Oracle8*i*

Java Stored Procedures Developer's Guide

Release 8.1.5

February 1999

Part No. A64686-01

ORACLE

Java Stored Procedures Developer's Guide, Release 8.1.5

Part No. A64686-01

Author: Tom Portfolio

Graphics Artist: Valarie Moore

Contributors: Dave Alpern, Gray Clossman, Matthieu Devin, Steve Harris, Hal Hildebrand, Thomas Kurian, Dave Rosenberg, Jerry Schwarz

Contents

Send Us Your Comments

Oracle8*i* Java Stored Procedures Developer's Guide, Release 8.1.5

Part No. A64686-01

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

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

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

jpgcomnt@us.oracle.com

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

_____

_____

_____

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

Preface

Welcome to the future of database programming. The rapid rise of Java has forever changed the art of software development. Now, using intranets, the Internet, and Java's cross-platform ability, you can develop applications with global reach. The Oracle8*i* database server and its Java Virtual Machine (JVM) provide an ideal platform on which to deploy such applications.

This guide gets you started building Java applications for Oracle8*i*. Working from simple examples, you quickly learn how to load, publish, and call Java stored procedures.

Major Topics

Who Should Read This Guide?

Anyone developing Java applications for Oracle8*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 Oracle8*i*.

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

How This Guide Is Organized

This guide is divided into the following five chapters:

Chapter 1, "Introduction" After discussing Java's synergy with the Oracle RDBMS, this chapter surveys the main features of stored procedures and points out the advantages they offer. Then, you learn how the Aurora JVM and its main components work with Oracle8*i*. The chapters ends with an overview of the Java stored procedures development process.

Chapter 2, "Loading Stored Procedures" This chapter shows you how to load Java source, class, and resource files into the RDBMS. You learn how to manage Java schema objects using the `loadjava` and `dropjava` utilities. Also, you learn about name resolution and invoker versus definer rights.

Chapter 3, "Publishing Stored Procedures" 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 ties together what you have learned. Step by step, it walks you through the development of a Java stored procedures application.

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 Java, PL/SQL, and SQL code, schema object names, program names, file names, and path names.

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`	Lower case is used for keywords and for one-word names of variables, methods, and packages.
`UPPER CASE`	Upper case is used for names of constants (static final variables) and for names of supplied classes that map to built-in SQL datatypes.
`Mixed Case`	Mixed case is used 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`	Lower case is used for names of constants, variables, cursors, exceptions, subprograms, and packages.
`UPPER CASE`	Upper case is used for keywords, names of predefined exceptions, and names of supplied PL/SQL packages.
`Mixed Case`	Mixed case is used 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:

Oracle8i Application Developer's Guide - Fundamentals
Oracle8i Java Developer's Guide
Oracle8i JDBC Developer's Guide and Reference
Oracle8i SQLJ Developer's Guide and Reference
Oracle8i SQL Reference
PL/SQL User's Guide and Reference
*SQL*Plus User's Guide and Reference*

Suggested Reading

The Java Programming Language by Arnold & Gosling, Addison-Wesley, 1998
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, 1998
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, 1996
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, 1997
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, 1998
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
```

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

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

1

Introduction

Oracle8*i* has the application development features needed to build a new generation of sophisticated applications at 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.

Major Topics

Java in the RDBMS: A Robust Combination

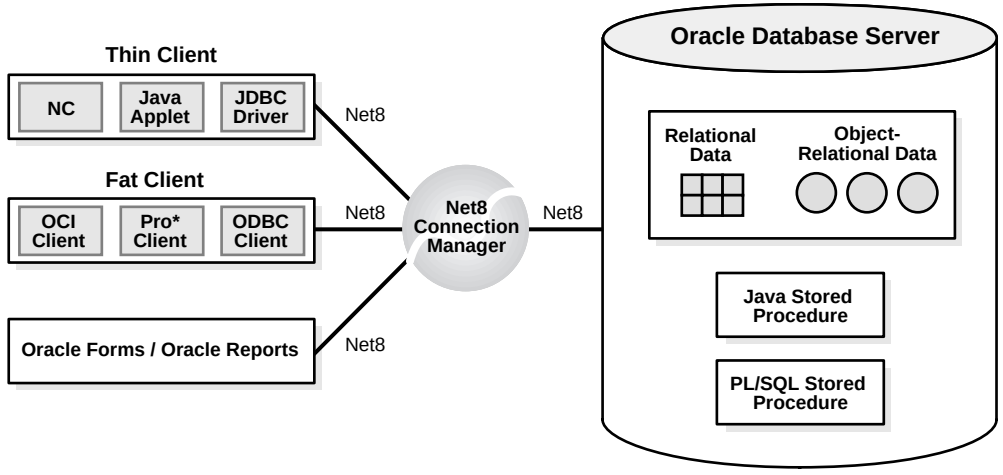
The Oracle RDBMS provides Java applications with a dynamic data-processing engine, which supports complex queries and various views of the same data. All client requests are assembled as data queries for immediate processing, and query results are generated on the fly.

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 Oracle8*i*.) The figure also shows how the Net8 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



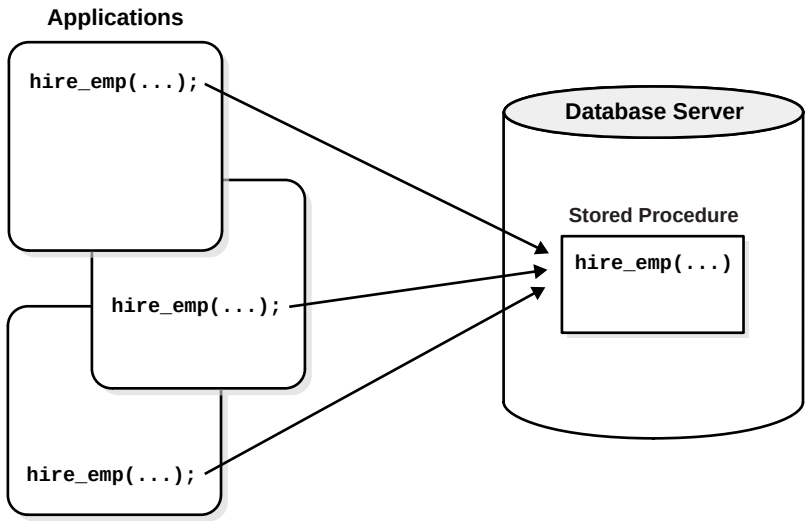
Stored Procedures and Run-Time Contexts

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 procedure stored in the database and called by various applications.

Figure 1–2 Calling a Stored Procedure



Except for graphical-user-interface (GUI) methods, any Java method can run in the RDBMS 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 packaged (not local) PL/SQL functions and procedures can be used as call specs. When you define them using the SQL `CREATE FUNCTION`, `CREATE PROCEDURE`, and/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 `CALL` statements
- PL/SQL blocks, subprograms, and packages

Java methods published as functions are also callable from:

- SQL DML statements (`INSERT`, `UPDATE`, `DELETE`, and `SELECT`).

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 you need.

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

To help you build powerful database applications, stored procedures provide 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, CORBA, and EJB 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 Multi-Threaded Server (MTS) enable Oracle8*i* to support more than 10,000 concurrent users on a single node. For more scalability, you can use the Net8 Connection Manager to multiplex Net8 connections.

Maintainability

Once it is validated, a stored procedure can be used 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 various 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, like 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.

Security

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.

Replication

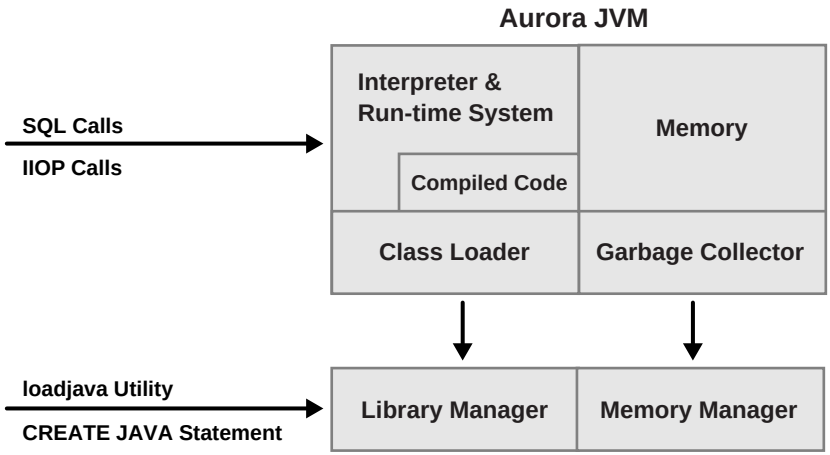
With Oracle Advanced Replication, stored procedures can be replicated (copied) from one Oracle8*i* database to another. This feature makes them ideal for implementing a central set of business rules. Once written, the stored procedures are replicated and distributed to work groups and branch offices throughout the company. In this way, policies can be revised on a central server rather than on individual servers.

The Aurora JVM and Its Components

Oracle's Java Virtual Machine (known as the "Aurora JVM") is a complete, JDK 1.1.6-compliant Java execution environment. The Aurora JVM 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 Aurora 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 Aurora JVM



The Aurora 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 RDBMS and middle-tier Java business objects, even when they have session-long state.

The Aurora JVM versus Client JVMs

This section discusses some important differences between the Aurora 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. Like 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 via top-level entrypoints, 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 provide the logic that drives them. For example, the Aurora 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 Aurora 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. A future release will integrate the Aurora JVM with a variety of client-side IDEs to enable remote debugging, profiling, test coverage analysis, and so on.

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

Multi-threading

Multi-threaded Java programs execute on the Oracle8*i* database server without modification. However, in a server environment, Java threads do *not* increase concurrency (and therefore throughput). Throughput is affected only by MTS mode, the number of OS processes used by the RDBMS, and various tuning methods.

Before porting a multi-threaded application to the server, make sure you understand how threads work with the Aurora JVM. The important differences are that on the server:

- Threads run sequentially, not concurrently.
- Threads within a call die when the call ends.
- Threads are cooperative, not preemptive, so if one thread enters an infinite loop, no other threads can run.

Oracle8*i* multi-threading refers to concurrent user sessions, *not* Java multi-threading. On the server, throughput is increased by supporting many concurrent user sessions. The scheduling of Java execution (of each call within a session for example) to maximize throughput is done by the RDBMS, not by Java.

Main Components of the Aurora JVM

This section briefly describes the main components of the Aurora 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 RDBMS. You never access these Java schema objects directly; only the Aurora JVM uses them.

Memory Manager

Automated storage management is one of Java's key features. In particular, the Java run-time system requires automatic *garbage collection* (deallocation of memory held by unused objects). The memory manager uses memory allocation techniques tuned to object lifetimes. Objects that survive beyond call boundaries are migrated to appropriate memory areas. Also, the memory manager minimizes the footprint per session by sharing immutable object state such as class definitions and final static variables.

Compiler

The Aurora JVM includes a standard JDK 1.1.6-compatible Java compiler. When invoked by the CREATE JAVA SOURCE statement, it translates Java source files into architecture-neutral instructions called *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 Aurora JVM includes a standard JDK1.1.6-compatible bytecode interpreter. The interpreter and associated Java run-time system execute standard Java class files. For high throughput, the interpreter runs on the Multi-Threaded Server, which manages sessions and schedules the execution of Java programs. The run-time system supports native methods and call-in / call-out from the host environment.

Note: Although your own code is interpreted, the Aurora JVM uses natively compiled versions of the core Java class libraries, object request broker (ORB), and JDBC drivers. See "JServer Accelerator" on page 1-13.

Class Loader

In response to requests from the run-time system, the Java class loader locates, loads, and initializes Java classes stored in the RDBMS. 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 per 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 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 JDBC driver complies fully with the Sun JDBC specification. Tightly integrated with the RDBMS, it supports Oracle-specific datatypes, NLS character sets, and stored procedures. Also, 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.

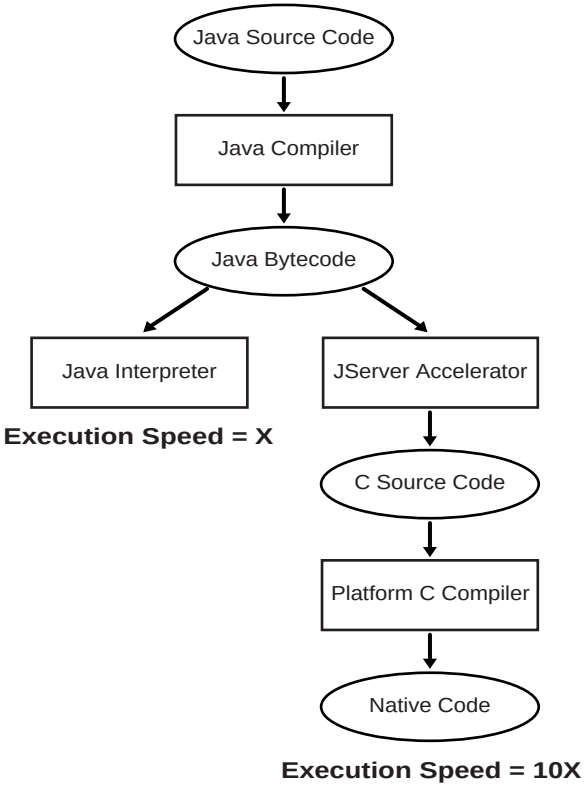
A highly optimized SQLJ Translator runs directly inside the RDBMS, where it provides run-time access to Oracle data via the server-side JDBC driver. SQLJ forms can include queries, DML, DDL, transaction control statements, and calls to stored procedures. The client-side and server-side SQLJ APIs are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

JServer Accelerator

The JServer Accelerator is a native-code compiler that speeds up the execution of Java programs by eliminating interpreter overhead. (See Figure 1–4 on page 1-14.) It translates standard Java class files into specialized C source files that are processed by a platform-dependent C compiler into native libraries, which the Aurora JVM can load dynamically.

Unlike just-in-time (JIT) compilers, the JServer Accelerator is portable to all OS and hardware platforms. To speed up your applications, the Aurora JVM is supplied with natively compiled versions of the core Java class libraries, embedded ORB, and JDBC drivers.

Figure 1–4 Interpreter versus JServer Accelerator



Developing Stored Procedures: An Overview

To develop Java stored procedures, take the four steps listed below. For a detailed example showing the design and implementation of a Java stored procedures application, see Chapter 5.

Write or Reuse the Procedures

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

Load the Procedures into the RDBMS

Load the Java source, class, and resource files into the Oracle RDBMS using the `loadjava` command-line utility, which allows you to specify several options. For more information, see Chapter 2.

Publish the Procedures in the Oracle Data Dictionary

For each Java method that is callable from SQL, write call a spec, which exposes the method's top-level entry point to Oracle. For more information, see Chapter 3.

Call the Procedures from SQL and PL/SQL

Call your Java stored procedures directly from SQL DML statements and from PL/SQL blocks and subprograms. Also, using the SQL `CALL` statement, call the stored procedures from the top level (in SQL*Plus for example) and from database triggers. For more information, see Chapter 4.

2

Loading Stored Procedures

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

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 RDBMS. You can upload Java files from OS 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 RDBMS. For more information, see the *Oracle8i SQL Reference*.

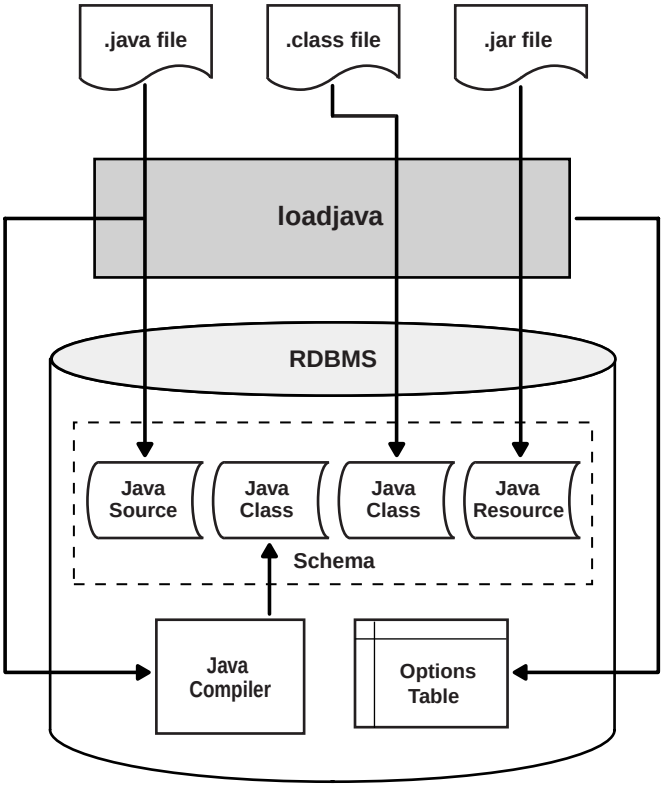
Major Topics

Java in the Database

To make Java files available to the Aurora JVM, you must load them into the RDBMS as schema objects. As Figure 2–1 shows, `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. For example, the option `-encoding` localizes Java source files by specifying their character-encoding scheme. (For more information about the options table, see "Passing Options to the Compiler" on page 2-8.)

Figure 2–1 Loading Java into the RDBMS



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 `OracleSimpleChecker` follows:

```
oracle.sqlj.checker.OracleSimpleChecker
```

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

```
oracle/sqlj/checker/OracleSimpleChecker
```

The 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.

Managing Java Schema Objects

To manage Java schema objects, you use the command-line utilities `loadjava` and `dropjava`. You can write, compile, and partially test and debug Java stored procedures on the client side in popular Java IDEs. Then, you can use `loadjava` to upload the resulting Java source, class, and resource files into the RDBMS as schema objects. In addition, you can use `dropjava` to drop given Java source, class, and resource schema objects from your schema.

What to Load

If you create Java class files on the client side, you can use `loadjava` to upload them into the RDBMS. Alternatively, you can upload Java source files and let the Aurora JVM compile them. In most cases, it is best to compile and debug programs on the client side, then upload the class files for final testing within the RDBMS.

Loading Java archives (JARs) or ZIP archives is the simplest way to use `loadjava`. Archives cannot be schema objects. Therefore, when passed a JAR or ZIP archive, `loadjava` loads the archived files individually. For efficiency, files not modified since the last time they were loaded are not reloaded.

Note: `loadjava` requires *uncompressed* JARs and ZIP archives that are *not nested* inside another JAR or ZIP archive.

Two objects in the same schema cannot define the same class. For example, suppose you define class `X` in the file `A.java`, load the file, then move the definition of `X` to the file `B.java`. An attempt to load `B.java` will fail. Instead, either drop `A.java` or load a new version of it (which no longer defines `X`), then load `B.java`.

How External References Are Resolved

All the classes in a Java program must be loaded before its external references can be resolved. One reason for this requirement is that Java programs have multiple source files because the JDK and other client-side tools require a separate source file for each public class. Another reason is that classes often refer to each other, so name resolution is not possible until all the files have been loaded.

You can use `loadjava` to force early resolution. When you specify the option `-resolve`, `loadjava` uses the SQL `ALTER JAVA CLASS ... RESOLVE` statement to resolve external references in uploaded Java classes. All references must be resolved before you can use those classes. If you do not specify the option, Oracle executes the SQL `ALTER JAVA` statement implicitly at run time.

If all its references to other classes cannot be resolved, a class is marked invalid. At run time, attempts to load an invalid class throw a `ClassNotFound` exception, as required by the *Java Language Specification*.

CLASSPATH versus Resolver Spec

Java classes are loaded dynamically at run time. In Sun's JDK, the class loader locates a class by searching sequentially through the list of directories specified by the environment variable CLASSPATH.

The Aurora JVM uses a similar list called a *resolver spec*, but instead of directories, it specifies SQL schemas. When the JVM searches for a class corresponding to a Java name, it searches the list of schemas sequentially until a Java schema object matching that name is found. If the object is not found, an error is generated.

Each class has its own resolver spec. For example, class A's resolver spec lists the schemas to be searched for the classes referenced by A. Reference lists are maintained by a facility called the *resolver*.

Resolution Modes

You can have `loadjava` resolve classes or defer resolution until run time. (The resolver runs automatically when the JVM tries to load a class that is marked invalid.) However, it is best to resolve classes before run time to learn of missing classes early.

Note: `loadjava` resolves references to classes but not to resources. So, make sure the resource files that your classes need are loaded correctly.

You can run `loadjava` in the following three resolution modes:

- **Load, then resolve:** If you specify the option `-resolve`, `loadjava` uploads all classes listed on the command line, marks them invalid, and then resolves them. Use this mode when loading classes that refer to each other, or when reloading isolated classes. Any class that depends on classes not yet loaded remains marked invalid.
- **Load and resolve:** If you specify the option `-andresolve`, `loadjava` resolves each class as it is loaded. In general, this mode is not recommended because it can leave classes that have unresolved references marked valid, causing an error at run time.

- **Defer resolution:** If you specify neither `-resolve` nor `-andresolve`, `loadjava` loads files but does not compile or resolve them. Instead, files are compiled (if necessary), and resolved at run time or when the JVM needs their definitions to resolve other classes. If you defer resolution until all classes for an application are loaded, the resolver never has to mark a class invalid because it refers to classes not yet loaded. For that reason, this is the preferred mode.

Kinds of Resolver Specs

The option `-resolver` binds a resolver spec to the class schema objects that `loadjava` creates or replaces. Alternatively, a predefined resolver spec can be bound to class schema objects by the option `-oracleresolver` (the default). In most cases, the predefined resolver spec will meet your needs.

A resolver spec lists one or more items, each consisting of a name spec and schema spec in the following form:

```
"((name_spec schema_spec) [(name_spec schema_spec)] ...)"
```

A *name spec* is similar to a name in a Java import statement. It can be a Java class full name, a package name whose final element is the wildcard "`*`", or just that wildcard. However, the elements of a name spec must be separated by slashes, not periods. For example, the name spec `a/b/*` matches all class names that begin with `a.b.` The special name spec, "`*`", matches all class names.

A *schema spec* can be a schema name or the wildcard "`-`". The wildcard does not identify a schema; it tells the resolver not to mark a class invalid even if a reference cannot be resolved. (Without the wildcard "`-`" in a resolver spec, an unresolved reference invalidates the class and generates an error.) Use wildcard "`-`" when you want unresolved references to be detected at run time rather than load time.

When searching for a schema object whose name matches the name spec, the resolver looks in the schema designated by the schema spec. The resolver searches schemas in the order they are listed in the resolver spec. For example,

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

means "Search for a reference in schema SCOTT, then in schema PUBLIC. If a reference is not resolved, mark the referring class *invalid* and display an error message. In other words, call attention to missing classes". For user SCOTT, this resolver spec is equivalent to the `-oracleresolver` spec.

For another example,

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

means "Search for a reference in schema SCOTT, then in schema PUBLIC. If the reference is not resolved, mark the referring class *valid* and do not display an error message. In other words, ignore missing classes".

What the Digest Table Does

`loadjava` uses the hash table JAVA$CLASS$MD5$TABLE (called "digest table" from here on) to track the loading of Java schema objects into a given schema. (MD5 refers to RSA Data Security's MD5 Message-Digest Algorithm, which does the hashing.) If you use `loadjava` to load a Java schema object, you must use `dropjava` to drop the object. Otherwise, the digest table is not updated properly. If that happens, specify the option `-force` to bypass the digest table lookup.

The digest table enables `loadjava` to skip files that have not changed since they were last loaded. This improves the performance of makefiles and scripts that invoke `loadjava` for a whole list of files.

`loadjava` detects unchanged files by maintaining a digest table in every schema. The digest table relates a filename to a *digest*, which is a hash of the file's content. Comparing digests computed for the same file at different times is a fast way to detect a change in the file's content.

For each file it processes, `loadjava` computes a digest of the file's content, then looks up the filename in the digest table. If there is a matching entry, `loadjava` does not load the file (because a corresponding schema object exists and is up to date). If you specify the option `-verbose`, `loadjava` shows you the results of its digest table lookups.

How Compilation Is Done

Loading a source file creates or updates a Java source schema object and invalidates the class schema objects derived previously from that source file (because they were not compiled from the newly loaded source file). If the class schema objects do not exist, they are created.

To force compilation when you upload a source file, specify the `loadjava` option `-resolve`. `loadjava` displays messages inserted by the compiler into the system database table USER_ERRORS. For a description of this table, see the *Oracle8i Reference*.

Passing Options to the Compiler

There are two ways to pass options to the compiler. If you specify the option `-resolve` (which might trigger compilation), then you can specify compiler options on the `loadjava` command line. You can also set compiler option values in the database table JAVA$OPTIONS (called "options table" from here on). Then, you can selectively override those settings using `loadjava` command-line options.

A row in the options table 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.

The compiler looks up options in the options table unless they are specified on the `loadjava` command line. If there is no options-table entry or command-line value for an option, the compiler uses the following default value (for the non-default values, see the *Oracle8i SQLJ Developer's Guide and Reference*):

```
encoding = latin1
online = true      // applies only to SQLJ source files
```

You can get and set options-table entries using the following functions and procedures, which are defined in the supplied 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 parameter name is the name of a Java package, the full name of a class, or the empty string. After searching the options table, the compiler selects the row in which name most closely matches the full name of the schema object. If name is the empty string, it matches the name of any schema object.

Initially, a schema does not have an options table. To create one, use the procedure dbms_java.set_compiler_option to set a value. The procedure creates the table if it does not exist. Enclose parameters in single quotes, as shown in the following example:

```
SQL> dbms_java.set_compiler_option('X.sqlj', 'online', 'false');
```

Using loadjava

Using a built-in package named LOADLOBS, the loadjava utility uploads Java files into a BLOB column in the database table CREATE$JAVA$LOB$TABLE, which the utility creates in the logon schema. Then, it uses the SQL CREATE JAVA statement to load the Java files into the RDBMS as schema objects.

On the command line, you can enter the names of Java source, class, and resource files, SQLJ input files (.sqlj files), and uncompressed JARs and ZIP archives in any order. Here is the syntax:

```
loadjava {-user | -u} username/password[@database]
  [-option_name -option_name ...] filename filename ...
```

where option_name stands for the following syntax:

```
{  {andresolve | a}
 | debug
 | {definer | d}
 | {encoding | e} encoding_scheme_name
 | {force | f}
 | {grant | g} {username | role_name}[,{username | role_name}]...
 | {oci8 | o}
 | oracleresolver
 | {resolve | r}
 | {resolver | R} "resolver_spec"
 | {schema | S} schema_name
 | {synonym | s}
 | {thin | t}
 | {verbose | v} }
```

To display a help screen, use this syntax:

```
loadjava {-help | -h}
```

In a list of options or files, names must be separated only by spaces:

```
-force, -resolve, -thin  // No
-force -resolve -thin    // Yes
```

However, in a list of users and/or roles, names must be separated only by commas:

```
SCOTT, PAYROLL, BLAKE  // No
SCOTT,PAYROLL,BLAKE    // Yes
```

Table 2–1 describes the `loadjava` command-line options.

Table 2–1 loadjava Options

Option	Description
`andresolve`	Compiles source files and resolves each class file as it is loaded. This option and `-resolve` are mutually exclusive. If neither is specified, files are loaded but not compiled or resolved. In general, this mode is not recommended because it can leave classes that have unresolved references marked valid, causing an error at run time. See "Resolution Modes" on page 2-5.
`debug`	Generates debug information. This option is equivalent to `javac -g`.
`definer`	Specifies that the methods of uploaded classes will execute with the privileges of their definer, not their invoker. By default, methods execute with the privileges of their invoker. Different definers can have different privileges, and an application can have many classes, so make sure the methods of a given class execute only with the privileges they need. For more information, see "Invoker Rights versus Definer Rights" on page 2-21.
`encoding`	Sets (or resets) the option `-encoding` in the database table `JAVA$OPTIONS` to the specified value, which must be the name of a standard JDK encoding-scheme (the default is `latin1`). The compiler uses this value, so the encoding of uploaded source files must match the specified encoding.
`force`	Forces the loading of Java class files whether or not they have been loaded before. By default, previously loaded class files are rejected. You cannot force the loading of a class file if you previously loaded the source file. You must drop the source schema object first.
`grant`	Grants the `EXECUTE` privilege on uploaded classes to the listed users and/or roles. (To call the methods of a class directly, users must have the `EXECUTE` privilege.) This option is cumulative. Users and roles are added to the list of those having the `EXECUTE` privilege. To revoke the privilege, either drop and reload the schema object without specifying `-grant`, or use the SQL `REVOKE` statement. To grant the privilege on an object in another user's schema, you must have the `CREATE PROCEDURE WITH GRANT` privilege.
`oci8`	Directs `loadjava` to communicate with the database using the OCI JDBC driver. This option (the default) and `-thin` are mutually exclusive.

Table 2–1 (Cont.) loadjava Options

Option	Description
`oracleresolver`	Binds newly created class schema objects to the following predefined resolver spec: `"((* definer's_schema) (* public))"` This option (the default) detects missing classes immediately. It and `-resolver` are mutually exclusive.
`resolve`	After all class files on the command line are loaded and compiled (if necessary), resolves all external references in those classes. If this option is not specified, files are loaded but not compiled or resolved until run time. Specify this option to compile (if necessary) and resolve a class that was loaded previously. You need not specify the option `-force` because resolution is done independently, after loading.
`resolver`	Binds newly created class schema objects to a user-defined resolver spec. Because it contains spaces, the resolver spec must be enclosed by double quotes. This option and `-oracleresolver` (the default) are mutually exclusive.
`schema`	Assigns newly created Java schema objects to the specified schema. If this option is not specified, then the logon schema is used. You must have the `CREATE ANY PROCEDURE` privilege to load into another user's schema.
`synonym`	Creates a public synonym for uploaded classes, making them accessible outside the schema into which they are loaded. To specify this option, you must have the `CREATE PUBLIC SYNONYM` privilege. If you specify this option for source files, it also applies to classes compiled from those source files.
`thin`	Directs `loadjava` to communicate with the database using the thin JDBC driver. This option and `-oci8` (the default) are mutually exclusive.
`verbose`	Enables verbose mode, in which progress messages are displayed.

Specifying the User

The argument `-user` specifies a username, password, and database connect string in the following format:

```
username/password[@database]
```

Files are loaded into the designated database instance. With option `-oci8` (the default), the connect string `database` is optional. If `database` is specified, it can be a TNS name or a Net8 name-value list. If it is not specified, then the user's default database is used.

With option `-thin`, the connect string `database` must be specified in this format:

```
@host:lport:SID
```

where `host` names the host computer, `lport` is the port configured to listen for Net8 connections (the default is `5521`), and `SID` is the database system identifier (the default is `ORCL`).

Specifying Filenames

On the command line, you can enter as many names of Java source, class, and resource files, SQLJ input files, and uncompressed JARs and ZIP archives as you like, in any order. Archives cannot be schema objects. Therefore, when passed a JAR or ZIP archive, `loadjava` loads the archived files individually.

The best way to upload files is to store them in a JAR or ZIP archive. By loading archives, you avoid schema object naming complications (discussed below). If you have a JAR or ZIP archive that works with the JDK, you can be sure that it will also work with `loadjava`.

The names of schema objects differ slightly from filenames, and different schema objects have different naming conventions. Class files are self-identifying, so `loadjava` can map their filenames to the names of schema objects automatically. `loadjava` can also map source filenames automatically. It simply gives the schema object the full name of the first class defined in the file. Likewise, JARs and ZIP archives include the names of the files they contain.

However, resource files are not self identifying; `loadjava` derives the names of Java resource schema objects from the literal names you enter on the command-line (or the literal names in a JAR or ZIP archive). Resource schema objects are used by running programs, so make sure you enter resource filenames correctly.

The best way to load individual resource files is to run `loadjava` from the top of the package tree, specifying resource filenames relative to that directory. If you decide not to follow that rule, the details of resource file naming follow.

When you load a resource file, `loadjava` derives the name of the resource schema object from the filename that you enter on the command line. Suppose you type the following relative and absolute pathnames on the command line:

```
alpha/beta/x.props
/home/scott/javastuff/alpha/beta/x.props
```

Although you specified the same file, loadjava creates two schema objects:

```
alpha/beta/x.props
ROOT/home/scott/javastuff/alpha/beta/x.props
```

`loadjava` prefixes `ROOT` to the second name because the names of schema objects cannot begin with a slash (`/`).

Classes can refer to resource files relatively (for example, `b.props`) or absolutely (for example, `/a/b.props`). To ensure that `loadjava` and the class loader use the same name for a resource schema object, enter the name that the class passes to method `getResource()` or `getResourceAsString()`.

To make sure the correct names are used, before uploading resource files, store them in a JAR, as shown in the following example:

```
> cd /home/scott/javastuff
> jar -cf alphaResources.jar alpha/*.props
> loadjava ... alphaResources.jar
```

Even better, store both the class and resource files in a JAR. That way, the following invocations are equivalent because you can use any pathname to load the contents of a JAR:

```
> loadjava ... alpha.jar
> loadjava ... /home/scott/javastuff/alpha.jar
```

Examples

In the following example, `loadjava` connects to the default database using the default OCI JDBC driver, loads files from a JAR into schema `BLAKE`, then resolves them:

```
> loadjava -user scott/tiger -resolve -schema BLAKE serverObjs.jar
```

In the next example, `loadjava` connects using the thin JDBC driver, loads a class and a resource file, then resolves them:

```
> loadjava -u scott/tiger@dbhost:5521:orcl -t -r Agent.class \
    images.dat
```

In the final example, `loadjava` adds `NILES` and `FORD` to the list of users who can execute `Manager.class`:

```
> loadjava -thin -user scott/tiger@localhost:5521:orcl \
    -grant NILES,FORD Manager.class
```

Reloading Files

If you upload the same file multiple times but specify different options, the options specified for the last upload prevail. The two exceptions are:

- If `loadjava` does not load a file because it matches a digest table entry, then most options have no effect on the schema object. However, the options `-grant` and `-resolve` are always obeyed. To have `loadjava` skip the digest table lookup, specify the option `-force`.
- The option `-grant` is cumulative. All the users and roles specified in every upload of a given class in a given schema are granted the `EXECUTE` privilege.

Also, if you use the SQL `DROP JAVA` statement to drop a Java class, and then use `loadjava` to reload the same class, you must specify the option `-force`. Otherwise, the upload fails.

Checking Upload Results

To check upload results, you can query the database view USER_OBJECTS, which contains information about schema objects owned by the user (SCOTT in this case). For example, the following SQL*Plus script formats and displays useful information about Java source, class, and resource schema objects:

```
SET SERVEROUTPUT ON
SET VERIFY OFF
PROMPT A)ll or J)ava only?
ACCEPT x CHAR PROMPT 'Choice: '

DECLARE
  choice CHAR(1) := UPPER('&x');
  printable BOOLEAN;
  bad_choice EXCEPTION;
BEGIN
  IF choice NOT IN ('A', 'J') THEN RAISE bad_choice; END IF;
  DBMS_OUTPUT.PUT_LINE(CHR(0));
  DBMS_OUTPUT.PUT_LINE('Object Name                    ' ||
    'Object Type   Status  Timestamp');
  DBMS_OUTPUT.PUT_LINE('------------------------------ ' ||
    '------------ ------- ----------------');
  FOR i IN (SELECT object_name, object_type, status, timestamp
    FROM user_objects ORDER BY object_type, object_name)
  LOOP
    /* Exclude objects generated for loadjava and dropjava. */
    printable := i.object_name NOT LIKE 'SYS_%'
      AND i.object_name NOT LIKE 'CREATE$%'
      AND i.object_name NOT LIKE 'JAVA$%'
      AND i.object_name NOT LIKE 'LOADLOB%';
    IF choice = 'J' THEN
      printable := i.object_type LIKE 'JAVA %';
    END IF;
    IF printable THEN
      DBMS_OUTPUT.PUT_LINE(RPAD(i.object_name,31) ||
        RPAD(i.object_type,14) ||
        RPAD(i.status,8) || SUBSTR(i.timestamp,1,16));
    END IF;
  END LOOP;
EXCEPTION
  WHEN bad_choice THEN
    DBMS_OUTPUT.PUT_LINE('Bad choice');
END;
/
```

You can choose to display all your schema objects or only the Java objects:

```
SQL> @usr_obj
A)ll or J)ava only?
Choice: a

Object Name                    Object Type   Status  Timestamp
------------------------------ ------------- ------- ----------------
Alerter                        JAVA CLASS    VALID   1998-10-08:13:42
POManager                      JAVA CLASS    VALID   1998-10-08:17:14
Alerter                        JAVA SOURCE   VALID   1998-10-08:13:42
POManager                      JAVA SOURCE   VALID   1998-10-08:17:11
BONUS                          TABLE         VALID   1998-10-08:14:02
DEPT                           TABLE         VALID   1998-10-08:14:02
EMP                            TABLE         VALID   1998-10-08:14:02
SALGRADE                       TABLE         VALID   1998-10-08:14:02

SQL> @usr_obj
A)ll or J)ava only?
Choice: j

Object Name                    Object Type   Status  Timestamp
------------------------------ ------------- ------- ----------------
Alerter                        JAVA CLASS    VALID   1998-10-08:13:42
POManager                      JAVA CLASS    VALID   1998-10-08:17:14
Alerter                        JAVA SOURCE   VALID   1998-10-08:13:42
POManager                      JAVA SOURCE   VALID   1998-10-08:17:11
```

The column `object_name` stores the full names of Java schema objects. However, if a name is longer than 30 characters or contains an untranslatable character, then the short name is stored instead. To convert short names to full names, you can use the function `longname` in the utility package `DBMS_JAVA`, as follows:

```
SQL> SELECT dbms_java.longname(object_name), ... FROM user_objects;
```

Table 2–2 describes all the columns in database view `USER_OBJECTS`.

Table 2–2 Columns in USER_OBJECTS

Column Name	Datatype	Description
`OBJECT_NAME`	`VARCHAR2(128)`	name of object
`SUBOBJECT_NAME`	`VARCHAR2(30)`	name of any sub-object (a partition for example)
`OBJECT_ID`	`NUMBER`	object number of object

Table 2–2 (Cont.) Columns in USER_OBJECTS

Column Name	Datatype	Description
DATA_OBJECT_ID	NUMBER	object number of segment that contains the object
OBJECT_TYPE	VARCHAR2(15)	type of object (a table or index for example)
CREATED	DATE	date on which object was created
LAST_DDL_TIME	DATE	date of last DDL operation on the object
TIMESTAMP	VARCHAR2(19)	character string containing date and time the object was created
STATUS	VARCHAR2(7)	status (valid or invalid) of object
TEMPORARY	VARCHAR2(1)	indicator (y/n) of whether current session sees only the data that it stores in the object
GENERATED	VARCHAR2(1)	indicator of whether name of the object was generated by the system
SECONDARY	VARCHAR2(1)	indicator of whether object is a secondary object created for domain indexes

Using dropjava

The `dropjava` utility converts filenames into the names of schema objects, drops the schema objects, then deletes their digest table rows. Dropping a class invalidates classes that depend on it directly or indirectly. Dropping a source also drops classes derived from it.

On the command line, you can enter the names of Java source, class, and resource files, SQLJ input files, and uncompressed JARs and ZIP archives in any order. Here is the syntax:

```
dropjava {-user | -u} username/password[@database]
  [-option_name -option_name ...] filename filename ...
```

where `option_name` stands for the following syntax:

```
{  {oci8 | o}
 | {schema | S} schema_name
 | {thin | t}
 | {verbose | v} }
```

Table 2–3 describes the `dropjava` command-line options.

Table 2–3 dropjava Options

Option	Description
`oci8`	Directs `dropjava` to communicate with the database using the OCI JDBC driver. This option (the default) and `-thin` are mutually exclusive.
`schema`	Drops Java schema objects from the specified schema. If this option is not specified, then the logon schema is used. You must have the `DROP ANY PROCEDURE` privilege to drop objects from another user's schema.
`thin`	Directs `dropjava` to communicate with the database using the thin JDBC driver. This option and `-oci8` (the default) are mutually exclusive.
`verbose`	Enables verbose mode, in which progress messages are displayed.

Specifying Filenames

`dropjava` interprets most filenames the same way `loadjava` does. With class files, `dropjava` finds the name of the class in the file, then drops the corresponding schema object. With source files and SQLJ input files, `dropjava` finds the name of the first class in the file, then drops the corresponding schema object. With uncompressed JARs and ZIP archives, `dropjava` processes the names of archived files as if you had entered them on the command line.

If a filename has an extension other than `.java`, `.class`, `.sqlj`, `.jar`, or `.zip`, or has no extension, `dropjava` assumes the filename is the name of a schema object, then drops all source, class, and resource schema objects with that name. If the filename begins with a slash, then `dropjava` prefixes `ROOT` to the name of the schema object.

If `dropjava` encounters a filename that does not match the name of any schema object, it displays an error message, then processes the remaining filenames.

Examples

In the following example, `dropjava` connects to the default database using the OCI JDBC driver, then drops all objects from schema `BLAKE` that were loaded from `serverObjs.jar`:

```
> loadjava -user scott/tiger -schema BLAKE serverObjs.jar
```

In the next example, `dropjava` connects using the thin JDBC driver, then drops a class and resource from the user's schema:

```
> loadjava -u scott/tiger@dbhost:5521:orcl -t Agent.class images.dat
```

Invoker Rights versus Definer Rights

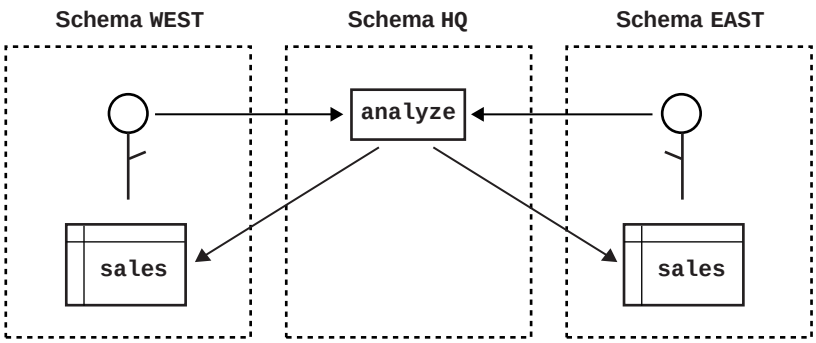
By default, Java stored procedures execute with the privileges of their invoker, not their definer. Invoker-rights procedures are not bound to a particular schema. Their unqualified references to schema objects (such as database tables) are resolved in the schema of the invoker, not the definer.

To override the default behavior, specify the `loadjava` option `definer`. 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 centralize data retrieval. 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.

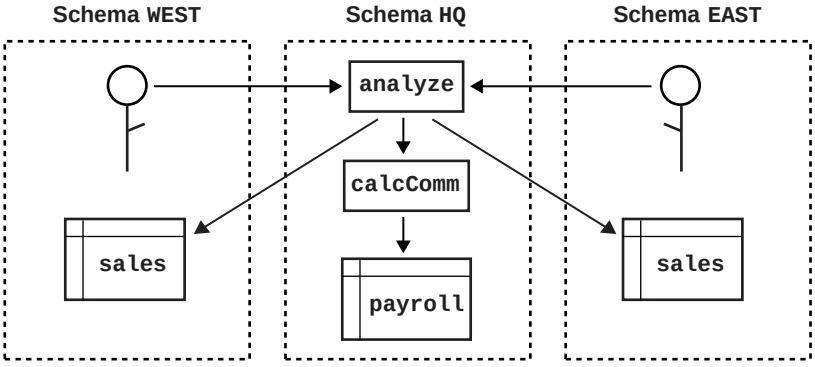
To solve the problem, the company installs an invoker-rights version of the procedure `analyze` at headquarters. Now, as Figure 2–2 shows, all regional sites can use the same procedure to query their own `sales` tables.

Figure 2–2 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 2–3 on page 2-22 shows, the solution is to have procedure `analyze` call the definer-rights procedure `calcComm`, which, in turn, updates the `payroll` table.

Figure 2–3 Indirect Access



3

Publishing Stored Procedures

Before calling Java methods from SQL, you must publish them in the Oracle data dictionary. When you load a Java class into the RDBMS, 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.

Major Topics

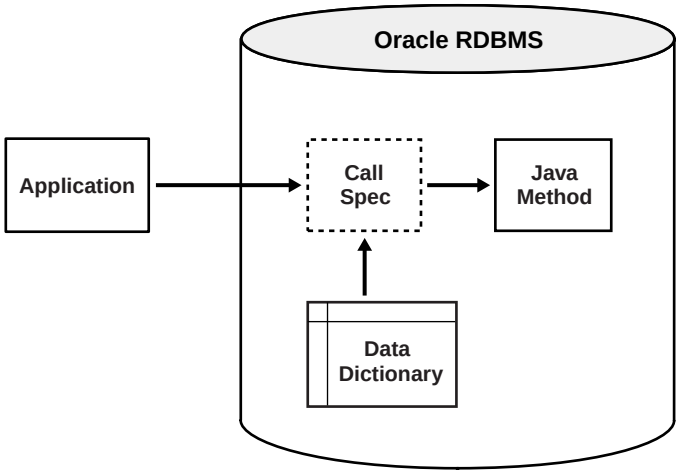
Understanding Call Specs

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 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.

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. You can define the call spec as a:

- stand-alone (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. So, 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 (for `IN OUT` parameters only) 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 gives 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, NCHAR, LONG, VARCHAR2, NVARCHAR2	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
RAW, LONG RAW	oracle.sql.RAW byte[]
ROWID	oracle.sql.CHAR oracle.sql.ROWID java.lang.String
BFILE	oracle.sql.BFILE
BLOB	oracle.sql.BLOB oracle.jdbc2.Blob

Table 3–1 (Cont.) Legal Datatype Mappings

SQL Type	Java Class
CLOB, NCLOB	`oracle.sql.CLOB` `oracle.jdbc2.Clob`

OBJECT	`oracle.sql.STRUCT`
	`oracle.SqljData`
	`oracle.jdbc2.Struct`

REF	`oracle.sql.REF` `oracle.jdbc2.Ref`
TABLE, VARRAY	`oracle.sql.ARRAY` `oracle.jdbc2.Array`
any of the above SQL types	`oracle.sql.CustomDatum` `oracle.sql.Datum`

Notes:

1. The type `UROWID` and the `NUMBER` subtypes (`INTEGER`, `REAL`, and so on) are not supported.
2. The Java wrapper classes (`java.lang.Byte`, `java.lang.Short`, and so on) are useful for returning nulls from SQL.
3. When the class `oracle.sql.CustomDatum` is used to declare parameters, it must define the following member:

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

4. `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.
5. 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.
6. For information about supplied packages `oracle.jdbc2` and `oracle.sql`, see the *Oracle8i JDBC Developer's Guide and Reference*.

Using the Server-Side 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 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. To get a `Connection` object, simply execute the following statement:

```
Connection conn = new OracleDriver().defaultConnection();
```

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. (A question mark serves as a placeholder.) 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 a variable 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 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.
- The method `defaultConnection()` always returns the same connection object. So, you need not supply a connect string.
- 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 JDBC driver does not support auto-commits. So, your application must explicitly commit or rollback database changes.
- You cannot close a default connect established by the server-side JDBC driver. Calling method `close()` on the connection has no effect.

For more information, see the *Oracle8i 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. For example, the SQLJ input file (`.sqlj` file) below embeds `SELECT` and `CALL` statements in the definition of the 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}
```

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 RDBMS. 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 some 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 register a driver, specify a default connection, or create a connection object for your `#sql` statements.
- You cannot connect to a remote database. You can only "connect" to the server that is running your Java program.
- A SQLJ connection-context instance communicates with the database through the session that runs your Java program, not through a true connection. So, closing the connection-context instance has no effect.
- 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 package `DBMS_JAVA`.
- The server-side SQLJ Translator does not support the option `-ser2class`. So, it always generates profiles as serialized resource files (`.ser` files), never as class files.
- 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 *Oracle8i 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 *Oracle8i SQL Reference*.

The `NAME`-clause string uniquely identifies the Java method. The fully qualified Java names and the call spec parameters, which are mapped by position, must correspond one to one. (That rule does not apply to method `main`. See Example 2 on page 3-12.) If the Java method takes no arguments, code an empty parameter list for it but *not* for the function or procedure.

As usual, fully qualified Java names are written 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 following Java class has been loaded into the RDBMS:

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

public class GenericDrop {
  public static void dropIt (String object_type, String object_name)
  throws SQLException {
    // Connect to Oracle using JDBC driver
    Connection conn = new OracleDriver().defaultConnection();
    // 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 the method `main`. Its `String[]` parameter can be mapped to multiple `CHAR` and / 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.driver.*;

public class RowCounter {
  public static int rowCount (String tabName) throws SQLException {
    Connection conn = new OracleDriver().defaultConnection();
    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 below 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.driver.*;

public class DeptManager {
  public static void addDept (String deptName, String deptLoc)
  throws SQLException {
    Connection conn = new OracleDriver().defaultConnection();
    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 = new OracleDriver().defaultConnection();
    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 = new OracleDriver().defaultConnection();
    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 datatypes that encapsulate a data structure along with the functions and procedures needed to manipulate the data. The variables that form the data structure are called *attributes*. The functions and procedures that characterize the behavior of the object type are called *methods*, which can be written in Java.

Like 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 and/or call specs. (For details, see the *PL/SQL User's Guide and Reference*.)

If an object type spec declares only attributes and/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 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.

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.

Like a Java variable, an attribute is declared 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 named 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` as shown below. 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` as shown below. 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, as shown below, 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;
```

Then, as shown below, 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 *Oracle8i JDBC Developer's Guide and Reference*.

Using Class oracle.sql.STRUCT

To continue, you write a Java stored procedure, as shown below. 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.driver.*;
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 attribute values more natural, you can create a Java class for the object that implements the SQLData interface. For details, see the *Oracle8i JDBC Developer's Guide and Reference*. If you choose to create a Java class that implements SQLData, 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. 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.driver.*;
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 `oralce.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`, as shown below. 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, as shown below. 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.driver.*;
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.

Major Topics

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

System.out and System.err print to the current trace files. To redirect output to the SQL*Plus text buffer, use this simple workaround:

```
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 example below, 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.driver.*;

public class Deleter {
  public static void main (String[] args) throws SQLException {
    Connection conn = new OracleDriver().defaultConnection();
    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 RDBMS:

```
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 *Oracle8i SQL Reference*. For a full discussion of database triggers, see the *Oracle8i 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.driver.*;

public class DBTrigger {
  public static void logSal (int empID, float oldSal, float newSal)
  throws SQLException {
    Connection conn = new OracleDriver().defaultConnection();
    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 UPDATE statement below, 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 = new OracleDriver().defaultConnection();
  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, and DELETE statements. For example, assume that the executable for the following Java class is stored in the RDBMS:

```
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 called from a `SELECT` statement or a parallelized `INSERT`, `UPDATE`, or `DELETE` statement, the method cannot modify any database tables.
- When called from an `INSERT`, `UPDATE`, or `DELETE` statement, the method cannot query or modify any database tables modified by that statement.
- When called 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`). Also, 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 RDBMS:

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

public class Adjuster {
  public static void raiseSalary (int empNo, float percent)
  throws SQLException {
    Connection conn = new OracleDriver().defaultConnection();
    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 3-12) from a stand-alone 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 "Examples" on page 3-20) from an anonymous PL/SQL block:

```
DECLARE
  emp_id NUMBER(4);
  v emp_type;
BEGIN
  -- assign 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 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 *Oracle8i JDBC Developer's Guide and Reference*. To learn more about SQLJ, see the *Oracle8i SQLJ Developer's Guide and Reference*.

How Exceptions Are Handled

Java exceptions are objects, so they have classes as their types. Like 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 that is not caught by Java, the caller gets an *exception thrown from Java* error message. 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.

Major Topics

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

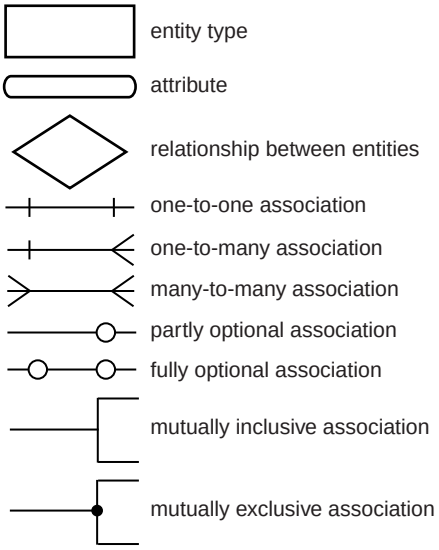
attribute a characteristic of an entity

relationship an association between entities

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

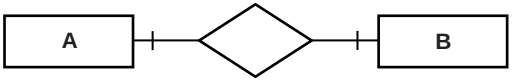
record an ordered set of attribute values that describe an instance of an entity type

Symbols:

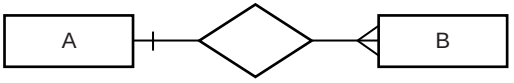


Examples:

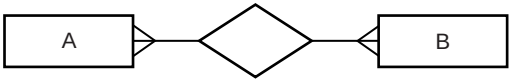
One A is associated with one B:



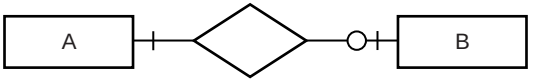
One A is associated with one or more B's:



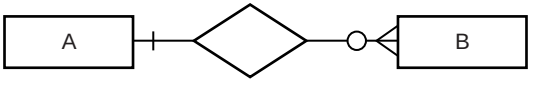
One or more A's are associated with one or more B's:



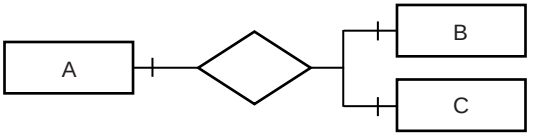
One A is associated with zero or one B:



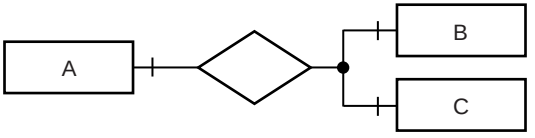
One A is associated with zero or more B's:



One A is associated with one B and one C:

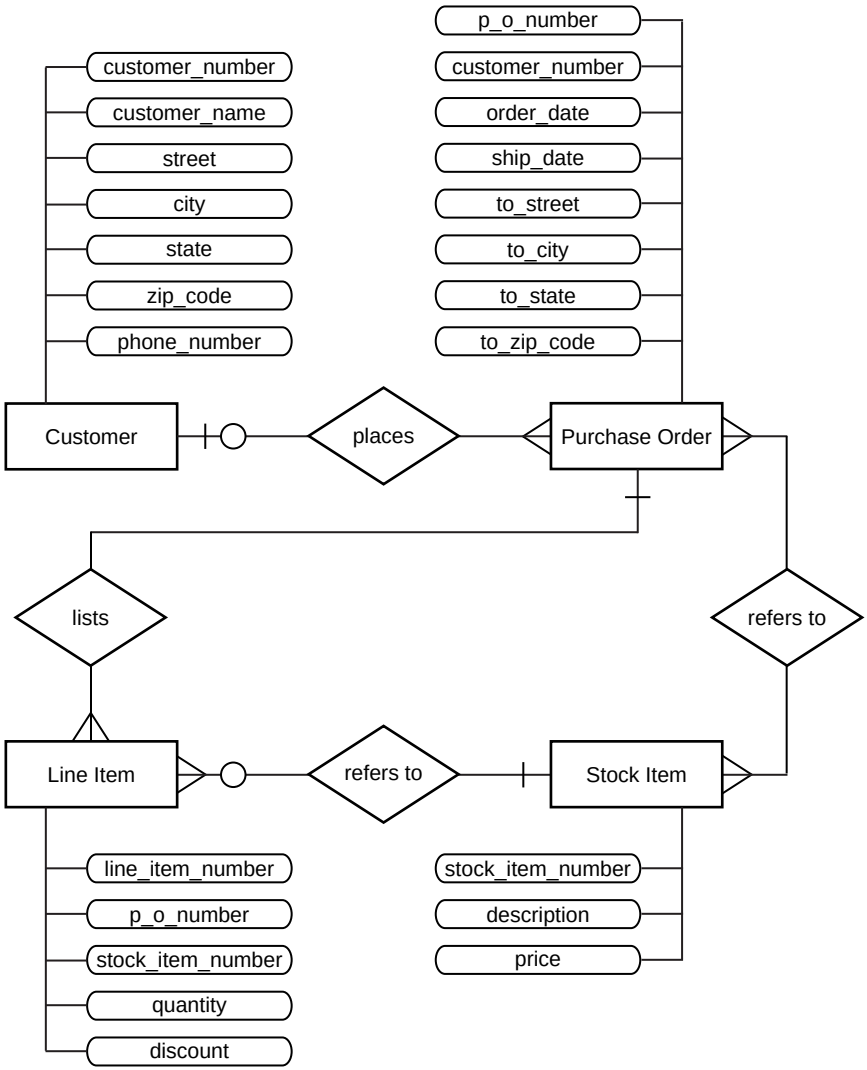


One A is associated with one B or one C (but not both):



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:

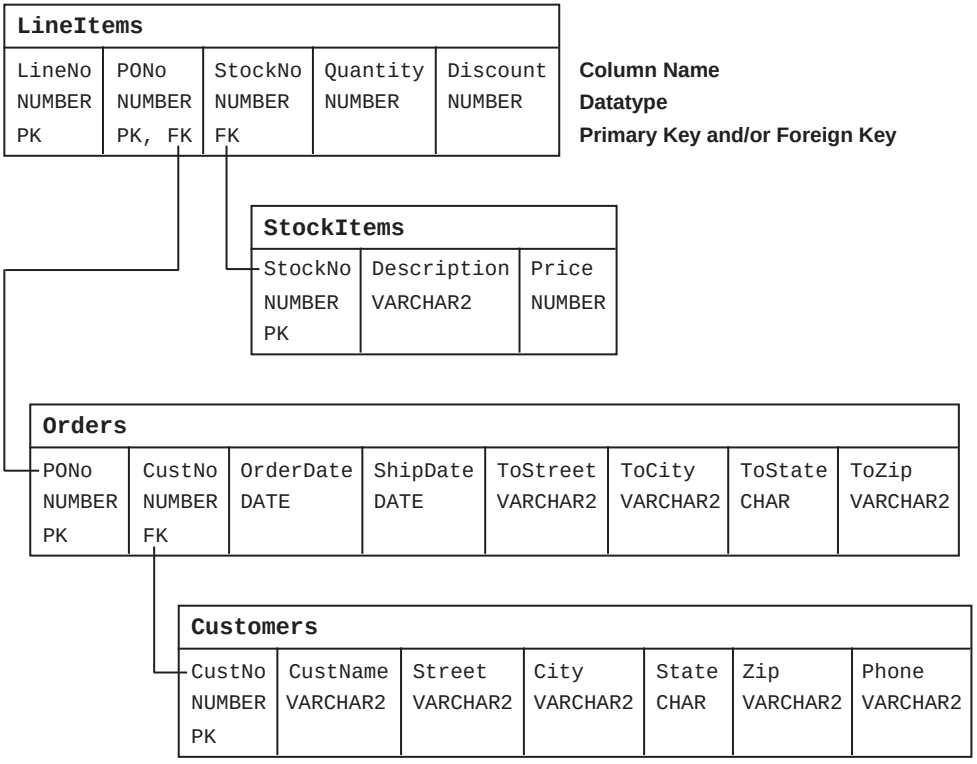
- `Customers`
- `Orders`
- `LineItems`
- `StockItems`

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 Stored Procedures

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.driver.*;

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 = new OracleDriver().defaultConnection();
        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 = new OracleDriver().defaultConnection();
        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 = new OracleDriver().defaultConnection();
      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 = new OracleDriver().defaultConnection();
      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 = new OracleDriver().defaultConnection();
    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 = new OracleDriver().defaultConnection();
      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 = new OracleDriver().defaultConnection();
        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 = new OracleDriver().defaultConnection();
      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 Stored Procedures

Next, you use the command-line utility `loadjava` to upload your Java stored procedures into the Oracle RDBMS, 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 thin JDBC driver.

Publishing the Java Stored Procedures

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);
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');
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);
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

E

F

G

I

J

K

L

M

N

O

P

R

S

T

U

V