Oracle9*i*

Java Stored Procedures Developer's Guide

Release 2 (9.2)

March 2002

Part No. A96659-01

ORACLE®

Oracle9*i* Java Stored Procedures Developer’s Guide, Release 2 (9.2)

Part No. A96659-01

Copyright © 2000, 2002 Oracle Corporation. All rights reserved.

Author: Tom Portfolio

Graphics Artist: Valarie Moore

Contributors: Sheryl Maring, Dave Alpern, Gray Clossman, Matthieu Devin, Steve Harris, Hal Hildebrand, Susan Kraft, Sunil Kunisetty, Thomas Kurian, Dave Rosenberg, Jerry Schwarz.

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 Oracle9*i*, Oracle8*i*, PL/SQL, Pro*C/C++ and SQL*Plus are trademarks or registered trademarks of Oracle Corporation. Other names may be trademarks of their respective owners.

Contents

Send Us Your Comments vii

Preface ix

1 Introduction

Java and the RDBMS: A Robust Combination 1-2
Stored Procedures and Run-Time Contexts 1-3
Functions and Procedures 1-4
Database Triggers 1-5
Object-Relational Methods 1-6
Advantages of Stored Procedures 1-6
Performance 1-6
Productivity and Ease of Use 1-7
Scalability 1-7
Maintainability 1-7
Interoperability 1-7
Replication 1-8
Security 1-8
The Oracle JVM and Its Components 1-9
The Oracle JVM versus Client JVMs 1-10
Main Components of the Oracle JVM 1-11
Java Stored Procedure Configuration 1-16
Developing Stored Procedures: An Overview 1-17

2 Loading Java Classes

Java in the Database 2-2
Java Code, Binaries, and Resources Storage 2-3
Preparing Java Class Methods for Execution 2-4
Compiling Java Classes 2-4
Resolving Class Dependencies 2-9
Loading Classes 2-12
How to Grant Execute Rights 2-15
Checking Java Uploads 2-16
User Interfaces on the Server 2-18
Shortened Class Names 2-19
Controlling the Current User 2-20

3 Publishing Java Classes

Understanding Call Specs 3-2
Defining Call Specs: Basic Requirements 3-3
Setting Parameter Modes 3-3
Mapping Datatypes 3-4
Using the Server-Side Internal JDBC Driver 3-7
Using the Server-Side SQLJ Translator 3-9
Writing Top-Level Call Specs 3-11
Writing Packaged Call Specs 3-15
Writing Object Type Call Specs 3-18
Declaring Attributes 3-19
Declaring Methods 3-19

4 Calling Stored Procedures

Calling Java from the Top Level 4-2
Redirecting Output 4-2
Calling Java from Database Triggers 4-6
Calling Java from SQL DML 4-10
Restrictions 4-11
Calling Java from PL/SQL 4-12
Calling PL/SQL from Java 4-14

How the JVM Handles Exceptions 4-15

5 Developing an Application

Drawing the Entity-Relationship Diagram 5-2
Planning the Database Schema 5-5
Creating the Database Tables 5-7
Writing the Java Classes 5-9
Loading the Java Classes 5-15
Publishing the Java Classes 5-16
Calling the Java Stored Procedures 5-18

Index

Send Us Your Comments

Oracle9*i* Java Stored Procedures Developer's Guide, Release 2 (9.2)

Part No. A96659-01

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

Who Should Read This Guide?

Anyone developing Java applications for Oracle9*i* will benefit from reading this guide. Written especially for programmers, it will also be of value to architects, systems analysts, project managers, and others interested in network-centric database applications. To use this guide effectively, you must have a working knowledge of Java, SQL, PL/SQL, and Oracle9*i*.

Note: This guide presumes you are an experienced Java programmer. If you are just learning Java, see "Suggested Reading" on page xiv.

How This Guide Is Organized

This guide enables you to build Java applications for Oracle. Working from simple examples, you quickly learn how to load, publish, and call Java stored procedures.

This guide is divided into the following five chapters:

- Chapter 1, "Introduction"—This chapter surveys the main features of stored procedures and points out the advantages they offer. Then, you learn how the Oracle JVM and its main components work together. The chapter ends with an overview of the Java stored procedures development process.
- Chapter 2, "Loading Java Classes"—This chapter shows you how to load Java source, class, and resource files into the Oracle database. You learn how to manage Java schema objects using the `loadjava` and `dropjava` utilities. In addition, you learn about name resolution and invoker rights versus definer rights.

- Chapter 3, "Publishing Java Classes"—This chapter shows you how to publish Java classes to SQL. Among other things, you learn how to write call specifications, map datatypes, and set parameter modes.
- Chapter 4, "Calling Stored Procedures"—This chapter shows you how to call Java stored procedures in various contexts. For example, you learn how to call Java from SQL DML statements, database triggers, and PL/SQL blocks.
- Chapter 5, "Developing an Application"—This chapter describes each step for the development of a Java stored procedures application.

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.

Notational Conventions

This guide follows these conventions:

Convention	Meaning
Italic	Italic font denotes terms being defined for the first time, words being emphasized, error messages, and book titles.
`Courier`	Courier font denotes program code, schema object names, file names, path names, and Internet addresses.

Java code examples follow these conventions:

Convention	Meaning
`{ }`	Braces enclose a block of statements.
`//`	A double slash begins a single-line comment, which extends to the end of a line.
`/*   */`	A slash-asterisk and an asterisk-slash delimit a multi-line comment, which can span multiple lines.
`...`	An ellipsis shows that statements or clauses irrelevant to the discussion were left out.
`lower case`	We use lower case for keywords and for one-word names of variables, methods, and packages.
`UPPER CASE`	We use upper case for names of constants (static final variables) and for names of supplied classes that map to built-in SQL datatypes.
`Mixed Case`	We use mixed case for names of classes and interfaces and for multi-word names of variables, methods, and packages. The names of classes and interfaces begin with an upper-case letter. In all multi-word names, the second and succeeding words begin with an upper-case letter.

PL/SQL code examples follow these conventions:

Convention	Meaning
`--`	A double hyphen begins a single-line comment, which extends to the end of a line.
`/*   */`	A slash-asterisk and an asterisk-slash delimit a multi-line comment, which can span multiple lines.
`...`	An ellipsis shows that statements or clauses irrelevant to the discussion were left out.
`lower case`	We use lower case for names of constants, variables, cursors, exceptions, subprograms, and packages.
`UPPER CASE`	We use upper case for keywords, names of predefined exceptions, and names of supplied PL/SQL packages.
`Mixed Case`	We use mixed case for names of user-defined datatypes and subtypes. The names of user-defined types begin with an upper-case letter.

Syntax definitions use a simple variant of Backus-Naur Form (BNF) that includes the following symbols:

Symbol	Meaning
`[ ]`	Brackets enclose optional items.
`{ }`	Braces enclose items of which only one is required.
`|`	A vertical bar separates alternatives within brackets or braces.
`...`	An ellipsis shows that the preceding syntactic element can be repeated.
`delimiters`	Delimiters other than brackets, braces, vertical bars, and ellipses must be entered as shown.

Sample Database Tables

Most programming examples in this guide use two sample database tables named `dept` and `emp`. Their definitions follow:

```
CREATE TABLE dept (deptno NUMBER(2) NOT NULL,
                   dname  VARCHAR2(14),
                   loc    VARCHAR2(13));

CREATE TABLE emp (empno    NUMBER(4) NOT NULL,
                  ename    VARCHAR2(10),
                  job      VARCHAR2(9),
                  mgr      NUMBER(4),
                  hiredate DATE,
                  sal      NUMBER(7,2),
                  comm     NUMBER(7,2),
                  deptno   NUMBER(2));
```

Respectively, the `dept` and `emp` tables contain the following rows of data:

```
DEPTNO  DNAME      LOC
------- ---------- ---------
10      ACCOUNTING NEW YORK
20      RESEARCH   DALLAS
30      SALES      CHICAGO
40      OPERATIONS BOSTON

EMPNO ENAME   JOB          MGR  HIREDATE    SAL   COMM  DEPTNO
----- ------- --------- ------ --------- ------ ------ -------
 7369 SMITH   CLERK       7902 17-DEC-80    800             20
 7499 ALLEN   SALESMAN    7698 20-FEB-81   1600    300      30
 7521 WARD    SALESMAN    7698 22-FEB-81   1250    500      30
 7566 JONES   MANAGER     7839 02-APR-81   2975             20
 7654 MARTIN  SALESMAN    7698 28-SEP-81   1250   1400      30
 7698 BLAKE   MANAGER     7839 01-MAY-81   2850             30
 7782 CLARK   MANAGER     7839 09-JUN-81   2450             10
 7788 SCOTT   ANALYST     7566 19-APR-87   3000             20
 7839 KING    PRESIDENT        17-NOV-81   5000             10
 7844 TURNER  SALESMAN    7698 08-SEP-81   1500      0      30
 7876 ADAMS   CLERK       7788 23-MAY-87   1100             20
 7900 JAMES   CLERK       7698 03-DEC-81    950             30
 7902 FORD    ANALYST     7566 03-DEC-81   3000             20
 7934 MILLER  CLERK       7782 23-JAN-82   1300             10
```

To create and load the tables, run the script `demobld.sql`, which can be found in the SQL*Plus demo directory.

Related Publications

Occasionally, this guide refers you to the following Oracle publications for more information:

Oracle9i Application Developer's Guide - Fundamentals
Oracle9i Java Developer's Guide
Oracle9i JDBC Developer's Guide and Reference
Oracle9i SQLJ Developer's Guide and Reference
Oracle9i SQL Reference
*SQL*Plus User's Guide and Reference*

Suggested Reading

The Java Programming Language by Arnold & Gosling, Addison-Wesley
Coauthored by the originator of Java, this definitive book explains the basic concepts, areas of applicability, and design philosophy of the language. Using numerous examples, it progresses systematically from basic to advanced programming techniques.

Thinking in Java by Bruce Eckel, Prentice Hall
This book offers a complete introduction to Java on a level appropriate for both beginners and experts. Using simple examples, it presents the fundamentals and complexities of Java in a straightforward, good-humored way.

Core Java by Cornell & Horstmann, Prentice-Hall
This book is a complete, step-by-step introduction to Java programming principles and techniques. Using real-world examples, it highlights alternative approaches to program design and offers many programming tips and tricks.

Java in a Nutshell by Flanagan, O'Reilly
This indispensable quick reference provides a wealth of information about Java's most commonly used features. It includes programming tips and traps, excellent examples of problem solving, and tutorials on important features.

Java Software Solutions by Lewis & Loftus, Addison-Wesley
This book provides a clear, thorough introduction to Java and object-oriented programming. It contains extensive reference material and excellent pedagogy including self-assessment questions, programming projects, and exercises that encourage experimentation.

Online Sources

There are many useful online sources of information about Java. For example, you can view or download guides and tutorials from the Sun Microsystems home page on the Web:

```
http://www.sun.com
```

Another popular Java Web site is:

```
http://www.gamelan.com
```

For Java API documentation, visit:

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

Also, the following Internet news groups are dedicated to Java:

```
comp.lang.java.programmer
comp.lang.java.misc
```

At the following site, you can get the latest Oracle JVM news, updates, and offerings:

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

In addition to try-and-buy tools, you can download JDBC drivers, SQLJ reference implementations, white papers on Java application development, and collections of frequently asked questions (FAQs).

1 Introduction

The Oracle JVM has all the features you need to build a new generation of enterprise-wide applications at a low cost. Chief among those features are stored procedures, which open the Oracle RDBMS to all Java programmers. With stored procedures, you can implement business logic at the server level, thereby improving application performance, scalability, and security.

This chapter contains the following information:

- Java and the RDBMS: A Robust Combination
- Stored Procedures and Run-Time Contexts
- Advantages of Stored Procedures
- The Oracle JVM and Its Components
- Java Stored Procedure Configuration
- Developing Stored Procedures: An Overview

Java and the RDBMS: A Robust Combination

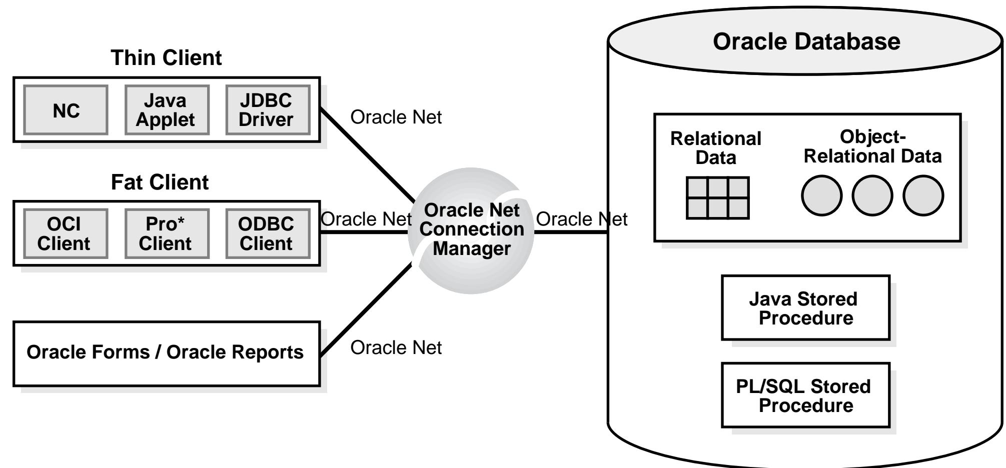
The Oracle RDBMS provides Java applications with a dynamic data-processing engine that supports complex queries and different views of the same data. All client requests are assembled as data queries for immediate processing, and query results are generated on the fly.

Several features make Java ideal for server programming. Java lets you assemble applications using off-the-shelf software components (JavaBeans). Its type safety and automatic memory management allow for tight integration with the RDBMS. In addition, Java supports the transparent distribution of application components across a network.

Thus, Java and the RDBMS support the rapid assembly of component-based, network-centric applications that can evolve gracefully as business needs change. In addition, you can move applications and data stores off the desktop and onto intelligent networks and network-centric servers. More important, you can access those applications and data stores from any client device.

Figure 1–1 shows a traditional two-tier, client/server configuration in which clients call Java stored procedures the same way they call PL/SQL stored procedures. (PL/SQL is an advanced 4GL tightly integrated with Oracle.) The figure also shows how the Oracle Net Services Connection Manager can funnel many network connections into a single database connection. This enables the RDBMS to support a large number of concurrent users.

Figure 1–1 Two-Tier Client/Server Configuration



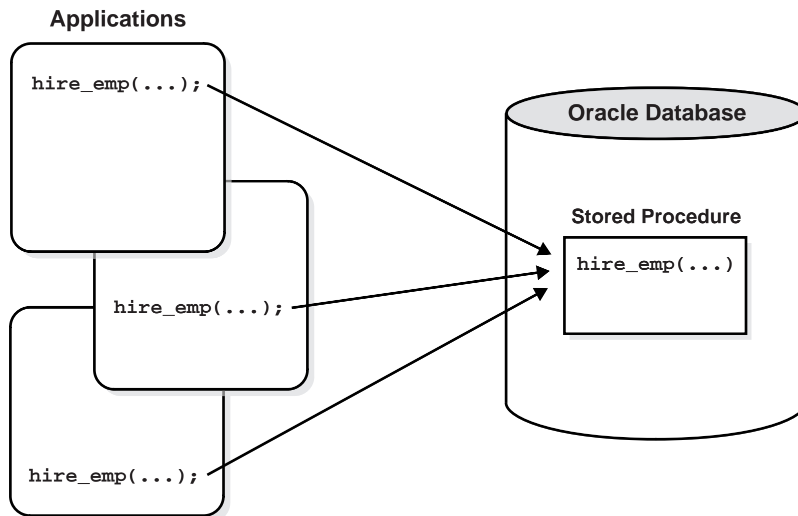
Stored Procedures and Run-Time Contexts

Stored procedures are Java methods published to SQL and stored in an Oracle database for general use. To publish Java methods, you write call specifications (*call specs* for short), which map Java method names, parameter types, and return types to their SQL counterparts.

Unlike a wrapper, which adds another layer of execution, a call spec simply publishes the existence of a Java method. So, when you call the method (through its call spec), the run-time system dispatches the call with minimal overhead.

When called by client applications, a stored procedure can accept arguments, reference Java classes, and return Java result values. Figure 1–2 shows a stored procedure being called by various applications.

Figure 1–2 Calling a Stored Procedure



Except for graphical-user-interface (GUI) methods, Oracle JVM can run any Java method as a stored procedure. The run-time contexts are:

- functions and procedures
- database triggers
- object-relational methods

The next three sections describe these contexts.

Functions and Procedures

Functions and procedures are named blocks that encapsulate a sequence of statements. They are like building blocks that you can use to construct modular, maintainable applications.

Generally, you use a procedure to perform an action, and a function to compute a value. So, for `void` Java methods, you use procedure call specs, and for value-returning methods, you use function call specs.

Only top-level and package (not local) PL/SQL functions and procedures can be used as call specs. When you define them using the SQL `CREATE FUNCTION`, `CREATE PROCEDURE`, or `CREATE PACKAGE` statement, they are stored in the database, where they are available for general use.

Java methods published as functions and procedures must be invoked explicitly. They can accept arguments and are callable from:

- SQL DML statements (INSERT, UPDATE, DELETE, SELECT, CALL , EXPLAIN PLAN, LOCK TABLE, and MERGE)
- SQL CALL statements
- PL/SQL blocks, subprograms, and packages

Database Triggers

A database trigger is a stored procedure associated with a specific table or view. Oracle invokes (fires) the trigger automatically whenever a given DML operation modifies the table or view.

A trigger has three parts: a triggering event (DML operation), an optional trigger constraint, and a trigger action. When the event occurs, the trigger fires and a CALL statement calls a Java method (through its call spec) to perform the action.

Database triggers, which you define using the SQL CREATE TRIGGER statement, let you customize the RDBMS. For example, they can restrict DML operations to regular business hours. Typically, triggers are used to enforce complex business rules, derive column values automatically, prevent invalid transactions, log events transparently, audit transactions, or gather statistics.

Object-Relational Methods

A SQL object type is a user-defined composite datatype that encapsulates a set of variables (*attributes*) with a set of operations (*methods*), which can be written in Java. The data structure formed by the set of attributes is public (visible to client programs). However, well-behaved programs do not manipulate it directly. Instead, they use the set of methods provided.

When you define an object type using the SQL `CREATE ... OBJECT` statement, you create an abstract template for some real-world object. The template specifies only those attributes and behaviors the object will need in the application environment. At run time, when you fill the data structure with values, you create an instance of the object type. You can create as many instances (objects) as necessary.

Typically, an object type corresponds to some business entity such as a purchase order. To accommodate a variable number of items, object types can use variable-length arrays (varrays) and nested tables. For example, this feature enables a purchase order object type to contain a variable number of line items.

Advantages of Stored Procedures

Stored procedures offer several advantages including better performance, higher productivity, ease of use, and increased scalability.

Performance

Stored procedures are compiled once and stored in executable form, so procedure calls are quick and efficient. Executable code is automatically cached and shared among users. This lowers memory requirements and invocation overhead.

By grouping SQL statements, a stored procedure allows them to be executed with a single call. This minimizes the use of slow networks, reduces network traffic, and improves round-trip response time. OLTP applications, in particular, benefit because result-set processing eliminates network bottlenecks.

Additionally, stored procedures enable you to take advantage of the computing resources of the server. For example, you can move computation-bound procedures from client to server, where they will execute faster. Likewise, stored functions called from SQL statements enhance performance by executing application logic within the server.

Productivity and Ease of Use

By designing applications around a common set of stored procedures, you can avoid redundant coding and increase your productivity. Moreover, stored procedures let you extend the functionality of the RDBMS. For example, stored functions called from SQL statements enhance the power of SQL.

You can use the Java integrated development environment (IDE) of your choice to create stored procedures. Then, you can deploy them on any tier of the network architecture. Moreover, they can be called by standard Java interfaces, such as JDBC, and by programmatic interfaces and development tools such as SQLJ, the OCI, Pro*C/C++, and JDeveloper.

This broad access to stored procedures lets you share business logic across applications. For example, a stored procedure that implements a business rule can be called from various client-side applications, all of which can share that business rule. In addition, you can leverage the server's Java facilities while continuing to write applications for your favorite programmatic interface.

Scalability

Stored procedures increase scalability by isolating application processing on the server. In addition, automatic dependency tracking for stored procedures aids the development of scalable applications.

The shared memory facilities of the Shared Server enable Oracle to support more than 10,000 concurrent users on a single node. For more scalability, you can use the Oracle Net Services Connection Manager to multiplex Oracle Net Services connections.

Maintainability

Once it is validated, you can use a stored procedure with confidence in any number of applications. If its definition changes, only the procedure is affected, not the applications that call it. This simplifies maintenance and enhancement. Also, maintaining a procedure on the server is easier than maintaining copies on different client machines.

Interoperability

Within the RDBMS, Java conforms fully to the *Java Language Specification* and furnishes all the advantages of a general-purpose, object-oriented programming

language. Also, as with PL/SQL, Java provides full access to Oracle data, so any procedure written in PL/SQL can be written in Java.

PL/SQL stored procedures complement Java stored procedures. Typically, SQL programmers who want procedural extensions favor PL/SQL, and Java programmers who want easy access to Oracle data favor Java.

The RDBMS allows a high degree of interoperability between Java and PL/SQL. Java applications can call PL/SQL stored procedures using an embedded JDBC driver; conversely, PL/SQL applications can call Java stored procedures directly.

Replication

With Oracle Advanced Replication, you can replicate (copy) stored procedures from one Oracle database to another. That feature makes them ideal for implementing a central set of business rules. Once you write them, you can replicate and distribute the stored procedures to work groups and branch offices throughout the company. In this way, you can revise policies on a central server rather than on individual servers.

Security

Security is a large arena that includes network security for the connection, access and execution control of operating system resources or of JVM and user-defined classes, and bytecode verification of imported JAR files from an external source.

Oracle uses Java 2 security to protect its Java virtual machine. All classes are loaded into a secure database, so they are untrusted. To access classes and operating system resources, a user needs the proper permissions. Likewise, all stored procedures are secured against other users (to whom you can grant the database privilege `EXECUTE`).

You can restrict access to Oracle data by allowing users to manipulate the data only through stored procedures that execute with their definer's privileges. For example, you can allow access to a procedure that updates a database table, but deny access to the table itself.

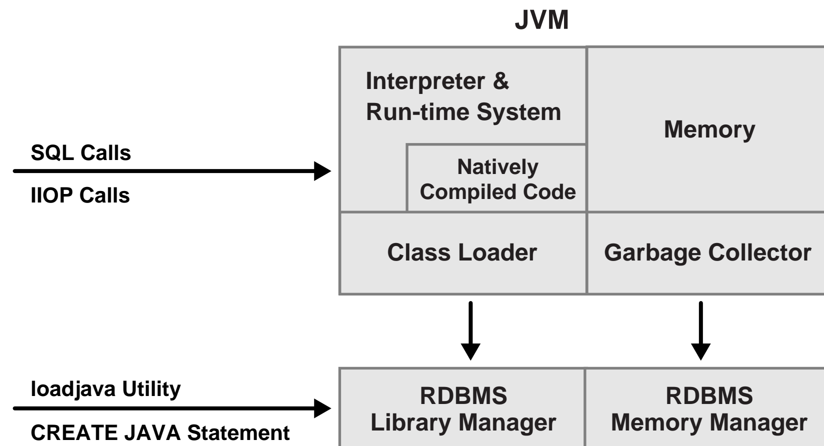
For a full discussion of Oracle JVM security, see the *Oracle9i Java Developer's Guide.*

The Oracle JVM and Its Components

The Oracle Java virtual machine (JVM) is a complete, Java 2-compliant Java execution environment. It runs in the same process space and address space as the RDBMS kernel, sharing its memory heaps and directly accessing its relational data. This design optimizes memory use and increases throughput.

The Oracle JVM provides a run-time environment for Java objects. It fully supports Java data structures, method dispatch, exception handling, and language-level threads. It also supports all the core Java class libraries including `java.lang`, `java.io`, `java.net`, `java.math`, and `java.util`. Figure 1–3 shows its main components.

Figure 1–3 Main Components of the Oracle JVM



The Oracle JVM embeds the standard Java namespace in RDBMS schemas. This feature lets Java programs access Java objects stored in Oracle databases and application servers across the enterprise.

In addition, the JVM is tightly integrated with the scalable, shared memory architecture of the RDBMS. Java programs use call, session, and object lifetimes efficiently without your intervention. So, you can scale Oracle JVM and middle-tier Java business objects, even when they have session-long state.

The Oracle JVM versus Client JVMs

This section discusses some important differences between the Oracle JVM and typical client JVMs.

Method main()

Client-based Java applications declare a single, top-level method (`main()`) that defines the profile of an application. As with applets, server-based applications have no such "inner loop". Instead, they are driven by logically independent clients.

Each client begins a session, calls its server-side logic modules through top-level entry points, and eventually ends the session. The server environment hides the managing of sessions, networks, and other shared resources from hosted Java programs.

The GUI

A server cannot provide GUIs, but it can supply the logic that drives them. For example, the Oracle JVM does not supply the basic GUI components found in the JDK's Abstract Windowing Toolkit (AWT). However, all AWT Java classes are available within the server environment. So, your programs can use AWT functionality, as long as they do not attempt to materialize a GUI on the server.

The IDE

The Oracle JVM is oriented to Java application deployment, not development. You can write and unit-test applications in your favorite IDE, then deploy them for execution within the RDBMS.

Java's binary compatibility enables you to work in any IDE, then upload Java class files to the server. You need not move your Java source files to the database. Instead, you can use powerful client-side IDEs to maintain Java applications that are deployed on the server.

Multithreading

Multithreading support is often cited as one of the key scalability features of the Java language. Certainly, the Java language and class libraries make it simpler to write multithreaded applications in Java than many other languages, but it is still a daunting task in any language to write reliable, scalable multithreaded code.

As a database server, Oracle9*i* efficiently schedules work for thousands of users. The Oracle JVM uses the facilities of the RDBMS server to concurrently schedule Java execution for thousands of users. Although Oracle9*i* supports Java language

level threads required by the Java Language Specification (JLS) and Java Compatibility Kit (JCK), using threads within the scope of the database will not increase your scalability. Using the embedded scalability of the database eliminates the need for writing multithreaded Java servers. You should use the database's facilities for scheduling users by writing single-threaded Java applications. The database will take care of the scheduling between each application; thus, you achieve scalability without having to manage threads. You can still write multithreaded Java applications, but multiple Java threads will not increase your server's performance.

One difficulty multithreading imposes on Java is the interaction of threads and automated storage management, or garbage collection. The garbage collector executing in a generic JVM has no knowledge of which Java language threads are executing or how the underlying operating system schedules them.

- Non-Oracle model—A single user maps to a single Java language level thread; the same single garbage collector manages all garbage from all users. Different techniques typically deal with allocation and collection of objects of varying lifetimes and sizes. The result in a heavily multithreaded application is, at best, dependent upon operating system support for native threads, which can be unreliable and limited in scalability. High levels of scalability for such implementations have not been convincingly demonstrated.
- Oracle JVM model—Even when thousands of users connect to the server and execute the same Java code, each user experiences it as if he is executing his own Java code on his own Java virtual machine. The responsibility of the Oracle JVM is to make use of operating system processes and threads, using the scalable approach of the Oracle RDBMS. As a result of this approach, the JVM's garbage collector is more reliable and efficient because it never collects garbage from more than one user at any time.

Main Components of the Oracle JVM

This section briefly describes the main components of the Oracle JVM and some of the facilities they provide.

Library Manager

To store Java classes in an Oracle database, you use the command-line utility `loadjava`, which employs SQL `CREATE JAVA` statements to do its work. When invoked by the `CREATE JAVA {SOURCE | CLASS | RESOURCE}` statement, the library manager loads Java source, class, or resource files into the database. You never access these Java schema objects directly; only the Oracle JVM uses them.

Garbage Collection of Memory

Garbage collection is a major feature of Java's automated storage management, eliminating the need for Java developers to allocate and free memory explicitly. Consequently, this eliminates a large source of memory leaks that commonly plague C and C++ programs. There is a price for such a benefit: garbage collection contributes to the overhead of program execution speed and footprint. Although many papers have been written qualifying and quantifying the trade-off, the overall cost is reasonable, considering the alternatives.

Garbage collection imposes a challenge to the JVM developer seeking to supply a highly scalable and fast Java platform. The Oracle9*i* JVM meets these challenges in the following ways:

- The Oracle JVM uses the Oracle9*i* scheduling facilities, which can manage multiple users efficiently.
- Garbage collection is performs consistently for multiple users because garbage collection is focused on a single user within a single session. The Oracle JVM enjoys a huge advantage because the burden and complexity of the memory manager's job does not increase as the number of users increases. The memory manager performs the allocation and collection of objects within a single session—which typically translates to the activity of a single user.
- The Oracle JVM uses different garbage collection techniques depending on the type of memory used. These techniques provide high efficiency and low overhead.

Compiler

The Oracle JVM includes a standard Java 2 (also known as JDK 1.2) Java compiler. When invoked by the `CREATE JAVA SOURCE` statement, it translates Java source files into architecture-neutral, one-byte instructions known as *bytecodes*. Each bytecode consists of an opcode followed by its operands. The resulting Java class files, which conform fully to the Java standard, are submitted to the interpreter at run time.

Interpreter

To execute Java programs, the Oracle JVM includes a standard Java 2 bytecode interpreter. The interpreter and associated Java run-time system execute standard Java class files. The run-time system supports native methods and call-in/call-out from the host environment.

Note: Although your own code is interpreted, the Oracle JVM uses natively compiled versions of the core Java class libraries, SQLJ translator, and JDBC drivers. For more information, see "Native Compiler (Accelerator)" on page 1-14.

Class Loader

In response to requests from the run-time system, the Java class loader locates, loads, and initializes Java classes stored in the database. The class loader reads the class, then generates the data structures needed to execute it. Immutable data and metadata are loaded into initialize-once shared memory. As a result, less memory is required for each session. The class loader attempts to resolve external references when necessary. Also, it invokes the Java compiler automatically when Java class files must be recompiled (and the source files are available).

Verifier

Java class files are fully portable and conform to a well-defined format. The verifier prevents the inadvertent use of "spoofed" Java class files, which might alter program flow or violate access restrictions. Oracle security and Java security work with the verifier to protect your applications and data.

Server-Side JDBC Internal Driver

JDBC is a standard set of Java classes providing vendor-independent access to relational data. Specified by Sun Microsystems and modeled after ODBC (Open Database Connectivity) and the X/Open SQL CLI (Call Level Interface), the JDBC classes supply standard features such as simultaneous connections to several databases, transaction management, simple queries, calls to stored procedures, and streaming access to `LONG` column data.

Using low-level entry points, a specially tuned JDBC driver runs directly inside the RDBMS, thereby providing the fastest access to Oracle data from Java stored procedures. The server-side internal JDBC driver complies fully with the Sun Microsystems JDBC specification. Tightly integrated with the RDBMS, it supports Oracle-specific datatypes, globalization character sets, and stored procedures. Additionally, the client-side and server-side JDBC APIs are the same, which makes it easy to partition applications.

Server-Side SQLJ Translator

SQLJ enables you to embed SQL statements in Java programs. It is more concise than JDBC and more amenable to static analysis and type checking. The SQLJ preprocessor, itself a Java program, takes as input a Java source file in which SQLJ clauses are embedded. Then, it translates the SQLJ clauses into Java class definitions that implement the specified SQL statements. The Java type system ensures that objects of those classes are called with the correct arguments.

A highly optimized SQLJ translator runs directly inside the RDBMS, where it provides run-time access to Oracle data using the server-side internal JDBC driver. SQLJ forms can include queries, DML, DDL, transaction control statements, and calls to stored procedures. The client-side and server-side SQLJ APIs are identical, which makes it easy to partition applications.

Native Compiler (Accelerator)

Java executes platform-independent bytecodes on top of a JVM, which in turn interacts with the specific hardware platform. Any time you add levels within software, your performance is degraded. Because Java requires going through an intermediary to interpret platform-independent bytecodes, a degree of inefficiency exists for Java applications that does not exists within a platform-dependent language, such as C. To address this issue, several JVM suppliers create native compilers. Native compilers translate Java bytecodes into platform-dependent native code, which eliminates the interpreter step and improves performance.

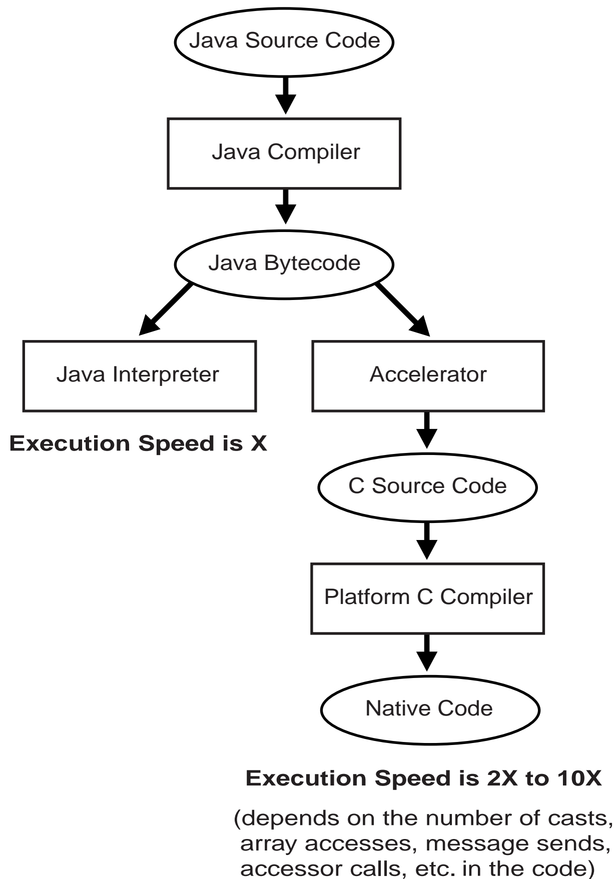
The following describes two methods for native compilation:

Compiler	Description
Just-In-Time (JIT) Compilation	JIT compilers quickly compile Java bytecodes to native (platform-specific) machine code during runtime. This does not produce an executable to be executed on the platform; instead, it provides platform-dependent code from Java bytecodes that is executed directly after it is translated. This should be used for Java code that is run frequently, which will be executed at speeds closer to languages such as C.
Ahead-of-Time Compilation	Compilation translates Java bytecodes to platform-independent C code before runtime. Then a standard C compiler compiles the C code into an executable for the target platform. This approach is more suitable for Java applications that are modified infrequently. This approach takes advantage of the mature and efficient platform-specific compilation technology found in modern C compilers.

Oracle9*i* uses Ahead-of-Time compilation to deliver its core Java class libraries: JDBC code in natively compiled form. It is applicable across all the platforms Oracle supports, whereas a JIT approach requires low-level, processor-dependent code to be written and maintained for each platform. You can use this native compilation technology with your own Java code.

As Figure 1–4 shows, natively compiled code executes up to ten times faster than interpreted code. So, the more native code your program uses, the faster it executes.

Figure 1–4 Interpreter versus Accelerator



You can natively compile your own code using the `ncomp` tool. See *Oracle9i Java Developer's Guide* for more information.

Java Stored Procedure Configuration

To configure the database to run Java stored procedures, you must decide whether you want the database to run in dedicated server mode or shared server mode.

- Dedicated server mode—You must configure the database and clients in dedicated server mode using Oracle Net Services connections.

- shared server mode—You must configure the server for shared server mode with the DISPATCHERS parameter, as Chapter 9 of the *Oracle9i Net Services Administrator's Guide* explains.

Java, SQL, or PL/SQL clients, which execute Java stored procedures on the server, connect to the database over a Oracle Net Services connection. For a full description of how to configure this connection, see the *Oracle9i Net Services Administrator's Guide.*

Developing Stored Procedures: An Overview

You execute Java stored procedures similarly to PL/SQL. Normally, calling a Java stored procedure is a by-product of database manipulation, because it is usually the result of a trigger or SQL DML call. To invoke a Java stored procedure, you must publish it through a call specification.

This section demonstrates how to develop a simple Java stored procedure. For more examples of a Java stored procedures application, see Chapter 5.

Step 1: Create or Reuse the Java Classes

Use your favorite Java IDE to create classes, or simply reuse existing classes that meet your needs. Oracle's Java facilities support many Java development tools and client-side programmatic interfaces. For example, the Oracle JVM accepts programs developed in popular Java IDEs such as Oracle's JDeveloper, Symantec's Visual Café, and Borland's JBuilder.

In the following example, you create the public class `Oscar`. It has a single method named `quote()`, which returns a quotation from Oscar Wilde.

```
public class Oscar {
  // return a quotation from Oscar Wilde
  public static String quote() {
    return "I can resist everything except temptation.";
  }
}
```

In the following example, using Sun Microsystems's JDK Java compiler, you compile class `Oscar` on your client workstation:

```
javac Oscar.java
```

The compiler outputs a Java binary file—in this case, `Oscar.class`.

Step 2: Load and Resolve the Java Classes

Using the utility `loadjava`, you can upload Java source, class, and resource files into an Oracle database, where they are stored as Java schema objects. You can run `loadjava` from the command line or from an application, and you can specify several options including a resolver.

In the following example, `loadjava` connects to the database using the default JDBC OCI driver. You must specify the username and password. By default, class `Oscar` is loaded into the logon schema (in this case, `scott`).

```
> loadjava -user scott/tiger Oscar.class
```

Later, when you call method `quote()`, the server uses a resolver (in this case, the default resolver) to search for supporting classes such as `String`. The default resolver searches first in the current schema, then in schema `SYS`, where all the core Java class libraries reside. If necessary, you can specify different resolvers.

For more information, see Chapter 2.

Step 3: Publish the Java Classes

For each Java method callable from SQL, you must write a call spec, which exposes the method's top-level entry point to Oracle. Typically, only a few call specs are needed, but if you like, Oracle's JDeveloper can generate them for you.

In the following example, from SQL*Plus, you connect to the database, then define a top-level call spec for method `quote()`:

```
SQL> connect scott/tiger

SQL> CREATE FUNCTION oscar_quote RETURN VARCHAR2
  2  AS LANGUAGE JAVA
  3  NAME 'Oscar.quote() return java.lang.String';
```

For more information, see Chapter 3.

Step 4: Call the Stored Procedures

You can call Java stored procedures from SQL DML statements, PL/SQL blocks, and PL/SQL subprograms. Using the SQL `CALL` statement, you can also call them from the top level (from SQL*Plus, for example) and from database triggers.

In the following example, you declare a SQL*Plus host variable:

```
SQL> VARIABLE theQuote VARCHAR2(50);
```

Then, you call the function `oscar_quote()`, as follows:

```
SQL> CALL oscar_quote() INTO :theQuote;

SQL> PRINT theQuote;

THEQUOTE
--------------------------------------------------
I can resist everything except temptation.
```

For more information, see Chapter 4.

Step 5: If Necessary, Debug the Stored Procedures

Your Java stored procedures execute remotely on a server, which typically resides on a separate machine. However, the JDK debugger (`jdb`) cannot debug remote Java programs.

Oracle9*i* furnishes a debugging capability that is useful for developers who use the JDK's `jdb` debugger. Two interfaces are supported.

- The debug Agent protocol that was introduced in Oracle8*i*, and is supported by JDK 1.2 and later versions of JDB. The class `DebugProxy` makes remote Java programs appear to be local. It lets any debugger that supports the `sun.tools.debug.Agent` protocol connect to a program as if the program were local. The proxy forwards requests to the server and returns results to the debugger.

  For detailed instructions, see the *Oracle9i Java Developer's Guide.*

- The Java Debug Wire Protocol supported by JDK 1.3 and later versions of the Sun Microsystems JDB debugger (`http://java.sun.com/j2se/1.3/docs/guide/jpda/`, `http://java.sun.com/j2se/1.4/docs/guide/jpda/`.) The use of this interface is documented on OTN. The JDWP protocol supports many new features, including the ability to listen for connections (no more `DebugProxy`), change the values of variables while debugging, and evaluate arbitrary Java expressions, including method evaluation.

Oracle's JDeveloper provides a user-friendly integration with these debugging features. See the JDeveloper documentation for more information on how to debug your Java application through JDeveloper. Other independent IDE vendors will be able to integrate their own debuggers with Oracle9*i*.

Another Example

The following example shows how to create, resolve, load, and publish a simple Java stored procedure that echoes "Hello world".

1. Write the Java class.

   Define a class, `Hello`, with one method, `Hello.world()`, that returns the string "`Hello world`".

   ```
   public class Hello
   {
      public static String world ()
      {
         return "Hello world";
      }
   }
   ```

2. Compile the class on your client system. Using the Sun Microsystems JDK, for example, invoke the Java compiler, `javac`, as follows:

   ```
   javac Hello.java
   ```

   Normally, it is a good idea to specify your `CLASSPATH` on the `javac` command line, especially when writing shell scripts or make files. The Java compiler produces a Java binary file—in this case, `Hello.class`.

   Keep in mind where this Java code will execute. If you execute `Hello.class` on your client system, it searches the CLASSPATH for all supporting core classes it must execute. This search should result in locating the dependent class in one of the following:

   - as an individual file in a directory, where the directory is specified in the CLASSPATH
   - within a `.jar` or `.zip` file, where the directory is specified in the CLASSPATH

3. Decide on the resolver for your class.

   In this case, you load `Hello.class` in the server, where it is stored in the database as a Java schema object. When you execute the `world()` method of the `Hello.class` on the server, it finds the necessary supporting classes, such as `String`, using a resolver—in this case, the default resolver. The default resolver looks for classes in the current schema first and then in PUBLIC. All core class libraries, including the `java.lang` package, are found in PUBLIC. You may need to specify different resolvers, and you can force resolution to

occur when you use `loadjava`, to determine if there are any problems earlier, rather than at runtime. Refer to the *Oracle9i Java Developer's Guide* for more details on resolvers and `loadjava`.

4. Load the class on the Oracle9*i* server using `loadjava`. You must specify the username and password.

```
loadjava -user scott/tiger Hello.class
```

5. Publish the stored procedure through a call specification.

   To invoke a Java static method with a SQL CALL, you must publish it with a call specification. A call specification defines for SQL which arguments the method takes and the SQL types it returns.

   In SQL*Plus, connect to the database and define a top-level call specification for `Hello.world()`:

```
SQL> connect scott/tiger
connected
SQL> create or replace function HELLOWORLD return VARCHAR2 as
  2  language java name 'Hello.world () return java.lang.String';
  3  /
Function created.
```

6. Invoke the stored procedure.

```
SQL> variable myString varchar2[20];
SQL> call HELLOWORLD() into :myString;
Call completed.
SQL> print myString;

MYSTRING
---------------------------------------
Hello world

SQL>
```

   The `call HELLOWORLD() into :myString` statement performs a top-level call in Oracle9*i*. The Oracle-specific `select HELLOWORLD from DUAL` also works. Note that SQL and PL/SQL see no difference between a stored procedure that is written in Java, PL/SQL, or any other language. The call specification provides a means to tie inter-language calls together in a consistent manner. Call specifications are necessary only for entry points invoked with triggers or SQL and PL/SQL calls. Furthermore, JDeveloper can automate the task of writing call specifications.

2

Loading Java Classes

Before you can call Java stored procedures, you must load them into the Oracle database and publish them to SQL. Loading and publishing are separate tasks. Many Java classes, referenced only by other Java classes, are never published.

To load Java stored procedures automatically, you use the command-line utility `loadjava`. It uploads Java source, class, and resource files into a system-generated database table, then uses the SQL `CREATE JAVA {SOURCE | CLASS | RESOURCE}` statement to load the Java files into the Oracle database. You can upload Java files from file systems, popular Java IDEs, intranets, or the Internet.

> **Note:** To load Java stored procedures manually, you use `CREATE JAVA` statements. For example, in SQL*Plus, you can use the `CREATE JAVA CLASS` statement to load Java class files from local `BFILE`s and `LOB` columns into the Oracle database.

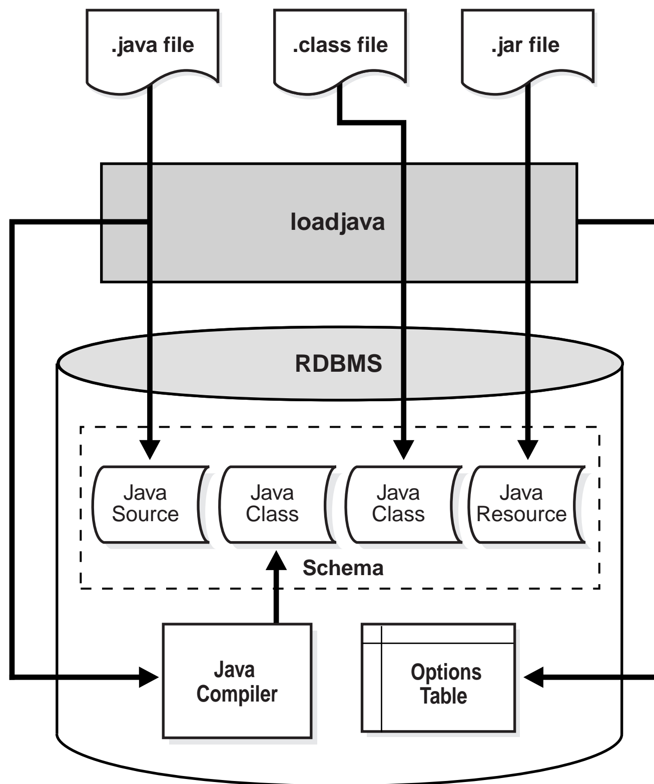
- Java in the Database
- Java Code, Binaries, and Resources Storage
- Preparing Java Class Methods for Execution
- User Interfaces on the Server
- Shortened Class Names
- Controlling the Current User

Java in the Database

To make Java files available to the Oracle JVM, you must load them into the Oracle database as schema objects. As Figure 2–1 illustrates, `loadjava` can invoke the JVM's Java compiler, which compiles source files into standard class files.

The figure also shows that `loadjava` can set the values of options stored in a system database table. Among other things, these options affect the processing of Java source files.

Figure 2–1 Loading Java into the Oracle Database



Each Java class is stored as a schema object. The name of the object is derived from the fully qualified name (*full name*) of the class, which includes the names of containing packages. For example, the full name of class `Handle` is:

```
oracle.aurora.rdbms.Handle
```

In the name of a Java schema object, slashes replace dots, so the full name of the class becomes:

```
oracle/aurora/rdbms/Handle
```

The Oracle RDBMS accepts Java names up to 4000 characters long. However, the names of Java schema objects cannot be longer than 30 characters, so if a name is longer than that, the system generates an alias (*short name*) for the schema object. Otherwise, the full name is used. You can specify the full name in any context that requires it. When needed, name mapping is handled by the RDBMS.

Java Code, Binaries, and Resources Storage

In the Sun Microsystems Java development environment, Java source code, binaries, and resources are stored as files in a file system.

- Source code files are known as `.java` files.
- Compiled Java binary files are known as `.class` files.
- Resources are any data files, such as `.properties` or `.ser` files that are held within the file system hierarchy, which are loaded or used at runtime.

In addition, when you execute Java, you specify a `CLASSPATH`, which is a set of a file system tree roots containing your files. Java also provides a way to group these files into a single archive form—a ZIP or JAR file.

Both of these concepts are different within the database. The following describes how Oracle9*i* handles Java classes and locates dependent classes:

Java code, binaries, and resources	In the Oracle JVM environment, source, classes, and resources reside within the Oracle9*i* database. Because they reside in the database, they are known as Java schema objects, where a schema corresponds to a database user. There are three types of Java objects: source, class, and resource. There are no `.java`, `.class`, `.sqlj`, `.properties`, or `.ser` files on the server; instead, these files map to source, class, and resource Java schema objects.

Locating Java classes	Instead of a CLASSPATH, you use a resolver to specify one or more schemas to search for source, class, and resource Java schema objects.

The call and session terms, used during our discussions, are not Java terms; but are server terms that apply to the Oracle JVM platform. The Oracle memory manager preserves Java program state throughout your session (that is, between calls). The JVM uses the Oracle database to hold Java source, classes, and resources within a schema—Java schema objects. You can use a resolver to specify how Java, when executed in the server, locates source code, classes, and resources.

Preparing Java Class Methods for Execution

For your Java methods to be executed, you must do the following:

1. Decide when your source is going to be compiled.
2. Decide if you are going to use the default resolver or another resolver for locating supporting Java classes within the database.
3. Load the classes into the database. If you do not wish to use the default resolver for your classes, you should specify a separate resolver on the load command.
4. Publish your class or method.

Compiling Java Classes

Compilation of your source can be performed in one of the following ways:

- You can compile the source explicitly on your client machine, before loading it into the database, through a Java compiler, such as `javac`.
- You can ask the database to compile the source during the loading process managed within the `loadjava` tool.
- You can force the compilation to occur dynamically at runtime.

Note: If you decide to compile through `loadjava`, you can specify compiler options. See "Specifying Compiler Options" on page 2-5 for more information.

Compiling Source Through javac

You can compile your Java with a conventional Java compiler, such as `javac`. After compilation, you load the compiled binary into the database, rather than the source itself. This is a better option, because it is normally easier to debug your Java code on your own system, rather than debugging it on the database.

Compiling Source Through loadjava

When you specify the `-resolve` option on `loadjava` for a source file, the following occurs:

1. The source file is loaded as a source schema object.
2. The source file is compiled.
3. Class schema objects are created for each class defined in the compiled `.java` file.
4. The compiled code is stored in the class schema objects.

Oracle9*i* logs all compilation errors both to `loadjava`'s logfile and the `USER_ERRORS` view. For more information on the `USER_ERRORS` view.

Compiling Source at Runtime

When you load the Java source into the database without the `-resolve` option, Oracle9*i* compiles the source automatically when the class is needed during runtime. The source file is loaded into a source schema object.

Oracle9*i* logs all compilation errors both to `loadjava`'s logfile and the `USER_ERRORS` view.

Specifying Compiler Options

There are two ways to specify options to the compiler.

- Specify compiler options on the `loadjava` command line. You can specify the `encoding` option on the `loadjava` command line.
- Specify persistent compiler options in a schema database table called `JAVA$OPTIONS`. Every time you compile, the compiler uses these options. However, any specified compiler options on the `loadjava` command override the options defined in this table.

  You must create this table yourself if you wish to specify compiler options this way. See "Compiler Options Specified in a Database Table" on page 2-6 for instructions on how to create the `JAVA$OPTIONS` table.

The following sections describe your compiler options:

- Default Compiler Options
- Compiler Options on the Command Line
- Compiler Options Specified in a Database Table

Default Compiler Options When compiling a source schema object for which there is neither a `JAVA$OPTIONS` entry nor a command line value for an option, the compiler assumes a default value as follows:

- `encoding` = `System.getProperty("file.encoding");`
- `online` = `true`: See the *Oracle9i SQLJ Developer's Guide and Reference* for a description of this option, which applies only to Java sources that contain SQLJ constructs.
- `debug` = `true`: This option is equivalent to `javac -g`.

Compiler Options on the Command Line The `loadjava` compiler option, `encoding`, identifies the encoding of the `.java` file. This option overrides any matching value in the `JAVA$OPTIONS` table. The values are identical to the `javac -encoding` option. This option is relevant only when loading a source file.

Compiler Options Specified in a Database Table Each `JAVA$OPTIONS` row contains the names of source schema objects to which an option setting applies; you can use multiple rows to set the options differently for different source schema objects.

You can set `JAVA$OPTIONS` entries by means of the following functions and procedures, which are defined in the database package `DBMS_JAVA`:

- `PROCEDURE set_compiler_option(name VARCHAR2, option VARCHAR2, value VARCHAR2);`
- `FUNCTION get_compiler_option(name VARCHAR2, option VARCHAR2) RETURNS VARCHAR2;`
- `PROCEDURE reset_compiler_option(name VARCHAR2, option VARCHAR2);`

The parameters for these methods are described as follows:

name	The `name` parameter is a Java package name, a fully qualified class name, or the empty string. When the compiler searches the `JAVA$OPTIONS` table for the options to use for compiling a Java source schema object, it uses the row whose `name` most closely matches the schema object's fully qualified class name. A `name` whose value is the empty string matches any schema object name.
option	The `option` parameter is either `'online'`, `'encoding'` or `'debug'`. For the `values` you can specify for these options, see the *Oracle9i SQLJ Developer's Guide and Reference.*

A schema does not initially have a `JAVA$OPTIONS` table. To create a `JAVA$OPTIONS` table, use the `DBMS_JAVA` package's `java.set_compiler_option` procedure to set a value. The procedure will create the table if it does not exist. Specify parameters in single quotes. For example:

```
SQL> execute dbms_java.set_compiler_option('x.y', 'online', 'false');
```

Table 2–1 represents a hypothetical `JAVA$OPTIONS` database table. The pattern match rule is to match as much of the schema name against the table entry as possible. The schema name with a higher resolution for the pattern match is the entry that applies. Because the table has no entry for the `encoding` option, the compiler uses the default or the value specified on the command line. The `online` option shown in the table matches schema object names as follows:

- The name `a.b.c.d` matches class and package names beginning with `a.b.c.d`; the packages and classes are compiled with `online` = `true`.
- The name `a.b` matches class and package names beginning with `a.b`. The name `a.b` does not match `a.b.c.d`; therefore, the packages and classes are compiled with `online` = `false`.
- All other packages and classes match the empty string entry and are compiled with `online` = `true`.

Table 2–1 Example JAVA$OPTIONS Table

Name	Option	Value	Match Examples
`a.b.c.d`	`online`	`true`	■ `a.b.c.d`—matches the pattern exactly. ■ `a.b.c.d.e`—first part matches the pattern exactly; no other rule matches full name.

Table 2–1 Example JAVA$OPTIONS Table

Name	Option	Value	Match Examples
`a.b`	`online`	`false`	▪ `a.b`—matches the pattern exactly ▪ `a.b.c.x`—first part matches the pattern exactly; no other rule matches beyond specified rule name.
(empty string)	`online`	`true`	▪ `a.c`—no pattern match with any defined name; defaults to (empty string) rule ▪ `x.y`—no pattern match with any defined name; defaults to (empty string) rule

Automatic Recompilation

Oracle9*i* provides a dependency management and automatic build facility that will transparently recompile source programs when you make changes to the source or binary programs upon which they depend. Consider the following cases:

```
public class A
{
      B b;
      public void assignB () {b = new B()}
}
public class B
{
      C c;
      public void assignC () {c = new C()}
}
public class C
{
      A a;
      public void assignA () {a = new A()}
}
```

The system tracks dependencies at a class level of granularity. In the preceding example, you can see that classes A, B, and C depend on one another, because A holds an instance of B, B holds an instance of C, and C holds an instance of A. If you change the definition of class A by adding a new field to it, the dependency mechanism in Oracle9*i* flags classes B and C as invalid. Before you use any of these classes again, Oracle9*i* attempts to resolve them again and recompile, if necessary. Note that classes can be recompiled only if source is present on the server.

The dependency system enables you to rely on Oracle9*i* to manage dependencies between classes, to recompile, and to resolve automatically. You must force

compilation and resolution yourself only if you are developing and you want to find problems early. The `loadjava` utility also provides the facilities for forcing compilation and resolution if you do not want to allow the dependency management facilities to perform this for you.

Resolving Class Dependencies

Many Java classes contain references to other classes, which is the essence of reusing code. A conventional Java virtual machine searches for classes, ZIP, and JAR files within the directories specified in the CLASSPATH. In contrast, the Oracle Java virtual machine searches database schemas for class objects. With Oracle, you load all Java classes within the database, so you might need to specify where to find the dependent classes for your Java class within the database.

All classes loaded within the database are referred to as class schema objects and are loaded within certain schemas. All JVM classes, such as `java.lang.*`, are loaded within `PUBLIC`. If your classes depend upon other classes you have defined, you will probably load them all within your own schema. For example, if your schema is `SCOTT`, the database resolver (the database replacement for `CLASSPATH`) searches the `SCOTT` schema before `PUBLIC`. The listing of schemas to search is known as a resolver spec. Resolver specs are for each class, whereas in a classic Java virtual machine, CLASSPATH is global to all classes.

When locating and resolving the interclass dependencies for classes, the resolver marks each class as valid or invalid, depending on whether all interdependent classes are located. If the class that you load contains a reference to a class that is not found within the appropriate schemas, the class is listed as invalid. Unsuccessful resolution at runtime produces a "class not found" exception. Furthermore, runtime resolution can fail for lack of database resources if the tree of classes is very large.

Note: As with the Java compiler, `loadjava` resolves references to classes, but not to resources. Be sure to correctly load the resource files that your classes need.

For each interclass reference in a class, the resolver searches the schemas specified by the resolver spec for a valid class schema object that satisfies the reference. If all references are resolved, the resolver marks the class valid. A class that has never been resolved, or has been resolved unsuccessfully, is marked invalid. A class that depends on a schema object that becomes invalid is also marked invalid.

To make searching for dependent classes easier, Oracle provides a default resolver and resolver spec that searches first the definer's schema and then PUBLIC. This covers most of the classes loaded within the database. However, if you are accessing classes within a schema other than your own or PUBLIC, you must define your own resolver spec.

- loading using Oracle's default resolver, which searches the definer's schema and `PUBLIC`:

  ```
  loadjava -resolve
  ```

- loading using your own resolver spec definition containing the SCOTT schema, OTHER schema, and PUBLIC:

  ```
  loadjava -resolve -resolver "((* SCOTT)(* OTHER)(* PUBLIC))"
  ```

The `-resolver` option specifies the objects to search within the schemas defined. In the previous example, all class schema objects are searched within SCOTT, OTHER, and PUBLIC. However, if you wanted to search for only a certain class or group of classes within the schema, you could narrow the scope for the search. For example, to search only for the classes "`my/gui/*`" within the OTHER schema, you would define the resolver spec as follows:

```
loadjava -resolve -resolver '((* SCOTT) ("my/gui/*" OTHER) (* PUBLIC))'
```

The first parameter within the resolver spec is for the class schema object; the second parameter defines the schema within which to search for these class schema objects.

Allowing References to Non-Existent Classes

You can specify a special option within a resolver spec that allows an unresolved reference to a non-existent class. Sometimes, internal classes are never used within a product. For example, some ISVs do not remove all references to internal test classes from the JAR file before shipping. In a normal Java environment, this is not a problem, because as long as the methods are not called, the Sun Microsystems JVM ignores them. However, the Oracle9*i* resolver tries to resolve all classes referenced within the JAR file—even unused classes. If the reference cannot be validated, the classes within the JAR file are marked as invalid.

To ignore references, you can specify the "-" wildcard within the resolver spec. The following example specifies that any references to classes within "`my/gui`" are to be allowed, even if it is not present within the resolver spec schema list.

```
loadjava -resolve -resolver '((* SCOTT) (* PUBLIC) ("my/gui/*" -))'
```

In addition, you can define that all classes not found are to be ignored. Without the wildcard, if a dependent class is not found within one of the schemas, your class is listed as invalid and cannot be run. However, this is also dangerous, because if there is a dependent class on a used class, you mark a class as valid that can never run without the dependent class. In this case, you will receive an exception at runtime.

To ignore all classes not found within SCOTT or PUBLIC, specify the following resolver spec:

```
loadjava -resolve -resolver "((* SCOTT) (* PUBLIC) (* -))"
```

Note: Never use a resolver containing "-" if you later intend to load the classes that were causing you to use such a resolver in the first place. Instead, include all referenced classes in the schema before resolving.

ByteCode Verifier

According to the JVM specification, `.class` files are subject to verification before the class they define is available in a JVM. In Oracle JVM, the verification process occurs at class resolution. The resolver might find one of the following problems and issue the appropriate Oracle error code:

ORA-29545	If the resolver determines that the class is malformed, the resolver does not mark it valid. When the resolver rejects a class, it issues an ORA-29545 error (badly formed class). The `loadjava` tool reports the error. For example, this error is thrown if the contents of a `.class` file are not the result of a Java compilation or if the file has been corrupted.
ORA-29552	In some situations, the resolver allows a class to be marked valid, but will replace bytecodes in the class to throw an exception at runtime. In these cases, the resolver issues an ORA-29552 (verification warning), which `loadjava` will report. The `loadjava` tool issues this warning when the Java Language Specification would require an `IncompatibleClassChangeError` be thrown. Oracle JVM relies on the resolver to detect these situations, supporting the proper runtime behavior that the JLS requires.

The resolver also issues the following warnings:

- Resolvers containing "-"

  This type of resolver marks your class valid regardless of whether classes it references are present. Because of inheritance and interfaces, you may want to write valid Java methods that use an instance of a class as if it were an instance of a superclass or of a specific interface. When the method being verified uses a reference to class A as if it were a reference to class B, the resolver must check that A either extends or implements B. For example, consider the following potentially valid method, whose signature implies a return of an instance of B, but whose body returns an instance of A:

  ```
  B myMethod(A a) { return a; }
  ```

  The method is valid only if A extends B, or A implements the interface B. If A or B have been resolved using a "-" term, the resolver does not know that this method is safe. It will replace the bytecodes of `myMethod` with bytecodes that throw an Exception if `myMethod` is ever called.

- Use of other resolvers

  The resolver ensures that the class definitions of A and B are found and resolved properly if they are present in the schemas they specifically identify. The only time you might consider using the alternative resolver is if you must load an existing JAR file containing classes that reference other non-system classes that are not included in the JAR file.

For more information on class resolution and loading your classes within the database, see the *Oracle9i Java Developer's Guide.*

Loading Classes

This section gives an overview of loading your classes into the database using the `loadjava` tool. You can also execute `loadjava` within your SQL. See the *Oracle9i Java Developer's Guide* for complete information on `loadjava`.

Unlike a conventional Java virtual machine, which compiles and loads from files, the Oracle Java virtual machine compiles and loads from database schema objects.

`.java` source files or `.sqlj` source files	correspond to Java source schema objects
`.class` compiled Java files	correspond to Java class schema objects
`.properties` Java resource files, `.ser` SQLJ profile files, or data files	correspond to Java resource schema objects

You must load all classes or resources into the database to be used by other classes within the database. In addition, at loadtime, you define who can execute your classes within the database.

The `loadjava` tool performs the following for each type of file:

Schema Object	loadjava Operations on Object
`.java` source files	**1.** It creates a source schema object within the definer's schema unless another schema is specified. **2.** It loads the contents of the source file into a schema object. **3.** It creates a class schema object for all classes defined in the source file. **4.** If `-resolve` is requested, it does the following: a. It compiles the source schema object. b. It resolves the class and its dependencies. c. It stores the compiled class into a class schema object.
`.sqlj` source files	**1.** It creates a source schema object within the definer's schema unless another schema is specified. **2.** It loads contents of the source file into the schema object. **3.** It creates a class schema object for all classes and resources defined in the source file. **4.** If `-resolve` is requested, it does the following: a. It translates and compiles the source schema object. b. It stores the compiled class into a class schema object. c. It stores the profile into a `.ser` resource schema object and customizes it.
`.class` compiled Java files	**1.** It creates a class schema object within the definer's schema unless another schema is specified. **2.** It loads the class file into the schema object. **3.** It resolves and verifies the class and its dependencies if `-resolve` is specified.
`.properties` Java resource files	**1.** It creates a resource schema object within the definer's schema unless another schema is specified. **2.** It loads a resource file into a schema object.

Schema Object	loadjava Operations on Object
.ser SQLJ profile	1. It creates a resource schema object within the definer's schema unless another schema is specified. 2. It loads the .ser resource file into a schema object and customizes it.

The dropjava tool performs the reverse of the loadjava tool: it deletes schema objects that correspond to Java files. Always use dropjava to delete a Java schema object created with loadjava. Dropping with SQL DDL commands will not update auxiliary data maintained by loadjava and dropjava. You can also execute dropjava from within SQL commands.

> **Note:** More options for loadjava are available. However, this section discusses only the major options. See the *Oracle9i Java Developer's Guide* for complete information on loadjava and dropjava.

You must abide by certain rules, which are detailed in the following sections, when loading classes into the database:

- Defining the Same Class Twice
- Designating Database Privileges and JVM Permissions
- Loading JAR or ZIP Files

After loading, you can access the USER_OBJECTS view in your database schema to verify that your classes and resources loaded properly. For more information, see "Checking Java Uploads" on page 2-16.

Defining the Same Class Twice

You cannot have two different definitions for the same class. This rule affects you in two ways:

- You can load either a particular Java .class file or its .java file, but not both.

  Oracle9*i* tracks whether you loaded a class file or a source file. If you wish to update the class, you must load the same type of file that you originally loaded. If you wish to update the other type, you must drop the first before loading the second. For example, if you loaded x.java as the source for class y, to load x.class, you must first drop x.java.

- You cannot define the same class within two different schema objects within the same schema. For example, suppose `x.java` defines class `y` and you want to move the definition of `y` to `z.java`. If `x.java` has already been loaded, `loadjava` rejects any attempt to load `z.java` (which also defines `y`). Instead, do either of the following:
  - Drop `x.java`, load `z.java` (which defines `y`), then load the new `x.java` (which does not define `y`).
  - Load the new `x.java` (which does not define `y`), then load `z.java` (which defines `y`).

Designating Database Privileges and JVM Permissions

You must have the following SQL database privileges to load classes:

- `CREATE PROCEDURE` and `CREATE TABLE` privileges to load into your schema.
- `CREATE ANY PROCEDURE` and `CREATE ANY TABLE` privileges to load into another schema.
- `oracle.aurora.security.JServerPermission.loadLibraryInClass.<classname>`. See the Security chapter in the *Oracle9i Java Developer's Guide* for more information.

Loading JAR or ZIP Files

The `loadjava` tool accepts `.class`, `.java`, `.properties`, `.sqlj`, `.ser`, `.jar`, or `.zip` files. The JAR or ZIP files can contain source, class, and data files. When you pass `loadjava` a JAR or ZIP file, `loadjava` opens the archive and loads its members individually. There is no JAR or ZIP schema object. If the JAR or ZIP content has not changed since the last time it was loaded, it is not reloaded; therefore, there is little performance penalty for loading JAR or ZIP files. In fact, loading JAR or ZIP files is the simplest way to use `loadjava`.

> **Note:** Oracle does not reload a class if it has not changed since the last load. However, you can force a class to be reloaded through the `loadjava -force` option.

How to Grant Execute Rights

If you load all classes within your own schema and do not reference any class outside of your schema, you already have execution rights. You have the privileges necessary for your objects to invoke other objects loaded in the same schema. That

is, the ability for class A to invoke class B. Class A must be given the right to invoke class B.

The classes that define a Java application are stored within the Oracle9*i* RDBMS under the SQL schema of their owner. By default, classes that reside in one user's schema are not executable by other users, because of security concerns. You can allow other users (schemas) the right to execute your class through the `loadjava -grant` option. You can grant execution rights to a certain user or schema. You cannot grant execution rights to a role, which includes the super-user DBA role. The setting of execution rights is the same as used to grant or revoke privileges in SQL DDL statements.

Figure 2–2 Execution Rights

Method invocation: Class A invokes class B; class B invokes class C.
Execution rights for classes:
* Class A needs execution rights for B.
* Class A does not need execution rights for C.
* Class B needs execution rights for C.

Checking Java Uploads

You can query the database view `USER_OBJECTS` to obtain information about schema objects—including Java sources, classes, and resources—that you own. This allows you, for example, to verify that sources, classes, or resources that you load are properly stored into schema objects.

Columns in `USER_OBJECTS` include those contained in Table 2–2.

Table 2–2 Key USER_OBJECT Columns

Name	Description
OBJECT_NAME	name of the object
OBJECT_TYPE	type of the object (such as `JAVA SOURCE`, `JAVA CLASS`, or `JAVA RESOURCE`)
STATUS	status of the object (`VALID` or `INVALID`) (always `VALID` for `JAVA RESOURCE`)

Object Name and Type

An OBJECT_NAME in USER_OBJECTS is the short name. The full name is stored as a short name if it exceeds 31 characters. See "Shortened Class Names" on page 2-19 for more information on full and short names.

If the server uses a short name for a schema object, you can use the LONGNAME() routine of the server DBMS_JAVA package to receive it from a query in full name format, without having to know the short name format or the conversion rules.

```
SQL*Plus> SELECT dbms_java.longname(object_name) FROM user_objects
          WHERE object_type='JAVA SOURCE';
```

This routine shows you the Java source schema objects in full name format. Where no short name is used, no conversion occurs, because the short name and full name are identical.

You can use the SHORTNAME() routine of the DBMS_JAVA package to use a full name as a query criterion, without having to know whether it was converted to a short name in the database.

```
SQL*Plus> SELECT object_type FROM user_objects
          WHERE object_name=dbms_java.shortname('known_fullname');
```

This routine shows you the OBJECT_TYPE of the schema object of the specified full name. This presumes that the full name is representable in the database character set.

```
SVRMGR> select * from javasnm;
SHORT                          LONGNAME
----------------------------------------------------------------------
/78e6d350_BinaryExceptionHandl sun/tools/java/BinaryExceptionHandler
/b6c774bb_ClassDeclaration     sun/tools/java/ClassDeclaration
/af5a8ef3_JarVerifierStream1   sun/tools/jar/JarVerifierStream$1
```

Status

STATUS is a character string that indicates the validity of a Java schema object. A source schema object is VALID if it compiled successfully; a class schema object is VALID if it was resolved successfully. A resource schema object is always VALID, because resources are not resolved.

Example: Accessing USER_OBJECTS The following SQL*Plus script accesses the USER_OBJECTS view to display information about uploaded Java sources, classes, and resources.

```
COL object_name format a30
COL object_type format a15
SELECT object_name, object_type, status
   FROM user_objects
   WHERE object_type IN ('JAVA SOURCE', 'JAVA CLASS', 'JAVA RESOURCE')
   ORDER BY object_type, object_name;
```

You can optionally use wildcards in querying `USER_OBJECTS`, as in the following example.

```
SELECT object_name, object_type, status
   FROM user_objects
   WHERE object_name LIKE '%Alerter';
```

This routine finds any `OBJECT_NAME` entries that end with the characters: `Alerter`.

User Interfaces on the Server

Oracle9*i* furnishes all core Java class libraries on the server, including those associated with presentation of user interfaces (`java.awt` and `java.applet`). It is, however, inappropriate for code executing in the server to attempt to bring up or materialize a user interface in the server. Imagine thousands of users worldwide exercising an Internet application that executes code that requires someone to click a dialog presented on the server hardware. You can write Java programs that reference and use `java.awt` classes as long as you do not attempt to materialize a user interface.

When building applets, you test them using the `java.awt` and the Peer implementation, which is a platform-specific set of classes for support of a specific windowing system. When the user downloads an applet, it dynamically loads the proper client Peer libraries, and the user sees a display appropriate for the operating system or windowing system in use on the client side. Oracle9*i* takes the same approach. We provide an Oracle-specific Peer implementation that throws an exception, `oracle.aurora.awt.UnsupportedOperation`, if you execute Java code on the Oracle9*i* server that attempts to materialize a user interface.

Oracle9*i*'s lack of support for materializing user interfaces in the server means that we do not pass the Java 2 Compatibility Kit tests for `java.awt`, `java.awt.manual`, and `java.applet`. In the Oracle RDBMS, all user interfaces are supported only on client applications, although they might be displayed on the same physical hardware that supports the server—for example, in the case of

Windows NT. Because it is inappropriate for the server to support user interfaces, we exclude these tests from our complete Java Compatibility Kit testing.

A similar issue exists for vendors of Java-powered embedded devices and in handheld devices (known as Personal Java). Future releases of Java and the Java Compatibility Kit will provide improved factorization of user interface support so that vendors of Java server platforms can better address this issue.

Shortened Class Names

Each Java source, class, and resource is stored in its own schema object in the server. The name of the schema object is derived from the fully qualified name, which includes relevant path or package information. Dots are replaced by slashes. These fully qualified names (with slashes)—used for loaded sources, loaded classes, loaded resources, generated classes, and generated resources—are referred to in this chapter as schema object *full names*.

Schema object names, however, have a maximum of only 31 characters, and all characters must be legal and convertible to characters in the database character set. If any full name is longer than 31 characters or contains illegal or non-convertible characters, the Oracle9*i* server converts the full name to a *short name* to employ as the name of the schema object, keeping track of both names and how to convert between them. If the full name is 31 characters or less and has no illegal or inconvertible characters, then the full name is used as the schema object name.

Because Java classes and methods can have names exceeding the maximum SQL identifier length, Oracle9*i* uses abbreviated names internally for SQL access. Oracle9*i* provides a method within the `DBMS_JAVA` package for retrieving the original Java class name for any truncated name.

```
FUNCTION longname (shortname VARCHAR2) RETURN VARCHAR2
```

This function returns the longname from a Java schema object. An example is to print the fully qualified name of classes that are invalid for some reason.

```
select dbms_java.longname (object_name) from user_objects
   where object_type = 'JAVA CLASS' and status = 'INVALID';
```

In addition, you can specify a full name to the database by using the `shortname()` routine of the `DBMS_JAVA` package, which takes a full name as input and returns the corresponding short name. This is useful when verifying that your classes loaded by querying the `USER_OBJECTS` view.

```
FUNCTION shortname (longname VARCHAR2) RETURN VARCHAR2
```

Controlling the Current User

During execution of Java or PL/SQL, there is always a current user. Initially, this is the user who creates the session.

Invoker's and definer's rights is a SQL concept that is used dynamically when executing SQL, PL/SQL, or JDBC. The current user controls the interpretation of SQL and determines privileges. For example, if a table is referenced by a simple name, it is assumed that the table belongs in the user's schema. In addition, the privileges that are checked when resources are requested are based on the privileges granted to the current user.

In addition, for Java stored procedures, the call specifications use a PL/SQL wrapper. So, you could specify definer's rights on either the call specification or on the Java class itself. If either is redefined to definer's rights, then the called method executes under the user that deployed the Java class.

By default, Java stored procedures execute without changing the current user—that is, with the privileges of their invoker, not their definer. Invoker-rights procedures are not bound to a particular schema. Their unqualified references to schema objects (such as database tables) are resolved in the schema of the current user, not the definer.

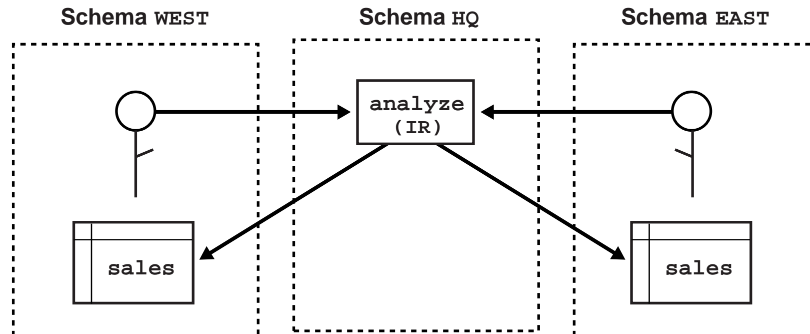
On the other hand, definer-rights procedures are bound to the schema in which they reside. They execute with the privileges of their definer, and their unqualified references to schema objects are resolved in the schema of the definer.

Invoker-rights procedures let you reuse code and centralize application logic. They are especially useful in applications that store data in different schemas. In such cases, multiple users can manage their own data using a single code base.

Consider a company that uses a definer-rights procedure to analyze sales. To provide local sales statistics, the procedure `analyze` must access `sales` tables that reside at each regional site. To do so, the procedure must also reside at each regional site. This causes a maintenance problem.

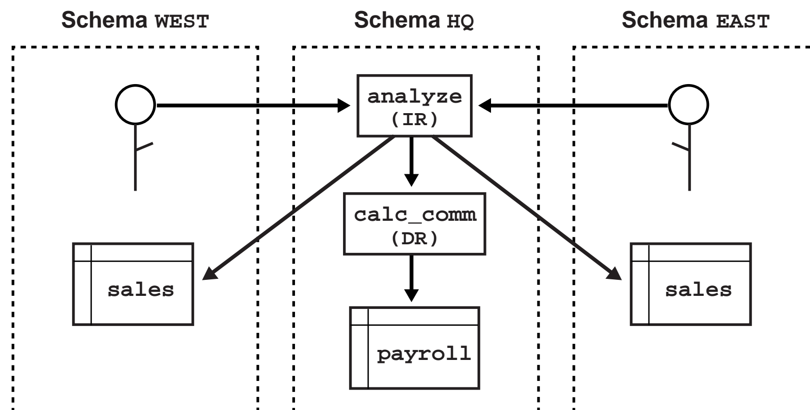
To solve the problem, the company installs an invoker-rights (IR) version of the procedure `analyze` at headquarters. Now, as Figure 2–3 shows, all regional sites can use the same procedure to query their own `sales` tables.

Figure 2–3 Invoker-Rights Solution



Occasionally, you might want to override the default invoker-rights behavior. Suppose headquarters would like the procedure `analyze` to calculate sales commissions and update a central `payroll` table. That presents a problem because invokers of `analyze` should not have direct access to the `payroll` table, which stores employee salaries and other sensitive data. As Figure 2–4 shows, the solution is to have procedure `analyze` call the definer-rights (DR) procedure `calcComm`, which, in turn, updates the `payroll` table.

Figure 2–4 Indirect Access



To override the default invoker-rights behavior, specify the `loadjava` option `-definer`, which is similar to the UNIX facility `setuid`, except that `-definer`

applies to individual classes, not whole programs. Alternatively, you can execute the SQL DDL that changes the AUTHID of the current user.

Different definers can have different privileges, and applications can consist of many classes. So, use the option `-definer` carefully, making sure that classes have only the privileges they need.

3 Publishing Java Classes

Before calling Java methods from SQL, you must publish them in the Oracle data dictionary. When you load a Java class into the database, its methods are not published automatically because Oracle does not know which methods are safe entrypoints for calls from SQL. To publish the methods, you must write call specifications (call specs), which map Java method names, parameter types, and return types to their SQL counterparts.

- Understanding Call Specs
- Defining Call Specs: Basic Requirements
- Writing Top-Level Call Specs
- Writing Packaged Call Specs
- Writing Object Type Call Specs

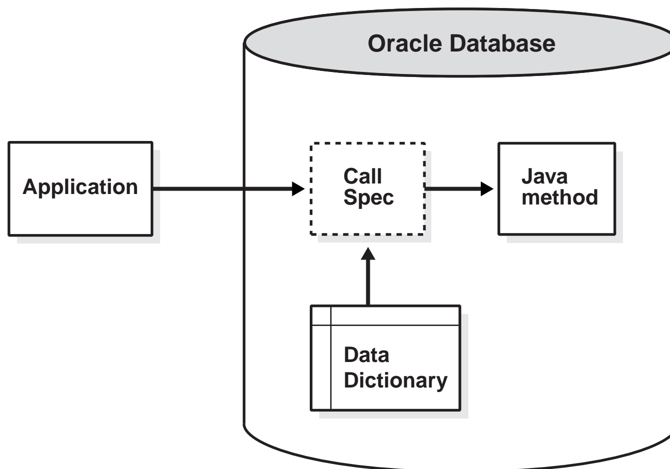
Understanding Call Specs

To publish Java methods, you write call specs. For a given Java method, you declare a function or procedure call spec using the SQL `CREATE FUNCTION` or `CREATE PROCEDURE` statement. Inside a PL/SQL package or SQL object type, you use similar declarations.

You publish value-returning Java methods as functions or procedures and `void` Java methods as procedures. The function or procedure body contains the `LANGUAGE JAVA` clause. This clause records information about the Java method including its full name, its parameter types, and its return type. Mismatches are detected only at run time.

As Figure 3–1 shows, applications call the Java method through its call spec, that is, by referencing the call-spec name. The run-time system looks up the call-spec definition in the Oracle data dictionary, then executes the corresponding Java method.

Figure 3–1 Calling a Java Method



Defining Call Specs: Basic Requirements

A call spec and the Java method it publishes must reside in the same schema (unless the Java method has a `PUBLIC` synonym). You can declare the call spec as a:

- standalone (top-level) PL/SQL function or procedure
- packaged PL/SQL function or procedure
- member method of a SQL object type

A call spec exposes a Java method's top-level entry point to Oracle. Therefore, you can publish only `public static` methods—with one exception. You can publish instance methods as member methods of a SQL object type.

Packaged call specs perform as well as top-level call specs. So, to ease maintenance, you might want to place call specs in a package body. That way, you can modify them without invalidating other schema objects. Also, you can overload them.

Setting Parameter Modes

In Java and other object-oriented languages, a method cannot assign values to objects passed as arguments. So, when calling a method from SQL or PL/SQL, to change the value of an argument, you must declare it as an `OUT` or `IN OUT` parameter in the call spec. The corresponding Java parameter must be a one-element array.

You can replace the element value with another Java object of the appropriate type, or (with `IN OUT` parameters) you can modify the value if the Java type permits. Either way, the new value propagates back to the caller. For example, you might map a call spec `OUT` parameter of type `NUMBER` to a Java parameter declared as `float[] p`, then assign a new value to `p[0]`.

Note: A function that declares `OUT` or `IN OUT` parameters cannot be called from SQL DML statements.

Mapping Datatypes

In a call spec, corresponding SQL and Java parameters (and function results) must have compatible datatypes. Table 3–1 contains all the legal datatype mappings. Oracle converts between the SQL types and Java classes automatically.

Table 3–1 Legal Datatype Mappings

SQL Type	Java Class
CHAR, LONG, VARCHAR2	oracle.sql.CHAR java.lang.String java.sql.Date java.sql.Time java.sql.Timestamp java.lang.Byte java.lang.Short java.lang.Integer java.lang.Long java.lang.Float java.lang.Double java.math.BigDecimal byte, short, int, long, float, double
DATE	oracle.sql.DATE java.sql.Date java.sql.Time java.sql.Timestamp java.lang.String
NUMBER	oracle.sql.NUMBER java.lang.Byte java.lang.Short java.lang.Integer java.lang.Long java.lang.Float java.lang.Double java.math.BigDecimal byte, short, int, long, float, double
OPAQUE	oracle.sql.OPAQUE
RAW, LONG RAW	oracle.sql.RAW byte[]
ROWID	oracle.sql.CHAR oracle.sql.ROWID java.lang.String
BFILE	oracle.sql.BFILE

Table 3–1 Legal Datatype Mappings (Cont.)

SQL Type	Java Class
BLOB	oracle.sql.BLOB oracle.jdbc2.Blob (oracle.jdbc2.Blob under JDK 1.1.x)
CLOB, NCLOB	oracle.sql.CLOB oracle.jdbc2.Clob (oracle.jdbc2.Clob under JDK 1.1.x)
OBJECT Object types and SQLJ types	oracle.sql.STRUCT java.sql.Struct (oracle.jdbc2.Struct under JDK 1.1.x) java.sql.SqlData oracle.sql.ORAData
REF Reference types	oracle.sql.REF java.sql.Ref (oracle.jdbc2.Ref under JDK 1.1.x) oracle.sql.ORAData
TABLE, VARRAY Nested table types and VARRAY types	oracle.sql.ARRAY java.sql.Array (oracle.jdbc2.Array under JDK 1.1.x) oracle.sql.ORAData
any of the preceding SQL types	oracle.sql.CustomDatum oracle.sql.Datum

Table 3–1 Legal Datatype Mappings (Cont.)

SQL Type	Java Class

Notes:

3. The type UROWID and the NUMBER subtypes (INTEGER, REAL, and so on) are not supported.
4. You cannot retrieve a value larger than 32KB from a LONG or LONG RAW database column into a Java stored procedure.
5. The Java wrapper classes (java.lang.Byte, java.lang.Short, and so on) are useful for returning nulls from SQL.
6. When you use the class oracle.sql.CustomDatum to declare parameters, it must define the following member:

```
public static oracle.sql.CustomDatumFactory.getFactory();
```

7. oracle.sql.Datum is an abstract class. The value passed to a parameter of type oracle.sql.Datum must belong to a Java class compatible with the SQL type. Likewise, the value returned by a method with return type oracle.sql.Datum must belong to a Java class compatible with the SQL type.
8. The mappings to oracle.sql classes are optimal because they preserve data formats and require no character-set conversions (apart from the usual network conversions). Those classes are especially useful in applications that "shovel" data between SQL and Java.

Using the Server-Side Internal JDBC Driver

Normally, with JDBC, you establish a connection to the database using the `DriverManager` class, which manages a set of JDBC drivers. Once the JDBC drivers are loaded, you call the method `getConnection`. When it finds the right driver, `getConnection` returns a `Connection` object, which represents a database session. All SQL statements are executed within the context of that session.

However, the server-side internal JDBC driver runs within a default session and default transaction context. So, you are already "connected" to the database, and all your SQL operations are part of the default transaction. You need not register the driver because it comes pre-registered. To get a `Connection` object, simply execute the following statement:

```
Connection conn =
  DriverManager.getConnection("jdbc:default:connection:");
```

Use class `Statement` for SQL statements that take no `IN` parameters and are executed only once. When invoked on a `Connection` object, method `createStatement` returns a new `Statement` object. An example follows:

```
String sql = "DROP " + object_type + " " + object_name;
Statement stmt = conn.createStatement();
stmt.executeUpdate(sql);
```

Use class `PreparedStatement` for SQL statements that take `IN` parameters or are executed more than once. The SQL statement, which can contain one or more parameter placeholders, is precompiled. (Question marks serve as placeholders.) When invoked on a `Connection` object, method `prepareStatement` returns a new `PreparedStatement` object, which contains the precompiled SQL statement. Here is an example:

```
String sql = "DELETE FROM dept WHERE deptno = ?";
PreparedStatement pstmt = conn.prepareStatement(sql);
pstmt.setInt(1, deptID);
pstmt.executeUpdate();
```

A `ResultSet` object contains SQL query results, that is, the rows that met the search condition. You use the method `next` to move to the next row, which becomes the current row. You use the `get`*`XXX`* methods to retrieve column values from the current row. An example follows:

```
String sql = "SELECT COUNT(*) FROM " + tabName;
int rows = 0;
Statement stmt = conn.createStatement();
```

```
ResultSet rset = stmt.executeQuery(sql);
while (rset.next()) {rows = rset.getInt(1);}
```

A `CallableStatement` object lets you call stored procedures. It contains the call text, which can include a return parameter and any number of `IN`, `OUT`, and `INOUT` parameters. The call is written using an escape clause, which is delimited by braces. As the following examples show, the escape syntax has three forms:

```
// parameterless stored procedure
CallableStatement cstmt = conn.prepareCall("{CALL proc}");

// stored procedure
CallableStatement cstmt = conn.prepareCall("{CALL proc(?,?)}");

// stored function
CallableStatement cstmt = conn.prepareCall("{? = CALL func(?,?)}");
```

Important Points

When developing JDBC stored procedure applications, keep the following points in mind:

- The server-side internal JDBC driver runs within a default session and default transaction context. So, you are already "connected" to the database, and all your SQL operations are part of the default transaction. Note that this transaction is a local transaction and not part of a global transaction, such as implemented by JTA or JTS.
- Statements and result sets persist across calls, and their finalizers do not release database cursors. So, to avoid running out of cursors, close all statements and result sets when you are done with them. Alternatively, you can ask your DBA to raise the limit set by the Oracle initialization parameter `OPEN_CURSORS`.
- The server-side internal JDBC driver does not support auto-commits. So, your application must explicitly commit or roll back database changes.
- You cannot connect to a remote database using the server-side internal JDBC driver. You can "connect" only to the server running your Java program. For server-to-server connections, use the server-side JDBC Thin driver. (For client/server connections, use the client-side JDBC Thin or JDBC OCI driver.)
- You cannot close the physical connection to the database established by the server-side internal JDBC driver. However, if you call method `close()` on the default connection, all connection instances (which, in fact, reference the same object) are cleaned up and closed. To get a new connection object, you must call method `getConnection()` again.

For more information, see the *Oracle9i JDBC Developer's Guide and Reference.*

Using the Server-Side SQLJ Translator

The SQLJ translator lets you embed SQL statements in your Java source files using an escape clause, which begins with `#sql`. For example, the SQLJ input file embeds `SELECT` and `CALL` statements in the definition of Java class `TodaysDate`. No explicit connection handling is required for the server-side execution of SQLJ programs.

```
import java.sql.*;
class TodaysDate {
  public static void main (String[] args) {
    try {
      Date today;
      #sql {SELECT SYSDATE INTO :today FROM dual};
      putLine("Today is " + today);
    } catch (Exception e) {putLine("Run-time error: " + e);}
  }

  static void putLine(String s) {
    try {
      #sql {CALL DBMS_OUTPUT.PUT_LINE(:s)};
    } catch (SQLException e) {}
  }
}
```

SQLJ provides the following convenient syntax for calling stored procedures and functions:

```
// parameterless stored procedure
#sql {CALL procedure_name()};

// stored procedure
#sql {CALL procedure_name(parameter, parameter, ...)};

// stored function
#sql result = {VALUES(function_name(parameter, parameter, ...))};
```

where `parameter` stands for the following syntax:

```
{literal | :[{IN | OUT | INOUT}] host_variable_name}
```

If `host_variable_name` is a dot-qualified expression (such as `max.Salary`), it must be enclosed in parentheses.

You can use the client-side SQLJ translator to compile source files and customize profiles. Then, you can upload the resulting class and resource file into the database. Alternatively, you can use the server-side SQLJ translator to compile source files after they are uploaded. If you are writing programs on the client side, the first method is more flexible because most SQLJ translator options are not available on the server side.

Important Points

When developing SQLJ stored procedure applications, keep the following points in mind:

- The SQLJ run-time packages are available automatically on the server. You need not import them to use the run-time classes.
- The current user has an implicit channel to the database. So, you need not create a SQLJ connection-context instance, register a driver, or specify a default connection for your `#sql` statements.
- You cannot connect to a remote database using the server-side internal JDBC driver. You can "connect" only to the server running your Java program. For server-to-server connections, use the server-side JDBC Thin driver. (For client/server connections, use the client-side JDBC Thin or JDBC OCI driver.)
- A SQLJ connection-context instance communicates with the database through the session that is running your Java program, not through a true connection. So, any attempt to close the connection-context instance is ignored, and no exception is thrown.
- Option settings for the server-side SQLJ translator are stored in the database table `JAVA$OPTIONS`. You can get and set the option values using functions and procedures in the utility package `DBMS_JAVA`.
- The server-side SQLJ translator does not support the option `-ser2class`. So, it always generates profiles as resource schema objects, never as class schema objects.
- On the server side (but not on the client side), the SQLJ translator lets you give different names to an input source file and its first public class. However, it is a poor programming practice to use different names.

For more information, see the *Oracle9i SQLJ Developer's Guide and Reference.*

Writing Top-Level Call Specs

In SQL*Plus, you can define top-level call specs interactively using the following syntax:

```
CREATE [OR REPLACE]
{  PROCEDURE procedure_name [(param[, param]...)]
 | FUNCTION function_name [(param[, param]...)] RETURN sql_type}
[AUTHID {DEFINER | CURRENT_USER}]
[PARALLEL_ENABLE]
[DETERMINISTIC]
{IS | AS} LANGUAGE JAVA
NAME 'method_fullname (java_type_fullname[, java_type_fullname]...)
  [return java_type_fullname]';
```

where `param` stands for the following syntax:

```
parameter_name [IN | OUT | IN OUT] sql_type
```

The `AUTHID` clause determines whether a stored procedure executes with the privileges of its definer or invoker (the default) and whether its unqualified references to schema objects are resolved in the schema of the definer or invoker. You can override the default behavior by specifying `DEFINER`. (However, you cannot override the `loadjava` option `-definer` by specifying `CURRENT_USER`.)

The `PARALLEL_ENABLE` option declares that a stored function can be used safely in the slave sessions of parallel DML evaluations. The state of a main (logon) session is never shared with slave sessions. Each slave session has its own state, which is initialized when the session begins. The function result should not depend on the state of session (`static`) variables. Otherwise, results might vary across sessions.

The hint `DETERMINISTIC` helps the optimizer avoid redundant function calls. If a stored function was called previously with the same arguments, the optimizer can elect to use the previous result. The function result should not depend on the state of session variables or schema objects. Otherwise, results might vary across calls. Only `DETERMINISTIC` functions can be called from a function-based index or a materialized view that has query-rewrite enabled. For more information, see the statements `CREATE INDEX` and `CREATE MATERIALIZED VIEW` in the *Oracle9i SQL Reference*.

The `NAME`-clause string uniquely identifies the Java method. The Java full names and the call spec parameters, which are mapped by position, must correspond one to one. (This rule does not apply to method `main`. See Example 2 on page 3-13.) If the Java method takes no arguments, code an empty parameter list for it but *not* for the function or procedure.

As usual, you write Java full names using dot notation. The following example shows that long names can be broken across lines at dot boundaries:

```
artificialIntelligence.neuralNetworks.patternClassification.
  RadarSignatureClassifier.computeRange()
```

Example 1

Assume that the executable for the following Java class has been loaded into the Oracle database:

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

public class GenericDrop {
  public static void dropIt (String object_type, String object_name)
  throws SQLException {
    // Connect to Oracle using JDBC driver
    Connection conn =
      DriverManager.getConnection("jdbc:default:connection:");
    // Build SQL statement
    String sql = "DROP " + object_type + " " + object_name;
    try {
      Statement stmt = conn.createStatement();
      stmt.executeUpdate(sql);
      stmt.close();
    } catch (SQLException e) {System.err.println(e.getMessage());}
  }
}
```

Class `GenericDrop` has one method named `dropIt`, which drops any kind of schema object. For example, if you pass the arguments `'table'` and `'emp'` to `dropIt`, the method drops database table `emp` from your schema. Let's write a call spec for this method.

```
CREATE OR REPLACE PROCEDURE drop_it (
  obj_type VARCHAR2,
  obj_name VARCHAR2)
AS LANGUAGE JAVA
NAME 'GenericDrop.dropIt(java.lang.String, java.lang.String)';
```

Notice that you must fully qualify the reference to class `String`. Package `java.lang` is automatically available to Java programs but must be named explicitly in call specs.

Example 2

As a rule, Java names and call spec parameters must correspond one to one. However, that rule does not apply to method `main`. Its `String[]` parameter can be mapped to multiple `CHAR` or `VARCHAR2` call spec parameters. Suppose you want to publish the following method `main`, which prints its arguments:

```
public class EchoInput {
  public static void main (String[] args) {
    for (int i = 0; i < args.length; i++)
      System.out.println(args[i]);
  }
}
```

To publish method `main`, you might write the following call spec:

```
CREATE OR REPLACE PROCEDURE echo_input (
  s1 VARCHAR2,
  s2 VARCHAR2,
  s3 VARCHAR2)
AS LANGUAGE JAVA
NAME 'EchoInput.main(java.lang.String[])';
```

You cannot impose constraints (such as precision, size, or `NOT NULL`) on call spec parameters. So, you cannot specify a maximum size for the `VARCHAR2` parameters, even though you must do so for `VARCHAR2` variables, as in:

```
DECLARE
  last_name VARCHAR2(20);  -- size constraint required
```

Example 3

Next, you publish Java method `rowCount`, which returns the number of rows in a given database table.

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

public class RowCounter {
  public static int rowCount (String tabName) throws SQLException {
    Connection conn =
      DriverManager.getConnection("jdbc:default:connection:");
    String sql = "SELECT COUNT(*) FROM " + tabName;
    int rows = 0;
```

```
    try {
      Statement stmt = conn.createStatement();
      ResultSet rset = stmt.executeQuery(sql);
      while (rset.next()) {rows = rset.getInt(1);}
      rset.close();
      stmt.close();
    } catch (SQLException e) {System.err.println(e.getMessage());}
    return rows;
  }
}
```

In the following call spec, the return type is `NUMBER`, not `INTEGER`, because `NUMBER` subtypes (such as `INTEGER`, `REAL`, and `POSITIVE`) are *not* allowed in a call spec:

```
CREATE FUNCTION row_count (tab_name VARCHAR2) RETURN NUMBER
AS LANGUAGE JAVA
NAME 'RowCounter.rowCount(java.lang.String) return int';
```

Example 4

Suppose you want to publish the following Java method named `swap`, which switches the values of its arguments:

```
public class Swapper {
  public static void swap (int[] x, int[] y) {
    int hold = x[0];
    x[0] = y[0];
    y[0] = hold;
  }
}
```

The call spec publishes Java method `swap` as call spec `swap`. The call spec declares `IN OUT` formal parameters because values must be passed in and out. All call spec `OUT` and `IN OUT` parameters must map to Java array parameters.

```
CREATE PROCEDURE swap (x IN OUT NUMBER, y IN OUT NUMBER)
AS LANGUAGE JAVA
NAME 'Swapper.swap(int[], int[])';
```

Notice that a Java method and its call spec can have the same name.

Writing Packaged Call Specs

A PL/SQL package is a schema object that groups logically related types, items, and subprograms. Usually, packages have two parts, a *specification* (*spec*) and a *body* (sometimes the body is unnecessary). The spec is the interface to your applications: it declares the types, constants, variables, exceptions, cursors, and subprograms available for use. The body fully defines cursors and subprograms, thereby implementing the spec. (For details, see the *PL/SQL User's Guide and Reference.*)

In SQL*Plus, you can define PL/SQL packages interactively using this syntax:

```
CREATE [OR REPLACE] PACKAGE package_name
  [AUTHID {CURRENT_USER | DEFINER}] {IS | AS}
  [type_definition [type_definition] ...]
  [cursor_spec [cursor_spec] ...]
  [item_declaration [item_declaration] ...]
  [{subprogram_spec | call_spec} [{subprogram_spec | call_spec}]...]
END [package_name];

[CREATE [OR REPLACE] PACKAGE BODY package_name {IS | AS}
  [type_definition [type_definition] ...]
  [cursor_body [cursor_body] ...]
  [item_declaration [item_declaration] ...]
  [{subprogram_spec | call_spec} [{subprogram_spec | call_spec}]...]
[BEGIN
  sequence_of_statements]
END [package_name];]
```

The spec holds public declarations, which are visible to your application. The body contains implementation details and private declarations, which are hidden from your application. Following the declarative part of the package body is the optional initialization part, which typically holds statements that initialize package variables. It is run only once, the first time you reference the package.

A call spec declared in a package spec cannot have the same signature (name and parameter list) as a subprogram in the package body. If you declare all the subprograms in a package spec as call specs, the package body is unnecessary (unless you want to define a cursor or use the initialization part).

The `AUTHID` clause determines whether all the packaged subprograms execute with the privileges of their definer (the default) or invoker, and whether their unqualified references to schema objects are resolved in the schema of the definer or invoker.

An Example

Consider the Java class `DeptManager`, which has methods for adding a new department, dropping a department, and changing the location of a department. Notice that method `addDept` uses a database sequence to get the next department number. The three methods are logically related, so you might want to group their call specs in a PL/SQL package.

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

public class DeptManager {
  public static void addDept (String deptName, String deptLoc)
  throws SQLException {
    Connection conn =
      DriverManager.getConnection("jdbc:default:connection:");
    String sql = "SELECT deptnos.NEXTVAL FROM dual";
    String sql2 = "INSERT INTO dept VALUES (?, ?, ?)";
    int deptID = 0;
    try {
      PreparedStatement pstmt = conn.prepareStatement(sql);
      ResultSet rset = pstmt.executeQuery();
      while (rset.next()) {deptID = rset.getInt(1);}
      pstmt = conn.prepareStatement(sql2);
      pstmt.setInt(1, deptID);
      pstmt.setString(2, deptName);
      pstmt.setString(3, deptLoc);
      pstmt.executeUpdate();
      rset.close();
      pstmt.close();
    } catch (SQLException e) {System.err.println(e.getMessage());}
  }

  public static void dropDept (int deptID) throws SQLException {
    Connection conn =
      DriverManager.getConnection("jdbc:default:connection:");
    String sql = "DELETE FROM dept WHERE deptno = ?";
    try {
      PreparedStatement pstmt = conn.prepareStatement(sql);
      pstmt.setInt(1, deptID);
      pstmt.executeUpdate();
      pstmt.close();
    } catch (SQLException e) {System.err.println(e.getMessage());}
  }
```

```
  public static void changeLoc (int deptID, String newLoc)
  throws SQLException {
    Connection conn =
      DriverManager.getConnection("jdbc:default:connection:");
    String sql = "UPDATE dept SET loc = ? WHERE deptno = ?";
    try {
      PreparedStatement pstmt = conn.prepareStatement(sql);
      pstmt.setString(1, newLoc);
      pstmt.setInt(2, deptID);
      pstmt.executeUpdate();
      pstmt.close();
    } catch (SQLException e) {System.err.println(e.getMessage());}
  }
}
```

Suppose you want to package methods `addDept`, `dropDept`, and `changeLoc`. First, you create the package spec, as follows:

```
CREATE OR REPLACE PACKAGE dept_mgmt AS
  PROCEDURE add_dept (dept_name VARCHAR2, dept_loc VARCHAR2);
  PROCEDURE drop_dept (dept_id NUMBER);
  PROCEDURE change_loc (dept_id NUMBER, new_loc VARCHAR2);
END dept_mgmt;
```

Then, you create the package body by writing call specs for the Java methods:

```
CREATE OR REPLACE PACKAGE BODY dept_mgmt AS
  PROCEDURE add_dept (dept_name VARCHAR2, dept_loc VARCHAR2)
  AS LANGUAGE JAVA
  NAME 'DeptManager.addDept(java.lang.String, java.lang.String)';

  PROCEDURE drop_dept (dept_id NUMBER)
  AS LANGUAGE JAVA
  NAME 'DeptManager.dropDept(int)';

  PROCEDURE change_loc (dept_id NUMBER, new_loc VARCHAR2)
  AS LANGUAGE JAVA
  NAME 'DeptManager.changeLoc(int, java.lang.String)';
END dept_mgmt;
```

To reference the stored procedures in the package `dept_mgmt`, you must use dot notation, as the following example shows:

```
CALL dept_mgmt.add_dept('PUBLICITY', 'DALLAS');
```

Writing Object Type Call Specs

In SQL, object-oriented programming is based on object types, which are user-defined composite data types that encapsulate a data structure along with the functions and procedures needed to manipulate the data. The variables that form the data structure are known as *attributes*. The functions and procedures that characterize the behavior of the object type are known as *methods*, which can be written in Java.

SQLJ Types are implemented to support automatic generation. You can create SQL types that correspond to Java types, which implement SQLData. SQLJ object types are discussed in *Oracle9i JDBC Developer's Guide and Reference*. In addition, you can use JPublisher as a convenient method for creating classes that implement SQL data. See *Oracle9i JPublisher User's Guide* for more information.

As with a package, an object type has two parts: a specification (spec) and a body. The spec is the interface to your applications; it declares a data structure (set of attributes) along with the operations (methods) needed to manipulate the data. The body implements the spec by defining PL/SQL subprogram bodies or call specs. (For details, see the *PL/SQL User's Guide and Reference*.)

If an object type spec declares only attributes or call specs, then the object type body is unnecessary. (You cannot declare attributes in the body.) So, if you implement all your methods in Java, you can place their call specs in the object type spec and omit the body.

In SQL*Plus, you can define SQL object types interactively using this syntax:

```
CREATE [OR REPLACE] TYPE type_name
  [AUTHID {CURRENT_USER | DEFINER}] {IS | AS} OBJECT (
  attribute_name datatype[, attribute_name datatype]...
  [{MAP | ORDER} MEMBER {function_spec | call_spec},]
  [{MEMBER | STATIC} {subprogram_spec | call_spec}
  [, {MEMBER | STATIC} {subprogram_spec | call_spec}]...]
);

[CREATE [OR REPLACE] TYPE BODY type_name {IS | AS}
  { {MAP | ORDER} MEMBER function_body;
   | {MEMBER | STATIC} {subprogram_body | call_spec};}
  [{MEMBER | STATIC} {subprogram_body | call_spec};]...
END;]
```

The `AUTHID` clause determines whether all member methods execute with the current user privileges—which determines invoker's or definer's rights.

Declaring Attributes

In an object type spec, all attributes must be declared before any methods. At least one attribute is required (the maximum is 1000). Methods are optional.

As with a Java variable, you declare an attribute with a name and datatype. The name must be unique within the object type but can be reused in other object types. The datatype can be any SQL type except LONG, LONG RAW, NCHAR, NVARCHAR2, NCLOB, ROWID, or UROWID.

You cannot initialize an attribute in its declaration using the assignment operator or DEFAULT clause. Furthermore, you cannot impose the NOT NULL constraint on an attribute. However, objects can be stored in database tables on which you can impose constraints.

Declaring Methods

MEMBER methods accept a built-in parameter known as SELF, which is an instance of the object type. Whether declared implicitly or explicitly, it is always the first parameter passed to a MEMBER method. In the method body, SELF denotes the object whose method was invoked. MEMBER methods are invoked on instances, as follows:

```
instance_expression.method()
```

However, STATIC methods, which cannot accept or reference SELF, are invoked on the object type, not its instances, as follows:

```
object_type_name.method()
```

If you want to call a non-static Java method, you specify the keyword MEMBER in its call spec. Likewise, if you want to call a static Java method, you specify the keyword STATIC in its call spec.

Map and Order Methods

The values of a SQL scalar datatype such as `CHAR` have a predefined order, which allows them to be compared. However, instances of an object type have no predefined order. To put them in order, SQL calls a user-defined *map method*.

SQL uses the ordering to evaluate Boolean expressions such as `x > y` and to make comparisons implied by the `DISTINCT`, `GROUP BY`, and `ORDER BY` clauses. A map method returns the relative position of an object in the ordering of all such objects. An object type can contain only one map method, which must be a parameterless function with one of the following return types: `DATE`, `NUMBER`, or `VARCHAR2`.

Alternatively, you can supply SQL with an *order method*, which compares two objects. Every order method takes just two parameters: the built-in parameter `SELF` and another object of the same type. If `o1` and `o2` are objects, a comparison such as `o1 > o2` calls the order method automatically. The method returns a negative number, zero, or a positive number signifying that `SELF` is respectively less than, equal to, or greater than the other parameter. An object type can contain only one order method, which must be a function that returns a numeric result.

You can declare a map method or an order method but not both. If you declare either method, you can compare objects in SQL and PL/SQL. However, if you declare neither method, you can compare objects only in SQL and solely for equality or inequality. (Two objects of the same type are equal if the values of their corresponding attributes are equal.)

Constructor Methods

Every object type has a *constructor method* (*constructor* for short), which is a system-defined function with the same name as the object type. The constructor initializes and returns an instance of that object type.

Oracle generates a default constructor for every object type. The formal parameters of the constructor match the attributes of the object type. That is, the parameters and attributes are declared in the same order and have the same names and datatypes. SQL never calls a constructor implicitly, so you must call it explicitly. Constructor calls are allowed wherever function calls are allowed.

Note: To invoke a Java constructor from SQL, you must wrap calls to it in a `static` method and declare the corresponding call spec as a `STATIC` member of the object type.

Examples

In this section, each example builds on the previous one. To begin, you create two SQL object types to represent departments and employees. First, you write the spec for object type `Department`. The body is unnecessary because the spec declares only attributes.

```
CREATE TYPE Department AS OBJECT (
  deptno NUMBER(2),
  dname  VARCHAR2(14),
  loc    VARCHAR2(13)
);
```

Then, you create object type `Employee`. Its last attribute, `deptno`, stores a handle, called a *ref*, to objects of type `Department`. A ref indicates the location of an object in an *object table*, which is a database table that stores instances of an object type. The ref does not point to a specific instance copy in memory. To declare a ref, you specify the datatype `REF` and the object type that the ref targets.

```
CREATE TYPE Employee AS OBJECT (
  empno    NUMBER(4),
  ename    VARCHAR2(10),
  job      VARCHAR2(9),
  mgr      NUMBER(4),
  hiredate DATE,
  sal      NUMBER(7,2),
  comm     NUMBER(7,2),
  deptno   REF Department
);
```

Next, you create SQL object tables to hold objects of type `Department` and `Employee`. First, you create object table `depts`, which will hold objects of type `Department`. You populate the object table by selecting data from the relational table `dept` and passing it to a constructor, which is a system-defined function with the same name as the object type. You use the constructor to initialize and return an instance of that object type.

```
CREATE TABLE depts OF Department AS
  SELECT Department(deptno, dname, loc) FROM dept;
```

Finally, you create the object table `emps`, which will hold objects of type `Employee`. The last column in object table `emps`, which corresponds to the last attribute of object type `Employee`, holds references to objects of type `Department`. To fetch the references into that column, you use the operator REF, which takes as its argument a table alias associated with a row in an object table.

```
CREATE TABLE emps OF Employee AS
  SELECT Employee(e.empno, e.ename, e.job, e.mgr, e.hiredate, e.sal,
    e.comm, (SELECT REF(d) FROM depts d WHERE d.deptno = e.deptno))
    FROM emp e;
```

Selecting a ref returns a handle to an object; it does not materialize the object itself. To do that, you can use methods in class `oracle.sql.REF`, which supports Oracle object references. This class, which is a subclass of `oracle.sql.Datum`, extends the standard JDBC interface `oracle.jdbc2.Ref`. For more information, see the *Oracle9i JDBC Developer's Guide and Reference.*

Using Class oracle.sql.STRUCT

To continue, you write a Java stored procedure. The class `Paymaster` has one method, which computes an employee's wages. The method `getAttributes()` defined in class `oracle.sql.STRUCT` uses the default JDBC mappings for the attribute types. So, for example, `NUMBER` maps to `BigDecimal`.

```
import java.sql.*;
import java.io.*;
import oracle.sql.*;
import oracle.jdbc.*;
import oracle.oracore.*;
import oracle.jdbc2.*;
import java.math.*;

public class Paymaster {
  public static BigDecimal wages(STRUCT e)
    throws java.sql.SQLException {
    // Get the attributes of the Employee object.
    Object[] attribs = e.getAttributes();
    // Must use numeric indexes into the array of attributes.
    BigDecimal sal = (BigDecimal)(attribs[5]);  // [5] = sal
    BigDecimal comm = (BigDecimal)(attribs[6]); // [6] = comm
    BigDecimal pay = sal;
    if (comm != null) pay = pay.add(comm);
    return pay;
  }
}
```

Because the method `wages` returns a value, you write a function call spec for it, as follows:

```
CREATE OR REPLACE FUNCTION wages (e Employee) RETURN NUMBER AS
  LANGUAGE JAVA
  NAME 'Paymaster.wages(oracle.sql.STRUCT) return BigDecimal';
```

This is a top-level call spec because it is not defined inside a package or object type.

Implementing the SQLData Interface

To make access to object attributes more natural, you can create a Java class that implements the `SQLData` interface. To do so, you must provide the methods `readSQL()` and `writeSQL()` as defined by the `SQLData` interface. The JDBC driver calls method `readSQL()` to read a stream of database values and populate an instance of your Java class. (For details, see the *Oracle9i JDBC Developer's Guide and Reference*) In the following example, you revise class `Paymaster`, adding a second method named `raiseSal()`:

```
import java.sql.*;
import java.io.*;
import oracle.sql.*;
import oracle.jdbc.*;
import oracle.oracore.*;
import oracle.jdbc2.*;
import java.math.*;

public class Paymaster implements SQLData {
  // Implement the attributes and operations for this type.
  private BigDecimal empno;
  private String ename;
  private String job;
  private BigDecimal mgr;
  private Date hiredate;
  private BigDecimal sal;
  private BigDecimal comm;
  private Ref dept;

  public static BigDecimal wages(Paymaster e) {
    BigDecimal pay = e.sal;
    if (e.comm != null) pay = pay.add(e.comm);
    return pay;
  }
```

```
  public static void raiseSal(Paymaster[] e, BigDecimal amount) {
    e[0].sal =              // IN OUT passes [0]
      e[0].sal.add(amount); // increase salary by given amount
  }

  // Implement SQLData interface.

  private String sql_type;

  public String getSQLTypeName() throws SQLException {
    return sql_type;
  }

  public void readSQL(SQLInput stream, String typeName)
    throws SQLException {
    sql_type = typeName;
    empno = stream.readBigDecimal();
    ename = stream.readString();
    job = stream.readString();
    mgr = stream.readBigDecimal();
    hiredate = stream.readDate();
    sal = stream.readBigDecimal();
    comm = stream.readBigDecimal();
    dept = stream.readRef();
  }

  public void writeSQL(SQLOutput stream) throws SQLException {
    stream.writeBigDecimal(empno);
    stream.writeString(ename);
    stream.writeString(job);
    stream.writeBigDecimal(mgr);
    stream.writeDate(hiredate);
    stream.writeBigDecimal(sal);
    stream.writeBigDecimal(comm);
    stream.writeRef(dept);
  }
}
```

You must revise the call spec for method `wages`, as follows, because its parameter has changed from `oracle.sql.STRUCT` to `Paymaster`:

```
CREATE OR REPLACE FUNCTION wages (e Employee) RETURN NUMBER AS
  LANGUAGE JAVA
  NAME 'Paymaster.wages(Paymaster) return BigDecimal';
```

Because the new method `raiseSal` is void, you write a procedure call spec for it, as follows:

```
CREATE OR REPLACE PROCEDURE raise_sal (e IN OUT Employee, r NUMBER)
  AS LANGUAGE JAVA
  NAME 'Paymaster.raiseSal(Paymaster[], java.math.BigDecimal)';
```

Again, this is a top-level call spec.

Implementing Object Type Methods

Later, you decide to drop the top-level call specs `wages` and `raise_sal` and redeclare them as methods of object type `Employee`. In an object type spec, all methods must be declared after the attributes. The object type body is unnecessary because the spec declares only attributes and call specs.

```
CREATE TYPE Employee AS OBJECT (
  empno    NUMBER(4),
  ename    VARCHAR2(10),
  job      VARCHAR2(9),
  mgr      NUMBER(4),
  hiredate DATE,
  sal      NUMBER(7,2),
  comm     NUMBER(7,2),
  deptno   REF Department
  MEMBER FUNCTION wages RETURN NUMBER
    AS LANGUAGE JAVA
    NAME 'Paymaster.wages() return java.math.BigDecimal',
  MEMBER PROCEDURE raise_sal (r NUMBER)
    AS LANGUAGE JAVA
    NAME 'Paymaster.raiseSal(java.math.BigDecimal)'
);
```

Then, you revise class `Paymaster` accordingly. You need not pass an array to method `raiseSal` because the SQL parameter `SELF` corresponds directly to the Java parameter `this`—even when `SELF` is declared as `IN OUT` (the default for procedures).

```
import java.sql.*;
import java.io.*;
import oracle.sql.*;
import oracle.jdbc.*;
import oracle.oracore.*;
import oracle.jdbc2.*;
import java.math.*;
```

```
public class Paymaster implements SQLData {
  // Implement the attributes and operations for this type.
  private BigDecimal empno;
  private String ename;
  private String job;
  private BigDecimal mgr;
  private Date hiredate;
  private BigDecimal sal;
  private BigDecimal comm;
  private Ref dept;

  public BigDecimal wages() {
    BigDecimal pay = sal;
    if (comm != null) pay = pay.add(comm);
    return pay;
  }

  public void raiseSal(BigDecimal amount) {
    // For SELF/this, even when IN OUT, no array is needed.
    sal = sal.add(amount);
  }

  // Implement SQLData interface.

  String sql_type;

  public String getSQLTypeName() throws SQLException {
    return sql_type;
  }

  public void readSQL(SQLInput stream, String typeName)
    throws SQLException {
    sql_type = typeName;
    empno = stream.readBigDecimal();
    ename = stream.readString();
    job = stream.readString();
    mgr = stream.readBigDecimal();
    hiredate = stream.readDate();
    sal = stream.readBigDecimal();
    comm = stream.readBigDecimal();
    dept = stream.readRef();
  }
```

```
  public void writeSQL(SQLOutput stream) throws SQLException {
    stream.writeBigDecimal(empno);
    stream.writeString(ename);
    stream.writeString(job);
    stream.writeBigDecimal(mgr);
    stream.writeDate(hiredate);
    stream.writeBigDecimal(sal);
    stream.writeBigDecimal(comm);
    stream.writeRef(dept);
  }
}
```

4

Calling Stored Procedures

After you load and publish a Java stored procedure, you can call it. This chapter demonstrates how to call Java stored procedures in various contexts. You learn how to call them from the top level and from database triggers, SQL DML statements, and PL/SQL blocks. You also learn how SQL exceptions are handled.

- Calling Java from the Top Level
- Calling Java from Database Triggers
- Calling Java from SQL DML
- Calling Java from PL/SQL
- Calling PL/SQL from Java
- How the JVM Handles Exceptions

Calling Java from the Top Level

The SQL `CALL` statement lets you call Java methods published at the top level, in PL/SQL packages, or in SQL object types. In SQL*Plus, you can execute the `CALL` statement interactively using the syntax:

```
CALL [schema_name.][{package_name | object_type_name}][@dblink_name]
{  procedure_name ([param[, param]...])
 | function_name ([param[, param]...]) INTO :host_variable};
```

where `param` stands for the following syntax:

```
{literal | :host_variable}
```

Host variables (that is, variables declared in a host environment) must be prefixed with a colon. The following examples show that a host variable cannot appear twice in the same `CALL` statement, and that a parameterless subprogram must be called with an empty parameter list:

```
CALL swap(:x, :x);  -- illegal, duplicate host variables
CALL balance() INTO :current_balance;  -- () required
```

Redirecting Output

On the server, the default output device is a trace file, not the user screen. As a result, `System.out` and `System.err` print to the current trace files. To redirect output to the SQL*Plus text buffer, call the procedure `set_output()` in package `DBMS_JAVA`, as follows:

```
SQL> SET SERVEROUTPUT ON
SQL> CALL dbms_java.set_output(2000);
```

The minimum (and default) buffer size is 2,000 bytes; the maximum size is 1,000,000 bytes. In the following example, the buffer size is increased to 5,000 bytes:

```
SQL> SET SERVEROUTPUT ON SIZE 5000
SQL> CALL dbms_java.set_output(5000);
```

Output is printed when the stored procedure exits.

For more information about SQL*Plus, see the *SQL*Plus User's Guide and Reference.*

Example 1

In the following example, the method `main` accepts the name of a database table (such as `'emp'`) and an optional `WHERE` clause condition (such as `'sal > 1500'`).

If you omit the condition, the method deletes all rows from the table. Otherwise, the method deletes only those rows that meet the condition.

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

public class Deleter {
  public static void main (String[] args) throws SQLException {
    Connection conn =
      DriverManager.getConnection("jdbc:default:connection:");
    String sql = "DELETE FROM " + args[0];
    if (args.length > 1) sql += " WHERE " + args[1];
    try {
      Statement stmt = conn.createStatement();
      stmt.executeUpdate(sql);
      stmt.close();
    } catch (SQLException e) {System.err.println(e.getMessage());}
  }
}
```

The method `main` can take either one or two arguments. Normally, the `DEFAULT` clause is used to vary the number of arguments passed to a PL/SQL subprogram. However, that clause is *not* allowed in a call spec. So, you must overload two packaged procedures (you cannot overload top-level procedures), as follows:

```
CREATE OR REPLACE PACKAGE pkg AS
  PROCEDURE delete_rows (table_name VARCHAR2);
  PROCEDURE delete_rows (table_name VARCHAR2, condition VARCHAR2);
END;

CREATE OR REPLACE PACKAGE BODY pkg AS
  PROCEDURE delete_rows (table_name VARCHAR2)
  AS LANGUAGE JAVA
  NAME 'Deleter.main(java.lang.String[])';

  PROCEDURE delete_rows (table_name VARCHAR2, condition VARCHAR2)
  AS LANGUAGE JAVA
  NAME 'Deleter.main(java.lang.String[])';
END;
```

Now, you are ready to call the procedure `delete_rows`:

```
SQL> CALL pkg.delete_rows('emp', 'sal > 1500');

Call completed.

SQL> SELECT ename, sal FROM emp;

ENAME           SAL
--------- --------
SMITH           800
WARD           1250
MARTIN         1250
TURNER         1500
ADAMS          1100
JAMES           950
MILLER         1300

7 rows selected.
```

Example 2

Assume that the executable for the following Java class is stored in the Oracle database:

```
public class Fibonacci {
  public static int fib (int n) {
    if (n == 1 || n == 2)
      return 1;
    else
      return fib(n - 1) + fib(n - 2);
  }
}
```

The class `Fibonacci` has one method named `fib`, which returns the *n*th Fibonacci number. The Fibonacci sequence (1, 1, 2, 3, 5, 8, 13, 21, . . .), which was first used to model the growth of a rabbit colony, is recursive. Each term in the sequence (after the second) is the sum of the two terms that immediately precede it. Because the method `fib` returns a value, you publish it as a function:

```
CREATE OR REPLACE FUNCTION fib (n NUMBER) RETURN NUMBER
AS LANGUAGE JAVA
NAME 'Fibonacci.fib(int) return int';
```

Next, you declare two SQL*Plus host variables, then initialize the first one:

```
SQL> VARIABLE n NUMBER
SQL> VARIABLE f NUMBER
SQL> EXECUTE :n := 7;

PL/SQL procedure successfully completed.
```

Finally, you are ready to call the function `fib`. Remember, in a `CALL` statement, host variables must be prefixed with a colon.

```
SQL> CALL fib(:n) INTO :f;

Call completed.

SQL> PRINT f

         F
----------
        13
```

Calling Java from Database Triggers

A *database trigger* is a stored program associated with a specific table or view. Oracle executes (fires) the trigger automatically whenever a given DML operation affects the table or view.

A trigger has three parts: a triggering event (DML operation), an optional trigger constraint, and a trigger action. When the event occurs, the trigger fires and either a PL/SQL block or a `CALL` statement performs the action. A *statement trigger* fires once, before or after the triggering event. A *row trigger* fires once for each row affected by the triggering event.

Within a database trigger, you can reference the new and old values of changing rows using the correlation names `new` and `old`. In the trigger-action block or `CALL` statement, column names must be prefixed with `:new` or `:old`.

To create a database trigger, you use the SQL `CREATE TRIGGER` statement. For the syntax of that statement, see the *Oracle9i SQL Reference.* For a full discussion of database triggers, see the *Oracle9i Application Developer's Guide - Fundamentals.*

Example 1

Suppose you want to create a database trigger that uses the following Java class to log out-of-range salary increases:

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

public class DBTrigger {
  public static void logSal (int empID, float oldSal, float newSal)
  throws SQLException {
    Connection conn =
      DriverManager.getConnection("jdbc:default:connection:");
    String sql = "INSERT INTO sal_audit VALUES (?, ?, ?)";
    try {
      PreparedStatement pstmt = conn.prepareStatement(sql);
      pstmt.setInt(1, empID);
      pstmt.setFloat(2, oldSal);
      pstmt.setFloat(3, newSal);
      pstmt.executeUpdate();
      pstmt.close();
    } catch (SQLException e) {System.err.println(e.getMessage());}
  }
}
```

The class `DBTrigger` has one method, which inserts a row into the database table `sal_audit`. Because `logSal` is a void method, you publish it as a procedure:

```
CREATE OR REPLACE PROCEDURE log_sal (
  emp_id NUMBER, old_sal NUMBER, new_sal NUMBER)
AS LANGUAGE JAVA
NAME 'DBTrigger.logSal(int, float, float)';
```

Next, you create the database table `sal_audit`, as follows:

```
CREATE TABLE sal_audit (
  empno  NUMBER,
  oldsal NUMBER,
  newsal NUMBER);
```

Finally, you create the database trigger, which fires when a salary increase exceeds twenty percent:

```
CREATE OR REPLACE TRIGGER sal_trig
AFTER UPDATE OF sal ON emp
FOR EACH ROW
WHEN (new.sal > 1.2 * old.sal)
CALL log_sal(:new.empno, :old.sal, :new.sal);
```

When you execute the following `UPDATE` statement, it updates all rows in the table `emp`. For each row that meets the trigger's `WHEN` clause condition, the trigger fires and the Java method inserts a row into the table `sal_audit`.

```
SQL> UPDATE emp SET sal = sal + 300;

SQL> SELECT * FROM sal_audit;

     EMPNO     OLDSAL     NEWSAL
---------- ---------- ----------
      7369        800       1100
      7521       1250       1550
      7654       1250       1550
      7876       1100       1400
      7900        950       1250
      7934       1300       1600

6 rows selected.
```

Example 2

Suppose you want to create a trigger that inserts rows into a database view defined as follows:

```
CREATE VIEW emps AS
  SELECT empno, ename, 'Sales' AS dname FROM sales
  UNION ALL
  SELECT empno, ename, 'Marketing' AS dname FROM mktg;
```

where the database tables `sales` and `mktg` are defined as:

```
CREATE TABLE sales (empno NUMBER(4), ename VARCHAR2(10));
CREATE TABLE mktg (empno NUMBER(4), ename VARCHAR2(10));
```

You must write an `INSTEAD OF` trigger because rows cannot be inserted into a view that uses set operators such as `UNION ALL`. Instead, your trigger will insert rows into the base tables.

First, you add the following Java method to the class `DBTrigger` (defined in the previous example):

```
public static void addEmp (
  int empNo, String empName, String deptName)
throws SQLException {
  Connection conn =
    DriverManager.getConnection("jdbc:default:connection:");
  String tabName = (deptName.equals("Sales") ? "sales" : "mktg");
  String sql = "INSERT INTO " + tabName + " VALUES (?, ?)";
  try {
    PreparedStatement pstmt = conn.prepareStatement(sql);
    pstmt.setInt(1, empNo);
    pstmt.setString(2, empName);
    pstmt.executeUpdate();
    pstmt.close();
  } catch (SQLException e) {System.err.println(e.getMessage());}
}
```

The method `addEmp` inserts a row into the table `sales` or `mktg` depending on the value of the parameter `deptName`. You write the call spec for this method as follows:

```
CREATE OR REPLACE PROCEDURE add_emp (
  emp_no NUMBER, emp_name VARCHAR2, dept_name VARCHAR2)
AS LANGUAGE JAVA
NAME 'DBTrigger.addEmp(int, java.lang.String, java.lang.String)';
```

Then, you create the `INSTEAD OF` trigger:

```
CREATE OR REPLACE TRIGGER emps_trig
INSTEAD OF INSERT ON emps
FOR EACH ROW
CALL add_emp(:new.empno, :new.ename, :new.dname);
```

When you execute each of the following `INSERT` statements, the trigger fires and the Java method inserts a row into the appropriate base table:

```
SQL> INSERT INTO emps VALUES (8001, 'Chand', 'Sales');
SQL> INSERT INTO emps VALUES (8002, 'Van Horn', 'Sales');
SQL> INSERT INTO emps VALUES (8003, 'Waters', 'Sales');
SQL> INSERT INTO emps VALUES (8004, 'Bellock', 'Marketing');
SQL> INSERT INTO emps VALUES (8005, 'Perez', 'Marketing');
SQL> INSERT INTO emps VALUES (8006, 'Foucault', 'Marketing');

SQL> SELECT * FROM sales;

     EMPNO ENAME
---------- ----------
      8001 Chand
      8002 Van Horn
      8003 Waters

SQL> SELECT * FROM mktg;

     EMPNO ENAME
---------- ----------
      8004 Bellock
      8005 Perez
      8006 Foucault

SQL> SELECT * FROM emps;

     EMPNO ENAME      DNAME
---------- ---------- ---------
      8001 Chand      Sales
      8002 Van Horn   Sales
      8003 Waters     Sales
      8004 Bellock    Marketing
      8005 Perez      Marketing
      8006 Foucault   Marketing
```

Calling Java from SQL DML

If you publish Java methods as functions, you can call them from SQL SELECT, INSERT, UPDATE, DELETE, CALL, EXPLAIN PLAN, LOCK TABLE, and MERGE statements. For example, assume that the executable for the following Java class is stored in the Oracle database:

```
public class Formatter {
  public static String formatEmp (String empName, String jobTitle) {
    empName = empName.substring(0,1).toUpperCase() +
      empName.substring(1).toLowerCase();
    jobTitle = jobTitle.toLowerCase();
    if (jobTitle.equals("analyst"))
      return (new String(empName + " is an exempt analyst"));
    else
      return (new String(empName + " is a non-exempt " + jobTitle));
  }
}
```

The class Formatter has one method named formatEmp, which returns a formatted string containing a staffer's name and job status. First, you write the call spec for this method as follows:

```
CREATE OR REPLACE FUNCTION format_emp (ename VARCHAR2, job VARCHAR2)
  RETURN VARCHAR2
AS LANGUAGE JAVA
NAME 'Formatter.formatEmp (java.lang.String, java.lang.String)
  return java.lang.String';
```

Then, you call the function format_emp to format a list of employees:

```
SQL> SELECT format_emp(ename, job) AS "Employees" FROM emp
  2    WHERE job NOT IN ('MANAGER', 'PRESIDENT') ORDER BY ename;

Employees
--------------------------------------------
Adams is a non-exempt clerk
Allen is a non-exempt salesman
Ford is an exempt analyst
James is a non-exempt clerk
Martin is a non-exempt salesman
Miller is a non-exempt clerk
Scott is an exempt analyst
Smith is a non-exempt clerk
Turner is a non-exempt salesman
Ward is a non-exempt salesman
```

Restrictions

To be callable from SQL DML statements, a Java method must obey the following "purity" rules, which are meant to control side effects:

- When you call it from a `SELECT` statement or a parallelized `INSERT`, `UPDATE`, or `DELETE` statement, the method cannot modify any database tables.
- When you call it from an `INSERT`, `UPDATE`, or `DELETE` statement, the method cannot query or modify any database tables modified by that statement.
- When you call it from a `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statement, the method cannot execute SQL transaction control statements (such as `COMMIT`), session control statements (such as `SET ROLE`), or system control statements (such as `ALTER SYSTEM`). In addition, it cannot execute DDL statements (such as `CREATE`) because they are followed by an automatic commit.

If any SQL statement inside the method violates a rule, you get an error at run time (when the statement is parsed).

Calling Java from PL/SQL

You can call Java stored procedures from any PL/SQL block, subprogram, or package. For example, assume that the executable for the following Java class is stored in the Oracle database:

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

public class Adjuster {
  public static void raiseSalary (int empNo, float percent)
  throws SQLException {
    Connection conn =
      DriverManager.getConnection("jdbc:default:connection:");
    String sql = "UPDATE emp SET sal = sal * ? WHERE empno = ?";
    try {
      PreparedStatement pstmt = conn.prepareStatement(sql);
      pstmt.setFloat(1, (1 + percent / 100));
      pstmt.setInt(2, empNo);
      pstmt.executeUpdate();
      pstmt.close();
    } catch (SQLException e) {System.err.println(e.getMessage());}
  }
}
```

The class `Adjuster` has one method, which raises the salary of an employee by a given percentage. Because `raiseSalary` is a `void` method, you publish it as a procedure, as follows:

```
CREATE OR REPLACE PROCEDURE raise_salary (empno NUMBER, pct NUMBER)
AS LANGUAGE JAVA
NAME 'Adjuster.raiseSalary(int, float)';
```

In the following example, you call the procedure `raise_salary` from an anonymous PL/SQL block:

```
DECLARE
   emp_id  NUMBER;
   percent NUMBER;
BEGIN
   -- get values for emp_id and percent
   raise_salary(emp_id, percent);
   ...
END;
```

In the next example, you call the function `row_count` (defined in "Example 3" on page 3-13) from a standalone PL/SQL stored procedure:

```
CREATE PROCEDURE calc_bonus (emp_id NUMBER, bonus OUT NUMBER) AS
  emp_count NUMBER;
  ...
BEGIN
  emp_count := row_count('emp');
  ...
END;
```

In the final example, you call the `raise_sal` method of object type `Employee` (defined in "Implementing Object Type Methods" on page 3-25) from an anonymous PL/SQL block:

```
DECLARE
  emp_id NUMBER(4);
  v emp_type;
BEGIN
  -- assign a value to emp_id
  SELECT VALUE(e) INTO v FROM emps e WHERE empno = emp_id;
  v.raise_sal(500);
  UPDATE emps e SET e = v WHERE empno = emp_id;
  ...
END;
```

Calling PL/SQL from Java

JDBC and SQLJ allow you to call PL/SQL stored functions and procedures. For example, suppose you want to call the following stored function, which returns the balance of a specified bank account:

```
FUNCTION balance (acct_id NUMBER) RETURN NUMBER IS
  acct_bal NUMBER;
BEGIN
  SELECT bal INTO acct_bal FROM accts
    WHERE acct_no = acct_id;
  RETURN acct_bal;
END;
```

From a JDBC program, your call to the function `balance` might look like this:

```
CallableStatement cstmt = conn.prepareCall("{? = CALL balance(?)}");
cstmt.registerOutParameter(1, Types.FLOAT);
cstmt.setInt(2, acctNo);
cstmt.executeUpdate();
float acctBal = cstmt.getFloat(1);
```

From a SQLJ program, the call might look like this:

```
#sql acctBal = {VALUES(balance(:IN acctNo))};
```

To learn more about JDBC, see the *Oracle9i JDBC Developer's Guide and Reference.* To learn more about SQLJ, see the *Oracle9i SQLJ Developer's Guide and Reference.*

How the JVM Handles Exceptions

Java exceptions are objects, so they have classes as their types. As with other Java classes, exception classes have a naming and inheritance hierarchy. Therefore, you can substitute a subexception (subclass) for its superexception (superclass).

All Java exception objects support the method `toString()`, which returns the fully qualified name of the exception class concatenated to an optional string. Typically, the string contains data-dependent information about the exceptional condition. Usually, the code that constructs the exception associates the string with it.

When a Java stored procedure executes a SQL statement, any exception thrown is materialized to the procedure as a subclass of `java.sql.SQLException`. That class has the methods `getErrorCode()` and `getMessage()`, which return the Oracle error code and message, respectively.

If a stored procedure called from SQL or PL/SQL throws an exception not caught by Java, the caller gets the following error message:

```
ORA-29532 Java call terminated by uncaught Java exception
```

This is how all uncaught exceptions (including non-SQL exceptions) are reported.

5

Developing an Application

This chapter demonstrates the building of a Java stored procedures application. The example is based on a simple business activity: managing customer purchase orders. By following along from design to implementation, you learn enough to start writing your own applications.

- Drawing the Entity-Relationship Diagram
- Planning the Database Schema
- Creating the Database Tables
- Writing the Java Classes
- Loading the Java Classes
- Publishing the Java Classes
- Calling the Java Stored Procedures

Drawing the Entity-Relationship Diagram

The objective is to develop a simple system for managing customer purchase orders. First, you must identify the business entities involved and their relationships. To do that, you draw an entity-relationship (E-R) diagram by following the rules and examples given in Figure 5–1.

Figure 5–1 Rules for Drawing an E-R Diagram

Definitions:

entity something about which data is collected, stored, and maintained

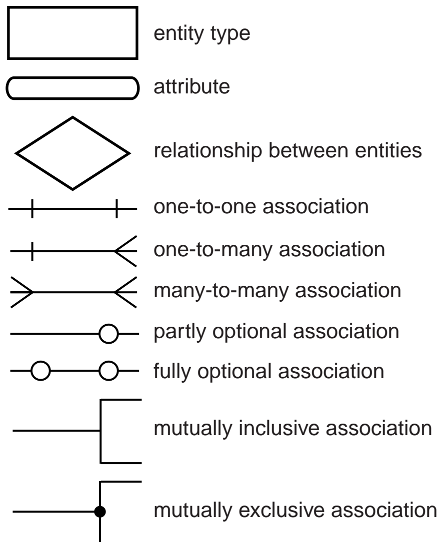
attribute a characteristic of an entity

relationship an association between entities

entity type a class of entities that have the same set of attributes

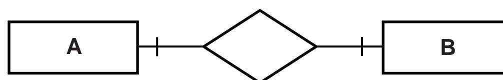
record an ordered set of attribute values that describe an instance of an entity type

Symbols:

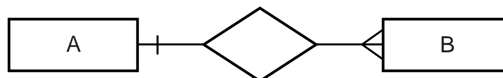


Examples:

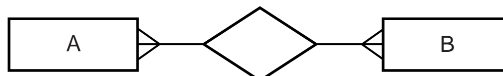
One A is associated with one B:



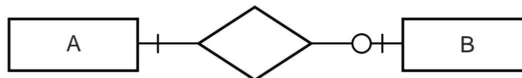
One A is associated with one or more B's:



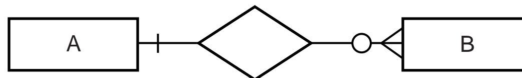
One or more A's are associated with one or more B's:



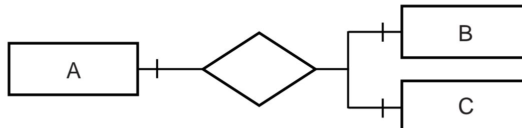
One A is associated with zero or one B:



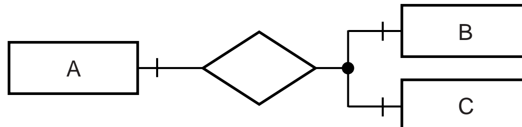
One A is associated with zero or more B's:



One A is associated with one B and one C:

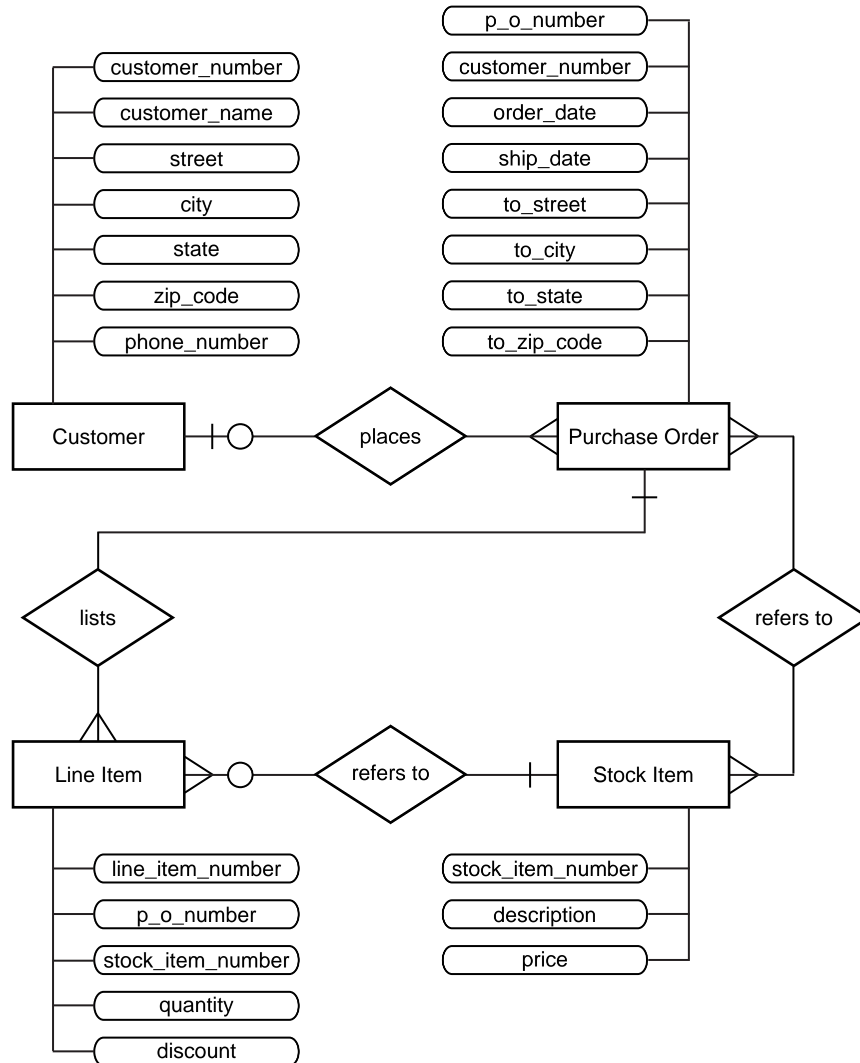


One A is associated with one B or one C (but not both):



As Figure 5–2 illustrates, the basic entities in this example are customers, purchase orders, line items, and stock items.

Figure 5–2 E-R Diagram for Purchase Order Application



A `Customer` has a one-to-many relationship with a `Purchase Order` because a customer can place many orders, but a given purchase order can be placed by only one customer. The relationship is optional because zero customers might place a given order (it might be placed by someone not previously defined as a customer).

A `Purchase Order` has a many-to-many relationship with a `Stock Item` because a purchase order can refer to many stock items, and a stock item can be referred to by many purchase orders. However, you do not know which purchase orders refer to which stock items.

Therefore, you introduce the notion of a `Line Item`. A `Purchase Order` has a one-to-many relationship with a `Line Item` because a purchase order can list many line items, but a given line item can be listed by only one purchase order.

A `LineItem` has a many-to-one relationship with a `StockItem` because a line item can refer to only one stock item, but a given stock item can be referred to by many line items. The relationship is optional because zero line items might refer to a given stock item.

Planning the Database Schema

Next, you must devise a schema plan. To do that, you decompose the E-R diagram into the following database tables:

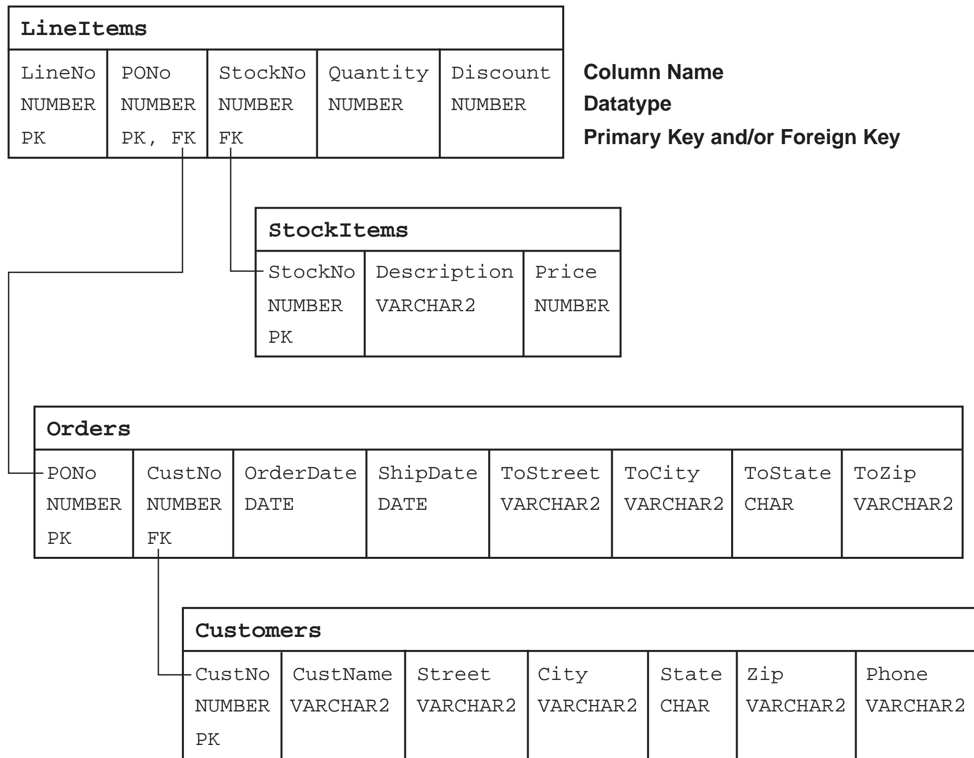
- `Customers`
- `Orders`
- `LineItems`
- `StockItems`

For example, you assign `Customer` attributes to columns in the table `Customers`.

Figure 5–3 on page 5-6 depicts the relationships between tables. The E-R diagram showed that a line item has a relationship with a purchase order and with a stock item. In the schema plan, you establish these relationships using primary and foreign keys.

A *primary key* is a column (or combination of columns) whose values uniquely identify each row in a table. A *foreign key* is a column (or combination of columns) whose values match the primary key in some other table. For example, column `PONo` in table `LineItems` is a foreign key matching the primary key in table `Orders`. Every purchase order number in column `LineItems.PONo` must also appear in column `Orders.PONo`.

Figure 5–3 Schema Plan for Purchase Order Application



Creating the Database Tables

Next, you create the database tables required by the schema plan. You begin by defining the table `Customers`, as follows:

```
CREATE TABLE Customers (
  CustNo   NUMBER(3) NOT NULL,
  CustName VARCHAR2(30) NOT NULL,
  Street   VARCHAR2(20) NOT NULL,
  City     VARCHAR2(20) NOT NULL,
  State    CHAR(2) NOT NULL,
  Zip      VARCHAR2(10) NOT NULL,
  Phone    VARCHAR2(12),
  PRIMARY KEY (CustNo)
);
```

The table `Customers` stores all the information about customers. Essential information is defined as `NOT NULL`. For example, every customer must have a shipping address. However, the table `Customers` does not manage the relationship between a customer and his or her purchase order. So, that relationship must be managed by the table `Orders`, which you define as:

```
CREATE TABLE Orders (
  PONo      NUMBER(5),
  Custno    NUMBER(3) REFERENCES Customers,
  OrderDate DATE,
  ShipDate  DATE,
  ToStreet  VARCHAR2(20),
  ToCity    VARCHAR2(20),
  ToState   CHAR(2),
  ToZip     VARCHAR2(10),
  PRIMARY KEY (PONo)
);
```

The E-R diagram in Figure 5–2 showed that line items have a relationship with purchase orders and stock items. The table `LineItems` manages these relationships using foreign keys. For example, the foreign key (FK) column `StockNo` in the table `LineItems` references the primary key (PK) column `StockNo` in the table `StockItems`, which you define as:

```
CREATE TABLE StockItems (
  StockNo     NUMBER(4) PRIMARY KEY,
  Description VARCHAR2(20),
  Price       NUMBER(6,2))
);
```

The table `Orders` manages the relationship between a customer and purchase order using the FK column `CustNo`, which references the PK column `CustNo` in the table `Customers`. However, the table `Orders` does not manage the relationship between a purchase order and its line items. So, that relationship must be managed by the table `LineItems`, which you define as:

```
CREATE TABLE LineItems (
  LineNo    NUMBER(2),
  PONo      NUMBER(5) REFERENCES Orders,
  StockNo   NUMBER(4) REFERENCES StockItems,
  Quantity  NUMBER(2),
  Discount  NUMBER(4,2),
  PRIMARY KEY (LineNo, PONo)
);
```

Writing the Java Classes

Next, you consider the operations needed in a purchase order system, then you write appropriate Java methods. In a simple system based on the tables defined in the previous section, you need methods for registering customers, stocking parts, entering orders, and so on. You implement these methods in the Java class `POManager`, as follows:

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

public class POManager {
  public static void addCustomer (int custNo, String custName,
    String street, String city, String state, String zipCode,
    String phoneNo) throws SQLException {
      String sql = "INSERT INTO Customers VALUES (?,?,?,?,?,?,?)";
      try {
        Connection conn =
          DriverManager.getConnection("jdbc:default:connection:");
        PreparedStatement pstmt = conn.prepareStatement(sql);
        pstmt.setInt(1, custNo);
        pstmt.setString(2, custName);
        pstmt.setString(3, street);
        pstmt.setString(4, city);
        pstmt.setString(5, state);
        pstmt.setString(6, zipCode);
        pstmt.setString(7, phoneNo);
        pstmt.executeUpdate();
        pstmt.close();
      } catch (SQLException e) {System.err.println(e.getMessage());}
  }
```

```
public static void addStockItem (int stockNo, String description,
  float price) throws SQLException {
    String sql = "INSERT INTO StockItems VALUES (?,?,?)";
    try {
      Connection conn =
        DriverManager.getConnection("jdbc:default:connection:");
      PreparedStatement pstmt = conn.prepareStatement(sql);
      pstmt.setInt(1, stockNo);
      pstmt.setString(2, description);
      pstmt.setFloat(3, price);
      pstmt.executeUpdate();
      pstmt.close();
    } catch (SQLException e) {System.err.println(e.getMessage());}
}

public static void enterOrder (int orderNo, int custNo,
  String orderDate, String shipDate, String toStreet,
  String toCity, String toState, String toZipCode)
  throws SQLException {
    String sql = "INSERT INTO Orders VALUES (?,?,?,?,?,?,?,?)";
    try {
      Connection conn =
        DriverManager.getConnection("jdbc:default:connection:");
      PreparedStatement pstmt = conn.prepareStatement(sql);
      pstmt.setInt(1, orderNo);
      pstmt.setInt(2, custNo);
      pstmt.setString(3, orderDate);
      pstmt.setString(4, shipDate);
      pstmt.setString(5, toStreet);
      pstmt.setString(6, toCity);
      pstmt.setString(7, toState);
      pstmt.setString(8, toZipCode);
      pstmt.executeUpdate();
      pstmt.close();
    } catch (SQLException e) {System.err.println(e.getMessage());}
}
```

```
public static void addLineItem (int lineNo, int orderNo,
  int stockNo, int quantity, float discount) throws SQLException {
    String sql = "INSERT INTO LineItems VALUES (?,?,?,?,?)";
    try {
      Connection conn =
        DriverManager.getConnection("jdbc:default:connection:");
      PreparedStatement pstmt = conn.prepareStatement(sql);
      pstmt.setInt(1, lineNo);
      pstmt.setInt(2, orderNo);
      pstmt.setInt(3, stockNo);
      pstmt.setInt(4, quantity);
      pstmt.setFloat(5, discount);
      pstmt.executeUpdate();
      pstmt.close();
    } catch (SQLException e) {System.err.println(e.getMessage());}
}

public static void totalOrders () throws SQLException {
  String sql =
    "SELECT O.PONo, ROUND(SUM(S.Price * L.Quantity)) AS TOTAL " +
    "FROM Orders O, LineItems L, StockItems S " +
    "WHERE O.PONo = L.PONo AND L.StockNo = S.StockNo " +
    "GROUP BY O.PONo";
  try {
    Connection conn =
      DriverManager.getConnection("jdbc:default:connection:");
    PreparedStatement pstmt = conn.prepareStatement(sql);
    ResultSet rset = pstmt.executeQuery();
    printResults(rset);
    rset.close();
    pstmt.close();
  } catch (SQLException e) {System.err.println(e.getMessage());}
}
```

```
static void printResults (ResultSet rset) throws SQLException {
  String buffer = "";
  try {
    ResultSetMetaData meta = rset.getMetaData();
    int cols = meta.getColumnCount(), rows = 0;
    for (int i = 1; i <= cols; i++) {
      int size = meta.getPrecision(i);
      String label = meta.getColumnLabel(i);
      if (label.length() > size) size = label.length();
      while (label.length() < size) label += " ";
      buffer = buffer + label + "  ";
    }
    buffer = buffer + "\n";
    while (rset.next()) {
      rows++;
      for (int i = 1; i <= cols; i++) {
        int size = meta.getPrecision(i);
        String label = meta.getColumnLabel(i);
        String value = rset.getString(i);
        if (label.length() > size) size = label.length();
        while (value.length() < size) value += " ";
        buffer = buffer + value + "  ";
      }
      buffer = buffer + "\n";
    }
    if (rows == 0) buffer = "No data found!\n";
    System.out.println(buffer);
  } catch (SQLException e) {System.err.println(e.getMessage());}
}
```

```
public static void checkStockItem (int stockNo)
  throws SQLException {
    String sql = "SELECT O.PONo, O.CustNo, L.StockNo, " +
      "L.LineNo, L.Quantity, L.Discount " +
      "FROM Orders O, LineItems L " +
      "WHERE O.PONo = L.PONo AND L.StockNo = ?";
    try {
      Connection conn =
        DriverManager.getConnection("jdbc:default:connection:");
      PreparedStatement pstmt = conn.prepareStatement(sql);
      pstmt.setInt(1, stockNo);
      ResultSet rset = pstmt.executeQuery();
      printResults(rset);
      rset.close();
      pstmt.close();
    } catch (SQLException e) {System.err.println(e.getMessage());}
}

public static void changeQuantity (int newQty, int orderNo,
  int stockNo) throws SQLException {
    String sql = "UPDATE LineItems SET Quantity = ? " +
      "WHERE PONo = ? AND StockNo = ?";
    try {
      Connection conn =
        DriverManager.getConnection("jdbc:default:connection:");
      PreparedStatement pstmt = conn.prepareStatement(sql);
      pstmt.setInt(1, newQty);
      pstmt.setInt(2, orderNo);
      pstmt.setInt(3, stockNo);
      pstmt.executeUpdate();
      pstmt.close();
    } catch (SQLException e) {System.err.println(e.getMessage());}
}
```

```
  public static void deleteOrder (int orderNo) throws SQLException {
    String sql = "DELETE FROM LineItems WHERE PONo = ?";
    try {
      Connection conn =
        DriverManager.getConnection("jdbc:default:connection:");
      PreparedStatement pstmt = conn.prepareStatement(sql);
      pstmt.setInt(1, orderNo);
      pstmt.executeUpdate();
      sql = "DELETE FROM Orders WHERE PONo = ?";
      pstmt = conn.prepareStatement(sql);
      pstmt.setInt(1, orderNo);
      pstmt.executeUpdate();
      pstmt.close();
    } catch (SQLException e) {System.err.println(e.getMessage());}
  }
}
```

Loading the Java Classes

Next, you use the command-line utility `loadjava` to upload your Java stored procedures into the Oracle database, as follows:

```
> loadjava -u scott/tiger@myPC:1521:orcl -v -r -t POManager.java
initialization complete
loading  : POManager
creating : POManager
resolver : resolver ( ("*" scott) ("*" public) ("*" -))
resolving: POManager
```

Recall that option `-v` enables verbose mode, that option `-r` compiles uploaded Java source files and resolves external references in the classes, and that option `-t` tells `loadjava` to connect to the database using the client-side JDBC Thin driver.

Publishing the Java Classes

Next, you must publish your Java stored procedures in the Oracle data dictionary. To do that, you write call specs, which map Java method names, parameter types, and return types to their SQL counterparts.

The methods in the Java class `POManager` are logically related, so you group their call specs in a PL/SQL package. First, you create the package spec, as follows:

```
CREATE OR REPLACE PACKAGE po_mgr AS
  PROCEDURE add_customer (cust_no NUMBER, cust_name VARCHAR2,
    street VARCHAR2, city VARCHAR2, state CHAR, zip_code VARCHAR2,
    phone_no VARCHAR2);
  PROCEDURE add_stock_item (stock_no NUMBER, description VARCHAR2,
    price NUMBER);
  PROCEDURE enter_order (order_no NUMBER, cust_no NUMBER,
    order_date VARCHAR2, ship_date VARCHAR2, to_street VARCHAR2,
    to_city VARCHAR2, to_state CHAR, to_zip_code VARCHAR2);
  PROCEDURE add_line_item (line_no NUMBER, order_no NUMBER,
    stock_no NUMBER, quantity NUMBER, discount NUMBER);
  PROCEDURE total_orders;
  PROCEDURE check_stock_item (stock_no NUMBER);
  PROCEDURE change_quantity (new_qty NUMBER, order_no NUMBER,
    stock_no NUMBER);
  PROCEDURE delete_order (order_no NUMBER);
END po_mgr;
```

Then, you create the package body by writing call specs for the Java methods:

```
CREATE OR REPLACE PACKAGE BODY po_mgr AS
  PROCEDURE add_customer (cust_no NUMBER, cust_name VARCHAR2,
    street VARCHAR2, city VARCHAR2, state CHAR, zip_code VARCHAR2,
    phone_no VARCHAR2) AS LANGUAGE JAVA
  NAME 'POManager.addCustomer(int, java.lang.String,
    java.lang.String, java.lang.String, java.lang.String,
    java.lang.String, java.lang.String)';

  PROCEDURE add_stock_item (stock_no NUMBER, description VARCHAR2,
    price NUMBER) AS LANGUAGE JAVA
  NAME 'POManager.addStockItem(int, java.lang.String, float)';
```

```
  PROCEDURE enter_order (order_no NUMBER, cust_no NUMBER,
    order_date VARCHAR2, ship_date VARCHAR2, to_street VARCHAR2,
    to_city VARCHAR2, to_state CHAR, to_zip_code VARCHAR2)
  AS LANGUAGE JAVA
  NAME 'POManager.enterOrder(int, int, java.lang.String,
    java.lang.String, java.lang.String, java.lang.String,
    java.lang.String, java.lang.String)';

  PROCEDURE add_line_item (line_no NUMBER, order_no NUMBER,
    stock_no NUMBER, quantity NUMBER, discount NUMBER)
  AS LANGUAGE JAVA
  NAME 'POManager.addLineItem(int, int, int, int, float)';

  PROCEDURE total_orders
  AS LANGUAGE JAVA
  NAME 'POManager.totalOrders()';

  PROCEDURE check_stock_item (stock_no NUMBER)
  AS LANGUAGE JAVA
  NAME 'POManager.checkStockItem(int)';

  PROCEDURE change_quantity (new_qty NUMBER, order_no NUMBER,
    stock_no NUMBER) AS LANGUAGE JAVA
  NAME 'POManager.changeQuantity(int, int, int)';

  PROCEDURE delete_order (order_no NUMBER)
  AS LANGUAGE JAVA
  NAME 'POManager.deleteOrder(int)';
END po_mgr;
```

Calling the Java Stored Procedures

Now, you can call your Java stored procedures from the top level and from database triggers, SQL DML statements, and PL/SQL blocks. To reference the stored procedures in the package `po_mgr`, you must use dot notation.

From an anonymous PL/SQL block, you might start the new purchase order system by stocking parts, as follows:

```
BEGIN
  po_mgr.add_stock_item(2010, 'camshaft', 245.00);
  po_mgr.add_stock_item(2011, 'connecting rod', 122.50);
  po_mgr.add_stock_item(2012, 'crankshaft', 388.25);
  po_mgr.add_stock_item(2013, 'cylinder head', 201.75);
  po_mgr.add_stock_item(2014, 'cylinder sleeve', 73.50);
  po_mgr.add_stock_item(2015, 'engine bearning', 43.85);
  po_mgr.add_stock_item(2016, 'flywheel', 155.00);
  po_mgr.add_stock_item(2017, 'freeze plug', 17.95);
  po_mgr.add_stock_item(2018, 'head gasket', 36.75);
  po_mgr.add_stock_item(2019, 'lifter', 96.25);
  po_mgr.add_stock_item(2020, 'oil pump', 207.95);
  po_mgr.add_stock_item(2021, 'piston', 137.75);
  po_mgr.add_stock_item(2022, 'piston ring', 21.35);
  po_mgr.add_stock_item(2023, 'pushrod', 110.00);
  po_mgr.add_stock_item(2024, 'rocker arm', 186.50);
  po_mgr.add_stock_item(2025, 'valve', 68.50);
  po_mgr.add_stock_item(2026, 'valve spring', 13.25);
  po_mgr.add_stock_item(2027, 'water pump', 144.50);
  COMMIT;
END;
```

Then, you register your customers:

```
BEGIN
  po_mgr.add_customer(101, 'A-1 Automotive', '4490 Stevens Blvd',
    'San Jose', 'CA', '95129', '408-555-1212');
  po_mgr.add_customer(102, 'AutoQuest', '2032 America Ave',
    'Hayward', 'CA', '94545', '510-555-1212');
  po_mgr.add_customer(103, 'Bell Auto Supply', '305 Cheyenne Ave',
    'Richardson', 'TX', '75080', '972-555-1212');
  po_mgr.add_customer(104, 'CarTech Auto Parts', '910 LBJ Freeway',
    'Dallas', 'TX', '75234', '214-555-1212');
  COMMIT;
END;
```

Next, you enter purchase orders placed by various customers:

```
BEGIN
  po_mgr.enter_order(30501, 103, '14-SEP-1998', '21-SEP-1998',
    '305 Cheyenne Ave', 'Richardson', 'TX', '75080');
  po_mgr.add_line_item(01, 30501, 2011, 5, 0.02);
  po_mgr.add_line_item(02, 30501, 2018, 25, 0.10);
  po_mgr.add_line_item(03, 30501, 2026, 10, 0.05);

  po_mgr.enter_order(30502, 102, '15-SEP-1998', '22-SEP-1998',
    '2032 America Ave', 'Hayward', 'CA', '94545');
  po_mgr.add_line_item(01, 30502, 2013, 1, 0.00);
  po_mgr.add_line_item(02, 30502, 2014, 1, 0.00);

  po_mgr.enter_order(30503, 104, '15-SEP-1998', '23-SEP-1998',
    '910 LBJ Freeway', 'Dallas', 'TX', '75234');
  po_mgr.add_line_item(01, 30503, 2020, 5, 0.02);
  po_mgr.add_line_item(02, 30503, 2027, 5, 0.02);
  po_mgr.add_line_item(03, 30503, 2021, 15, 0.05);
  po_mgr.add_line_item(04, 30503, 2022, 15, 0.05);

  po_mgr.enter_order(30504, 101, '16-SEP-1998', '23-SEP-1998',
    '4490 Stevens Blvd', 'San Jose', 'CA', '95129');
  po_mgr.add_line_item(01, 30504, 2025, 20, 0.10);
  po_mgr.add_line_item(02, 30504, 2026, 20, 0.10);
  COMMIT;
END;
```

Finally, in SQL*Plus, after redirecting output to the SQL*Plus text buffer, you might call the Java method `totalOrders` as follows:

```
SQL> SET SERVEROUTPUT ON
SQL> CALL dbms_java.set_output(2000);
...
SQL> CALL po_mgr.total_orders();
PONO   TOTAL
30501  1664
30502  275
30503  4149
30504  1635

Call completed.
```

Index

A

Accelerator, 1-14
application
  compiling, 2-4
  developing, 5-1
  execution rights, 2-16
attributes, 1-6, 3-18
  declaring, 3-19
AUTHID clause, 3-11, 3-15, 3-18

B

body
  package, 3-15
  SQL object type, 3-18
bytecode
  definition, 1-12
  verification, 2-11

C

call specs, 1-3
  basic requirements for defining, 3-3
  definition, 1-21
  example, 1-21
  understanding, 3-2
  writing object type, 3-18
  writing packaged, 3-15
  writing top-level, 3-11
class
  loader, 1-13
  loading, 2-12
  marking valid, 2-9
  name, 2-19
  resolving references, 2-9
  schema object, 2-3, 2-9, 2-12, 2-13
.class files, 2-3, 2-12, 2-13
CLASSPATH, 2-3
compiling, 1-12, 2-4
  error messages, 2-5
  options, 2-5
  runtime, 2-4
components, Oracle JVM, 1-9
constructor methods, 3-20
contexts, stored procedure run-time, 1-3
conventions, notational, xi
CREATE JAVA statement, 2-1

D

database
  Java, 2-2
  sample tables, xiii
  schema plan, 5-5
  triggers, 1-5, 4-6
database triggers
  calling Java from, 4-6
datatypes
  mapping, 3-4
DBMS_JAVA package, 4-2
  longname method, 2-17, 2-19
  shortname method, 2-17, 2-19
debug
  compiler option, 2-6
  stored procedures, 1-19
definer rights, 2-20
  versus invoker rights, 2-20

DETERMINISTIC hint, 3-11
dropjava tool, 2-14

E

ease of use, 1-7
encoding
  compiler option, 2-6
entity-relationship (E-R) diagram, drawing an, 5-2
errors
  compilation, 2-5
exceptions, how handled, 4-15
execution rights, 2-16

F

foreign key, 5-5
full name, Java, 2-3
functions, 1-4

G

garbage collection, 1-10, 1-12
get_compiler_option method, 2-6
Graphical User Interface, see GUI
GUI, 1-10, 2-18

I

IDE (integrated development environment), 1-10
interfaces
  user, 2-18
interoperability, 1-7
interpreter, 1-12
invoker rights, 2-20
  advantages, 2-20
  versus definer rights, 2-20

J

Java
  applications, 2-4
    loading, 2-12
  calling from database triggers, 4-6
  calling from PL/SQL, 4-12
  calling from SQL DML, 4-10
  calling from the top level, 4-2
  calling restrictions, 4-11
  compiling, 2-4
  development environment, 2-3
  execution rights, 2-16
  full name, 2-3
  in the RDBMS, 2-2
  loading classes
    checking results, 2-16
  Oracle database execution, 1-2
  resolving classes, 2-9
  schema objects, managing, 2-20
  short name, 2-3
.java files, 2-3, 2-12, 2-13
Java stored procedures
  calling, 4-1
  configuring, 1-16
  defined, 1-17
  developing, 5-1
  introduction to, 1-1
  loading, 2-1
  publishing, 3-1
Java virtual machine. See JVM
JAVA$OPTIONS table, 2-5
JDBC driver. See server-side JDBC driver
JVM, 1-10
  client JVMs versus Oracle JVM, 1-10
  components, 1-9
  garbage collection, 1-10, 1-12
  multithreading, 1-10

K

key
  foreign, 5-5
  primary, 5-5

L

library manager, 1-11
loader, class, 1-13
loading, 2-12 to ??
  checking results, 2-14, 2-16
  class, 2-4
  compilation option, 2-4

granting execution, 2-16
JAR or ZIP files, 2-15
necessary privileges and permissions, 2-15
reloading classes, 2-15
restrictions, 2-14
loadjava tool, 2-13 to 2-15, 2-20
compiling source, 2-4
example, 1-21
execution rights, 2-16
loading class, 2-12
loading ZIP or JAR files, 2-15
restrictions, 2-14
logging, 2-5
longname method, 2-17, 2-19

M

main method, 1-10
maintainability, 1-7
manager
library, 1-11
memory, 1-12
map methods, 3-20
memory manager, 1-12, 2-4
methods, 1-6, 3-18
constructor, 3-20
declaring, 3-19
map and order, 3-20
object-relational, 1-6
modes, parameter, 3-3
multithreading, 1-10

N

NAME clause, 3-11
notational conventions, xi

O

object
full to short name conversion, 2-17
schema, 2-3
short name, 2-17
SQL type, 1-6
table, 3-21
type
call specs, writing, 3-18
object-relational methods, 1-6
online
compiler option, 2-6
Oracle Net Services Connection Manager, 1-2
order methods, 3-20
output, redirecting, 4-2

P

package DBMS_JAVA, 4-2
packaged call specs, writing, 3-15
PARALLEL_ENABLE option, 3-11
parameter modes, 3-3
performance, 1-6
PL/SQL
calling Java from, 4-12
packages, 3-15
primary key, 5-5
procedures, 1-4
advantages of stored, 1-6
productivity, 1-7
.properties files, 2-3, 2-12, 2-13
publications, related, xiv
publishing, 2-4
example, 1-21
purity rules, 4-11

R

redirecting output, 4-2
ref, 3-21
replication, 1-8
reset_compiler_option method, 2-6
resolver, 2-9 to 2-12
default, 2-10
defined, 2-3, 2-4, 2-10
example, 1-20
ignoring non-existent references, 2-10, 2-12
resource schema object, 2-3, 2-12, 2-13
rights, invoker versus definer, 2-20
row trigger, 4-6
rules, purity, 4-11
run-time contexts, stored procedure, 1-3

S

sample database tables, xiii
scalability, 1-7
schema object
  defined, 2-12
  managing Java, 2-20
  name, 2-19
  names, maximum length, 2-3
  using, 2-3
security, 1-8
.ser files, 2-3, 2-12, 2-13
server-side JDBC driver, 1-14
  using, 3-7
server-side SQLJ translator, 1-14
  using, 3-9
set_compiler_option method, 2-6
shared server, 1-7
short name, Java, 2-3
shortname method, 2-17, 2-19
side effects
  controlling, 4-11
source schema object, 2-3, 2-12, 2-13
spec
  package, 3-15
  SQL object type, 3-18
SQL
  DML, calling Java from, 4-10
  object type, 1-6, 3-18
.sqlj files, 2-3, 2-12, 2-13
SQLJ translator. See server-side SQLJ translator
statement trigger, 4-6
stored procedures
  advantages of, 1-6
  calling, 4-1
  developing, 1-17, 5-1
  introduction to, 1-1
  loading, 2-1
  publishing, 3-1

T

tables, sample database, xiii
threading
  model, 1-10
top-level call specs, writing, 3-11
trigger
  database, 1-5, 4-6
  row, 4-6
  statement, 4-6
  using Java stored procedures, 1-17

U

user interface, 2-18
USER_ERRORS view, 2-5
USER_OBJECTS view, 2-14, 2-16
utilities
  loadjava, 2-20

V

verifier, 1-13