext();
ec.setBatching(true);
ec.setBatchLimit(10);
...
double[] sals = ...;
String[] empnos = ...;
for (int i = 0; i < 20; i++)
{
    #sql [ec] { UPDATE emp1 SET sal = :(sals[i]) WHERE empno = :(empnos[i]) };
}
```

This loop is executed 20 times, with the statements being batched and the batch being executed during the 11th time through the loop, before an 11th statement would be added to the batch. Note that the batch would not be executed a second time in the loop, however. When your application exits the loop, the last ten statements would still be in the batch and would not be executed until another statement is encountered or you execute the batch explicitly.

You can use two special static `int` constants of the `ExecutionContext` class as input to the `setBatchLimit()` method:

- `AUTO_BATCH`—Allows the SQLJ runtime to determine the batch limit.
- `UNLIMITED_BATCH` (default)—Specifies that there is no batch limit.

For example:

```
...
ExecutionContext ec = new ExecutionContext();
ec.setBatching(true);
ec.setBatchLimit(ExecutionContext.AUTO_BATCH);
...
```

or:

```
ec.setBatchLimit(ExecutionContext.UNLIMITED_BATCH);  
...
```

To check the current batch limit, use the `getBatchLimit()` method of the execution context instance.

Batching Incompatible Statements

If you want to batch a statement that is incompatible with statements in an existing batch, without implicitly executing the existing batch, you will have to use a separate execution context instance. Following is an example:

```
...  
ExecutionContext ec1 = new ExecutionContext();  
ec1.setBatching(true);  
ExecutionContext ec2 = new ExecutionContext();  
ec2.setBatching(true);  
...  
double[] sals = ...;  
String[] empnos = ...;  
for (int i = 0; i < empnos.length; i++)  
{  
    #sql [ec1] { UPDATE emp1 SET sal = :(sals[i]) WHERE empno = :(empnos[i]) };  
    #sql [ec2] { UPDATE emp2 SET sal = :(sals[i]) WHERE empno = :(empnos[i]) };  
}  
int[] updateCounts1 = ec1.executeBatch();  
int[] updateCounts2 = ec2.executeBatch();  
...
```

Note: This example assumes that the two UPDATE statements are completely independent of each other. Do not batch interdependent statements in different execution contexts because you cannot completely assure the order in which they will be executed.

An alternative is to use a single execution context and separate loops so that all the EMP1 updates are batched and executed prior to the EMP2 updates:

```
...  
ExecutionContext ec = new ExecutionContext();  
ec.setBatching(true);  
...
```

```

double[] sals = ...;
String[] empnos = ...;
for (int i = 0; i < empnos.length; i++)
{
    #sql [ec] { UPDATE emp1 SET sal = :(sals[i]) WHERE empno = :(empnos[i]) };
}
for (int i = 0; i < empnos.length; i++)
{
    #sql [ec] { UPDATE emp2 SET sal = :(sals[i]) WHERE empno = :(empnos[i]) };
}
ec.executeBatch();
...

```

This example executes the first batch implicitly and the second batch explicitly.

Using Implicit Execution Contexts for Update Batching

All the update batching examples so far have created and specified explicit execution context instances. This is not necessary, however, given that every connection context instance has an implicit execution context instance. For example, you can access the implicit execution context instance of the default connection as follows:

```

DefaultContext.getDefaultContext().getExecutionContext().setBatching(true);
...
double[] sals = ...;
String[] empnos = ...;
for (int i = 0; i < empnos.length; i++)
{
    #sql { UPDATE emp SET sal = :(sals[i]) WHERE empno = :(empnos[i]) };
}
// implicitly execute the batch and commit
#sql { COMMIT };

```

or, alternatively, you could execute the batch explicitly:

```

DefaultContext.getDefaultContext().getExecutionContext().executeBatch();

```

General Cautions Regarding Update Batching

If you use update batching, especially if you mix statements using an unbatched execution context instance with statements using a batched execution context instance, remember the following points:

- If an unbatched statement depends on a batched statement, be sure the batch is executed prior to the unbatched statement.
- A JDBC `COMMIT` or `ROLLBACK` operation—that is, an auto-commit or any explicit use of the `commit()` method or `rollback()` method of a JDBC `Connection` instance—does not execute pending statements in a batch.

It is important to note, however, that using a SQLJ `COMMIT` or `ROLLBACK` statement, such as follows, *will* execute pending statements in a batch:

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

or:

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

This is another reason that you should always commit or roll back changes using `#sql` syntax, which cleans up both SQLJ resources and JDBC resources.

- When a batch is implicitly executed as a result of an unbatchable or incompatible statement being encountered, the batch is executed *before* the unbatchable/incompatible statement is executed, but *after* the input parameters of that statement have been evaluated and passed to the statement.
- If you no longer intend to use a particular batch-enabled execution context instance, then explicitly execute or cancel its pending batch to free resources.

Error Conditions During Batch Execution

In the event that a statement causes an exception in the middle of a batch execution, be aware of the following:

- Batched statements following the statement that caused the exception are *not* executed.
- Batched statements that had already been executed prior to the exception are *not* rolled back.
- If the batch where the exception occurred was executed implicitly as the result of another (unbatchable or incompatible) statement being encountered, that statement is *not* executed.

Note: Presumably you have disabled auto-commit mode when using update batching. This gives you commit/rollback control in case of an error during batch execution.

When an exception occurs during batch execution under JDBC 2.0 or higher, it is typically an instance of the standard `java.sql.BatchUpdateException` class, a subclass of the `java.sql.SQLException` class. (Under JDK 1.1.x, the Oracle JDBC drivers support update-batching exceptions with the `oracle.jdbc2.BatchUpdateException` class.)

The `BatchUpdateException` class has a `getUpdateCounts()` method that, for batched statements successfully executed before the exception occurred, returns an array of update counts equivalent to what would be returned by the `ExecutionContext` class `executeBatch()` or `getBatchUpdateCounts()` method.

Recursive Call-ins and Update Batching

As discussed in "[Recursive SQLJ Calls in the Server](#)" on page 11-26, execution of SQLJ stored procedures where one calls the other can result in situations where the two procedures are simultaneously using the same execution context instance. The update-batching flag (set using the `setBatching()` method of the execution context instance) would behave in the same way as other execution context attributes—regardless of which stored procedure sets it, it would affect the next executable statement in either stored procedure.

For this reason, update batching is automatically disabled in the server whenever a recursive call-in occurs. The pending batch is executed, and no batching occurs in the recursively invoked procedure.

To avoid this behavior, use explicit execution context instances in batch-enabled stored procedures.

Column Definitions

Oracle SQLJ reflects Oracle JDBC support for column type and size definitions. Depending on the driver implementation, which differs somewhat among the different Oracle JDBC drivers, registering column types and sizes can save a trip to the database for each query. In particular, this is true for the Oracle JDBC Thin driver and use of positional iterators.

Oracle SQLJ Implementation of Column Definitions

If you enable column definitions, Oracle SQLJ takes the following steps to automatically register column types and sizes:

- During customization, or during translation when the default Oracle-specific code generation is used (`-codegen=oracle`), Oracle SQLJ connects to a specified database schema to determine types and sizes of columns being retrieved. With ISO standard SQLJ code generation, the column defaults become part of the SQLJ profile. This can be accomplished during the customization step of source code translation, or during separate customization of an existing profile.
- When your application executes, the SQLJ runtime will use the column information to register the column types and sizes with the JDBC driver, using a call to the `defineColumnType()` method available in the Oracle JDBC statement classes.

Customizer and Translator Options for Column Definitions

To enable column definitions, set SQLJ options as follows:

- Enable the `optcols` flag. For Oracle-specific code generation, use the SQLJ translator `-optcols` option. For ISO standard code generation, use either the translator option or the Oracle customizer option (`-P-Coptcols` on the SQLJ command line).
- Set the user, password, and URL for a database connection. For Oracle-specific code generation, this is through the SQLJ translator `-user`, `-password`, and `-url` options. For ISO standard code generation, this can be through the translator options or you can separately use the customizer options (`-P-user`, `-P-password`, and `-P-url` on the SQLJ command line). In addition, set the JDBC driver class (`-P-driver` on the SQLJ command line) if you are not using the default `OracleDriver` class.

For information about the translator options, see "[Column Definitions \(-optcols\)](#)" on page 8-53 and "[Connection Options](#)" on page 8-34.

For information about the customizer options, see the `optcols` section under "[Overview of Customizer-Specific Options](#)" on page A-23, and the `user`, `password`, `url`, and `driver` sections under "[Overview of Customizer Harness Options](#)" on page A-12.

Parameter Size Definitions

Oracle JDBC and Oracle SQLJ allow you to optimize JDBC resource allocation by defining parameter sizes—sizes of Java host variables used as any of the following:

- input or output parameters in stored procedure or function calls
- return values from stored function calls
- input or output parameters in SET statements
- input or output parameters in PL/SQL blocks

Oracle SQLJ Implementation of Parameter Size Definitions

Oracle SQLJ implements parameter size definitions through option settings, in combination with "hints" embedded in source code comments. For ISO standard SQLJ code generation, Oracle customizer options are available. For the default Oracle-specific code generation, equivalent SQLJ translator options are available.

Use options and hints as follows:

- Enable parameter size definitions through the SQLJ translator or Oracle customizer parameter definition flag.
- Specify default sizes for particular datatypes through the SQLJ translator or Oracle customizer parameter default size option.
- Override datatype default sizes for particular parameters by embedding hints in source code comments, following a prescribed format.

For any given host variable, when parameter size definitions are enabled, resources are allocated according to the source code hint if there is one. If there is no source code hint, then the default size for the corresponding datatype is used if one was specified. If there is no source code hint or appropriate default size, then maximum resources are allocated according to the JDBC implementation.

When your application executes, the parameter sizes are registered through calls to the `defineParameterType()` and `registerOutParameter()` methods available in the Oracle JDBC statement classes.

Note: If you do not enable the parameter definition flag, then parameter size defaults and source code hints will be ignored, and maximum or default resources will be allocated according to the JDBC implementation.

Customizer and Translator Options for Parameter Size Definitions

Use the following SQLJ options for parameter size definitions:

- Use the `optparams` flag to enable parameter size definitions. For Oracle-specific code generation, use the SQLJ translator `-optparams` option. For ISO standard code generation, use either the translator option or the Oracle customizer option (`-P-Coptparams` on the SQLJ command line).
- Use `optparamdefaults` to set default sizes for particular datatypes. For Oracle-specific code generation, use the SQLJ translator `-optparamdefaults=xxxx` option. For ISO standard code generation, use either the translator option or the Oracle customizer option, `-P-Coptparamdefaults=xxxx` on the SQLJ command line.

For information about the translator options, see "[Parameter Definitions \(-optparams\)](#)" on page 8-55 and "[Parameter Default Size \(-optparamdefaults\)](#)" on page 8-56.

For information about the customizer options, see the appropriate sections under "[Overview of Customizer-Specific Options](#)" on page A-23.

Source Code Hints for Parameter Size Definitions

Embed source code hints for parameter size definitions within your SQLJ statements in the following format (you can add white space within the comment, as desired):

```
/*(size)*/
```

The size is in bytes. Hints are ignored if the `optparams` flag is disabled.

You can override the default parameter size, without specifying a new size (leaving size allocation to the JDBC implementation), as follows:

```
/*()*/
```

Here is an example:

```
byte[] hash;  
String name=Tyrone;  
String street=2020 Meryl Street;  
String city=Wichita;  
String state=Kansas;  
String zipcode=77777;  
#sql hash = { /* (5) */ VALUES (ADDR_HASH(:name /* (20) */ , :street /* () */ ,  
                                :city, :state, :INOUT zipcode /* (10) */ ) };
```


A hint for a result expression, such as the result expression `hash` in the example, must be the first item appearing inside the brackets of the SQLJ statement, as shown. Hints for input and output host variables must immediately follow the variables, as shown.

The example sets parameter sizes as follows:

- `hash`—5 bytes
- `name`—20 bytes
- `street`—override default, but with no setting (leave allocation up to JDBC)
- `city`—none (use appropriate datatype default, if any)
- `state`—none (use appropriate datatype default, if any)
- `zipcode`—10 bytes

Note: If any parameter size is altered such that its actual size exceeds its registered size at runtime, a SQL exception will be thrown.

SQLJ Debugging Features

This section summarizes debugging features in Oracle SQLJ and refers to other parts of this manual where these features are described in more detail.

- If you run SQLJ from the command line, the `-linemap` option (or `-jdblinemap` option if you are using the `jdb` debugger) 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.
- If you use ISO standard code generation, you can use the specialized customizer `AuditorInstaller` to install "auditors" for use in debugging.
- SQLJ is integrated into the Oracle9i JDeveloper integrated development environment, allowing access to JDeveloper debugging facilities.

SQLJ -linemap Flag for Debugging

The `-linemap` flag instructs SQLJ to map line numbers from a SQLJ source code file to locations in the corresponding `.class` file. This will be the `.class` file created during compilation of the `.java` file generated by the SQLJ translator. As a result of this, when Java runtime errors occur, the line number reported by the Java virtual machine (JVM) is the line number in the SQLJ source code, making it much easier to debug.

If you are using the Sun Microsystems `jdb` debugger, then use the `-jdblinemap` option instead of the `-linemap` option. The options are equivalent, except that `-jdblinemap` does some special processing, necessitated by the fact that `jdb` does not support Java source files with file name extensions other than the `.java` extension.

For more information, see "[Line-Mapping to SQLJ Source File \(-linemap\)](#)" on page 8-49 and "[Line-Mapping to SQLJ Source File for jdb Debugger \(-jdblinemap\)](#)" on page 8-50.

Note: If you are translating in the server, class schema objects created during server-side translation automatically reference line numbers that map to the SQLJ source code. This is equivalent to enabling the `-linemap` option when you translate on a client.

Server-Side debug Option

If you are loading SQLJ source into the server and using the server-side embedded translator to translate it, the server-side `debug` option instructs the server-side compiler to output debugging information when a `.sqlj` or `.java` source file is compiled in the server. This is equivalent to using the `-g` option when running the standard `javac` compiler on a client. This does not aid in debugging your SQLJ code in particular, but aids in debugging your Java code in general.

See "[Option Support in the Server Embedded Translator](#)" on page 11-18 for more information about this option and information about how to set options in the server.

For general information about debugging in the Oracle JVM, see the *Oracle9i Java Developer's Guide*.

Introduction to the AuditorInstaller Specialized Customizer

For ISO code generation, SQLJ provides a special customizer, `AuditorInstaller`. This customizer will insert sets of debugging statements, known as *auditors*, into profiles specified on the SQLJ command line. These profiles must already exist from previous customization.

The debugging statements will execute during SQLJ runtime (when someone runs your application), displaying a trace of method calls and values returned.

Use the customizer harness `debug` option, preceded by `-P-` as with any general customization option, to insert the debugging statements.

`AuditorInstaller` is described in detail in "[AuditorInstaller Customizer for Debugging](#)" on page A-44.

Introduction to Developing and Debugging in Oracle9i JDeveloper

Oracle SQLJ is fully integrated into the Oracle9i JDeveloper visual programming tool.

JDeveloper also includes an integrated debugger that supports SQLJ. SQLJ statements, as with standard Java statements, can be debugged in-line as your application executes. Reported line numbers are according to the line numbers in your SQLJ source code (as opposed to in the generated Java code).

See "[SQLJ in Oracle9i JDeveloper and Other IDEs](#)" on page 1-32 for an introduction to JDeveloper.

SQLJ in the Server

SQLJ applications can be stored and run directly in the Oracle9i server. You have the option of either translating and compiling them on a client and loading the generated classes and resources into the server, or loading SQLJ source code into the server and having it translated and compiled by the embedded translator of the server.

This chapter discusses features and usage of SQLJ in the server, including additional considerations such as multithreading and recursive SQLJ calls.

The following topics are discussed:

- [Introduction to Server-Side SQLJ](#)
- [Creating SQLJ Code for Use in the Server](#)
- [Translating SQLJ Source on a Client and Loading Components](#)
- [Loading SQLJ Source and Translating in the Server](#)
- [Dropping Java Schema Objects](#)
- [Additional Considerations](#)

Introduction to Server-Side SQLJ

SQLJ code, as with any Java code, can run in Oracle9i in stored procedures, stored functions, or triggers. Data access is through a server-side implementation of the SQLJ runtime (with all SQLJ runtime packages automatically available) in combination with the Oracle JDBC server-side internal driver. (You will sometimes hear this referred to as the "KPRB driver".)

In addition, an embedded SQLJ translator in Oracle9i is available to translate SQLJ source files directly in the server.

Considerations for running SQLJ in the server include several server-side coding issues as well as decisions about where to translate your code and how to load it into the server. You must also be aware of how the server determines the names of generated output. You can either translate and compile on a client and load the class and resource files into the server, or you can load `.sqlj` source files into the server and have the files automatically translated by the embedded SQLJ translator.

The embedded translator has a different user interface than the client-side translator. Supported options can be specified using a database table, and error output is to a database table. Output files from the translator are transparent to the developer.

Notes:

- In Oracle9i release 2, the server uses a JDK 1.3 J2SE environment. The server-side SQLJ environment is roughly equivalent to a client-side environment using the `runtime12ee` library, except for SQLJ-specific connection bean support and considering any relevant exceptions noted in ["Creating SQLJ Code for Use in the Server"](#) on page 11-4.
 - This manual presumes that system configuration issues are outside the duties of most SQLJ developers. Therefore, configuration of the Oracle9i Java virtual machine (JVM) is not covered here. For information about setting Java-related configuration parameters (such as `JAVA_POOL_SIZE`), see the *Oracle9i Java Developer's Guide*. If you need information about configuring the multithreaded server, dispatcher, or listener, see the *Oracle9i Net Services Administrator's Guide*.
-
-

Note Regarding Desupport of J2EE in the Oracle9i Database:

With the introduction of Oracle9i Application Server Containers for J2EE (OC4J)—a new, lighter-weight, easier-to-use, faster, and certified J2EE container—Oracle will desupport the Java 2 Enterprise Edition (J2EE) and CORBA stacks from the database, starting with Oracle9i database release 2. However, the database-embedded Java VM (Oracle JVM) will still be present and will continue to be enhanced to offer Java 2 Standard Edition (J2SE) features, Java stored procedures, JDBC, and SQLJ in the database. As of Oracle9i database release 2 (9.2.0), Oracle will no longer support the following technologies in the database:

- the J2EE stack, consisting of:
 - Enterprise Beans (EJB) container
 - JavaServer Pages (JSP) container
 - Oracle9i Servlet Engine (OSE)
- the embedded Common Object Request Broker Architecture (CORBA) framework, based on Visibroker for Java

Customers will no longer be able to deploy servlets, JSP pages, EJBs, and CORBA objects in Oracle databases. Oracle9i database release 1 (9.0.1) is the last database release to support the J2EE and CORBA stack. Oracle is encouraging customers to migrate existing J2EE applications running in the database to OC4J.

Creating SQLJ Code for Use in the Server

With few exceptions, writing SQLJ code for use within the target Oracle9i database is identical to writing SQLJ code for client-side use. The few differences are due to Oracle JDBC characteristics or general Java characteristics in the server, rather than being specific to SQLJ. There are a few considerations to be aware of, however:

- There is an implicit connection to the server itself.
- There are coding issues such as lack of auto-commit functionality.
- In the server, the default output device is the current trace file.
- Name resolution functions differently in the server than on a client.
- SQL names must be interpreted and processed differently from Java names.
- There is no JSP, EJB, or CORBA functionality in the server. Because there is no JSP container, you cannot use the SQLJ JSP connection beans in server-side code.

Note: Writing SQLJ code to connect from one server to another through the server-side Thin driver is identical to writing code for an application that uses a client-side Thin driver. The points in this discussion do not apply.

Database Connections within the Server

The concept of connecting to a server is different when your SQLJ code is running within this server itself—there is no explicit database connection. By default, an implicit channel to the database 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 instance, specify a default connection context, specify any connection objects for any of your `#sql` statements, or close the connection.

Note: In the server, setting the default connection context to `null`, as follows, will reinstall the default connection context (the implicit connection to the server):

```
DefaultContext.setDefaultContext(null);
```

Coding Issues within the Server

There are a few coding issues to consider when your code will run within the target server using the server-side internal driver. Note the following:

- Result sets issued by the internal driver persist across calls, and their finalizers do not release their cursors. Because of this, it is especially important to close all iterators to avoid running out of available cursors, unless you have a particular reason for keeping an iterator open (such as when it is actually used across calls).
- The internal driver does not support auto-commit functionality—the auto-commit setting is ignored within the server. Use explicit `COMMIT` or `ROLLBACK` statements to implement or cancel your data updates:

```
#sql { COMMIT };
...
#sql { ROLLBACK };
```

Note: If you are using any kind of XA transactions, such as Java Transaction Service (JTS) transactions, you cannot use SQLJ or JDBC `COMMIT/ROLLBACK` statements or methods.

- For ISO standard code generation, if you use SQLJ code that interacts with JDBC code, and you use a nondefault connection context instance, then you must eventually close the connection context instance in order to clean up statements cached there (unless you use the same connection context instance for the duration of your session). Following is an example:

```
DefaultContext ctx = new DefaultContext(conn); // conn is JDBC connection
#sql [ctx] { SQL operation };
...
ctx.close(sqlj.runtime.ConnectionContext.KEEP_CONNECTION);
...
```

If you do not close the connection context instance, you are likely to run out of statement handles in your session. Also be aware that simply closing the underlying JDBC connection object does *not* reclaim statement handles, which differs from the behavior when the application executes on a client.

(For the default Oracle-specific code generation, statements are cached in the underlying JDBC statement cache and can be automatically reclaimed.)

- With Oracle-specific code generation for code that will run in the server, use an explicit `ExecutionContext` instance. This ensures that your application can fully interoperate with applications translated with ISO standard SQLJ code generation.

If you use one thread per connection (which translates to one thread per Oracle session), it is sufficient to use one static instance, as in the following example:

```
public static ExecutionContext ec = new ExecutionContext();
...
#sql [ec] { SQL operation };    // use ec for all operations
```

If you use multiple threads per connection, you must use a separate execution context instance for each method invocation.

For more information about server-side JDBC and the server-side internal and Thin drivers, see the *Oracle9i JDBC Developer's Guide and Reference*.

Default Output Device in the Server

The default standard output device in the Oracle Java virtual machine (JVM) is the current trace file.

If you want to reroute all standard output from a program executing in the server—output from any `System.out.println()` calls, for example—to a user screen, you can execute the `SET_OUTPUT()` procedure of the `DBMS_JAVA` package as in the following example. Input the buffer size in bytes (10,000 bytes in this case).

```
sqlplus> execute dbms_java.set_output(10000);
```

Output exceeding the buffer size will be lost.

If you want your code executing in the server to expressly 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 *Oracle9i Supplied PL/SQL Packages and Types Reference*.

Name Resolution in the Server

Class loading and name resolution in the server follow a very different paradigm than on a client, because the environments themselves are very different. This section gives only an overview; the topic is discussed in detail in the *Oracle9i Java Developer's Guide*.

Java name resolution in the Oracle JVM includes the following:

- class *resolver specs*, which are schema lists to search in resolving a class schema object (functionally equivalent to the classpath on a client)
- the *resolver*, which maintains mappings between class schema objects that reference each other in the server

A class schema object is said to be resolved when all of its external references to Java names are bound. In general, all the classes of a Java program should be compiled or loaded before they can be resolved. This is because Java programs are typically written in multiple source files that can reference each other recursively.

When all the class schema objects of a Java program in the server are resolved and none of them have been modified since being resolved, the program is effectively 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. For ISO standard code (which has to be translated on the client), be careful how you load any resources into resource schema objects in the server, as discussed in "[Naming of Loaded Class and Resource Schema Objects](#)" on page 11-11. If you enabled the SQLJ `-ser2class` flag for your client-side translation, then your SQLJ profiles will be in class files, and you will typically not have any resource files. If you did not enable `-ser2class`, then your profiles will be in `.ser` resource files.

SQL Names Versus Java Names

SQL names—such as names of source, class, and resource schema objects—are not global in the way that Java names are global. The Java Language Specification 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 processed differently than Java names. SQL names are relative names and are interpreted from the point of view of the schema where a program is executed. This is central to how the program binds local data stored at that schema. Java names are global names, and the classes that they designate can be loaded at any execution site, with reasonable expectation that those classes will be classes that were used to compile the program.

Translating SQLJ Source on a Client and Loading Components

One approach to deploying SQLJ code for the Oracle9i server is to run the SQLJ translator on a client machine to take care of translation, compilation, and profile customization (if applicable). Then load the resulting class and resource files (if any) into the server, typically using a Java archive (.jar) file. In fact, this is the only way to use ISO standard code in the server, because the server-side translator supports only Oracle-specific code generation.

If you are developing your source on a client 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.

For ISO standard code (`-codegen=iso`), it might also be advisable to use the SQLJ `-ser2class` option during translation when you intend to load an application into the server. This results in SQLJ profiles being converted from `.ser` serialized resource files to `.class` files and simplifies their naming. 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 8-68.

When you load `.class` files and `.ser` resource files into Oracle9i, either directly or using a `.jar` file, the resulting library units are referred to as Java *class schema objects* (for Java classes) and Java *resource schema objects* (for Java resources). Your SQLJ profiles (if any) will be in resource schema objects if you load them as `.ser` files, or in class schema objects if you enabled `-ser2class` during translation and load them as `.class` files.

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. This utility is discussed in detail in the *Oracle9i Java Developer's Guide*.

Either specify the class and resource files (if any) individually on the `loadjava` command line, or put them into a `.jar` file and specify the `.jar` file on the command line. A separate schema object is created for each `.class` or `.ser` file in the `.jar` file or on the command line.

Consider an example where you do the following:

1. Translate and compile `Foo.sqlj`, which includes an iterator declaration for `MyIter`, using ISO standard code generation.
2. Enable the `-ser2class` option when you translate `Foo.sqlj`.
3. Archive the resulting files (`Foo.class`, `MyIter.class`, `Foo_SJProfileKeys.class`, and `Foo_SJProfile0.class`) into `Foo.jar`.

Then run `loadjava` with the following command line (plus any options you want to specify). This examples uses the default OCI driver:

```
loadjava -user scott/tiger Foo.jar
```

or, alternatively, to use the original files:

```
loadjava -user scott/tiger Foo.class MyIter.class Foo_SJProfileKeys.class Foo_SJProfile0.class
```

or:

```
loadjava -user scott/tiger Foo*.class MyIter.class
```

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
```

For information about files generated by the SQLJ translator, see "[Code Generation](#)" on page 9-5 and "[Java Compilation](#)" on page 9-9.

Notes:

- When you use the `-codegen=iso` setting during translation, generating profile files, and then load these profiles into the server as `.ser` files, they are first customized if they were not already customized on the client. If they were already customized, then they are loaded as is.
 - You can access the `USER_OBJECTS` view in your schema to verify that your classes and resources are loaded properly. This is discussed in the *Oracle9i Java Developer's Guide*.
-
-

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 *Oracle9i SQL Reference* for more information about the `CREATE JAVA` commands.

Naming of Loaded Class and Resource Schema Objects

This section discusses how schema objects for classes and profiles are named when you load classes and profiles into the server. Remember, however, that profiles are created only for ISO standard code generation (`-codegen=iso`).

For ISO standard code generation, if the SQLJ `-ser2class` option was enabled when you translated your application on the client, then profiles were converted to `.class` files and will be loaded into class schema objects in the server. If `-ser2class` was not enabled, then profiles were generated as `.ser` serialized resource files and will be loaded into resource schema objects in the server.

In the following discussion, it is assumed that you use only the default connection context class for any application that will run in the server; therefore, there will be only one profile.

Full Names and Short Names

There are two forms of schema object names in the server: *full names* and *short names*.

Full names are fully qualified and are used as the schema object names whenever possible. If any full name is longer than 31 characters, however, or contains characters that are illegal or cannot be converted to characters in the database character set, then Oracle9i 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.

For more information about these and about other file naming considerations, including `DBMS_JAVA` procedures to retrieve a full name from a short name, and vice versa, see the *Oracle9i Java Developer's Guide*.

Full Names of Loaded Classes

Loaded classes will include profile files if you use ISO standard code generation (`-codegen=iso`) and enable the `-ser2class` flag.

The full name of the class schema object produced when you load a `.class` file into the server is determined by the package and class name in the original source code. Any path information you supply on the command line (so that `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 as follows (unless it is a nested class, in which case it will not have its own schema object):

```
x/y/MyIter
```

Furthermore, if you are using ISO standard code generation:

- The related profile-keys class file, generated by SQLJ when you translate `Foo.sqlj`, is `Foo_SJProfileKeys.class`. Therefore, the full name of its class schema object is:

```
x/y/Foo_SJProfileKeys
```

- If the `-ser2class` option was enabled when you translated your application, then the resulting profile was generated in file `Foo_SJProfile0.class`. Therefore, the full name of the class schema object is:

```
x/y/Foo_SJProfile0
```

Full Names of Loaded Resources (including profiles if `-ser2class` not enabled)

This discussion is relevant only if you are using ISO standard code generation (`-codegen=iso`) and did not enable the `-ser2class` option when you translated your application, or if you use other Java serialized resource (`.ser`) files in your application.

The naming of resource schema objects is handled differently from class schema objects—their names are not determined from the contents of the resources. Instead,

their full names are identical to the names that appear in a `.jar` file or on the `loadjava` command line, including path information. Note also 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 identical to the relative path and file name that Java would use to look it up on a client.

In the case of a SQLJ profile, this is a subdirectory under the directory specified by the translator `-d` option, according to the package name. If the `-d` option, used to specify the top-level output directory for generated `.class` and `.ser` files, is set to `/mydir` and the application is in package `abc.def`, then `.class` and `.ser` files generated during translation will be placed in the `/mydir/abc/def` directory. For more information about the SQLJ `-d` option, including the default value, see "[Output Directory for Generated .ser and .class Files \(-d\)](#)" on page 8-30.

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 -cvf myjar.jar abc/def/*.class abc/def/*.ser
loadjava <...options...> myjar.jar
```

If your application is `App` and your profile is `App_SJProfile0.ser`, then either of the above examples will correctly result in the following full name of the created resource schema object:

```
abc/def/App_SJProfile0.ser
```

Note that `".ser"` is retained.

Note also that if you set `-d` to a directory whose hierarchy has no other contents (which is advisable), you can simply run `jar` as follows to recursively get your application components:

```
cd /mydir
jar -cvf myjar.jar *
loadjava <...options...> myjar.jar
```

Publishing the Application After Loading Class and Resource Files

Before using your SQLJ code in the server, you must publish the top-level methods, as is true of any Java code you use in the server. Publishing includes writing call descriptors, mapping datatypes, and setting parameter modes. For information, see the *Oracle9i Java Stored Procedures Developer's Guide*.

Summary: Running a Client Application in the Server

This section summarizes the typical steps of running a client application in the server. As an example, it uses a demo application called `NamedIterDemo`.

1. Create a `.jar` file for your application components. For `NamedIterDemo`, the components include `SalesRec.class` as well as the application class and profile (if any).

You can create a `.jar` file `niter-server.jar` as follows:

```
jar cvf niter-server.jar Named*.class Named*.ser SalesRec.class connect.properties
```

But remember that `.ser` files are only relevant for ISO standard code generation.

2. Load the `.jar` file into the server.

Use `loadjava` as follows. This example instructs `loadjava` to use the OCI driver in loading the files. The `-resolve` option results in the class files being resolved.

```
loadjava -oci -resolve -force -user scott/tiger niter-server.jar
```

3. Create a SQL wrapper in the server for your application.

For example, run a SQL*Plus script that executes the following:

```
set echo on
set serveroutput on
set termout on
set flush on

execute dbms_java.set_output(10000);

create or replace procedure SQLJ_NAMED_ITER_DEMO as language java
name 'NamedIterDemo.main (java.lang.String[])';
/
```

The `DBMS_JAVA.SET_OUTPUT()` routine reroutes default output to your screen, instead of to a trace file; the input parameter is the buffer size in bytes.

4. Execute the wrapper.

For example:

```
sqlplus> call SQLJ_NAMED_ITER_DEMO();
```

Loading SQLJ Source and Translating in the Server

Another approach to developing SQLJ code for the server is loading the source code into the server and translating it directly in the server. This employs the embedded SQLJ translator in the Oracle JVM. This discussion still assumes you created the source on a client machine.

Note: As of Oracle9i release 2, the server-side SQLJ translator does not support ISO standard code generation (the `-codegen=iso` setting). If you want to use such code in the server, you must translate on a client and load the individual class files and resources into the server, as described in "[Translating SQLJ Source on a Client and Loading Components](#)" on page 11-9.

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 Oracle9i, either directly or using a `.jar` file, the resulting library units containing the source code are referred to as Java *source schema objects*. A separate schema object is created for each source file.

When translation and compilation take place, the resulting library units for the generated classes are referred to as Java *class schema objects*, just as they are when loaded directly into the server from `.class` files created on a client. A separate schema object is created for each class.

Resource schema objects are used for properties files that you load into the server.

Loading SQLJ Source Code into the Server

Use the Oracle `loadjava` client-side utility on a `.sqlj` file (instead of on a `.class` file) to load source into the server. This utility is discussed in detail in the *Oracle9i Java Developer's Guide*.

If you enable the `loadjava -resolve` option in loading a `.sqlj` file, then the server-side embedded translator is run to perform the translation and compilation of your application as it is loaded. Otherwise, the source is loaded into a source schema object without any translation. In this case, however, the source *is* implicitly translated and compiled the first time an attempt is made to use a class defined in the source. Such implicit translation might seem surprising at first, because there is nothing comparable in client-side SQLJ.

For example, run `loadjava` as follows:

```
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 being created in addition to the source schema object. For information, see "[Naming of Loaded Source and Generated Class and Resource Schema Objects](#)" on page 11-22.

Before running `loadjava`, however, you must set SQLJ options appropriately. For more information, see "[Option Support in the Server Embedded Translator](#)" on page 11-18. Note that encoding can be set on the `loadjava` command line, instead of through the server-side SQLJ encoding option, as follows:

```
loadjava -user scott/tiger -resolve -encoding SJIS Foo.sqlj
```

The `loadjava` script, which runs the actual utility, is in the `bin` subdirectory under your `[Oracle_Home]` directory. This directory should already be in your path once Oracle has been installed.

Notes:

- In processing a `.jar` file, `loadjava` first processes `.sqlj`, `.java`, and `.class` files. It then makes a second pass and processes everything else as Java resource files.
 - You cannot load a `.sqlj` file along with `.class` files that were generated from processing of the same `.sqlj` file. This would create an obvious conflict, because the server would be trying to load the same classes that it would also be trying to generate.
 - You can put multiple `.sqlj` files into a `.jar` file and specify the `.jar` file to `loadjava`.
 - You can access the `USER_OBJECTS` view in your schema to verify that your classes are loaded properly. This is discussed in the *Oracle9i Java Developer's Guide*.
-
-

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 and compiled at that time, creating class schema objects as appropriate in addition to the source schema object. Otherwise, it is not translated and compiled—in this case only the source schema object is created. In this latter case, however, the source *is* implicitly translated and compiled the first time an attempt is made to use a class contained in the source.

See the *Oracle9i SQL Reference* for more information about the `CREATE JAVA` commands.

Note: When you first load a source file, some checking of the source code is performed, such as determining what classes are defined. If any errors are detected at this time, the load fails.

Option Support in the Server Embedded Translator

The following options are available in the server-side SQLJ translator:

- `encoding`
- `online`
- `debug`

This section discusses these options, after leading off with some discussion of fixed settings in server-side SQLJ.

There is also discussion of the `loadjava` utility and its `-resolve` option. For more information, see the *Oracle9i Java Developer's Guide*.

Fixed Settings in the Server-Side SQLJ Translator

The following settings, supported by SQLJ translator options on a client, are fixed in the server-side translator:

- Both online semantics-checking and offline parsing are enabled in the server by default, equivalent to the default `-parse=both` setting on a client. You can override this to disable online semantics-checking through the `online` option,

but cannot disable offline parsing. See "[Online Semantics-Checking Versus Offline Parsing](#)" on page 8-73 for information about these features.

- Oracle-specific code generation is used in the server, equivalent to the default `-codegen=oracle` setting on a client. This is a fixed setting. See "[Oracle-Specific Code Generation \(No Profiles\)](#)" on page 4-39 for information about this feature.
- Class schema objects created during server-side translation reference line numbers that map to the SQLJ source code. This is equivalent to enabling the `-linemap` option when you translate on a client. For a discussion of this option, see "[Line-Mapping to SQLJ Source File \(-linemap\)](#)" on page 8-49.

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 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 11-16. Any `loadjava` command-line setting for encoding overrides this `encoding` option.

See "[Encoding for Input and Output Source Files \(-encoding\)](#)" on page 8-29 for general information about this option.

Note: If no encoding is specified, either through this option or through `loadjava`, then encoding is performed according to the `file.encoding` setting of the client from which you run `loadjava`.

The online Option

A `true` setting for the `online` option (the default value) enables online semantics-checking. Semantics-checking is performed relative to the schema in which the source is loaded. You do not specify an exemplar schema, as you do for online-checking on a client.

If the `online` option is set to `false`, offline checking is performed.

In either case, the default checker is `oracle.sqlj.checker.OracleChecker`, which will choose an appropriate checker according to your JDBC driver version and Oracle version. For information about `OracleChecker`, see

"Semantics-Checkers and the OracleChecker Front End (default checker)" on page 8-71.

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. This will occur immediately if you load it with the `loadjava -resolve` option enabled—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()`

See the *Oracle9i Supplied Java Packages Reference* for more information about the `dbms_java` package and these functions.

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');
```

Server-Side Option Notes Be aware of the following:

- The `set_compiler_option()` parameter for package and source names uses dotted names (such as `abc.def` as a package name) even though schema object names use slash syntax (such as `abc/def` as a package name).
- When you specify a package name, be aware that the option will apply to any included packages as well. A setting of `a.b.MyPackage` sets the option for any source schema objects whose names are of the following form:

```
a/b/MyPackage/subpackage/...
```

- Specifying `' '` (empty set of single-quotes) as a package name makes the option apply to the root and all subpackages, effectively making it apply to all packages in your schema.

Naming of Loaded Source and Generated Class and Resource Schema Objects

When you use the server-side SQLJ translator, such as when you use `loadjava` on a `.sqlj` file with the `-resolve` option enabled, the output generated by the server-side translator is essentially identical to what would be generated on a client—a compiled class for each class you defined in the source and a compiled class for each iterator and connection context class.

As a result, the following schema objects will be produced when you load a `.sqlj` file into the server with `loadjava` and have it translated and compiled:

- a source schema object for the original source code
- a class schema object for each class you defined in the source
- a class schema object for each iterator or connection context class you declared in the source

But presumably you will not need to declare connection context classes in code that will run in the server, unless it is to specify type maps for user-defined types. (See ["Requirements for Classes Implementing SQLData"](#) on page 6-13.)

The full names of these schema objects are determined as described in the following subsections. Use the `loadjava -verbose` option for a report of schema objects produced and what they are named.

Note: There are two forms of schema object names in the server: *full names* and *short names*. See ["Full Names and Short Names"](#) on page 11-11.

Full Name of Source

When you load a source file into the server, regardless of whether it is translated and compiled, a source schema object is produced. The full name of this schema object is determined by the package and class names in the source code. Any path information you supply to `loadjava` on the command line is irrelevant to the determination of the name of the schema object.

For example, if `Foo.sqlj` defines 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`

Note that `".sqlj"` is dropped.

If you define additional classes or declare iterator or connection context classes, then the source schema object is named according to the first public class definition or declaration encountered, or, if there are no public classes, the first class definition. In the server, there can be more than one public class definition in a single source.

For example, if `Foo.sqlj` is still in package `x.y`, defines public class `Bar` first and then class `Foo`, and has no public iterator or connection context class declarations preceding the definition of `Bar`, then the full name of the resulting source schema object is:

```
x/y/Bar
```

If, however, the declaration of public iterator class `MyIter` precedes the `Bar` and `Foo` class definitions, then the full name of the resulting source schema object is:

```
x/y/MyIter
```

Full Names of Generated Classes

Class schema objects are generated for each class you defined in the source, each iterator you declared, and the profile-keys class. The naming of the class schema objects is based on the class names and the package name from the source code.

This discussion continues the example in "[Full Name of Source](#)" above. Presume your source code specifies package `x.y`, defines public class `Bar` then class `Foo`, then declares public iterator class `MyIter`. The full names of the class schema objects for the classes you define and declare are as follows:

```
x/y/Bar  
x/y/Foo  
x/y/MyIter
```

Note that ".class" is not appended.

Note: It is recommended that the source name always match the first public class defined, or, if there are no public classes, the first class defined. This will avoid possible differences between client-side and server-side behavior.

The name of the original source file, as well as any path information you specify when loading the source into the server, is irrelevant in determining the names of the generated classes.

If you define inner classes or anonymous classes in your code, they are named according to the conventions of the standard `javac` compiler.

Error Output from the Server Embedded Translator

SQLJ error processing in the server is similar to general Java error processing in the server. SQLJ errors are directed into the `USER_ERRORS` table of the user schema. You can `SELECT` from the `TEXT` column of this table to get the text of a given error message.

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 suppressible warnings are withheld by the server-side translator in a way that is equivalent to the operation of the client-side translator with a `-warn=noportable,noverbose` setting (which is the default). See "[Translator Warnings \(-warn\)](#)" on page 8-45 for information about the meanings of these settings.

Publishing the Application After Loading Source Files

Before using your SQLJ code in the server, you must publish the top-level methods, as is true of any Java code you use in the server. Publishing includes writing call descriptors, mapping datatypes, and setting parameter modes. For information, see the *Oracle9i Java Stored Procedures 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. This section presents only an overview of `dropjava`; it is discussed in detail in the *Oracle9i Java Developer's Guide*.

The `dropjava` utility transforms command-line file names and `.jar` file contents to schema object names, then removes the schema objects. You can enter `.sqlj`, `.java`, `.class`, `.ser`, and `.jar` files on the command line in any order.

You should always remove Java schema objects in the same way that you first loaded them. If you load a `.sqlj` source file and translate it in the server, then run `dropjava` on the same source file. If you translate on a client and load classes and resources directly, then run `dropjava` on the same classes and resources.

For example, if you run `loadjava` on `Foo.sqlj`, then execute `dropjava` on the same file name, as follows:

```
dropjava -user scott/tiger Foo.sqlj
```

If you translate your program on the client and load it using a `.jar` file containing the generated components, then use the same `.jar` file name to remove the program:

```
dropjava -user scott/tiger Foo.jar
```

If you translate your program on the client and load the generated components using the `loadjava` command line, then remove them using the `dropjava` command line, as follows. For this example, assume `-codegen=oracle` and no iterator classes:

```
dropjava -user scott/tiger Foo*.class
```

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 can execute in Oracle9i 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 Oracle JVM, as opposed to in client-side JVMs:

- Threads in the server run sequentially, not simultaneously.
- In the server, threads within a call die at the end of the call.
- Threads in the server are not preemptively scheduled. If one thread goes into an infinite loop, then no other threads can run.

Do not confuse Java multithreading in Oracle9i with general Oracle server multithreading. The latter refers to simultaneous Oracle sessions, not Java multithreading. In the server, scalability and throughput are gained by having many individual users, each with his own session, executing simultaneously. The scheduling of Java execution for maximum throughput (such as for each call within a session) is performed by the Oracle server, not by Java.

For general information about Java multithreading in SQLJ, see "[Multithreading in SQLJ](#)" on page 7-35.

Recursive SQLJ Calls in the Server

As discussed in "[Execution Context Synchronization](#)" on page 7-27, 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 might use a SQLJ statement to call another stored procedure that uses SQLJ statements. When these stored procedures are first created, there is probably no way of knowing when such situations might arise, so it

is doubtful that particular execution context instances are specified for any of the SQLJ statements.

To address this situation, SQLJ *does* allow multiple SQLJ statements to use the same execution context instance simultaneously if this results from recursive calls.

Consider an example of a recursive situation to see what happens to status information in the execution context instance. Presume that all statements use the default connection context instance and its default execution context instance. If stored procedure `proc1` has a SQLJ statement that calls stored procedure `proc2`, which also has SQLJ statements, then the statements in `proc2` will each be using the execution context instance while the procedure call in `proc1` is also using it.

Each SQLJ statement in `proc2` results in status information for that statement being written to the execution context instance, with the opportunity to retrieve that information after completion of each statement as desired. The status information from the statement in `proc1` that calls `proc2` is written to the execution context instance only after `proc2` has finished executing, program flow has returned to `proc1`, and the operation in `proc1` that called `proc2` has completed.

To avoid confusion about execution context status information in recursive situations, execution context methods are carefully defined to update status information about a SQL operation only after the operation has completed.

Notes:

- To avoid confusion, use distinct execution context instances as appropriate whenever you plan to use execution context status or control methods in code that will run in the server.
 - Be aware that if the above example does not use distinct execution context instances, and `proc2` has any method calls to the execution context instance to change control parameters, then this will affect operations subsequently executed in `proc1`.
 - Update batching is not supported across recursive calls. By default, only the top-level procedure will perform batching (if enabled). This limitation can be avoided by using explicit execution context instances.
-
-

For information about `ExecutionContext` methods, see ["Execution Context Methods"](#) on page 7-28.

Verifying that Code is Running in the Server

A convenient way to verify that your code is actually running in the server is to use the static `getProperty()` method of the `java.lang.System` class to retrieve the `oracle.server.version` Java property. If this property contains a version number, then you are running in the Oracle server. If it is `null`, then you are not. Here is an example:

```
...
if (System.getProperty("oracle.server.version") != null
{
    // (running in server)
}
...
```

Note: Do not use the `getProperties()` method, as this causes a security exception in the server.

Customization and Specialized Customizers

Profiles and profile customization are introduced in "[SQLJ Profiles \(ISO Standard Code\)](#)" on page 1-6. This appendix presents more technical detail and discusses customizer options and how to use customizers other than the default Oracle customizer.

There is also discussion of Oracle specialized customizers, particularly the `SQLCheckerCustomizer` for semantics-checking profiles, and the `AuditorInstaller` for installing "auditors" for debugging.

The following topics are covered:

- [More About Profiles](#)
- [More About Profile Customization](#)
- [Customization Options and Choosing a Customizer](#)
- [JAR Files for Profiles](#)
- [SQLCheckerCustomizer for Profile Semantics-Checking](#)
- [AuditorInstaller Customizer for Debugging](#)

Important: If you use the default Oracle-specific code generation (`-codegen=oracle`), the discussion in this appendix does not pertain to your application.

More About Profiles

SQLJ profiles contain information about your embedded SQL operations, with a separate profile being created for each connection context class that your application uses. Profiles are created during the SQLJ translator code generation phase and customized during the customization phase. Customization enables your application to use vendor-specific database features. Separating these vendor-specific operations into your profiles enables the rest of your generated code to remain generic.

Each profile contains a series of entries for the SQLJ statements that use the relevant connection context class, where each entry corresponds to one SQL operation in your application.

Profiles exist as serialized objects stored in resource files packaged with your application. Because of this, profiles can be loaded, read, and modified (added to or 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. It creates a profile object as an instance of the `sqlj.runtime.profile.Profile` class.
2. It inserts information about your embedded SQL operations into the profile object, for SQLJ statements that use the relevant connection context class.
3. It serializes the profile object into a Java resource file, referred to as a *profile file*, with a `.ser` file name extension.

Note: Oracle SQLJ provides an option to have the translator automatically convert these `.ser` files to `.class` files. The `.ser` files are not supported by some browsers, and can be cumbersome when loading translated applications into the server. However, this prevents any further customization of the profile. For information, see "[Conversion of .ser File to .class File \(-ser2class\)](#)" on page 8-68.

As discussed in ["Code Generation"](#) on page 9-5, profile file names for application Foo are of the form:

```
Foo_SJProfile0.ser
```

SQLJ generates `Foo_SJProfile0.ser`, `Foo_SJProfile1.ser`, and so on, as needed, depending on how many connection context classes you use in your code. Or, if the `-ser2class` option is enabled, then SQLJ generates `Foo_SJProfile0.class`, `Foo_SJProfile1.class`, and so on.

Each profile has a `getConnectedProfile()` method that is called during SQLJ runtime. This method returns something equivalent to a JDBC `Connection` object, but with added functionality. This is further discussed in ["Functionality of a Customized Profile at Runtime"](#) on page A-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 empname = 'Smith';
Iter it;
...
#sql it = { SELECT ename, sal FROM emp WHERE ename = :empname };
```

Corresponding SQLJ Profile Entry

```
=====
...
#sql { SELECT ename, sal FROM emp WHERE ename = ? };
...
PREPARED_STATEMENT executed via EXECUTE_QUERY
role is QUERY
descriptor is null
contains one parameter
1. mode: IN, java type: java.lang.String (java.lang.String),
   sql type: VARCHAR, name: ename, ...
result set type is NAMED_RESULT
result set name is Iter
contains 2 result columns
1. mode: OUT, java type: double (double),
   sql type: DOUBLE, name: sal, ...
2. mode: OUT, java type: java.lang.String (java.lang.String),
   sql type: VARCHAR, name: ename, ...
=====
```

Note: This profile entry is presented here as text for convenience only; profiles are not actually in text format. They can be printed as text, however, using the SQLJ `-P-print` option, as discussed in ["Overview of Customizer Harness Options"](#) on page A-12.

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 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 Oracle9i.

You can also run the `sqlj` script to customize profiles created previously. On the SQLJ command line, you can specify `.ser` files individually, `.jar` files containing `.ser` files, or both.

Notes:

- Whenever you use the default Oracle customizer during translation, your application will require the Oracle SQLJ runtime and an Oracle JDBC driver when it runs, even if you do not use Oracle extensions in your code.
 - If an application has no customizations, or none suitable for the connection, then the generic SQLJ runtime is used.
 - You can run SQLJ to process `.sqlj` and `.java` files (for translation, compilation, and customization) or to process `.ser` and `.jar` files (for customization only), but not both categories at once.
-
-

Overview of the Customizer Harness and Customizers

Regardless of whether you use the Oracle customizer or an alternative customizer, SQLJ uses a front-end customization utility known as the *customizer harness* in accomplishing your customizations.

When you run SQLJ, you can specify customization options for the customizer harness (for general customization settings that apply to any customizer you use) and for your customizer (for settings used by the particular customizer). In either case, you can specify these options either on the command line or in a properties file. This is discussed in "[Customization Options and Choosing a Customizer](#)" on page A-11.

A customizer is required to be a JavaBeans component adhering to the standard JavaBeans API to expose its properties, and must implement the `sqlj.runtime.profile.util.ProfileCustomizer` interface, which specifies

a `customize()` method. For each profile to be customized, the customizer harness calls the `customize()` method of the customizer object.

The Oracle customizer meets the preceding requirements and is defined in the `oracle.sqlj.runtime.OraCustomizer` class.

Steps in the Customization Process

The SQLJ customization process during translation consists of the following steps, as applicable, either during the customization stage of an end-to-end SQLJ run, or when you run SQLJ to customize existing profiles only:

1. SQLJ instantiates and invokes the customizer harness and passes it any general customization options you specified.
2. The customizer harness instantiates the customizer you are using and passes it any customizer-specific options you specified.
3. When you run SQLJ for customization only, specifying one or more `.jar` files on the command line, the customizer harness discovers and extracts the profile files within these `.jar` files.
4. The customizer harness deserializes each profile file into a profile object (`.ser` files automatically created during an end-to-end SQLJ run, `.ser` files specified on the command line for customization only, or `.ser` files extracted from `.jar` files specified on the command line for customization only).
5. If the customizer you use requires a database connection, the customizer harness establishes that connection.
6. For each profile, the harness calls the `customize()` method of the customizer object instantiated in step 2 (customizers used with 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 might have a specialized purpose that does not require a customization to be created and registered in this way.
8. The customizer harness reserializes each profile and puts it back into a `.ser` file.
9. When you run SQLJ for customization only, specifying one or more `.jar` files on the command line, the customizer harness recreates the `.jar` contents,

inserting each customized `.ser` file to replace the original corresponding uncustomized `.ser` file.

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 SQLJ connection context object (if you are using a customizer that requires a connection), and an error log object (which is used in logging error messages during the customization).

The same error log object is used for all customizations throughout a single running of SQLJ, but its use is transparent. The customizer harness reads messages written to the error log object, and reports them in real-time to the standard output device (whatever SQLJ uses, typically your screen).

Recall that each profile has a set of entries, where each entry corresponds to a SQL operation. (These would be the SQL operations in your application that use instances of the connection context class associated with this profile.)

A `customize()` method implements special processing on these entries. It could be as simple as checking each entry to verify its syntax, or it could be more complicated, such as creating new entries that are equivalent to the original entries but are modified to use features of your particular database.

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 Oracle9i. 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 called at runtime to determine if the customization can create a connected profile object for a given SQLJ connection context 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 (underlying a SQLJ connection context 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 suppressible, however.

Error messages reported by the customizer harness fall into four categories:

- unrecognized or illegal option
- connection instantiation error
- profile instantiation error
- customizer instantiation error

Status messages reported by the customizer harness during customization 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 might be reported by the `customize()` method of the particular customizer.

During customization, the profile customizer writes messages to its error log, and the customizer harness reads the log contents in real-time and outputs these messages to the SQLJ output device, along with any other harness output. You never have to access error log contents directly.

Functionality of a Customized Profile at 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 `getConnectionProfile()` method.
2. The connection context `getConnectionProfile()` method calls the `getConnectionProfile()` method of the profile object 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 `getConnectionProfile()` 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:

- options you specify to the customizer harness, which apply to whatever customizer you use

This includes general options, connection options, and options that invoke specialized customizers.
- customizer-specific options you specify to your customizer through the customizer harness
- SQLJ options, which determine basic aspects of customization, such as whether to customize at all and which customizer to use

All categories of options are specified through the SQLJ command line or properties files.

The following topics are included in this section:

- [Overview of Customizer Harness Options](#)
- [General Customizer Harness Options](#)
- [Customizer Harness Options for Connections](#)
- [Customizer Harness Options that Invoke Specialized Customizers](#)
- [Overview of Customizer-Specific Options](#)
- [Oracle Customizer Options](#)
- [SQLJ Translator Options for Profile Customization](#)

To choose a customizer other than the default Oracle customizer, you can use either the customizer harness `customizer` option (discussed in "[Overview of Customizer Harness Options](#)" on page A-12) or the SQLJ `-default-customizer` option (discussed in "[SQLJ Translator Options for Profile Customization](#)" on page A-37).

Overview of Customizer Harness Options

The customizer harness provided with Oracle SQLJ offers a number of options that are not specific to a particular customizer. The harness uses these options in its front-end coordination of the customization process.

Syntax for Customizer Harness Options

Customizer harness option settings on the SQLJ command line have the following syntax:

```
-P-option=value
```

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 the Profile Customizer \(-P\)"](#) on page 8-63 and ["Properties File Syntax"](#) on page 8-17.

Options Supported by the Customizer Harness

The customizer harness supports the following general options:

- `backup`—Save a backup copy of the profile before customizing it.
- `context`—Limit customizations to profiles associated with the listed connection context classes.
- `customizer`—Specify the customizer to use.
- `digests`—Specify digests for `.jar` file manifests (relevant only if specifying `.jar` files to customize).
- `help`—Display customizer options (specified in SQLJ command-line only).
- `verbose`—Display status messages during customization.

The customizer harness supports the following options for customizer database connections. Currently, these are used by the Oracle customizer if you enable its `optcols` option for column definitions (for performance optimization). In addition, they are used by the `SQLCheckerCustomizer` if you use this specialized customizer to perform online semantics-checking on profiles.

- `user`—Specify the user name for the connection used in this customization.
- `password`—Specify the password for the connection used in this customization.
- `url`—Specify the URL for the connection used in this customization.
- `driver`—Specify the JDBC driver for the connection used in this customization.

For information about the Oracle customizer `optcols` flag, see "[Oracle Customizer Column Definition Option \(optcols\)](#)" on page A-27. For information about the `SQLCheckerCustomizer`, see "[SQLCheckerCustomizer for Profile Semantics-Checking](#)" on page A-40.

The following commands function as customizer harness options, but are implemented through specialized customizers provided with Oracle SQLJ.

- `debug`—Insert debugging information into the specified profiles, to be output at runtime. This is a shortcut to invoke the Oracle SQLJ `AuditorInstaller`, which is described in "[AuditorInstaller Customizer for Debugging](#)" on page A-44.
- `print`—Output the contents of the specified profiles, in text format.
- `verify`—Perform semantics-checking on a profile that was produced during a previous execution of the SQLJ translator (equivalent to semantics-checking performed on source code during translation). This is a shortcut to invoke the Oracle SQLJ `SQLCheckerCustomizer`, which is described in "[SQLCheckerCustomizer for Profile Semantics-Checking](#)" on page A-40.

General Customizer Harness Options

This section describes general options supported by the customizer harness.

Profile Backup Option (`backup`)

Use the `backup` flag to instruct the harness to save a backup copy of each `.jar` file and standalone `.ser` file before replacing the original. (Separate backups of `.ser` files that are within `.jar` files are not necessary.)

Backup file names are given the extension `.bakn`, 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`

Customization Connection Context Option (`context`)

Use the `context` option to limit customizations to profiles that correspond to the specified connection context classes. Fully qualify the class names and use a comma-delimited list to specify multiple classes. For example:

```
-P-context=sqlj.runtime.ref.DefaultContext,foo.bar.MyCtxtClass
```

There must be no space on either side of the comma.

If this option is not specified, then all profiles are customized, regardless of their associated connection context classes.

Command-line syntax `-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 qualify the class name, such as in the following example:

```
-P-customizer=oracle.sqlj.runtime.util.OraCustomizer
```

If you do not set this option, then SQLJ will use the customizer specified in the SQLJ `-default-customizer` option. Unless set otherwise, this is the following:

```
oracle.sqlj.runtime.util.OraCustomizer
```

Command-line syntax `-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 none (use default, set in SQLJ `-default-customizer` option)

Customization JAR File Digests Option (digests)

When a `.jar` file is produced, the `jar` utility can optionally include one or more *digests* for each entry, based on one or more specified algorithms, so that the integrity of the `.jar` file entries can later be verified. Digests are similar conceptually to checksums, for readers familiar with those.

If you are customizing profiles in a `.jar` file and want the `jar` utility to add new digests (or update existing digests) when the `.jar` file is updated, use the `digests` option to specify a comma-delimited list of one or more algorithms. These are the algorithms that `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 one of the following Web sites:

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

or:

<http://java.sun.com/j2se/1.3/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`

Customization Help Option (help)

Use the `help` option to display the option lists of the customizer harness and the default customizer or a specified customizer. For the harness and Oracle customizer, this includes a brief description and the current setting of each option.

Display the option lists for the harness and default customizer as follows (where the default customizer is the Oracle customizer or whatever you have specified in the `SQLJ -default-customizer` option):

```
-P-help
```

Use the `help` option in conjunction with the `customizer` option to display the option list of a particular customizer, as follows:

```
-P-help -P-customizer=sqlj.runtime.profile.util.AuditorInstaller
```

Notes:

- You can use the `-P-help` option on the SQLJ command line only, not in a SQLJ properties file.
 - No customizations are performed if the `-P-help` flag is enabled, even if you specify profiles to customize on the command line.
-

Command-line syntax `-P-help <-P-customizer=customizer_class>`

Command-line example `-P-help`

Properties file syntax `n/a`

Properties file example `n/a`

Default value `none`

Customization Verbose Option (verbose)

Use the `verbose` flag to instruct the harness to display status messages during customizations. These messages are written to the standard output device—wherever SQLJ writes its other messages.

Command-line syntax `-P-verbose<=true|false>`

Command-line example `-P-verbose`

Properties file syntax `profile.verbose<=true|false>`

Properties file example `profile.verbose`

Default value `false`

Customizer Harness Options for Connections

This section describes connection options supported by the customizer harness. These are used as follows:

- The Oracle customizer uses database connections only for column definitions. If you do not enable the Oracle customizer `optcols` option, then there is no need to set the customizer harness `user`, `password`, `url`, and `driver` options.
- The `SQLCheckerCustomizer`, a specialized customizer that performs semantics-checking on profiles, uses the customizer harness `user`, `password`, `url`, and `driver` settings for online checking.

Use `-P-verify` on the SQLJ command line to invoke this customizer.

Note: Do not confuse the customizer harness `user`, `password`, `url`, and `driver` options with the translator options of the same names, which are for semantics-checking during the translation step. However, the translator settings are passed to the customizer for convenience, in case customization is to use the same connection as translation. Override these initial settings through the customizer harness options if you wish.

Customization User Option (`user`)

Set the `user` option to specify a database schema if your customizer uses database connections.

In addition to specifying the schema, you can optionally specify the password, URL, or both in your `user` option setting. The password is preceded by a forward-slash (/), and the URL is preceded by an "at" sign (@), as in the following examples:

```
-P-user=scott/tiger
-P-user=scott@jdbc:oracle:oci:@
-P-user=scott/tiger@jdbc:oracle:oci:@
```

Note: When you use column definitions (`optcols` option), the `user` setting for the SQLJ translator is forwarded to the profile customizer as well, but you can use the customizer `user` option to override the translator setting.

Command-line syntax `-P-user=username</password><url>`

Command-line examples

```
-P-user=scott
-P-user=scott/tiger
-P-user=scott/tiger@jdbc:oracle:oci:@
```

Properties file syntax `profile.user=username</password><url>`

Properties file examples

```
profile.user=scott
profile.user=scott/tiger
profile.user=scott/tiger@jdbc:oracle:oci:@
```

Default value `null`

Customization Password Option (password)

Use the `password` option if your customizer uses database connections.

The password can also be set with the `user` option, as described in "[Customization User Option \(user\)](#)" on page A-18.

Note: When you use column definitions (`optcols` option), the password setting for the SQLJ translator is forwarded to the profile customizer as well, but you can use the customizer `password` option to override the translator setting.

Command-line syntax `-P-password=password`

Command-line example `-P-password=tiger`

Properties file syntax `profile.password=password`

Properties file example `profile.password=tiger`

Default value `null`

Customization URL Option (`url`)

Use the `url` option if your customizer uses database connections.

The URL can also be set with the `user` option, as described in "[Customization User Option \(`user`\)](#)" on page A-18.

Note: When you use column definitions (`optcols` option), the URL setting for the SQLJ translator is forwarded to the profile customizer as well, but you can use the customizer `url` option to override the translator setting.

Command-line syntax `-P-url=url`

Command-line example `-P-url=jdbc:oracle:oci:@`

Properties file syntax `profile.url=url`

Properties file example `profile.url=jdbc:oracle:oci:@`

Default value `jdbc:oracle:oci:@`

Customization JDBC Driver Option (`driver`)

Use the `driver` option to register a comma-delimited list of JDBC driver classes if your customizer uses database connections. For example:

`-P-driver=sun.jdbc.odbc.JdbcOdbcDriver,oracle.jdbc.OracleDriver`

There must be no space on either side of the comma.

Command-line syntax `-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.OracleDriver`

Customizer Harness Options that Invoke Specialized Customizers

The customizer harness supports the following options that invoke specialized customizers:

- `debug`—This invokes the `AuditorInstaller` customizer, used in debugging.
- `print`—This invokes a customizer that prints a text version of a profile.
- `verify`—This invokes the `SQLCheckerCustomizer` customizer, which performs semantics-checking on a profile.

Important: Because each of these options invokes a customizer, and only one customizer can run in a single execution of SQLJ, you cannot perform any other customization when you use any of these options.

You also cannot use more than one of `print`, `debug`, or `verify` simultaneously.

Specialized Customizer: Profile Debug Option (`debug`)

The `debug` option runs a specialized customizer, called the `AuditorInstaller`, that inserts debugging statements into profiles. Use this option in conjunction with a SQLJ command line file list to insert debugging statements into the specified profiles. These profiles must already be customized from a previous SQLJ run.

For detailed information about this customizer, including additional options that it supports, see "[AuditorInstaller Customizer for Debugging](#)" on page A-44.

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
```

Command-line syntax `sqlj -P-debug profile_list`

Command-line example `sqlj -P-debug Foo_SJProfile*.ser`

Properties file syntax `profile.debug`

(Also specify profiles in the SQLJ file list.)

Properties file example `profile.debug`

Default value `n/a`

Specialized Customizer: Profile Print Option (`print`)

The `print` option runs a specialized customizer that prints profiles in text format. Use this option in conjunction with a SQLJ command line file list to output the contents of one or more specified profiles. The output goes to the standard SQLJ output device, typically the user screen.

Following are examples of how to specify the `print` option:

```
sqlj -P-print Foo_SJProfile0.ser Bar_SJProfile0.ser
```

```
sqlj -P-print *.ser
```

For sample output, see ["Sample Profile Entry"](#) on page A-3.

Command-line syntax `sqlj -P-print profile_list`

Command-line example `sqlj -P-print Foo_SJProfile*.ser`

Properties file syntax `profile.print`

(Also specify profiles in SQLJ file list.)

Properties file example `profile.print`

Default value `n/a`

Specialized Customizer: Profile Semantics-Checking Option (`verify`)

The `verify` option runs a specialized customizer, called the `SQLCheckerCustomizer`, that performs semantics-checking on a profile. This is equivalent to the semantics-checking that is performed on source code during translation. The profile will have been created during a previous execution of the SQLJ translator.

This option is useful for checking semantics against the runtime database, after deployment, and after the source code may no longer be available.

For detailed information about this customizer, including additional options that it supports, see ["SQLCheckerCustomizer for Profile Semantics-Checking"](#) on page A-40.

Note: For online semantics-checking of the profile, you must also use the customizer harness `user`, `password`, and `url` options.

Following are examples of how to specify the `verify` option. Both of these examples use the `SQLCheckerCustomizer` default semantics-checker, which employs online checking through the specified database connection. (The first is a single wraparound command.)

```
sqlj -P-verify -P-user=scott -P-password=tiger -P-url=jdbc:oracle:oci:@  
Foo_SJProfile0.ser Bar_SJProfile0.ser
```

```
sqlj -P-verify -P-user=scott -P-password=tiger -P-url=jdbc:oracle:oci:@ *.ser
```

Command-line syntax `sqlj -P-verify <conn params> profile_list`

Command-line example `sqlj -P-verify <conn params> Foo_SJProfile*.ser`

Properties file syntax `profile.verify`

(You must also specify profiles, and typically customizer harness connection options, in the SQLJ command line.)

Properties file example `profile.verify`

Default value `n/a`

Overview of Customizer-Specific Options

You can set customizer-specific options, such as options for the Oracle customizer, on the SQLJ command line or in a SQLJ properties file. The syntax is similar to that for setting customizer harness options.

Set a customizer option on the SQLJ command line by preceding it with:

```
-P-C
```

Alternatively, you can set it in a SQLJ properties file by preceding it with:

```
profile.C
```

This option syntax is also discussed in ["Options to Pass to the Profile Customizer \(-P\)"](#) on page 8-63 and ["Properties File Syntax"](#) on page 8-17.

The remainder of this section discusses features of the Oracle customizer, which supports several options. Most of these options are boolean and are enabled as follows:

```
-P-Option
```

or:

```
-P-Option=true
```

Boolean options are disabled by default, but you can explicitly disable them with:

```
-P-Option=false
```

Numeric or string options are set similarly:

```
-P-Option=value
```

Oracle Customizer Options

This section describes options that are specific to the Oracle customizer, beginning with an overview of the options supported.

Options Supported by the Oracle Customizer

The Oracle customizer implements the following options:

- `compat`—Display version compatibility information.
- `force`—Instruct the customizer to customize even if a valid customization already exists.
- `optcols`—Enable iterator column type and size definitions to optimize performance.
- `optparams`—Enable parameter size definitions to optimize JDBC resource allocation (used in conjunction with `optparamdefaults`).
- `optparamdefaults`—Set parameter size defaults for particular datatypes (used in conjunction with `optparams`).

- `fixedchar`—Enable CHAR comparisons with blank padding for WHERE clauses.
- `showSQL`—Display SQL statement transformations.
- `stmtcache`—Set the statement cache size (the number of statements that can be cached for each connection during runtime) for performance optimization, or set it to zero to disable statement caching.
- `summary`—Display a summary of Oracle features used in your application.

Any output displayed by these options is written to the standard output device, wherever SQLJ writes its other messages.

Oracle Customizer Version Compatibility Option (`compat`)

Use the `compat` flag to instruct the Oracle customizer to display information about compatibility of your application with different versions of the Oracle database and Oracle JDBC drivers. This can be accomplished either during a full SQLJ translation run or on profiles previously created.

For example, to see compatibility output when translating and customizing the application `MyApp`:

```
sqlj <...SQLJ options...> -P-Ccompat MyApp.sqlj
```

In this example, the `MyApp` profiles will be created, customized, and checked for compatibility in a single running of SQLJ.

To see compatibility output for `MyApp` profiles previously created:

```
sqlj <...SQLJ options...> -P-Ccompat MyApp_SJProfile*.ser
```

In this example, the `MyApp` profiles were created (and possibly customized) in a previous running of SQLJ and will be customized (if needed) and checked for compatibility in the above running of SQLJ.

Following are two output samples from a `-P-Ccompat` setting when using the default Oracle customizer. The first example indicates that the application can be used with all Oracle JDBC driver versions:

```
MyApp_SJProfile0.ser: Info: compatible with all Oracle JDBC drivers
```

This second example indicates that the application can be used only with Oracle JDBC release 8.1 or later:

```
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 the `MyApp` profiles, regardless of whether they have already been customized. Otherwise, by default, the Oracle customizer will not reinstall over a previously existing customization unless the previous one had been installed with an older version of the customizer.

Command-line syntax `-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 Column Definition Option (`optcols`)

Use the `optcols` flag to instruct the Oracle customizer to determine types and sizes of iterator or result set columns and add this information to the profile. This enables the SQLJ runtime to automatically register the columns with the Oracle JDBC driver when your application runs, saving round trips to Oracle depending on the particular driver implementation. Specifically, this is effective for the Thin driver and positional iterators.

For an overview of column definitions, see "[Column Definitions](#)" on page 10-23.

An error will be generated if you enable the Oracle customizer `optcols` option without setting the user name, password, and URL for a database connection. You can accomplish this through the translator `-user`, `-password`, and `-url` options, which are forwarded to the customizer during ISO standard code generation, or directly through the customizer `user`, `password`, and `url` options.

The customizer does not have to connect to the same schema or even the same database that your application will connect to at runtime, but the relevant columns will have to be in the same order and of identical types and sizes to avoid runtime errors.

For information about the customizer harness connection options, see the `user`, `password`, `url`, and `driver` sections under "[Overview of Customizer Harness Options](#)" on page A-12.

Note: You can use the SQLJ translator `-optcols` option instead—this sets the customizer option automatically. (And for Oracle-specific code generation, which uses no profiles, you *must* use the translator option instead.) See "[Column Definitions \(-optcols\)](#)" on page 8-53.

That section also has some additional conceptual information.

You can enable or disable the customizer `optcols` flag on the SQLJ command line or in a properties file.

Enable it on the command line as follows:

```
-P-Coptcols
```

or:

```
-P-Coptcols=true
```

This flag is disabled by default, but you can also disable it explicitly. Disable it on the command line as follows:

```
-P-Coptcols=false
```

Column definitions require the customizer to make a database connection to examine columns of tables being queried, so the customizer harness `user`, `password`, and `url` options must be set appropriately (as well as the customizer harness `driver` option if you are not using the default `OracleDriver` class). For example:

```
sqlj <...SQLJ options...> -P-user=scott/tiger@jdbc:oracle:oci:@ -P-Coptcols MyApp.sqlj
```

Note that as with the SQLJ translator, you can optionally set the password and URL in the `user` option instead of in the `password` and `url` options.

Alternatively, you can insert column definitions into a previously existing profile. In this case you must also use the Oracle customizer `force` option to force a recustomization:

```
sqlj -P-user=scott/tiger@jdbc:oracle:oci:@ -P-Cforce -P-Coptcols MyApp_SJProfile*.ser
```

You also can insert column definitions into previously existing profiles in a `.jar` file:

```
sqlj -P-user=scott/tiger@jdbc:oracle:oci:@ -P-Cforce -P-Coptcols MyAppProfiles.jar
```

When you run the Oracle customizer with its `optcols` flag enabled, either during translation and creation of a new profile or during customization of an existing profile, you can also enable the customizer harness `verbose` flag. This will instruct the Oracle customizer to display information about what iterators and result sets are being processed and what their column type and size definitions are. For example:

```
sqlj -P-user=scott/tiger@jdbc:oracle:oci:@ -P-verbose -P-Cforce -P-Coptcols MyApp_SJProfile*.ser
```

For general information about the `verbose` flag, see that section under "[Overview of Customizer Harness Options](#)" on page A-12.

You can execute the Oracle customizer with its `summary` flag enabled on an existing profile to determine if column definitions have been added to that profile:

```
sqlj -P-Csummary MyApp_SJProfile*.ser
```

For general information about the `summary` flag, see that section under "[Overview of Customizer-Specific Options](#)" on page A-23.

Command-line syntax `-P-Coptcols<=true|false>`

Command-line example `-P-Coptcols`

Properties file syntax `profile.Coptcols<=true|false>`

Properties file example `profile.Coptcols`

Default value `false`

Oracle Customizer Parameter Definition Option (optparams)

Use the `optparams` flag to enable parameter size definitions. If this flag is enabled, SQLJ will register your input and output parameters (host variables) to optimize JDBC resource allocations according to sizes you specify.

For an overview of parameter size definitions and a discussion of source code hints, see "[Parameter Size Definitions](#)" on page 10-25.

Note: You can use the SQLJ translator `-optparams` option instead—this sets the customizer option automatically. (And for Oracle-specific code generation, which uses no profiles, you *must* use the translator option instead.) See "[Parameter Definitions \(-optparams\)](#)" on page 8-55.

That section also has some additional conceptual information.

You can enable or disable the `optparams` flag on the command line or in a SQLJ properties file.

Enable it on the command line as follows:

```
-P-Coptparams
```

or:

```
-P-Coptparams=true
```

This flag is disabled by default, but you can also disable it explicitly. Disable it on the command line as follows:

```
-P-Coptparams=false
```

Note: Unlike the `optcols` option, the `optparams` option does not require a database connection by the customizer, because you are providing the size specifications yourself.

Following is a command-line example (omitting a setting for the `optparamdefaults` option, which is discussed in the next section):

```
sqlj <...SQLJ options...> -P-Coptparams -P-Coptparamdefaults=defaults_string MyApp.sqlj
```

or, to enable parameter size definitions for a previously existing profile:

```
sqlj -P-Coptparams -P-Coptparamdefaults=defaults_string MyApp_SJProfile*.ser
```

or, for previously existing profiles in a `.jar` file:

```
sqlj -P-Coptparams -P-Coptparamdefaults=defaults_string MyAppProfiles.jar
```

Command-line syntax `-P-Coptparams<=true|false>`

Command-line example `-P-Coptparams`

Properties file syntax `profile.Coptparams<=true|false>`

Properties file example `profile.Coptparams`

Default value `false`

Oracle Customizer Parameter Default Size Option (`optparamdefaults`)

If you enable the `optparams` option to set parameter sizes, use the `optparamdefaults` option as desired to set default sizes for specified datatypes. If `optparams` is *not* enabled, then any `optparamdefaults` setting is ignored.

For an overview of parameter size definitions and a discussion of source code hints, see "[Parameter Size Definitions](#)" on page 10-25.

Note: You can use the SQLJ translator `-optparamdefaults` option instead—this sets the customizer option automatically. (And for Oracle-specific code generation, which uses no profiles, you *must* use the translator option instead.) See "[Parameter Default Size \(-optparamdefaults\)](#)" on page 8-56.

That section also has important additional conceptual and syntax information. Functionality of the two options is equivalent.

You can set the `optparamdefaults` flag on the command line or in a SQLJ properties file.

Set it on the command line as follows:

```
-P-Coptparamdefaults=datatype1(size1),datatype2(size2),...
```

Following is a command-line example, including the `optparams` setting as well:

```
sqlj <...SQLJ options...> -P-Coptparams -P-Coptparamdefaults=CHAR_TYPE(50),RAW_TYPE(500),CHAR(10) MyApp.sqlj
```

The syntax is explained in "[Parameter Default Size \(-optparamdefaults\)](#)" on page 8-56.

Alternatively, you can specify parameter size defaults for a previously existing profile, in which case you must also use the Oracle customizer `force` option to force a recustomization:

```
sqlj -P-Cforce -P-Coptparams -P-Coptparamdefaults=CHAR_TYPE(50),RAW_TYPE(500),CHAR(10) MyApp_SJProfile*.ser
```

You also can specify parameter size defaults for previously existing profiles in a `.jar` file:

```
sqlj -P-Cforce -P-Coptparams -P-Coptparamdefaults=CHAR_TYPE(50),RAW_TYPE(500),CHAR(10) MyAppProfiles.jar
```

Note: If at runtime the actual size exceeds the registered size of any parameter, runtime errors will occur.

Command-line syntax `-P-Coptparamdefaults=defaults_string`

Command-line example `-P-Coptparamdefaults=VAR%(50),LONG%(500),RAW_TYPE()`

Properties file syntax `profile.Coptparamdefaults=defaults_string`

Properties file example `profile.Coptparamdefaults=VAR%(50),LONG%(50),RAW_TYPE()`

Default value `null`

Oracle Customizer CHAR Comparisons with Blank Padding (`fixedchar`)

Set this flag to `true` to account for blank padding in `CHAR` database columns when binding character strings for `WHERE` clause comparisons. This way, for example, "mystring" would compare positively against "mystring ".

Here is an example of Oracle customizer `fixedchar` usage:

```
% sqlj -P-Cfixedchar MyProgram.sqlj AnotherProg.java ...
```

Notes:

- You can use the SQLJ translator `-fixedchar` option instead—this sets the customizer option automatically. (And for Oracle-specific code generation, which uses no profiles, you *must* use the translator option instead.) See "[CHAR Comparisons with Blank Padding \(-fixedchar\)](#)" on page 8-58. That section also has some additional conceptual information.
 - If you also enable the Oracle customizer `summary` flag, the number of usages of the Oracle `setFixedCHAR()` API (used behind the scenes for `fixedchar` functionality) will be displayed. See "[Oracle Customizer Summary Option \(summary\)](#)" on page A-35 for an example.
-
-

Command-line syntax `-P-Cfixedchar<=true|false>`

Command-line example `-P-Cfixedchar`

Properties file syntax `profile.Cfixedchar<=true|false>`

Properties file example `profile.Cfixedchar`

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 Oracle9i does not. For example, 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 Oracle9i 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 Statement Cache Size Option (stmtcache)

Use the Oracle customizer `stmtcache` option to set the statement cache size—the number of statements that can be cached for each database connection as your application runs—or to disable statement caching.

The default statement cache size is 5. For an overview of statement caching, see ["Statement Caching"](#) on page 10-4.

Important: With the default Oracle-specific code generation (`-codegen=oracle`), SQLJ does not produce profiles and skips the customization step. In this case, use connection context methods to control SQLJ statement caching. See ["Connection Context Methods for Statement Caching \(Oracle-Specific Code\)"](#) on page 10-5.

You can set the statement cache size on the command line or in a properties file.

To use the command line to set the statement cache size to 15 (for example) for the application `MyApp`:

```
sqlj <...SQLJ options...> -P-Cstmtcache=15 MyApp.sqlj
```

To disable statement caching, set the cache size to 0:

```
sqlj <...SQLJ options...> -P-Cstmtcache=0 MyApp.sqlj
```

You also can alter the statement cache size in an existing profile without re-translating the application, but you must also use the Oracle customizer `force` option to force a recustomization, as follows:

```
sqlj -P-Cforce -P-Cstmtcache=15 MyApp_SJProfile0.ser
```

If you have multiple profiles, you can set their statement cache sizes individually by running SQLJ separately for each profile, after you have translated your application:

```
sqlj -P-Cforce -P-Cstmtcache=10 MyApp_SJProfile0.ser  
sqlj -P-Cforce -P-Cstmtcache=15 MyApp_SJProfile1.ser  
sqlj -P-Cforce -P-Cstmtcache=0 MyApp_SJProfile2.ser
```

Of course, you must determine which profile corresponds to each of your connection context classes. This is determined as follows: profile 0 will correspond to the connection context class used for the first executable statement in your application; profile 1 will correspond to the connection context class used for the

first executable statement that does not use the first connection context class, and so on. You can verify the correlation by using the customizer harness `print` option to examine each profile.

Command-line syntax `-P-Cstmtcache=value`

Command-line example `-P-Cstmtcache=10`

Properties file syntax `profile.Cstmtcache=value`

Properties file example `profile.Cstmtcache=10`

Default value 5

Oracle Customizer Summary Option (summary)

Use the `summary` flag to instruct the Oracle customizer to display a summary of Oracle features used in an application being translated, or in specified profile files. This is useful in identifying features that would prevent portability to other platforms and can be accomplished either during a full SQLJ translation run or on profiles previously created.

To see summary output when translating and customizing the application `MyApp`:

```
sqlj <...SQLJ options...> -P-Csummary MyApp.sqlj
```

In this example, the `MyApp` profiles will be created, customized, and summarized in a single running of SQLJ.

To see summary output for `MyApp` profiles previously created:

```
sqlj <...SQLJ options...> -P-Csummary MyApp_SJProfile*.ser
```

In this example, the `MyApp` profiles were created (and possibly customized) in a previous running of SQLJ and will be customized (if needed) and summarized in the above running of SQLJ.

Following are two samples resulting from a `-P-Csummary` setting when using the default Oracle customizer. The first example indicates no Oracle features are used:

```
MyApp_SJProfile0.ser: Info: Oracle features used:  
MyApp_SJProfile0.ser: Info: * none
```

This second example indicates that Oracle features *are* used—namely, several Oracle extended datatypes from the `oracle.sql` package—and lists them:

```
MyApp_SJProfile0.ser: Info: Oracle features used:
MyApp_SJProfile0.ser: Info: * oracle.sql.NUMBER: 2
MyApp_SJProfile0.ser: Info: * oracle.sql.DATE: 2
MyApp_SJProfile0.ser: Info: * oracle.sql.CHAR: 2
MyApp_SJProfile0.ser: Info: * oracle.sql.RAW: 2
```

The following example prints out the number of usages of the Oracle `setFixedCHAR()` API (enabled through the Oracle customizer `fixedchar` option, to account for blank padding when binding a string into a `WHERE` clause for comparison against `CHAR` data):

```
% sqlj -P-Cfixedchar -P-Csummary -P-Cforce *.ser
FC_SJProfile0.ser: Info: re-installing Oracle customization
FC_SJProfile0.ser: Info: Oracle features used:
FC_SJProfile0.ser: Info: * setFixedCHAR(): 4
```

Note: If customization does not take place because a valid previous customization is detected, the `summary` option produces a summary regardless.

Command-line syntax `-P-Csummary<=true|false>`

Command-line example `-P-Csummary`

Properties file syntax `profile.Csummary<=true|false>`

Properties file example `profile.Csummary`

Default value `false`

Options for Other Customizers

Oracle SQLJ provides additional, specialized customizers described later in this chapter. These customizers also have command-line options:

- `SQLCheckerCustomizer` (for profile semantics-checking)—See "[SQLCheckerCustomizer for Profile Semantics-Checking](#)" on page A-40 for general information, and "[SQLCheckerCustomizer Options](#)" on page A-41 for information about its options.
- `AuditorInstaller` (for debugging)—See "[AuditorInstaller Customizer for Debugging](#)" on page A-44 for general information, and "[AuditorInstaller Options](#)" on page A-47 for information about its options.

SQLJ Translator Options for Profile Customization

The following SQLJ translator options relate to profile customization and are described elsewhere in this manual:

- `-default-customizer`—Specify the default profile customizer to use if none is specified in the customizer harness `-customizer` option.
See "[Default Profile Customizer \(-default-customizer\)](#)" on page 8-88.
- `-profile`—Specify whether to customize during this running of SQLJ.
See "[Profile Customization Flag \(-profile\)](#)" on page 8-66.

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` or `.java` files or both on the SQLJ command line for normal SQLJ processing, or you can specify `.ser` or `.jar` files or both on the command line for customization only, but not both categories.
 - It is permissible for the `.jar` file to contain files that are not profiles. Any file whose manifest entry indicates that the file is not a profile will be ignored during customization.
 - The `.jar` file is used as the class-loading context for each profile it contains. If a profile contains a reference to a class contained within the `.jar` file, then that class is loaded from the `.jar` file. If a profile contains a reference to a class not in the `.jar` file, then the system class loader will find and load the class according to your classpath, as usual.
-
-

JAR File Requirements

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). Presume you call this file `MyAppJarEntries.txt`.

```
Name: foo/bar/MyApp_SJProfile0.ser  
SQLJProfile: TRUE
```

```
Name: foo/bar/MyApp_SJProfile1.ser  
SQLJProfile: TRUE
```

```
Name: foo/bar/MyApp_SJProfile2.ser  
SQLJProfile: TRUE
```

2. When you run `jar` to create the `.jar` file, use the `-m` option to input your text file as follows (presume you want to call the `.jar` file `myjarfile.jar`):

```
jar -cvfm myjarfile.jar MyAppJarEntries.txt foo/bar/MyApp_SJProfile*.ser foo/bar/*.class
```

As the `jar` utility constructs the manifest during creation of the `.jar` file, it reads your text file and inserts the `SQLJProfile: TRUE` line into the manifest entry of each profile. It accomplishes this by matching the names in the manifest with the names you specify in your text file.

JAR File Logistics

When you specify a `.jar` file on the SQLJ command line, each profile in the `.jar` file is deserialized and customized.

A `.jar` file is successfully customized only if all the profiles it contains are successfully customized. After a successful customization, each profile has been reserialized into a `.ser` file, the `.jar` file has been modified to replace the original `.ser` files with the customized `.ser` files, and the `.jar` file manifest has been updated to indicate the new entries.

If any error is encountered in the customization of any profile in a `.jar` file, then the `.jar` file customization has failed, and the original `.jar` file is left completely unchanged.

Note: If you use signature files for authentication, the signature files that appeared in the original `.jar` file will appear unchanged in the updated `.jar` file. You are responsible for re-signing the new `.jar` file if the profiles require signing.

SQLCheckerCustomizer for Profile Semantics-Checking

Oracle provides a special customizer, `SQLCheckerCustomizer`, that will perform semantics-checking on a profile that was produced during previous execution of the translator. This semantics-checking is similar to what is normally performed during translation of the source code.

This is particularly valuable when the database to be used at runtime is not available for semantics-checking during translation. In these circumstances, you can use `SQLCheckerCustomizer` after deployment, against the runtime database, typically in a scenario where the source code is no longer available.

You can specify the checker to use. If you accept the default `OracleChecker` front end, `SQLCheckerCustomizer` will perform online semantics-checking using an appropriate online checker.

Note: For online semantics-checking of the profile, you must also specify connection parameters using the customizer harness connection options.

Invoking SQLCheckerCustomizer with the Customizer Harness `verify` Option

Following are examples of how to specify the Oracle customizer harness `verify` option to run `SQLCheckerCustomizer` in its default mode. Because it defaults to an online checker, you typically must provide connection parameters through the customizer harness `user`, `password`, and `url` options. (The first example is a single wraparound command line.)

```
sqlj -P-verify -P-user=scott -P-password=tiger -P-url=jdbc:oracle:oci:@  
Foo_SJProfile0.ser Bar_SJProfile0.ser
```

```
sqlj -P-verify -P-user=scott -P-password=tiger -P-url=jdbc:oracle:oci:@ *.ser
```

The `verify` option results in the customizer harness instantiating and invoking the following class:

```
sqlj.runtime.profile.util.SQLCheckerCustomizer
```

This class coordinates semantics-checking of the SQL operations in the profile. You can specify a semantics-checker or accept the default `OracleChecker` semantics-checker front end.

The `-P-verify` option is equivalent to the following:

```
-P-customizer=sqlj.runtime.profile.util.SQLCheckerCustomizer
```

This overrides the customizer specified in the SQLJ `-default-customizer` option.

Notes:

- As with any Oracle customizer, help output and an option list will be provided if you specify `-P-verify` together with `-P-help` on the SQLJ command line.
 - It is important to realize that because the `verify` option invokes a customizer, and only one customizer can run in any single running of SQLJ, you cannot do any other customization when you use this option.
 - You also cannot use more than one of `-P-print`, `-P-debug`, and `-P-verify` simultaneously, because each of these invokes a specialized customizer.
-
-

Command-line syntax `sqlj -P-verify <conn params> profile_list`

Command-line example `sqlj -P-verify <conn params> Foo_SJProfile*.ser`

Properties file syntax `profile.verify`

(You must also specify profiles, and typically customizer harness connection options, in the SQLJ command line.)

Properties file example `profile.verify`

Default value n/a

SQLCheckerCustomizer Options

Like any customizer, `SQLCheckerCustomizer` has its own options, which can be set using the `-P-C` prefix on the SQLJ command line or the `profile.C` prefix in a SQLJ properties file.

SQLCheckerCustomizer supports the following options:

- `checker`—Specify the semantics-checker to use. The default is the OracleChecker front end, as for checking during SQLJ translation.
- `warn`—Specify the categories of warnings and messages to display during semantics-checking of the profile. This is equivalent to the SQLJ `-warn` flag for warning categories during translation-time semantics-checking, supports the same settings, and uses the same defaults. See "[Translator Warnings \(-warn\)](#)" on page 8-45.

SQLCheckerCustomizer Semantics-Checker Option (`checker`)

The `checker` option allows you to specify the semantics-checker to use in checking the SQL operations in a profile.

This defaults to the Oracle semantics-checker front end, `oracle.sqlj.checker.OracleChecker`, which for SQLCheckerCustomizer chooses an appropriate online checker for your environment. For more information about OracleChecker, see "[Semantics-Checkers and the OracleChecker Front End \(default checker\)](#)" on page 8-71.

Following is a full command-line example, showing how to use the SQLCheckerCustomizer `checker` option, in conjunction with the customizer harness `verify` option and connection options.

```
sqlj -P-verify -P-user=scott -P-password=tiger -P-url=jdbc:oracle:oci:@  
-P-Cchecker=abc.def.MyChecker *.ser
```

(This is a single wraparound command line.)

Command-line syntax `-P-Cchecker=checker_class`

Command-line example `-P-Cchecker=a.b.c.MyChecker`

Properties file syntax `profile.Cchecker=checker_class`

Properties file example `profile.Cchecker=a.b.c.MyChecker`

Default value `oracle.sqlj.checker.OracleChecker`

SQLCheckerCustomizer Warnings Option (warn)

The `warn` option is equivalent to the SQLJ translator `-warn` option, allowing you to choose the categories of warnings and messages to be displayed as semantics-checking is performed on a profile.

For a complete description of the functionality and possible settings of these options, see "[Translator Warnings \(-warn\)](#)" on page 8-45.

This defaults to the `all,noverbose,noportable` settings, resulting in all warning categories except `verbose` and `portable` being enabled. You will receive any warnings regarding inheritance hierarchy requirements, data precision, conversion loss for nullable data, and strict matching for named iterators. These are the same defaults as for warnings during SQLJ translation.

Following is a full command-line example showing how to use the SQLCheckerCustomizer `warn` option, in conjunction with the customizer harness `verify` option and connection options. This would result in only portability warnings being displayed.

```
sqlj -P-verify -P-user=scott -P-password=tiger -P-url=jdbc:oracle:oci:@  
-P-Cwarn=none,portable *.ser
```

(This is a single wraparound command line.)

Command-line syntax `-P-Cwarn=comma-delimited_list_of_flags`

Command-line example `-P-Cwarn=none,verbose`

Properties file syntax `profile.Cwarn=comma-delimited_list_of_flags`

Properties file example `profile.Cwarn=none,verbose`

Default value `all,noverbose,noportable`

AuditorInstaller Customizer for Debugging

For ISO code generation, SQLJ provides a special customizer, `AuditorInstaller`. This customizer will insert sets of debugging statements, known as *auditors*, into profiles specified on the SQLJ command line. These profiles must already exist from previous customization.

The debugging statements will execute during SQLJ runtime (when someone runs your application), displaying a trace of method calls and values returned.

Use the customizer harness `debug` option, preceded by `-P-` as with any general customization option, to insert the debugging statements. (Syntax for this option is discussed in "[Invoking AuditorInstaller with the Customizer Harness debug Option](#)" on page A-45.)

Overview of Auditors and Code Layers

When an application is customized, the Oracle customizer implements profiles in *layers* of code (typically less than five) for different levels of runtime functionality. The deepest layer uses straight Oracle JDBC calls and implements any of your SQLJ statements that can be executed through JDBC functionality. Each higher layer is a specialized layer for some category of SQLJ functionality that is not supported by JDBC and so must be handled specially by the SQLJ runtime. For example, a layer for iterator conversion statements (`CAST`) is used to convert JDBC result sets to SQLJ iterators. Another layer is used for assignment statements (`SET`).

At 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 process it (usually the deepest layer, which executes all JDBC calls).

You can install debugging statements at only one layer during a single execution of `AuditorInstaller`. Each set of debugging statements installed at a particular layer of code is referred to as an individual *auditor*. During 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 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 your SQLJ executable statements that result in JDBC calls.

Use multiple executions of `AuditorInstaller` to install auditors at different levels. You might want to do that to install auditors at both the deepest layer and the shallowest layer, for example.

See "[AuditorInstaller Depth Option \(depth\)](#)" on page A-48 for information about how to specify the layer at which to install an auditor.

Invoking AuditorInstaller with the Customizer Harness debug Option

Following are examples of how to specify the Oracle customizer harness debug option to run `AuditorInstaller` in its default mode:

```
sqlj -P-debug Foo_SJProfile0.ser Bar_SJProfile0.ser
```

```
sqlj -P-debug *.ser
```

```
sqlj -P-debug myappjar.jar
```

The `debug` option results in the customizer harness instantiating and invoking the following class:

```
sqlj.runtime.profile.util.AuditorInstaller
```

This class performs the work of inserting the debugging statements.

The `-P-debug` option is equivalent to the following:

```
-P-customizer=sqlj.runtime.profile.util.AuditorInstaller
```

This overrides the customizer specified in the `SQLJ -default-customizer` option.

Be aware of the following:

- To run an application with auditors installed, the Oracle SQLJ file `translator.jar` (or `.zip`) must be in your classpath. (Normally, running a pre-translated SQLJ application requires only a runtime library.)
- As with any Oracle customizer, help output and an option list will be provided if you specify `-P-debug` together with `-P-help` on the SQLJ command line.
- It is important to realize that because the `debug` option invokes a customizer, and only one customizer can run in any single running of SQLJ, you cannot perform any other customization when you use this option.
- You also cannot use more than one of `-P-print`, `-P-debug`, and `-P-verify` simultaneously, because each of these invokes a specialized customizer.

Command-line syntax `sqlj -P-debug profile_list`

Command-line example `sqlj -P-debug Foo_SJProfile*.ser`

Properties file syntax `profile.debug`

(You must also specify profiles in the file list.)

Properties file example `profile.debug`

Default value `n/a`

AuditorInstaller Runtime Output

During runtime, debugging statements placed by AuditorInstaller result in a trace of methods called and values returned. This happens for all profile layers that had debugging statements installed. There is no means of selective debug output at 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` interface, containing equivalent attributes and methods.

AuditorInstaller Options

As with any customizer, `AuditorInstaller` has its own options that can be set using the `-P-C` prefix on the SQLJ command line (or `profile.C` in a SQLJ properties file).

`AuditorInstaller` supports the following options:

- `depth`—Specify how deeply you want to go into the layers of runtime functionality in your profiles.
- `log`—Specify the target file for runtime output of the debugging statements of the installed auditor.
- `prefix`—Specify a prefix for each line of runtime output that will result from this installation of debugging statements.
- `showReturns`—Enable the installed auditor to include return arguments in its runtime call tracing.
- `showThreads`—Enable the installed auditor to include thread names in its runtime call tracing (relevant only for multithreaded applications).
- `uninstall`—Remove the debugging statements placed into the profiles during the most recent previous invocation of `AuditorInstaller` on those profiles.

AuditorInstaller Depth Option (depth)

As discussed in "[Overview of Auditors and Code Layers](#)" on page A-44, `AuditorInstaller` can install a set of debugging statements, known as an auditor, at only a single layer of code during any one execution. The `AuditorInstaller` `depth` option 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; 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 File Option (log)

Use the `log` option to specify an output file for runtime output that will result from the auditor that you are currently installing. Otherwise, standard output will be used—debug output will go to wherever SQLJ messages go.

When auditors write messages to an output file, they append; they do not overwrite. Therefore, you can specify the same log file for multiple auditors without conflict. In fact, it is typical in this way to have debug information from all layers of your application go to the same log file.

Command-line syntax `-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 (prefix)

Use the `prefix` option to specify a prefix for each line of runtime output resulting from the debugging statements installed during this invocation of AuditorInstaller.

This option is often used in conjunction with the `depth` option. By running AuditorInstaller multiple times with different prefixes for different layers, you can easily see at runtime what information is coming from which layers.

Command-line syntax `-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 Return Arguments Option (showReturns)

Use the `showReturns` option to enable or disable the display of return arguments as part of the runtime call tracing. This is enabled by default.

The following few lines show sample output with `showReturns` enabled (default):

```
oracle.sqlj.runtime.OraRTStatement@3 . executeComplete ( )
oracle.sqlj.runtime.OraRTStatement@3 . executeComplete returned
oracle.sqlj.runtime.OraRTResultSet@6 . next ( )
oracle.sqlj.runtime.OraRTResultSet@6 . next returned true
oracle.sqlj.runtime.OraRTResultSet@6 . getBigDecimal ( 1 )
oracle.sqlj.runtime.OraRTResultSet@6 . getBigDecimal returned 5
```

```
oracle.sqlj.runtime.OraRTResultSet@6 . getDate ( 7 )
oracle.sqlj.runtime.OraRTResultSet@6 . getDate returned 1998-03-28
```

With showReturns disabled, the output would appear as follows:

```
oracle.sqlj.runtime.OraRTStatement@3 . executeComplete ( )
oracle.sqlj.runtime.OraRTResultSet@6 . next ( )
oracle.sqlj.runtime.OraRTResultSet@6 . getBigDecimal ( 1 )
oracle.sqlj.runtime.OraRTResultSet@6 . getDate ( 7 )
```

Instead of both a call line and a return line for each method call, there is only a call line.

Command-line syntax `-P-CshowReturns<=true|false>`

Command-line example `-P-CshowReturns=false`

Properties file syntax `profile.CshowReturns<=true|false>`

Properties file example `profile.CshowReturns=false`

Default value `true`

AuditorInstaller Thread Names Option (showThreads)

Use the `showThreads` option to enable or disable the display of thread names as part of the runtime call tracing (relevant only for multithreaded applications). This is disabled by default.

When this option is enabled, thread names prefix the method names in the trace output.

Command-line syntax `-P-CshowThreads<=true|false>`

Command-line example `-P-CshowThreads`

Properties file syntax `profile.CshowThreads<=true|false>`

Properties file example `profile.CshowThreads`

Default value `false`

AuditorInstaller Uninstall Option (uninstall)

Use the `uninstall` option to remove debugging statements placed during previous invocations of `AuditorInstaller`. Each time you use the `uninstall` option, it will remove the auditor most recently installed.

To remove all auditors from a profile, run `AuditorInstaller` repeatedly until you get a message indicating that the profile was unchanged.

Command-line syntax `-P-Cuninstall`

Command-line example `-P-Cuninstall`

Properties file syntax `profile.Cuninstall`

Properties file example `profile.Cuninstall`

Default value `disabled`

Full Command-Line Examples

Following are some full SQLJ command-line examples showing the specification of `AuditorInstaller` options.

Insert a set of debugging statements, or auditor, into the deepest layer (which is the default layer), with runtime output to standard output:

```
sqlj -P-debug MyApp_SJProfile*.ser
```

Insert an auditor into the deepest layer, with runtime output to `log.txt`:

```
sqlj -P-debug -P-Clog=foo/bar/log.txt MyApp_SJProfile*.ser
```

Insert an auditor into the deepest layer, with runtime output to standard output, showing thread names but not return arguments:

```
sqlj -P-debug -P-CshowThreads=true -P-CshowReturns=false MyApp_SJProfile*.ser
```

Insert an auditor into layer 0 (the shallowest layer). Send runtime output to `log.txt`; prefix each line of runtime output with "Layer 0: " (the following command is a single wraparound line):

```
sqlj -P-debug -P-Clog=foo/bar/log.txt -P-Cdepth=0 -P-Cprefix="Layer 0: "  
MyApp_SJProfile*.ser
```

Uninstall an auditor (this uninstalls the auditor most recently installed; do it repeatedly to uninstall all auditors):

```
sqlj -P-debug -P-Cuninstall MyApp_SJProfile*.ser
```

SQLJ Error Messages

This appendix lists error messages that might be output by the SQLJ translator and SQLJ runtime. Cause and action information is also provided, as well as the SQL state for runtime errors.

- [SQLJ Translation Time Messages](#)
- [SQLJ Runtime Messages](#)

Note: Error messages, causes, and actions in this appendix were copied directly from SQLJ source code without alteration.

SQLJ 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 for Translator Errors \(-explain\)](#)" on page 8-49.

<<<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 error occurred when determining result set column sizes: *message*

Cause: User specified the `-P-Coptcols` option. An error occurred when the profile customizer was trying to determine the types and sizes the columns in a result set columns.

Action: Check your SQL statement. You may want to perform connected translation to better determine the cause of the error.

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 load type map from resource *map name*.

Action: Ensure that the type map resource *map name* is present on the CLASSPATH.

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 retrieve type map for context class *context class*: error message

Cause: An error occurred when trying to retrieve a type map for the connection context class *context class*.

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 *classname* not found.

Cause: The program contained a reference to a class named *classname*. The class definition was not found in any source file currently being translated or in the classpath.

Action: Check the name of the class. Verify that it is defined either in class format in the classpath or in a source file passed to the translator.

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 has errors - unable to resolve method *method()*.

Cause: SQLJ translation is not able to determine if the Java class properly implements the oracle.sql.CustomDatum or java.sql.SQLData interface. This is due to an error in the Java class.

Action: Fix the problem in the Java class. You may want to compile it separately to obtain errors on the class.

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.

Class not found: *mesg*. The problem is likely due to the fact that either your program or the SQLJ runtime references javax.sql.DataSource.

Cause: You are probably using the WITH attribute "dataSource" on a connection context and/or a SQLJ runtime version, such as runtime12ee.zip, that is statically linked with javax.sql.DataSource.

Action: Ensure that the javax.sql.* and javax.naming.* packages are in your CLASSPATH. Or remove the "dataSource" attribute from the connection context declaration and do not use runtime12ee.zip.

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 command line 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 or NEXT, or PRIOR, or FIRST, or LAST, or ABSOLUTE, or RELATIVE.

Cause: A host variable representing an iterator type or a keyword 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 *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 *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 *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 *fileName* does not contain type *className* as expected. Please adjust the class path so that the file does not appear in the unnamed package.

Cause: Ensure that the class *className* is defined in file *fileName* that you pass to SQLJ.

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_.

Ignored type map entries: *entry list*.

Cause: One or more non-standard, non-portable entries on the connection context type map were found and ignored.

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 FETCH clause: expected *expected token or expression*.

Cause: A particular syntactic keyword or expression was expected in the FETCH clause.

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 *CustomDatum* or *SQLData* implementation in *type*: *mesg*

Cause: You are employing a user-defined Java type *type* that implements the `oracle.sql.CustomDatum` or the `java.sql.SQLData` 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 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 iterator declaration.

Cause: There is a syntax error in the SQL declaration.

Action: Check the SQL declaration syntax.

Invalid Java type *classname* in entry "*typemap entry*".

Cause: The Java classname must be the name of a valid Java class that is present in the Java environment.

Invalid Java type *java type* in *map* at entry "*entry*".

Cause: The type *java type* is not the name of a valid Java class.

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 *#position*: *classname*. Oracle SQL does not support this type.

Cause: The Oracle SQLJ runtime does not support writing instances of this type to the database.

Action: If the problem type is `oracle.sql.STRUCT`, `oracle.sql.REF`, or `oracle.sql.ARRAY`, you can use a JPublisher-generated wrapper class instead of the `oracle.XXX` type.

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 SQL type in entry "*entry*" of type map *type map message*.

Cause: The SQL type in entry *entry* was not given properly, or it has duplicate entries.

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 *iterator* must implement the interface *interface*.

Cause: Because of the motion command used on this iterator, it must implement the *interface* interface.

Action: Declare the iterator type as follows: #sql iterator *iterator* implements *interface* (...);

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 class *class* specified in *type map* does not implement *interface*.

Cause: According to the context type map *type map*, the class *class* must implement the interface *type map*. This is not the case.

Java class *class* specified in *type map* implements neither *interface1* nor *interface2*.

Cause: According to the context type map *type map*, the class *class* must implement interface *interface1* or it must implement interface *interface2*. This is not the case.

Java class *classname* in entry "*entry*" must implement *interface*

Cause: Instances of Java objects that are read from or written to the database must implement a particular Java interface.

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.

Must be connected online to perform optimization for result set columns.

Cause: User specified the -P-Coptcols option. The profile customizer must be able to log on to the database in order to determine the types and sizes of all result set columns.

Action: Specify connection information through the -P-user, -P-password, and -P-url options.

Must specify STRUCT or JAVA_OBJECT in entry "*entry*"

Cause: The SQL type to which Java class maps must either be a structured type (STRUCT xxx) or a SQL type capable of holding Java object instances (JAVA_OBJECT xxx).

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.

No variable *name* defined in class *classname*

Cause: A variable named *name* could not be found in class *classname*.

Action: Verify that the variable exists and is accessible in the named class.

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.

Option `optparamdefaults`: Invalid JDBC type in *size hint*

Cause: User specified the `-P-Coptparamdefaults` option which contains a comma-separated list of size hints of the form `<JDBC-type>(<number>)` or `<JDBC-type>()`. `<JDBC-type>` was not one of CHAR, VARCHAR, VARCHAR2, LONG, LONGVARCHAR, BINARY, RAW, VARBINARY, LONGVARBINARY, LONGRAW, or a wildcard `XXX%` matching one or more of these, or CHAR_TYPE, or RAW_TYPE.

Option `optparamdefaults`: Invalid or missing size indicator in *size hint*

Cause: User specified the `-P-Coptparamdefaults` option which contains a comma-separated list of size hints. One or more of the hints did not have the form `<JDBC-type>(<number>)` or `<JDBC-type>()`.

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.

Size designation *size hint* for parameter *param* ignored.

Cause: A size hint was given for parameter *param*. However, this parameter does not have a variable size type. Therefore the size hint will be ignored.

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).

SQLJ runtime library is missing. You need to provide *sqlj runtime library* on the CLASSPATH.

Cause: From version 8.1.7 on, the translator.zip library does not contain the SQLJ runtime classes any longer.

Action: Ensure that one of runtime.zip, runtime11.zip, or runtime12.zip is available on your CLASSPATH or via the -classpath option. Depending on your JDBC and Java environment, the error message suggests a particular runtime version.

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 keyword *non-portable keyword* for iterator movement is not portable - use *portable expression* instead.

Cause: The syntax used here is not part of the ISO SQLJ standard.

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 SQLJ runtime version requires JDK version 1.2 or later.

Cause: You are using `runtime12.zip` under JDK 1.1.x.

Action: Either run in a JDK 1.2 environment, or use a JDK 1.1.x compatible runtime, such as `runtime.zip` or `runtime11.zip`.

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 map *map* value at *key* is not a String.

Cause: You specified the type map resource *map* with your connection context. The entry for the key *key* is not an instance of java.lang.String.

Action: Ensure that every key maps to a non-null String value.

Type map *map* value at *key* is null.

Cause: You specified a type map resource *map* with your connection context. The entry for the key *key* is null.

Action: Ensure that every key maps to a non-null String value.

Type map *map*: inner Java class *java type* must be specified as *required type* at entry "*entry*"

Cause: When referencing an inner class in a type map, you wrote the class name the same way it would be written in Java source: <package name>.<outer class>.<inner class>. However, at runtime the Java VM will not be able to load this class with Class.forName.

Action: In the type map make reference to inner classes as follows: <package name>.<outer class>.\$<inner class>.

Type map resource *resource* appears to have the same name as a class. You should rename the resource.

Cause: The resource name *resource* coincides with an existing class name. This can cause problems when you run your program.

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 connect to data source "*data source*". Will attempt to use JDBC connection instead.

Cause: The connection context has the `dataSource` attribute value *data source*. Since the translator was unable to connect to this data source, it now attempts to use a JDBC connection instead.

Unable to convert *profile* to a class file.

Cause: SQLJ could not convert the profile file *profile* to a class file.

Action: Ensure that the profile file is present, that directory specified in the `-d` option is writable, and that the Java compiler is accessible.

Unable to create a connection context instance of *context name*: *message*.

Cause: The SQLJ customizer is unable to instantiate the connection context type *context name*.

Action: Ensure that the context class *context name* is declared as public and is available on the CLASSPATH. This is particularly important if this context has declared a type map.

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 determine type of WITH-clause attribute *name*: circular reference.

Cause: The value of the WITH-clause attribute *name* directly or indirectly referenced itself. The type of the attribute cannot be determined in such cases.

Action: Update the WITH-clause value so it does not refer to itself.

unable to find input file *filename*

Action: Ensure that file *filename* exists.

Unable to initialize system classes: *error*. This can be caused by a version mismatch between the SQLJ runtime and the Java environment.

Cause: The SQLJ runtime does not appear to be compatible with the Java environment.

Action: Under JDK 1.1.x use `runtime.zip` or `runtime11.zip`, under JDK 1.2 or later use (preferably) `runtime12.zip` or `runtime.zip`.

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 class file *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 obtain line mapping information from Java file *args*: *message*

Cause: SQLJ could not obtain line mapping information from the Java file *args* due to some error.

Action: Ensure that the Java file is present, that it is not corrupt, and that it is readable.

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 or value 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 and value 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.

Undefined variable or class name: *name*

Cause: The name *name* was used in an expression but did not correspond to any accessible variable or class name.

Action: Verify that the name refers to an accessible variable or class name.

Undefined variable, class, or package name: *name*

Cause: The name *name* was used in an expression but did not correspond to any accessible variable or class name.

Action: Verify that the name refers to an accessible variable or class name.

Undefined variable: *name*

Cause: The name *name* was used in an expression but did not correspond to any accessible variable.

Action: Verify that the name refers to an accessible variable.

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.

WITH attribute *attribute* must be of type *Java type expected*, not *Java type seen*.

Cause: You used a `WITH` attribute with your iterator or context declaration. The Java type of this attribute should be *Java type expected*. The actual type of the attribute, however, was *Java type seen*.

Action: Use the Java type *Java type expected* for this attribute.

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 a 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.

SQLJ Runtime Messages

This section provides a list of error messages that users may encounter from the SQLJ runtime, including SQL state, cause, and action information.

See ["Retrieving SQL States and Error Codes"](#) on page 4-24 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 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: closed connection

SQL State: 08000

Cause: A attempt was made to close a connection context object whose underlying JDBC connection object was already closed.

Action: Ensure that the underlying JDBC connection was not inadvertently closed. Also, if several SQLJ connection contexts share the same underlying JDBC connection, you have to ensure that when you close the connection context objects all but the last `close()` invokes the method `close(ConnectionContext.KEEP_CONNECTION)`. This ensures that the underlying JDBC connection remains open for the duration and all associated JDBC resources -such as JDBC statement objects- can be properly released.

java.sql.SQLException: could not establish connection to DataSource *data source name: message*

SQL State: 08003

Cause: Unable to establish a connection with data source *data source name*.

Action: Examine the message text *message* to determine the action required for connecting with data source *data source name*.

java.sql.SQLException: could not load context type map *type map resource: message*

SQL State: 08000

Cause: Either the type map could not be found or loaded from the resource *type map resource*, or the type map contained an invalid entry, or a Java class referenced in the type map could not be found.

Action: Examine the message to determine the nature of the failure, and how it could be remedied.

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 batch limit: *batch limit*

SQL State: 08000

Cause: The value *batch limit* is not a valid batch limit.

Action: Ensure that the batch limit is either non-negative or that it is the constant `sqlj.runtime.ExecutionContext.AUTO_BATCH`.

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: Java serialization not possible into SQL type: *code*

SQL State: 08000

Cause: The SQL type -as reflected in the OracleTypes code *code* is not suitable for serialization/deserialization of Java objects.

Action: Ensure that you serialize/deserialize Java objects into a SQL type that supports this, such as RAW or BLOB.

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: only fetch forward direction permitted

SQL State: 46110

Cause: This Oracle JDBC driver only supports FETCH FORWARD result sets.

Action: Update to an 8.1.6 or later JDBC driver with full support for scrollable result sets.

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: SQL operation currently in use

SQL State: 46000

java.sql.SQLException: streaming of BLOB data not supported in this driver

SQL State: 46110

Cause: Your JDBC driver does not support the streaming of raw data into BLOBS.

Action: Update to a newer Oracle JDBC driver version, or use a RAW column to hold serialized Java objects.

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.

A

access mode settings (transactions), 7-50
alternative environments, support, 8-80
applets
 using SQLJ, 1-25
ASENSITIVE (cursor state), 3-7
assignment statements (SET), 3-58
assumptions, environment, 2-2
AuditorInstaller
 command-line examples, A-51
 customizer for debugging, A-44
 invoking, A-45
 options, A-47
 runtime output, A-46
auditors in profiles for debugging, A-44
auto-commit
 modifying in existing connection, 4-28
 not supported in server, 11-5
 specifying in new connection, 4-27

B

backup option (customizer harness), A-14
backward compatibility
 Oracle SQLJ, general, 2-9
 to Oracle8i, 5-11
 to Oracle8/Oracle7, 5-12
batch updates
 batch limit, 10-19
 batchable and compatible statements, 10-13
 batching incompatible statements, 10-20
 canceling a batch, 10-17
 cautions, 10-22

 enabling and disabling, 10-14
 error conditions during execution, 10-22
 explicit and implicit batch execution, 10-14
 overview, 10-12
 update counts, 10-18
 using implicit execution contexts, 10-21
 with respect to recursive call-ins, 10-23
BetterDate (custom Java class), 6-19
BFILEs
 as stored function results, 5-34
 BFILE support, 5-30
BigDecimal
 mapping (for attributes), 6-38
 support, 5-42
binary portability of profiles, 1-7
bind-by-identifier option (sqlj
 -bind-by-identifier), 8-70
BLOB support, 5-30
BOOLEAN type (PL/SQL), 5-9
builtintypes option (JPublisher -builtintypes), 6-37

C

C prefix (sqlj -C-x), 8-62
cache option (sqlj -cache), 8-78
caching online checker results, 8-78
caching statements, 10-4
CALL syntax for stored procedures, 3-60
calling stored functions, 3-61
calling stored procedures, 3-60
calls to runtime, generated, 9-8
case option (JPublisher -case), 6-36
case-sensitive SQL UDT names, 6-13, 6-14, 6-34,
 6-43

- cause/action output for errors, 8-49
- CHAR comparisons, blank padding, 8-58, A-32
- character encoding
 - command line example, 9-25
 - for messages, 9-23
 - for source, 9-23
 - overview, 9-20
 - setting at runtime, 9-29
 - using native2ascii, 9-30
- check source name against. public class, 8-85
- check sources, expand resolution search, 8-69
- checker option (SQLCheckerCustomizer), A-42
- checkfilename option (sqlj -checkfilename), 8-85
- checksource option (sqlj -checksource), 8-69
- class loading in server, 11-7
- class schema object naming
 - generated, 11-23
 - loaded, 11-12
- classpath and path, 2-12
- classpath option (sqlj -classpath), 8-23
- clauses, SQLJ executable statements, 3-10
- client-side translation to run in server, 11-9
- CLOB support, 5-30
- close() method (DefaultContext), 4-17
- close() method (ExecutionContext), 7-32
- close() method (Oracle class), 4-14, 4-17
- CLOSE_CONNECTION, 7-57
- code generation
 - general information, 9-5
 - Oracle-specific vs. ISO standard, 4-39
 - translator -codegen option, 8-52
- code layers in profiles, A-44
- codegen option (SQLJ -codegen), 8-52
- collections
 - about custom Java classes, 6-6
 - creating collection types, 6-25
 - datatypes, 6-5
 - fundamentals, 6-4
 - introduction to collection support, 6-2
 - mapping to alternative classes, 6-41
 - ORADData specifications, 6-7
 - specifying type mapping, 6-34, 6-37
 - strongly typed, 6-62
 - weak types, restrictions, 6-81
 - weak types, support, 6-80
- column definitions (types/sizes)
 - general information, 10-23
 - Oracle customizer optcols option, A-27
 - SQLJ -optcols option, 8-53
- command line (translator)
 - echoing without executing, 8-16
 - example, 8-15
 - overview, 8-2
 - syntax and arguments, 8-12
- commit
 - automatic vs. manual, 4-26
 - effect on iterators and result sets, 4-29
 - manual, 4-28
 - modifying auto-commit in existing
 - connection, 4-28
 - specifying auto-commit in new connection, 4-27
- compat(ibility) option (Oracle customizer), A-25
- compatible option (JPublisher -compatible), 6-33
- compilation
 - compiling in two passes, 8-86
 - debug option in server, 11-20
 - during translation, 9-9
 - enabling/disabling, 8-65
 - in server, 11-7
- compile option (sqlj -compile), 8-65
- compiler
 - classpath option, 8-23
 - option support for javac, 8-10
 - options through SQLJ, 8-62
 - related options, 8-80
 - required behavior, 8-82
 - specifying name, 8-82
- compiler encoding support option (sqlj), 8-83
- compiler executable option (sqlj), 8-82
- compiler message output pipe option (sqlj), 8-84
- compiler output file option (sqlj -compiler...), 8-83
- configuration and installation verification, 2-11
- connect() method (Oracle class), 4-13
- connection contexts
 - close connection, 7-9
 - concepts, 7-2
 - converting from JDBC connection, 7-56
 - converting to JDBC connection, 7-53
 - declaration with IMPLEMENTS clause, 7-11
 - declarations, 3-4

- declaring connection context class, 7-5
- from SQLJ data sources, 7-16, 7-20
- get default connection, 7-10
- get execution context, 7-9
- get JDBC connection, 7-9
- implementation and functionality, 7-9
- instantiating connection object, 7-6
- methods, 7-9
- multiple connections, example, 7-7
- relation to execution contexts, 7-25
- semantics-checking, 7-12
- set default connection, 7-10
- specifying connection for statement, 7-7
- specifying for executable statement, 3-11
- connections
 - closing, 4-11
 - closing shared connections with JDBC, 7-57
 - database connection in server, 11-4
 - from SQLJ data sources, 7-16, 7-20
 - JDBC transaction methods, 7-51
 - modifying auto-commit, 4-28
 - multiple, using declared connect contexts, 4-12
 - Oracle class to connect, 4-12
 - set up, 2-14
 - shared connections with JDBC, 7-56
 - single or multiple using default context, 4-6
 - specifying auto-commit, 4-27
 - translator options, 8-34
 - verify, 2-16
- context expressions
 - evaluation at runtime, 3-22
 - overview, 3-21
- context option (customizer harness), A-14
- converting .ser profiles to .class, 8-68
- CURSOR syntax (nested tables), 6-62
- custom Java classes
 - about custom Java classes, 6-6
 - compiling, 6-17
 - creation by JPublisher, 6-28
 - examples, 6-47
 - extending, 6-51
 - generation by JPublisher, 6-32
 - mapping to alternative classes, 6-41
 - reading and writing data, 6-17
 - requirements, 6-11
 - sample class, 6-19
 - specifying member names, 6-45
 - strongly typed, definition, 6-2
 - support for object methods, 6-10
 - using to serialize object, 6-74
 - weakly typed, definition, 6-2
- CustomDatum (deprecated), 6-8
- customization
 - converting .ser profiles to .class, 8-68
 - creation and registration, A-7
 - customizer harness connection options, A-18
 - customizer harness general options, A-14
 - customizer harness options overview, A-12
 - defining column types/sizes, A-27
 - defining parameter sizes, A-29
 - during translation, 9-10
 - enabling/disabling, 8-66
 - error and status messages, A-9
 - force customization, A-26
 - jar file usage, A-38
 - more about customization, A-5
 - options, A-11
 - options to invoke special customizers, A-21
 - Oracle customizer options, A-24
 - overview/syntax of customizer-specific options, A-23
 - parameter default sizes, A-30
 - related SQLJ options, A-37
 - show SQL transformations, A-32
 - statement cache size, A-34
 - steps in process, A-6
 - summary of Oracle features used, A-35
 - version compatibility, A-25
- customizer harness
 - connection options, A-18
 - general options, A-14
 - invoke special customizers, A-21
 - options overview, A-12
 - overview, A-5
- customizer option (customizer harness), A-15
- customizers
 - choosing, A-11
 - option to choose customizer, A-15
 - overview, A-5
 - passing options through SQLJ, 8-63

specifying default, 8-88

D

d option (sqlj -d), 8-30

data source support

 associating a connection, 7-13

 associating a default context, 7-15

 auto-commit mode, 7-14

 dataSource (connection context WITH clause), 3-7

 general overview, 7-13

 requirements, 7-15

 SQLJ data source classes, 7-17

 SQLJ data source interfaces, 7-16

 SQLJ-specific data sources, 7-16

database connection, verify, 2-16

DBMS_JAVA package

 set server output device, 11-6

 set server-side options, 11-20

DBMS_LOB package, 5-31

debug option (customizer harness), A-21

debug option for compile (in server), 11-20

debugging

 AuditorInstaller command-line examples, A-51

 AuditorInstaller customizer, A-44

 AuditorInstaller options, A-47

 AuditorInstaller runtime output, A-46

 debug option for compile (in server), 11-20

 debug option, customizer harness, A-21

 in JDeveloper, 10-29

 invoking AuditorInstaller, A-45

 line-mapping, SQLJ source to class, 8-49

 line-mapping, SQLJ source to class for jdb, 8-50

declarations

 connection context declarations, 3-4

 IMPLEMENTS clause, 3-5

 iterator declarations, 3-3

 overview, 3-2

 WITH clause, 3-6

default connection

 setting with Oracle.connect(), 4-6

 setting with setDefaultContext(), 4-10

default customizer option (sqlj), 8-88

default output device in server, 11-6

default properties files (translator), 8-19

default semantics-checker, 8-71

default URL prefix option (sqlj), 8-43

DefaultContext class

 close() method parameters, 4-17

 constructors, 4-15

 key methods, 4-15

 use for single or multiple connections, 4-6

defining column types/sizes, 10-23

defining parameter sizes, 10-25

depth option (AuditorInstaller), A-48

digests option, jar (customizer harness), A-15

dir option (sqlj -dir), 8-32

directory

 for generated .class and .ser, 8-30

 for generated .java, 8-32

dirty reads, 7-50

driver option (customizer harness), A-20

driver registration option (sqlj -driver), 8-44

dropjava, 11-25

dropping Java schema objects, 11-25

dynamic SQL

 defined, 1-2

 in JDBC code, 7-53

 in PL/SQL code, 3-14

dynamic SQL support in SQLJ

 examples, 7-65

 introduction, 7-63

 meta bind expressions, 7-63

 runtime behavior, 7-65

 translation-time behavior, 7-65

E

echo option, without execution, 8-27

echoing command line without executing, 8-16

encoding

 character encoding for messages, 9-23

 character encoding for source, 9-23

 command line example, 9-25

 do not pass option to compiler, 8-83

 overview of character encoding, 9-20

 setting at runtime, 9-29

 specifying in server, 11-19

 using native2ascii, 9-30

- encoding option (in server), 11-19
- encoding option, source files (sqlj -encoding), 8-29
- environment assumptions and requirements, 2-2
- environment variable, translator options, 8-19
- environments--scenarios and limitations, 2-4
- errors
 - character encoding for messages, 9-23
 - customization messages, A-9
 - messages, codes, and SQL states, 4-24
 - outputting cause and action, 8-49
 - runtime categories, 9-18
 - runtime error list, B-47
 - server-side error output, 11-24
 - translator error list, B-2
 - translator error, warning, info messages, 9-12
- exceptions
 - exception-handling requirements, 4-22
 - processing, 4-23
 - set up exception-handling, 4-32
 - using SQLException subclasses, 4-25
- executable statements
 - examples, 3-12
 - overview, 3-9
 - rules, 3-9
 - specifying connection/execution contexts, 3-11
 - SQLJ clauses, 3-10
 - using PL/SQL blocks, 3-14
- execution contexts
 - cancellation method, 7-30
 - close() method, 7-32
 - control methods, 7-28
 - creating and specifying, 7-26
 - method usage, example, 7-33
 - overview, 7-24
 - relation to connection contexts, 7-25
 - relation to multithreading, 7-33
 - savepoint methods, 7-31
 - specifying for executable statement, 3-11
 - status methods, 7-28
 - synchronization, 7-27
 - update-batching methods, 7-30
- exemplar schema, 4-17
- exit codes, translator, 9-15
- explain option (sqlj -explain), 8-49
- extending JPub-generated classes, 6-51

- extensions
 - overview, 1-8
 - performance extensions, 10-2
 - summary of features used, A-35
 - type extensions, 5-29

F

- FETCH CURRENT syntax (iterators), 7-46
- FETCH syntax (scrollable positional iterators), 7-45
- file name requirements and restrictions, 4-47
- fixedchar option (Oracle customizer), A-32
- fixedchar option (SQLJ -fixedchar), 8-58
- flags for special processing, 8-65
- force option (Oracle customizer), A-26
- ForUpdate/updateColumns (WITH clause), 3-7
- full names (schema names), 11-11
- function calls, stored, 3-61

G

- getConnection() method (Oracle class), 4-13
- globalization support
 - character encoding, language support, 9-19
 - outside of SQLJ, 9-29
 - overview, 1-32
 - related datatypes, 5-5
 - related Java types, 9-26
 - related SQLJ and Java settings, 9-22
 - support for Unicode characters, 9-25

H

- help option (customizer harness), A-16
- help options (sqlj -help-xxxx), 8-24
- hints in code, parameter sizes, 10-26
- holdability (cursor states, WITH clause), 3-7
- host expressions
 - basic syntax, 3-17
 - bind by identifier, 8-70
 - evaluation at runtime, 3-22
 - examples, 3-20
 - examples of evaluation at runtime, 3-24
 - iterators and result sets as host variables, 3-52
 - overview, 3-16

- restrictions, 3-33
- selecting a nested table, 6-64
- supported types for JDBC 2.0, 5-8
- type support for Oracle8i, 5-11
- type support for Oracle8/Oracle7, 5-12
- type support summary, 5-2

I

- IDE SQLJ integration, 1-32
- IMPLEMENTS clause
 - in connection context declarations, 7-11
 - in iterator declarations, 7-40
 - syntax, 3-5
- importing required classes, 4-31
- informational messages, translator, 9-12
- input to translator, 1-13
- INSENSITIVE (cursor state), 3-7
- installation and configuration verification, 2-11
- instrumenting class file (linemap), 8-50
- interoperability with JDBC
 - connection contexts and connections, 7-53
 - iterators and result sets, 7-58
- introduction to SQLJ, 1-2
- isolation level settings (transactions), 7-50
- iterators
 - accessing named iterators, 3-46
 - accessing positional iterators, 3-49
 - as host variables, 3-52
 - as iterator columns (nested), 3-55
 - as stored function returns, 3-63
 - commit/rollback effect, 4-29
 - concepts, 3-37
 - converting from result sets, 7-58
 - converting to result sets, 7-60
 - declarations, 3-3
 - declaring named iterators, 3-44
 - declaring positional iterators, 3-48
 - declaring with IMPLEMENTS clause, 7-40
 - general steps in using, 3-41
 - instantiating/populating named iterators, 3-45
 - instantiating/populating positional iterators, 3-49
 - iterator class functionality, 7-38
 - named vs. positional, 3-42

- nested iterators for nested tables, 6-67
- overview, 3-37
- positional iterators, using next(), 3-51
- result set iterators (weakly typed), 3-41, 7-41
- scrollable, 7-42
- scrollable result set iterators, 7-47
- selecting objects and references, 6-55
- set up named iterator (example), 4-35
- subclassing, 7-40
- using named iterators, 3-43
- using positional iterators, 3-48
- using weakly typed iterators, 7-61
- with serialized objects, 6-77

J

- J prefix (sqlj -J-x), 8-60
- jar file digests option, customization, A-15
- jar files for profiles, A-38
- Java bind expressions (dynamic SQL), 7-64
- Java names vs. SQL names in server, 11-8
- Java properties, getProperty(), 9-29
- Java VM
 - classpath option, 8-23
 - options through SQLJ, 8-60
 - specifying name, 8-81
- JavaBeans for SQLJ connections, 7-20
- javac compatibility, 8-10
- JDBC 2.0
 - support for LOB types, 5-29
 - support for weakly typed Struct, Ref, Array, 6-80
 - types supported, 5-8
- JDBC connection methods (transactions), 7-51
- JDBC considerations in server, 11-5
- JDBC driver registration option (sqlj -driver), 8-44
- JDBC drivers
 - Oracle drivers, 4-2
 - select for translation, 4-4
 - select/register for customization, A-20
 - select/register for runtime, 4-5
 - verify, 2-16
- JDBC interoperability
 - connection contexts and connections, 7-53
 - iterators and result sets, 7-58

JDBC mapping (for attributes), 6-37
JDBC vs. SQLJ, sample application, 1-18
jdblinemap option (sqlj -jdblinemap), 8-50
JDeveloper
 debugging with, 10-29
 SQLJ integration, 1-32
JDK
 appropriate JDBC class files, 2-12
 supported versions, 2-4
JNDI
 name of default data source, 7-15
 use for data sources, connections, 7-13
JPublisher
 builtintypes option, 6-37
 case option, 6-36
 compatible option, 6-33
 creation of custom Java classes, 6-28
 custom Java class examples, 6-47
 extending generated classes, 6-51
 generating custom Java classes, 6-32
 generating wrapper methods, 6-39
 implementation of method wrappers, 6-46
 input files, 6-42
 lobtypes option, 6-37
 mapping to alternative classes, 6-41
 numbertypes option, 6-37
 properties files, 6-44
 specifying member names, 6-45
 specifying type mapping, 6-34
 sql option, 6-34
 type categories and mapping options, 6-37
 type mapping, 6-37
 type mapping modes and option settings, 6-37
 types option, 6-34
 user option, 6-35
 what JPublisher produces, 6-29

K

KEEP_CONNECTION, 7-57

L

language support (globalization support), 9-22
linemap option (sqlj -linemap), 8-49

line-mapping
 SQLJ source to class file, 8-49
 SQLJ source to class for jdb, 8-50
loading classes/resources into server, 11-9
loading/translating source in server, 11-16
loadjava
 compatibility options, SQLJ, 8-9
 loading classes/resources, 11-9
 loading source, translating, 11-16
 output from loading source, 11-22
LOBs
 as iterator columns, 5-36
 as stored function results, 5-34
 FETCH INTO LOB host variables, 5-36
 SELECT INTO LOB host variables, 5-35
 support (oracle.sql and DBMS_LOB), 5-30
lobtypes option (JPublisher -lobtypes), 6-37
locale
 command line example, 9-25
 for messages, 9-23
 setting at runtime, 9-29
log option (AuditorInstaller), A-48

M

mapping to alternative classes (UDTs), 6-41
member names (objects), 6-45
message pipe, compiler, 8-84
meta bind expressions (dynamic SQL), 7-63
method support for objects, 6-10
method wrappers (JPub), implementation, 6-46
middle-tier considerations, 4-48
multithreading
 in server, 11-26
 in SQLJ, overview, 7-35
 relation to execution contexts, 7-33
 sample application, 7-36

N

n option (sqlj -n) (echo without execution), 8-27
name of compiler, 8-82
name of Java VM, 8-81
named iterators
 accessing, 3-46

- declaring, 3-44
 - instantiating and populating, 3-45
 - scrollable, 7-44
 - using, 3-43
 - naming requirements and restrictions
 - file names, 4-47
 - local variables, classes (Java namespace), 4-45
 - SQL namespace, 4-47
 - SQLJ namespace, 4-46
 - naming schema objects
 - generated class, 11-23
 - loaded classes, 11-12
 - loaded resources, 11-12
 - source, 11-22
 - National Language Support--see Globalization Support
 - native2ascii for encoding, 9-30
 - NCHAR class (globalization support), 9-25
 - NcharAsciiStream class (globalization support), 9-25
 - NcharUnicodeStream class (globalization support), 9-25
 - NCLOB class (globalization support), 9-25
 - nested iterators, 6-67
 - nested tables
 - accessing, 6-62
 - inserting in SQLJ, 6-63
 - manipulating, 6-65
 - selecting into host expression, 6-64
 - types, 6-4
 - using nested iterator, 6-67
 - NLS--see Globalization Support
 - non-repeatable reads, 7-50
 - NString class (globalization support), 9-25
 - null-handling
 - examples, 4-20
 - wrapper classes for null-handling, 4-19
 - numbertypes option (JPublisher
 - numbertypes), 6-37
- O**
-
- object method wrappers (JPub), 6-46
 - object references
 - selecting into iterators, 6-55
 - strongly typed in SQLJ, 6-55
 - updating in SQLJ, 6-60
 - weak types, restrictions, 6-81
 - weak types, support, 6-80
 - object-JDBC mapping (for attributes), 6-37
 - objects
 - about custom Java classes, 6-6
 - creating object types, 6-23
 - datatypes, 6-5
 - fundamentals, 6-4
 - inserting in SQLJ, 6-59
 - introduction to object support, 6-2
 - mapping to alternative classes, 6-41
 - method support, 6-10
 - ORADATA specifications, 6-7
 - selecting into iterators, 6-55
 - serializing (overview), 6-71
 - serializing RAW and BLOB columns, 6-71
 - serializing with custom Java class, 6-74
 - specifying type mapping, 6-34, 6-37
 - SQLData specifications, 6-9
 - strongly typed in SQLJ, 6-55
 - updating a reference in SQLJ, 6-60
 - updating in SQLJ, 6-57
 - weak types, restrictions, 6-81
 - weak types, support, 6-80
 - wrapper methods, 6-39
 - OCI driver (JDBC), 4-3
 - offline checking
 - default checker, Oracle checkers, 8-71
 - specifying checker, 8-75
 - offline option (sqlj -offline), 8-75
 - offline parsing
 - sqlj -parse option, 8-79
 - steps involved, 9-2
 - vs. online checking, 8-73
 - online checking
 - caching results, 8-78
 - default checker, Oracle checkers, 8-71
 - enabling in server, 11-19
 - enabling, setting user schema, 8-34
 - registering drivers, 8-44
 - setting default URL prefix, 8-43
 - setting password, 8-39
 - setting URL, 8-41

- specifying checker, 8-76
 - vs. offline parsing, 8-73
- online option (in server), 11-19
- online option (sqlj -online), 8-76
- opaque types, 6-82
- optcols option (Oracle customizer), A-27
- optcols option (SQLJ -optcols), 8-53
- optimizer, SQL, 10-2
- options (translator)
 - command line only, 8-22
 - flags for special processing, 8-65
 - for connections, 8-34
 - for customization, 8-87
 - for javac compatibility, 8-10
 - for loadjava compatibility, 8-9
 - for output files and directories, 8-29
 - for reporting and line-mapping, 8-45
 - for semantics-checking, offline parsing, 8-71
 - for VM and compiler, 8-80
 - help, 8-24
 - order of precedence, 8-20
 - overview, 8-3
 - prefixes for passing options, 8-60
 - summary list, 8-3
 - support for alternative environments, 8-80
- options for customizer harness
 - connection options, A-18
 - general options, A-14
 - invoke special customizers, A-21
 - overview, A-12
- options for Oracle customizer, A-24
- options for translation in server
 - fixed settings, 11-18
 - setting options, 11-20
 - supported options, 11-18
- optparamdefaults option (Oracle customizer), A-30
- optparamdefaults option (SQLJ -optparamdefaults), 8-56
- optparams option (Oracle customizer), A-29
- optparams option (SQLJ -optparams), 8-55
- Oracle class
 - close() method parameters, 4-14
 - connect() method, 4-13
 - for DefaultContext instances, 4-12
 - getConnection() method, 4-13
- Oracle customizer
 - blank padding for CHAR comparisons, A-32
 - define column types/sizes, A-27
 - define parameter sizes, A-29
 - force customization, A-26
 - options, A-24
 - set default parameter sizes, A-30
 - show SQL transformation, A-32
 - statement cache size, A-34
 - summary of Oracle features used, A-35
 - version compatibility, A-25
- Oracle extensions
 - overview, 1-8
 - performance extensions, 10-2
 - summary of features used, A-35
 - type extensions, 5-29
- Oracle mapping (for attributes), 6-37
- Oracle optimizer, 10-2
- Oracle9i Lite with SQLJ, 1-30
- OracleChecker default checker, 8-71
- Oracle-specific code generation
 - advantages and disadvantages, 4-39
 - coding considerations, limitations, 4-41
 - environment requirements, 4-40
 - introduction, 1-3, 4-39
 - server-side considerations, 4-44
 - translator/customizer usage changes, 4-42
- oracle.sql package, 5-30
- ORADData
 - additional uses, 6-18
 - specifications, 6-7
 - use in custom Java classes, 6-6
 - versus CustomDatum, 6-8
- output device in server, default, 11-6
- output directory
 - for generated .class and .ser, 8-30
 - for generated .java, 8-32
- output file and directory options (translator), 8-29
- output file for compiler, 8-83
- output from server-side translator, 11-22
- output from translator, 1-13
- output pipe, compiler messages, 8-84
- output, server-side translator errors, 11-24

P

- P prefix (sqlj -P-x), 8-63
- parameter definitions (sizes)
 - general information, 10-25
 - Oracle customizer optparamdefaults option, A-30
 - Oracle customizer optparams option, A-29
 - SQLJ -optparamdefaults option, 8-56
 - SQLJ -optparams option, 8-55
- parse option (sqlj -parse), 8-79
- passes option (sqlj -passes), 8-86
- passes, two-pass compiling, 8-86
- passing options to other executables, 8-60
- password option (customizer harness), A-19
- password option for checking (sqlj), 8-39
- path (connection context WITH clause), 3-8
- path and classpath, 2-12
- performance enhancements, 10-2
- phantom reads, 7-50
- pipe, compiler output messages, 8-84
- PL/SQL
 - blocks in executable statements, 3-14
 - BOOLEAN type, 5-9
 - RECORD type, 5-9
 - TABLE type, 5-9
- positional iterators
 - accessing, 3-49
 - declaring, 3-48
 - instantiating and populating, 3-49
 - navigation with next(), 3-51
 - scrollable, 7-45
 - using, 3-48
- positioned delete, 5-38
- positioned update, 5-38
- prefetching rows, 10-3
- prefix option (AuditorInstaller), A-49
- prefixes
 - to pass options to customizer, 8-63
 - to pass options to Java compiler, 8-62
 - to pass options to Java VM, 8-60
- print option (customizer harness), A-22
- procedure calls, stored, 3-60
- profile customization (see customization), 9-10
- profile option (sqlj -profile), 8-66

- profile-keys class, 9-5
- profiles
 - auditors for debugging, A-44
 - binary portability, 1-7
 - code layers, A-44
 - creation during code generation, A-2
 - debug option, A-21
 - functionality at runtime, A-9
 - generated profiles, 9-6
 - more about profiles, A-2
 - overview, 1-6
 - print option, A-22
 - sample profile entry, A-3
 - use of jar files, A-38
 - verify option, A-22
- properties files (translator)
 - default properties files, 8-19
 - overview, 8-16
 - setting input file, 8-23
 - syntax, 8-17
- properties, Java, getProperty(), 9-29
- props option (sqlj -props), 8-23
- public class name / source name check, 8-85

R

- READ COMMITTED transactions, 7-50
- READ ONLY transactions, 7-50
- READ UNCOMMITTED transactions, 7-50
- READ WRITE transactions, 7-50
- RECORD type (PL/SQL), 5-9
- recursive SQLJ calls in server, 11-26
- REF CURSOR
 - about REF CURSOR types, 5-40
 - example, 5-41
 - SQLJ support, 5-41
- register JDBC drivers
 - for runtime, 4-5
 - for translation, 8-44
- registering column types/sizes, 10-23
- registering parameter sizes, 10-25
- REPEATABLE READ transactions, 7-50
- reporting options (translator), 8-45
- requirements, environment, 2-3
- resource schema object naming

- loaded, 11-12
- result expressions
 - evaluation at runtime, 3-22
 - overview, 3-21
- result set iterators (weakly typed)
 - general information, 7-41
 - introduction, 3-41
 - scrollable, 7-47
- result sets
 - as host variables, 3-52
 - as iterator columns, 3-55
 - as stored function returns, 3-63
 - commit/rollback effect, 4-29
 - converting from iterators, 7-60
 - converting to iterators, 7-58
 - persistence across calls in server, 11-5
- ResultSetIterator type, 7-41
- returnability (cursor states, WITH clause), 3-7
- rollback
 - effect on iterators and result sets, 4-29
 - manual, 4-28
 - with savepoint, 4-30
- row prefetching, 10-3
- ROWID
 - as stored function results, 5-39
 - FETCH INTO ROWID host variable, 5-39
 - SELECT INTO ROWID host variable, 5-39
 - support, 5-37
- runtime
 - categories of errors, 9-18
 - debugging output (AuditorInstaller), A-46
 - error list, B-47
 - functionality, 9-16
 - functionality of profiles, A-9
 - generated calls to runtime, 9-8
 - globalization support, 9-19
 - JDBC driver selection and registration, 4-5
 - overview, 1-5
 - packages, 9-16
 - set up connection, 2-14
 - steps in runtime processing, 1-16
 - test, 2-17

S

- sample applications
 - JDBC vs. SQLJ, 1-18
 - multiple connection contexts, 7-7
 - multiple-row query (named iterator), 4-36
 - multithreading, 7-36
 - single-row query (SELECT INTO), 4-34
- sample classes
 - custom Java class (BetterDate), 6-19
 - SerializableDatum class, 6-78
- savepoints
 - ExecutionContext savepoint methods, 7-31
 - savepoint statements, 4-30
- schema objects
 - naming generated classes, 11-23
 - naming loaded classes, 11-12
 - naming loaded resources, 11-12
 - naming sources, 11-22
- scrollable iterators
 - declaring, 7-42
 - scrollable named iterators, 7-44
 - scrollable positional iterators, 7-45
 - sensitivity, 7-42
 - the scrollable interface, 7-43
- ScrollableResultSetIterator type, 7-47
- SELECT INTO statements
 - error conditions, 3-36
 - examples, 3-35
 - syntax, 3-34
- semantics-checking
 - caching online results, 8-78
 - default checker, Oracle checkers, 8-71
 - enabling online in server, 11-19
 - enabling online, setting user schema, 8-34
 - invoking SQLCheckerCustomizer, A-40
 - of profiles, via customizer harness, A-22
 - options, 8-71
 - registering drivers, 8-44
 - setting default URL prefix, 8-43
 - setting password, 8-39
 - setting URL, 8-41
 - specifying offline checker, 8-75
 - specifying online checker, 8-76
 - SQLCheckerCustomizer options, A-41

- steps involved, 9-2
- SENSITIVE (cursor state), 3-7
- sensitivity (cursor states, WITH clause), 3-7
- ser profiles (.ser)
 - converting to .class, 8-68
 - generated profiles, 9-6
- ser2class option (sqlj -ser2class), 8-68
- SERIALIZABLE transactions, 7-50
- serialized objects
 - as host variables, 6-77
 - in iterator columns, 6-77
 - overview, 6-71
 - SerializableDatum class (sample), 6-78
 - through custom Java class, 6-74
 - to RAW and BLOB columns, 6-71
- server-side internal driver (JDBC), 4-4
- server-side SQLJ
 - class loading, 11-7
 - coding considerations, 11-4
 - compilation, 11-7
 - connection to database, 11-4
 - default output device, 11-6
 - dropjava, 11-25
 - dropping Java schema objects, 11-25
 - error output, 11-24
 - fixed settings, 11-18
 - generated output from translation, 11-22
 - introduction, 11-2
 - Java multithreading, 11-26
 - JDBC differences, 11-5
 - loading classes/resources into server, 11-9
 - loading source into server, translating, 11-16
 - naming generated class schema objects, 11-23
 - naming loaded class schema objects, 11-12
 - naming loaded resource schema objects, 11-12
 - naming source schema objects, 11-22
 - options, 11-18
 - overview, 1-29
 - recursive calls, 11-26
 - running client program in server, 11-14
 - setting options, 11-20
 - SQL names vs. Java names, 11-8
 - translating in server, 11-16
 - translating on client, 11-9
 - verifying code is running in server, 11-28
 - server-side Thin driver (JDBC), 4-3
 - SET (assignment) statements, 3-58
 - SET TRANSACTION syntax, 7-49
 - setup of SQLJ, testing, 2-14
 - short names (schema names), 11-11
 - showReturns option (AuditorInstaller), A-49
 - showSQL option (Oracle customizer), A-32
 - showThreads option (AuditorInstaller), A-50
 - source check for type resolution, 8-69
 - source file line-mapping
 - for jdb, 8-50
 - general, 8-49
 - source files encoding option, 8-29
 - source name / public class name check, 8-85
 - source schema object naming, 11-22
 - SQL names vs. Java names in server, 11-8
 - SQL optimizer, 10-2
 - sql option (JPublisher -sql), 6-34
 - SQL replacement code (dynamic SQL), 7-65
 - SQL states (for errors), 4-24
 - SQLCheckerCustomizer
 - for semantics-checking of profiles, A-40
 - invoking, A-40
 - options, A-41
 - SQLData
 - specifications, 6-9
 - use in custom Java classes, 6-6
 - SQLException subclasses, using, 4-25
 - SQLJ vs. JDBC, sample application, 1-18
 - SQLJ_OPTIONS environment variable, 8-19
 - SqljConnBean for simple connection, 7-21
 - SqljConnCacheBean for connection caching, 7-22
 - sqljutil package, 2-13
 - statement caching, 10-4
 - static SQL, defined, 1-2
 - status messages
 - for customization, A-9
 - for translation, 9-15
 - translator, enabling/disabling, 8-48
 - status option (sqlj -status), 8-48
 - stmtcache option (Oracle customizer), A-34
 - stored function calls, 3-61
 - stored procedure calls, 3-60
 - streams
 - as function return values, 5-28

- as output parameters, 5-27
- classes and methods, 5-22
- examples, 5-24
- general use in SQLJ, 5-14
- precautions, 5-19
- retrieving data, 5-20
- sending data to database, 5-16
- supporting classes, 5-14
- strongly typed collections, 6-62
- strongly typed custom Java classes, 6-2
- strongly typed objects and references, 6-55
- subclassing iterator classes, 7-40
- summary option (Oracle customizer), A-35
- Sun JDK
 - appropriate JDBC class files, 2-12
 - supported versions, 2-4
- synchronization of execution contexts, 7-27
- syntax
 - translator command line, 8-12
 - translator properties files, 8-17

T

- TABLE syntax (nested tables), 6-62, 6-65
- TABLE type (PL/SQL), 5-9
- Thin driver (JDBC), 4-3
- transactions
 - access mode settings, 7-50
 - advanced transaction control, 7-49
 - automatic commit vs. manual commit, 4-26
 - basic transaction control, 4-26
 - isolation level settings, 7-50
 - JDBC Connection methods, 7-51
 - manual commit and rollback, 4-28
 - modifying auto-commit, 4-28
 - overview, 4-26
 - savepoints for rollbacks, 4-30
 - specifying auto-commit, 4-27
- transformGroup (connection context WITH clause), 3-8
- TRANSLATE (object member names), 6-45
- translating in server to run in server, 11-16
- translating on client to run in server, 11-9
- translator
 - basic translation steps, 1-10

- code generation, 9-5
- compilation, 9-9
- customization, 9-10
- error list, B-2
- error, warning, info messages, 9-12
- exit codes, 9-15
- globalization support, 9-19
- input and output, 1-13
- internal operations, 9-2
- Java and SQLJ code-parsing,
 - syntax-checking, 9-2
- output, server-side, 11-22
- overview, 1-5
- SQL semantics-checking and offline parsing, 9-2
- status messages, 9-15
- support for alternative environments, 8-80
- test, 2-17

Type, 5-1

type extensions, 5-29

type mapping

- BigDecimal mapping, 6-38

- JDBC mapping, 6-37

- JPublisher mapping option, 6-34

- object JDBC mapping, 6-37

- Oracle mapping, 6-37

- type categories and mapping modes, 6-37

type resolution, expand search, 8-69

typeMap (connection context WITH clause), 3-7

types option (JPublisher -types), 6-34

types supported

- for JDBC 2.0, 5-8

- for Oracle8i, 5-11

- for Oracle8/Oracle7, 5-12

- summary of types, 5-2

U

uninstall option (AuditorInstaller), A-51

update batching

- batch limit, 10-19

- batchable and compatible statements, 10-13

- batching incompatible statements, 10-20

- canceling a batch, 10-17

- cautions, 10-22

- enabling and disabling, 10-14

- error conditions during execution, 10-22
- explicit and implicit batch execution, 10-14
- overview, 10-12
- update counts, 10-18
- using implicit execution contexts, 10-21
- with respect to recursive call-ins, 10-23
- updateColumns/ForUpdate (WITH clause), 3-7
- url option (customizer harness), A-20
- url option for checking (sqlj -url), 8-41
- URL, default prefix for online checking, 8-43
- user option (customizer harness), A-18
- user option (JPublisher -user), 6-35
- user option for checking (sqlj -user), 8-34
- user-defined types, 6-23

- WITH clause syntax, 3-6
- wrapper classes for null-handling, 4-19
- wrapper methods (JPub), generating, 6-39

V

- VALUES syntax for stored functions, 3-61
- VARRAYs
 - inserting a row, 6-70
 - selecting into host expression, 6-69
 - VARRAY types, 6-4
- verbose option (customizer harness), A-17
- verify option (customizer harness), A-22
- version compatibility (Oracle customizer), A-25
- version number options (sqlj -version-xxxx), 8-26
- VM
 - classpath option, 8-23
 - options through SQLJ, 8-60
 - specifying name, 8-81
- vm option (sqlj -vm), 8-81

W

- warn option (SQLCheckerCustomizer), A-43
- warn option (sqlj -warn), 8-45
- warning messages, translator, 9-12
- warnings, translator, enabling/disabling, 8-45
- weak object/collection types
 - restrictions, 6-81
 - support, 6-80
- weakly typed custom Java classes, 6-2
- weakly typed iterators, 7-41
- WHERE CURRENT OF, 5-38
- Windows, SQLJ development in, 1-33