Oracle® Database

Java Developer's Guide

10*g* Release 1 (10.1)

Part No. B12021-01

December 2003

ORACLE®

Oracle Database Java Developer's Guide, 10*g* Release 1 (10.1)

Part No. B12021-01

Primary Authors: Sheryl Maring, Rick Sapir, Michael Wiesenberg

Contributing Authors: Brian Wright, Timothy Smith

Contributors: Malik Kalfane, Steve Harris, Ellen Barnes, Peter Benson, Greg Colvin, Bill Courington, Matthieu Devin, Jim Haungs, Hal Hildebrand, Mark Jungerman, Susan Kraft, Thomas Kurian, Scott Meyer, Tom Portfolio, Dave Rosenberg, Jerry Schwarz, Harlan Sexton, Tim Smith, David Unietis, Brian Wright.

Contents

2 Java Applications on Oracle Database

7 Calling Stored Procedures

8 Java Stored Procedures Application Example

9 Security For Oracle Database Java Applications

10 Oracle Database Java Application Performance

11 Schema Object Tools

12 Database Web Services

Send Us Your Comments

Oracle Database Java Developer's Guide, 10*g* Release 1 (10.1)

Part No. B12021-01

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

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

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

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

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

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

Preface

Who Should Read This Book?

This book has been written for the following audiences:

- Management—You may have purchased Oracle Database for reasons other than Java development within the database. However, if you want to know more about Oracle Database Java features, see "Oracle's Java Application Strategy" on page 1-24 for a management perspective.
- Non-Java Developers—Oracle database programming consists of PL/SQL and other non-Java programming. For experienced PL/SQL developers who are not familiar with Java, a brief overview of Java and object-oriented concepts is discussed in the first part of Chapter 1, "Introduction to Java in Oracle Database". For more detailed information on Java, see "Suggested Reading" at the end of this Preface.
- Java Developers—Pure Java developers are used to a Java environment that follows the Sun Microsystems specification. However, when Java is combined in the database, both Java and database concepts merge. Thus, the Java environment within Oracle Database is expanded to include database concerns. The bulk of this book discusses how to execute Java in the database. The following outlines the two viewpoints that arise from this merge:
    * Java environment—Note that Oracle Database delivers a compliant Java implementation—any 100% pure Java code will work. OracleJVM affects your Java development in the way you manage your classes, and the environment in which your classes exist. For example, the classes must be loaded into the database. In addition, there is a clearer separation of client and server in the Oracle Database model.

* Database environment—You need to be aware of database concepts for managing your Java objects. This book gives you a comprehensive view of how the two well-defined realms—the Oracle Database database and the Java environment—fit together. For example, when deciding on your security policies, you must consider both database security and Java security for a comprehensive security policy.

Organization

This document contains the following chapters:

Chapter 1, "Introduction to Java in Oracle Database"

Gives an overview of how to develop, load, and execute Java applications in the database.

Chapter 2, "Java Applications on Oracle Database"

Describes the basic differences for writing, installing, and deploying Java applications within Oracle Database.

Chapter 3, "Invoking Java in the Database"

Gives an overview and examples of how to invoke Java within the database.

Chapter 4, "Java Installation and Configuration"

Describes what you need to know to install and configure OracleJVM within your database.

Chapter 5, "Developing Java Stored Procedures"

Describes stored procedures, which open the Oracle RDBMS to all Java programmers.

Chapter 6, "Publishing Java Classes With Call Specs"

Describes how to publish the methods with call specifications (call specs), which map Java method names, parameter types, and return types to their SQL counterparts.

Chapter 7, "Calling Stored Procedures"

Demonstrates how to call Java stored procedures in various contexts.

Chapter 8, "Java Stored Procedures Application Example"

Demonstrates the building of a Java stored procedures application.

Chapter 9, "Security For Oracle Database Java Applications"

Details the security support available for Java applications within Oracle Database.

Chapter 10, "Oracle Database Java Application Performance"

Describes how to increase Java application performance with natively compiled code Java memory usage.

Chapter 11, "Schema Object Tools"

Describes the schema object tools to use in the Oracle Database Java environment.

Chapter 12, "Database Web Services"

Describes Database Web Services and Web Services callouts.

Appendix A, "DBMS_JAVA Package"

Describes the DBMS_JAVA package.

Glossary

Defines specialized terms.

Java API Programming Models

The building blocks that Java developers use in Oracle Database are as follows:

- Java stored procedures—You can develop Java applications that are stored in the database. Once loaded, these procedures can be invoked from SQL, PL/SQL, or as triggers. See Chapter 5, "Developing Java Stored Procedures" for more information.
- JDBC—You can write a Java application that accesses SQL data from the client, or directly on the server.

Each of these models is briefly discussed in Chapter 1, "Introduction to Java in Oracle Database" and examples are given in Chapter 3, "Invoking Java in the Database". Both of these chapters should help you decide which model to use for your particular application. Once you decide on the appropriate model, examine the appropriate developer's guide for in-depth information on each model.

Suggested Reading

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

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

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

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

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

Online Sources

There are many useful online sources of information about Java. For example, you can view or download documentation, guides, and tutorials from the JavaSoft Web site:

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

Another popular Java Web site is:

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

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

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

You can get the latest OracleJVM news, updates, and offerings from the Oracle Technology Network (OTN) at the following site:

```
http://otn.oracle.com/tech/java/java_db/content.html
```

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

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

```
http://otn.oracle.com/membership/
```

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

```
http://otn.oracle.com/documentation/content.html
```

Documentation Accessibility

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

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

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

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

1

Introduction to Java in Oracle Database

This book provides an overview on how to develop, load, and execute your Java applications in the Oracle Database.

This chapter contains the following information:

Chapter Contents

This chapter:

- Introduces the Java language for Oracle Database programmers. Oracle PL/SQL developers are accustomed to developing server-side applications that have tight integration with SQL data. You can develop Java server-side applications that take advantage of the scalability and performance of the Oracle Database. If you are not familiar with Java, see "Overview of Java" on page 1-5.
- Examines why may consider using Java within Oracle Database. See "Why Use Java in Oracle Database?" on page 1-14. In addition, a brief description is given for each of the Java APIs supported within Oracle Database. The list of APIs include JDBC and Java stored procedures. See "Oracle's Java Application Strategy" on page 1-24.

What's New in this Release?

The following sections describe the additions to this release:

- Upgrading to J2SE 1.4.1
- New Memory Model for Dedicated Mode Sessions
- Database Web Services Callouts
- Native Java Interface
- EJB Call-out

Upgrading to J2SE 1.4.1

In this release, the system classes are upgraded from J2SE 1.3 to J2SE 1.4.1. J2SE 1.4.1 is compatible with J2SE 1.3. Sun Microsystems publishes the list of incompatibilities between J2SE 1.4.1 and previous versions at the following Web site:

```
http://java.sun.com/products/j2se/1.4.1/compatibility.html
```

As part of the upgrade of the system classes to J2SE 1.4.1, the OracleJVM supports Headless AWT. Headless AWT allows AWT computation, which does not rely on the native display and input devices of the platform to occur, but, instead, disallows attempts to access those native resources. Methods that attempt to display a graphical user interface or to read from keyboard or mouse input instead throw the new runtime exception `java.awt.HeadlessException`. Similarly, the

OracleJVM disallows attempts to play or record sound using the server's native sound devices, but still allows applications to read, write and manipulate supported sound files. For more information, see "User Interfaces on the Server" on page 2-25.

New Memory Model for Dedicated Mode Sessions

In Oracle Database, the OracleJVM has a new memory model for sessions that connect to the database through a dedicated server. Since a session using a dedicated server is guaranteed to use the same process for every database call, the Process Global Area is used for session specific memory and object allocations. This means that some of the objects and resources that used to be reclaimed at the end of each call can now live across calls. In particular, resources specific to a particular operating system, such as threads and open files, now are no longer cleaned up at the end of each database call.

For sessions that use shared servers, the restrictions across calls that applied in previous releases are still present. The reason is that a session that uses a shared server is not guaranteed to connect to the same process on a subsequent database call, and hence the session-specific memory and objects that need to live across calls are saved in the System Global Area. This means that process-specific resources, such as threads, open files and sockets must be cleaned up at the end of each call, and hence will not be available for the next call. For more details on OracleJVM behavior when using shared servers, see "Special Considerations for Shared Servers" on page 2-36.

Database Web Services Callouts

In Oracle Database, you can load a Web Services client stack into the OracleJVM to support callouts to external Web Services from Java as well as from PL/SQL. You can use the JPublisher tool to generate static Java client-proxies as well as PL/SQL call specifications on these proxies that are loaded into the OracleJVM to enable access to Web Services from Java, PL/SQL, and SQL code.

See Chapter 12, "Database Web Services" for more information and the *Oracle Database JPublisher User's Guide* for more information. For more details, see "Support for Calling Java Stored Procedures Directly" on page 3-8.

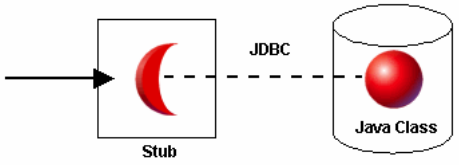
Native Java Interface

In Oracle Database, you can now invoke public static methods of Java classes in the OracleJVM directly from Java clients without defining PL/SQL call specifications and calling these through JDBC. Instead, you can use the JPublisher utility to

generate a client-proxy class with the same signature as the server-side Java class. Once you have instantiated a client-proxy instance with a JDBC connection, you can call the proxy methods directly.

Figure 1–1 demonstrates a client-side stub API for direct invocation of static server-side Java methods. JPublisher transparently takes care of stub generation.

Figure 1–1 Native Java Interface



For example, to call the following method in the server

```
public String oracle.sqlj.checker.JdbcVersion.to_string();
```

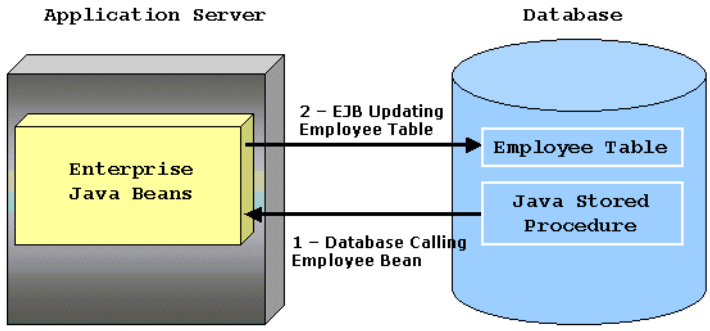
use

```
jpub -java=oracle.sqlj.checker.JdbcVersion
```

EJB Call-out

In certain enterprise applications it becomes essential to access Enterprise Java Beans (EJB) which are deployed on a remote server, from within the database. Oracle Database provides a means to access the remotely deployed EJBs over RMI.

Figure 1–2 EJB Call-out



Overview of Java

Java has emerged as the object-oriented programming language of choice. It includes the following concepts:

- a Java virtual machine (JVM), which provides the fundamental basis for platform independence
- automated storage management techniques, the most visible of which is garbage collection
- language syntax that borrows from C and enforces strong typing

The result is a language that is object-oriented and efficient for application-level programs.

Java and Object-Oriented Programming Terminology

This section covers some basic terminology of Java application development in the Oracle Database environment. The terms should be familiar to experienced Java programmers. A detailed discussion of object-oriented programming or of the Java language is beyond the scope of this book. Many texts, in addition to the complete language specification, are available at your bookstore and on the Internet. See "Suggested Reading" in the Preface for pointers to reference materials and for places to find Java-related information on the Internet.

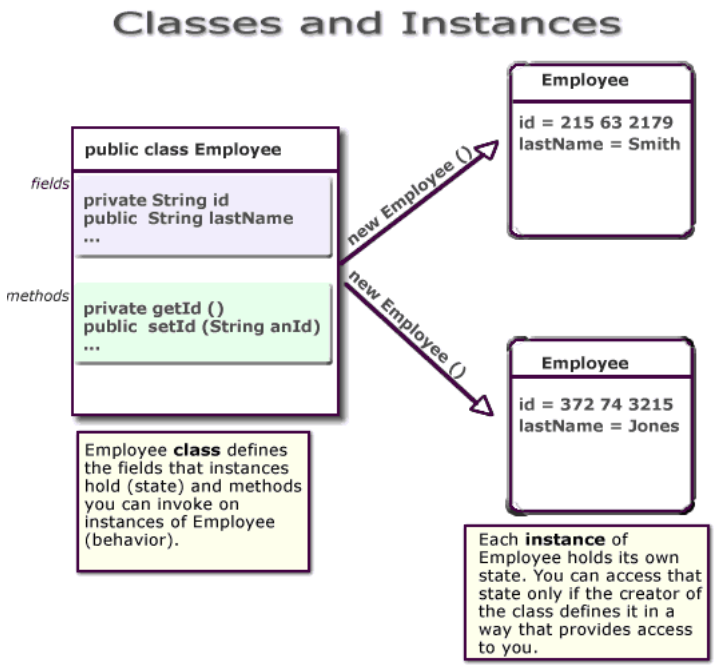
Classes

All object-oriented programming languages support the concept of a class. As with a table definition, a class provides a template for objects that share common characteristics. Each class can contain the following:

- Attributes—static or instance variables that each object of a particular class possesses.
- Methods—you can invoke methods defined by the class or inherited by any classes extended from the class.

When you create an object from a class, you are creating an instance of that class. The instance contains the fields of an object, which are known as its data, or state. Figure 1–3 shows an example of an `Employee` class defined with two attributes: last name (`lastName`) and employee identifier (`ID`).

Figure 1–3 Classes and Instances



When you create an instance, the attributes store individual and private information relevant only to the employee. That is, the information contained

within an employee instance is known only for that single employee. The example in Figure 1–3 shows two instances of employee—Smith and Jones. Each instance contains information relevant to the individual employee.

Attributes

Attributes within an instance are known as fields. Instance fields are analogous to the fields of a relational table row. The class defines the fields, as well as the type of each field. You can declare fields in Java to be static, public, private, protected, or default access.

- Public, private, protected, or default access fields are created within each instance.
- Static fields are like global variables in that the information is available to all instances of the employee class.

The language specification defines the rules of visibility of data for all fields. Rules of visibility define under what circumstances you can access the data in these fields.

Methods

The class also defines the methods you can invoke on an instance of that class. Methods are written in Java and define the behavior of an object. This bundling of state and behavior is the essence of encapsulation, which is a feature of all object-oriented programming languages. If you define an `Employee` class, declaring that each employee's `id` is a private field, other objects can access that private field only if a method returns the field. In this example, an object could retrieve the employee's identifier by invoking the `Employee.getId()` method.

In addition, with encapsulation, you can declare that the `Employee.getId()` method is private, or you can decide not to write an `Employee.getId()` method. Encapsulation helps you write programs that are reusable and not misused. Encapsulation makes public only those features of an object that are declared public; all other fields and methods are private. Private fields and methods can be used for internal object processing.

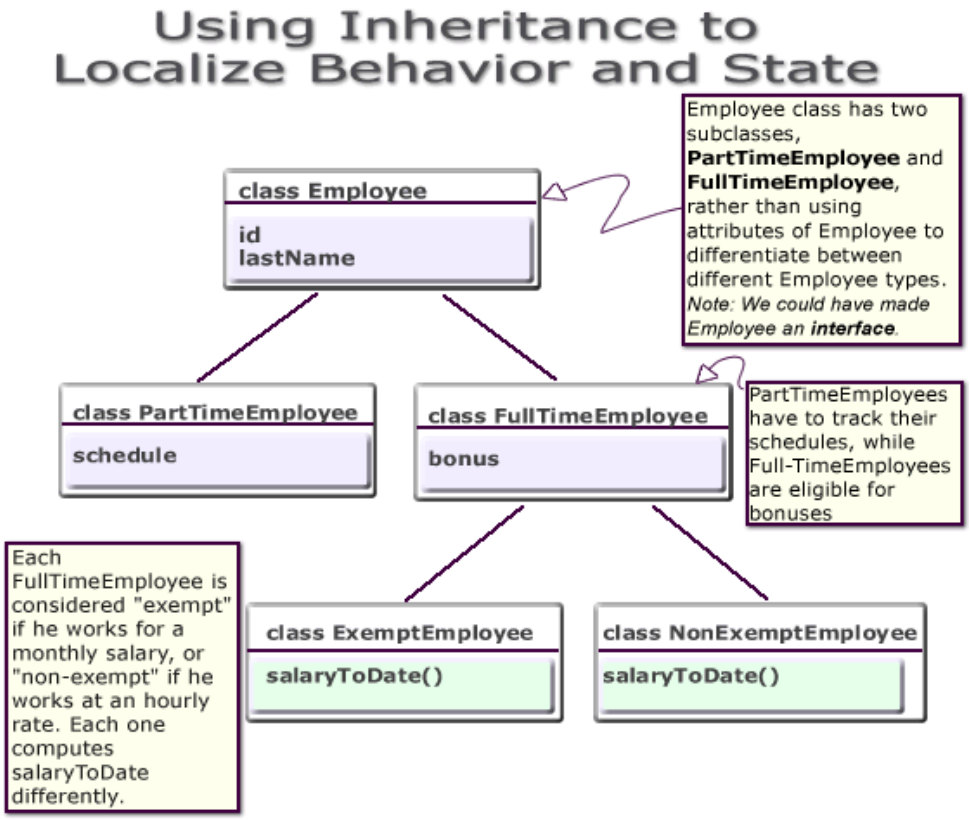
Class Hierarchy

Java defines classes within a large hierarchy of classes. At the top of the hierarchy is the `Object` class. All classes in Java inherit from the `Object` class at some level, as you walk up through the inheritance chain of superclasses. When we say Class B inherits from Class A, each instance of Class B contains all the fields defined in class B, as well as all the fields defined in Class A. For example, in Figure 1–4, the

`FullTimeEmployee` class contains the `id` and `lastName` fields defined in the `Employee` class, because it inherits from the `Employee` class. In addition, the `FullTimeEmployee` class adds another field, `bonus`, which is contained only within `FullTimeEmployee`.

You can invoke any method on an instance of Class B that was defined in either Class A or B. In our employee example, the `FullTimeEmployee` instance can invoke methods defined only within its own class, or methods defined within the `Employee` class.

Figure 1–4 Class Hierarchy



Instances of Class B are substitutable for instances of Class A, which makes inheritance another powerful construct of object-oriented languages for improving code reuse. You can create new classes that define behavior and state where it

makes sense in the hierarchy, yet make use of pre-existing functionality in class libraries.

Interfaces

Java supports only single inheritance; that is, each class has one and only one class from which it inherits. If you must inherit from more than one source, Java provides the equivalent of multiple inheritance, without the complications and confusion that usually accompany it, through interfaces. Interfaces are similar to classes; however, interfaces define method signatures, not implementations. The methods are implemented in classes declared to implement an interface. Multiple inheritance occurs when a single class simultaneously supports many interfaces.

Polymorphism

Assume in our `Employee` example that the different types of employees must be able to respond with their compensation to date. Compensation is computed differently for different kinds of employees.

- `FullTimeEmployees` are eligible for a bonus
- `NonExemptEmployees` get overtime pay

In traditional procedural languages, you would write a long switch statement, with the different possible cases defined.

```
switch: (employee.type) {
          case: Employee
            return employee.salaryToDate;
          case: FullTimeEmployee
            return employee.salaryToDate + employee.bonusToDate
         ...
```

If you add a new kind of `Employee`, you must update your switch statement. If you modify your data structure, you must modify all switch statements that use it. In an object-oriented language such as Java, you implement a method, `compensationToDate()`, for each subclass of `Employee` class that requires any special treatment beyond what is already defined in `Employee` class. For example, you could implement the `compensationToDate()` method of `NonExemptEmployee`, as follows:

```
private float compensationToDate() {
         return super.compensationToDate() + this.overtimeToDate();
}
```

You implement `FullTimeEmployee`'s method, as follows:

```
private float compensationToDate() {
        return super.compensationToDate() + this.bonusToDate();
}
```

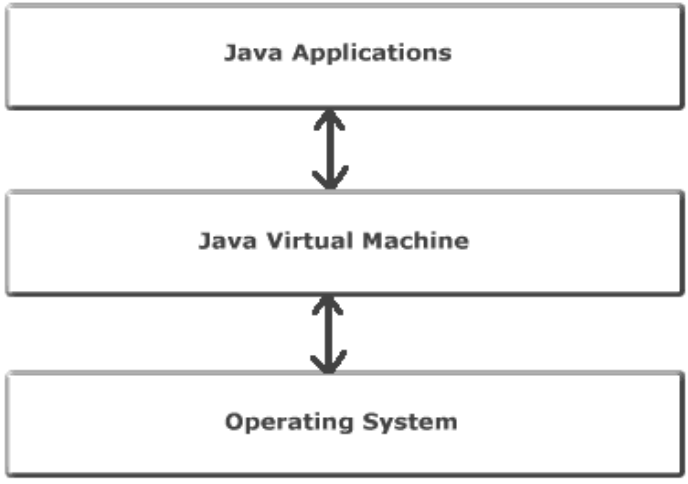
The common usage of the method name `compensationToDate()` allows you to invoke the identical method on different classes and receive different results, without knowing the type of employee you are using. You do not have to write a special method to handle `FullTimeEmployees` and `PartTimeEmployees`. This ability for the different objects to respond to the identical message in different ways is known as polymorphism.

In addition, you could create an entirely new class that does not inherit from `Employee` at all—`Contractor`—and implement a `compensationToDate()` method in it. A program that calculates total payroll to date would iterate over all people on payroll, regardless of whether they were full-time, part-time, or contractors, and add up the values returned from invoking the `compensationToDate()` method on each. You can safely make changes to the individual `compensationToDate()` methods with the knowledge that callers of the methods will work correctly. For example, you can safely add new fields to existing classes.

The Java Virtual Machine (JVM)

As with other high-level computer languages, your Java source compiles to low-level machine instructions. In Java, these instructions are known as bytecodes (because their size is uniformly one byte of storage). Most other languages—such as C—compile to machine-specific instructions—such as instructions specific to an Intel or HP processor. Your Java source compiles to a standard, platform-independent set of bytecodes, which interacts with a Java virtual machine (JVM). The JVM is a separate program that is optimized for the specific platform on which you execute your Java code. Figure 1–5 illustrates how Java can maintain platform independence. Your Java source is compiled into bytecodes, which are platform independent. Each platform has installed a JVM that is specific to its operating system. The Java bytecodes from your source get interpreted through the JVM into appropriate platform dependent actions.

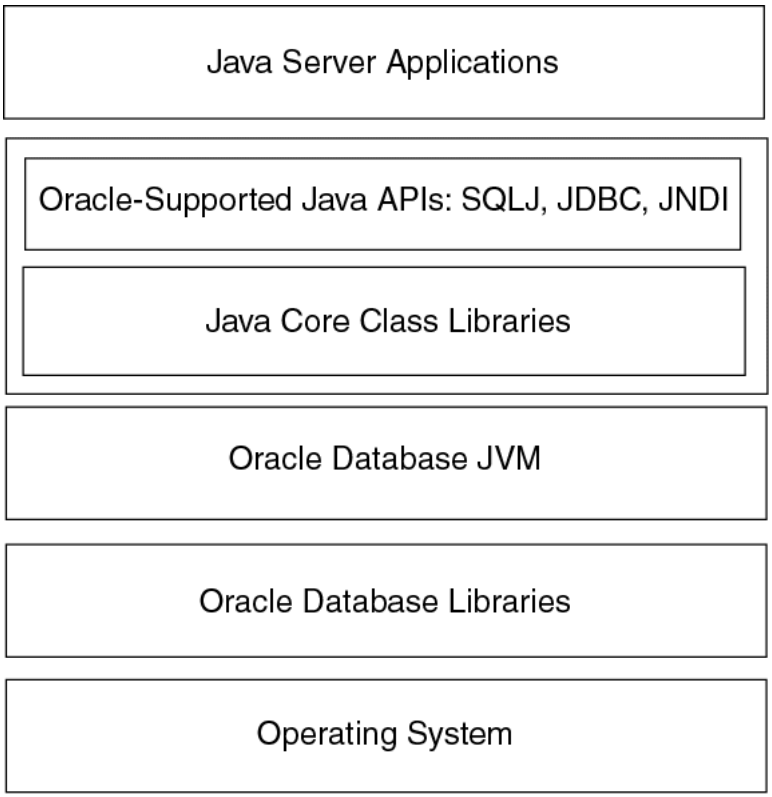
Figure 1–5 Java Component Structure



When you develop a Java program, you use predefined core class libraries written in the Java language. The Java core class libraries are logically divided into packages that provide commonly-used functionality, such as basic language support (`java.lang`), input/output (`java.io`), and network access (`java.net`). Together, the JVM and core class libraries provide a platform on which Java programmers can develop with the confidence that any hardware and operating system that supports Java will execute their program. This concept is what drives the "write once, run anywhere" idea of Java.

Figure 1–6 illustrates how Oracle Java applications sit on top of the Java core class libraries, which in turn sit on top of the JVM. Because the Oracle Java support system is located within the database, the JVM interacts with the Oracle database libraries, instead of directly with the operating system.

Figure 1–6 Oracle Database Java Component Structure



Sun Microsystems furnishes publicly available specifications for both the Java language and the JVM. The Java Language Specification (JLS) defines things such as syntax and semantics; the JVM specification defines the necessary low-level behavior for the "machine" that executes the bytecodes. In addition, Sun Microsystems provides a compatibility test suite for JVM implementors to determine if they have complied with the specifications. This test suite is known as the Java Compatibility Kit (JCK). The OracleJVM implementation complies fully with JCK. Part of the overall Java strategy is that an openly specified standard, together with a simple way to verify compliance with that standard, allows vendors to offer uniform support for Java across all platforms.

Key Features of the Java Language

The Java language has key features that make it ideal for developing server applications. These features include:

- **Simplicity**—Java is a simpler language than most others used in server applications because of its consistent enforcement of the object model. The large, standard set of class libraries brings powerful tools to Java developers on all platforms.
- **Portability**—Java is portable across platforms. It is possible to write platform-dependent code in Java, but it is also simple to write programs that move seamlessly across machines. Oracle server applications, which do not support graphical user interfaces directly on the platform that hosts them, also tend to avoid the few platform portability issues that Java has.
- **Automatic Storage Management**—The Java virtual machine automatically performs all memory allocation and deallocation during program execution. Java programmers can neither allocate nor free memory explicitly. Instead, they depend on the JVM to perform these bookkeeping operations, allocating memory as they create new objects and deallocating memory when the objects are no longer referenced. The latter operation is known as garbage collection.
- **Strong Typing**—Before you use a Java variable, you must declare the class of the object it will hold. Java's strong typing makes it possible to provide a reasonable and safe solution to inter-language calls between Java and PL/SQL applications, and to integrate Java and SQL calls within the same application.
- **No Pointers**—Although Java retains much of the flavor of C in its syntax, it does not support direct pointers or pointer manipulation. You pass all parameters, except primitive types, by reference (that is, object identity is preserved), not by value. Java does not provide C's low level, direct access to pointers, which eliminates memory corruption and leaks.
- **Exception Handling**—Java exceptions are objects. Java requires developers to declare which exceptions can be thrown by methods in any particular class.
- **Flexible Namespace**—Java defines classes and holds them within a hierarchical structure that mirrors the Internet's domain namespace. You can distribute Java applications and avoid name collisions. Java extensions such as the Java Naming and Directory Interface (JNDI) provide a framework for multiple name services to be federated. Java's namespace approach is flexible enough for Oracle to incorporate the concept of a schema for resolving class names, while fully complying with the language specification.

- **Security**—The design of Java bytecodes and the JVM allow for built-in mechanisms to verify that the Java binary code was not tampered with. Oracle Database is installed with an instance of `SecurityManager`, which, when combined with Oracle database security, determines who can invoke any Java methods.
- **Standards for Connectivity to Relational Databases**—JDBC enable Java code to access and manipulate data resident in relational databases. Oracle provides drivers that allow vendor-independent, portable Java code to access the relational database.

Why Use Java in Oracle Database?

The only reason that you are allowed to write and load Java applications within the database is because it is a safe language. Java has been developed to prevent anyone from tampering with the operating system that the Java code resides in. Some languages, such as C, can introduce security problems within the database; Java, because of its design, is a safe language to allow within the database.

Although the Java language presents many advantages to developers, providing an implementation of a JVM that supports Java server applications in a scalable manner is a challenge. This section discusses some of these challenges.

- Java and the RDBMS: A Robust Combination
- Multithreading
- Automated Storage Management With Garbage Collection
- Footprint
- Performance
- Dynamic Class Loading

Java and the RDBMS: A Robust Combination

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

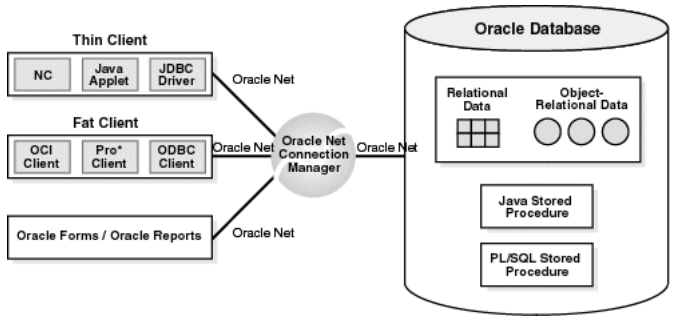
Several features make Java ideal for server programming. Java lets you assemble applications using off-the-shelf software components (JavaBeans). Its type safety and automatic memory management allow for tight integration with the RDBMS. In

addition, Java supports the transparent distribution of application components across a network.

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

Figure 1–7 shows a traditional two-tier, client/server configuration in which clients call Java stored procedures the same way they call PL/SQL stored procedures. (PL/SQL is an advanced 4GL tightly integrated with Oracle Database.) The figure also shows how the Oracle Net Services Connection Manager can funnel many network connections into a single database connection. This enables the RDBMS to support a large number of concurrent users.

Figure 1–7 Two-Tier Client/Server Configuration



Multithreading

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

As a database server, Oracle Database efficiently schedules work for thousands of users. The OracleJVM uses the facilities of the RDBMS server to concurrently schedule Java execution for thousands of users. Although Oracle Database supports Java language level threads required by the JLS and JCK, using threads within the scope of the database will not increase your scalability. Using the embedded scalability of the database eliminates the need for writing multithreaded Java servers. You should use the database's facilities for scheduling users by writing

single-threaded Java applications. The database will take care of the scheduling between each application; thus, you achieve scalability without having to manage threads. You can still write multithreaded Java applications, but multiple Java threads will not increase your server's performance.

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

- Non-Oracle Database model—A single user maps to a single Java language level thread; the same single garbage collector manages all garbage from all users. Different techniques typically deal with allocation and collection of objects of varying lifetimes and sizes. The result in a heavily multithreaded application is, at best, dependent upon operating system support for native threads, which can be unreliable and limited in scalability. High levels of scalability for such implementations have not been convincingly demonstrated.
- OracleJVM model—Even when thousands of users connect to the server and execute the same Java code, each user experiences it as if he is executing his own Java code on his own Java virtual machine. The responsibility of the OracleJVM is to make use of operating system processes and threads, using the scalable approach of the Oracle RDBMS. As a result of this approach, the JVM's garbage collector is more reliable and efficient because it never collects garbage from more than one user at any time. Refer to "Threading in Oracle Database" on page 2-34 for more information on the thread model implementation in OracleJVM.

Automated Storage Management With Garbage Collection

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

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

- The OracleJVM uses the Oracle Database scheduling facilities, which can manage multiple users efficiently.

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

Footprint

The footprint of an executing Java program is affected by many factors:

- Size of the program itself—how many classes and methods and how much code they contain.
- Complexity of the program—the amount of core class libraries that the OracleJVM uses as the program executes, as opposed to the program itself.
- Amount of state the JVM uses—how many objects the JVM allocates, how large they are, and how many must be retained across calls.
- Ability of the garbage collector and memory manager to deal with the demands of the executing program, which is often non-deterministic. The speed with which objects are allocated and the way they are held on to by other objects influences the importance of this factor.

From a scalability perspective, the key to supporting many concurrent clients is a minimum per-user session footprint. The OracleJVM keeps the per-user session footprint to a minimum by placing all read-only data for users, such as Java bytecodes, in shared memory. Appropriate garbage collection algorithms are applied against call and session memories to maintain a small footprint for the user's session. The OracleJVM uses three types of garbage collection algorithms to maintain the user's session memory:

- generational scavenging for short-lived objects
- mark and lazy sweep collection for objects that exist for the life of a single call
- copying collector for long-lived objects—objects that live across calls within a session

Performance

OracleJVM performance is enhanced by implementing a native compiler.

How Native Compilers Improve Performance

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

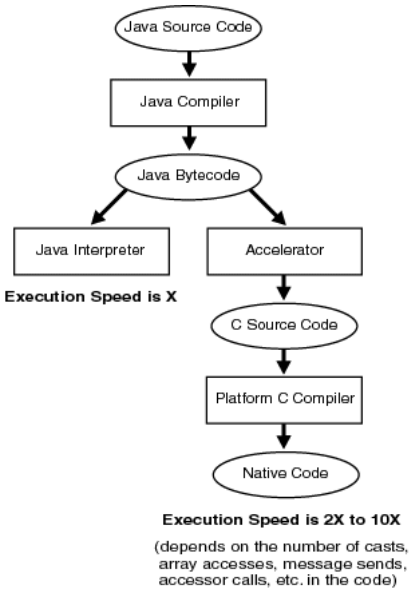
The following describes two methods for native compilation:

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

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

As Figure 1–8 shows, natively compiled code executes up to ten times faster than interpreted code. So, the more native code your program uses, the faster it executes.

Figure 1–8 Interpreter versus Accelerator



Refer to "Natively Compiled Code" on page 10-2 for more information.

Dynamic Class Loading

Another strong feature of Java is dynamic class loading. The class loader loads classes from the disk (and places them in the JVM-specific memory structures necessary for interpretation) only as they are used during program execution. The class loader locates the classes in the `CLASSPATH` and loads them during program execution. This approach, which works well for applets, poses the following problems in a server environment:

Problem	Description	Solution
Predictability	The class loading operation places a severe penalty on first-time execution. A simple program can cause the OracleJVM to load many core classes to support its needs. A programmer cannot easily predict or determine the number of classes loaded.	The OracleJVM loads classes dynamically, just as with any other Java virtual machine. The same one-time class loading speed hit is encountered. However, because the classes are loaded into shared memory, no other users of those classes will cause the classes to load again—they will simply use the same pre-loaded classes.

Problem	Description	Solution
Reliability	A benefit of dynamic class loading is that it supports program updating. For example, you would update classes on a server, and clients who download the program and load it dynamically see the update whenever they next use the program. Server programs tend to emphasize reliability. As a developer, you must know that every client executes a specific program configuration. You do not want clients to inadvertently load some classes that you did not intend them to load.	Oracle Database separates the upload and resolve operation from the class loading operation at runtime. You upload Java code you developed to the server using the `loadjava` utility. Instead of using `CLASSPATH`, you specify a resolver at installation time. The resolver is analogous to `CLASSPATH`, but allows you to specify the schemas in which the classes reside. This separation of resolution from class loading means you always know what program users execute. Refer to Chapter 11, "Schema Object Tools" for details on `loadjava` and resolvers.

What is Different With OracleJVM?

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

Method main()

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

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

The GUI

A server cannot provide GUIs, but it can supply the logic that drives them. The OracleJVM supports only the headless mode of the Abstract Windowing Toolkit (AWT). All AWT Java classes are available within the server environment and your programs can use AWT functionality, as long as they do not attempt to materialize a GUI on the server. For more information, see "User Interfaces on the Server" on page 2-25.

The IDE

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

> **Note:** See "Development Tools" on page 1-26 for more information.

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

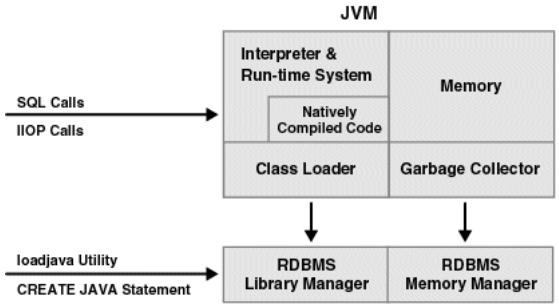
Main Components of the OracleJVM

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

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

The OracleJVM 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 1–9 shows its main components.

Figure 1–9 Main Components of the OracleJVM



The OracleJVM 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 OracleJVM 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 OracleJVMand middle-tier Java business objects, even when they have session-long state.

The garbage collector is described in "Automated Storage Management With Garbage Collection" on page 1-16. The native compiler is discussed in "Performance" on page 1-18. The rest of the components are described in the following sections:

- Library Manager
- Compiler
- Interpreter
- Class Loader
- Verifier

In addition, the following sections give an overview of the JDBC driver:

- Server-Side JDBC Internal Driver

Library Manager

To store Java classes in an Oracle database, you use the command-line utility `loadjava`, which employs SQL `CREATE JAVA` statements to do its work. When invoked by the `CREATE JAVA {SOURCE | CLASS | RESOURCE}` statement, the

library manager loads Java source, class, or resource files into the database. You never access these Java schema objects directly; only the OracleJVM uses them.

Compiler

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

Interpreter

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

Note: You can also compile your code for faster execution. The OracleJVM uses natively compiled versions of the core Java class libraries and JDBC drivers. For more information, see "Natively Compiled Code" on page 10-2.

Class Loader

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

Verifier

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

Server-Side JDBC Internal Driver

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

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

Oracle's Java Application Strategy

One appeal of Java is its ubiquity and the growing number of programmers capable of developing applications using it. Oracle furnishes enterprise application developers with an end-to-end Java solution for creating, deploying, and managing Java applications. The total solution consists of client-side and server-side programmatic interfaces, tools to support Java development, and a Java virtual machine integrated with the Oracle Database server. All these products are 100 percent compatible with Java standards.

Java Programming Environment

In addition to the OracleJVM, the Java programming environment consists of:

- Java stored procedures as the Java equivalent and companion for PL/SQL. Java stored procedures are tightly integrated with PL/SQL. You can call a Java stored procedure from a PL/SQL package; you can call PL/SQL procedures from a Java stored procedure.
- SQL data can be accessed through JDBC.
- Tools and scripts used in assisting in development, class loading, and class management.

To help you decide which Java APIs to use, examine the following table:

Type of functionality you need	Java API to use
To have a Java procedure invoked from SQL, such as a trigger.	Java Stored Procedures
To invoke dynamic, complex SQL statements from a Java object.	JDBC

Java Stored Procedures

If you are a PL/SQL programmer exploring Java, you will be interested in Java stored procedures. A Java stored procedure is a program you write in Java to execute in the server, exactly as a PL/SQL stored procedure. You invoke it directly with products like SQL*Plus, or indirectly with a trigger. You can access it from any Oracle Net client—OCI, PRO* or JDBC. Chapter 5, "Developing Java Stored Procedures" explains how to write stored procedures in Java, how to access them from PL/SQL, and how to access PL/SQL functionality from Java.

In addition, you can use Java to develop powerful programs independently of PL/SQL. Oracle Database provides a fully-compliant implementation of the Java programming language and JVM.

PL/SQL Integration and Oracle RDBMS Functionality

You can invoke existing PL/SQL programs from Java and invoke Java programs from PL/SQL. This solution protects and leverages your existing investment while opening up the advantages and opportunities of Java-based Internet computing.

Oracle offers two different application programming interfaces (APIs) for Java developers to access SQL data—JDBC. Both APIs are available on client and server, so you can deploy the same code in either place.

- JDBC Drivers
- JPublisher

JDBC Drivers

JDBC is a database access protocol that enables you to connect to a database and then prepare and execute SQL statements against the database. Core Java class libraries provide only one JDBC API. JDBC is designed, however, to allow vendors to supply drivers that offer the necessary specialization for a particular database. Oracle delivers the following three distinct JDBC drivers.

Driver	Description
JDBC Thin Driver	You can use the JDBC Thin driver to write 100% pure Java applications and applets that access Oracle SQL data. The JDBC Thin driver is especially well-suited to Web browser-based applications and applets, because you can dynamically download it from a Web page just like any other Java applet.
JDBC Oracle Call Interface Driver	The JDBC Oracle Call Interface (OCI) driver accesses Oracle-specific native code (that is, non-Java) libraries on the client or middle tier, providing some performance boost compared to the JDBC Thin driver, at the cost of significantly larger size and client-side installation.
JDBC Server-side Internal Driver	Oracle Database uses the server-side internal driver when Java code executes on the server. It allows Java applications executing in the server's Java virtual machine to access locally defined data (that is, on the same machine and in the same process) with JDBC. It provides a further performance boost because of its ability to use underlying Oracle RDBMS libraries directly, without the overhead of an intervening network connection between your Java code and SQL data. By supporting the same Java-SQL interface on the server, Oracle Database does not require you to rework code when deploying it.

For more information on JDBC, see "Utilizing JDBC for Querying the Database" on page 3-5.

JPublisher

JPublisher provides a simple and convenient tool to create Java programs that access existing Oracle relational database tables. See the *Oracle Database JPublisher User's Guide* for more information.

Development Tools

The introduction of Java to the Oracle Database server allows you to use several Java Integrated Development Environments. The adherence of Oracle Database to Java compatibility and open Internet standards and protocols ensures that your 100% pure Java programs work when you deploy them on Oracle Database. Oracle delivers many tools or utilities, all written in Java, that make development and deployment of Java server applications easier. Oracle's JDeveloper has many features designed specifically to make deployment of Java stored procedures and Enterprise JavaBeans easier. You can download JDeveloper at the following site:
`http://otn.oracle.com/software/products/jdev/content.html`.

Desupport of J2EE Technologies in the Oracle Database

With the introduction of Oracle Application Server Containers for J2EE (OC4J)--a new, lighter-weight, easier-to-use, faster, and certified J2EE container--Oracle began desupport of the Java 2 Enterprise Edition (J2EE) and CORBA stacks from the database, starting with Oracle9*i* database release 2. However, the database-embedded Java VM (OracleJVM) is still present and will continue to be enhanced to offer Java 2 Standard Edition (J2SE) features, Java stored procedures and JDBC in the database.

As of Oracle9*i* database release 2 (9.2.0), Oracle no longer supports the following technologies in the database:

- the J2EE stack, consisting of:
  - Enterprise Beans (EJB) container
  - JavaServer Pages (JSP) container
  - Oracle9*i* Servlet Engine (OSE)
- the embedded Common Object Request Broker Architecture (CORBA) framework, based on Visibroker for Java

Customers will no longer be able to deploy servlets, JSP pages, EJBs, and CORBA objects in Oracle databases. Oracle9*i* database release 1 (9.0.1) will be the last database release to support the J2EE and CORBA stack. Oracle is encouraging customers to migrate existing J2EE applications running in the database to OC4J now.

2

Java Applications on Oracle Database

Oracle Database executes standard Java applications. However, by integrating Java classes within the database server, your environment is different from a typical Java development environment. This chapter describes the basic differences for writing, installing, and deploying Java applications within Oracle Database.

> **Note:** To fully explore the usage for each API, refer to the documentation for each API. The intent of this chapter is to place the Java APIs in an overall context, with enough detail for you to see how they fit together and how you use them in the Oracle Database environment.

Overview

The OracleJVM platform is a standard, compatible Java environment, which will execute any 100% pure Java application. It has been implemented by Oracle to be compatible with the Java Language Specification and the Java virtual machine specification. It supports the standard Java binary format and the standard Java APIs. In addition, Oracle Database adheres to standard Java language semantics, including dynamic class loading at runtime.

However, unlike other Java environments, the OracleJVM is embedded within the Oracle Database and, therefore, introduces a number of new concepts. This section summarizes the differences between the Sun Microsystems J2SE environment and the environment that occurs when you combine Java with the Oracle Database.

Terminology

Table 2–1 Terminology Definitions

Term	Definition
OracleJVM	Java-enabled Oracle Database database server with JVM.
Session	As a user who executes Java code, you must establish a session in the server. The word *session* as we employ it here is identical to the standard Oracle (or any other database server) usage. A session is typically, although not necessarily, bounded by the time a single user connects to the server.
Call	When a user causes Java code to execute within a session, we refer to it as a *call*. You can initiate a call in different ways. ■ A SQL client program executes a Java stored procedure. ■ A trigger can execute a Java stored procedure. ■ A PL/SQL program calls some Java code. In all cases, a call begins, some combination of Java, SQL, or PL/SQL code is executed to completion, and the call ends.

In your standard Java environment, you run a Java application through the interpreter by executing `java <classname>`. This causes the application to execute within a process on your operating system.

With the OracleJVM, you must load the application into the database, publish the interface, and then run the application within a database session. This book

discusses how to run your Java applications within the database. Specifically, see the following sections for instructions on executing Java in the database:

- Load and publish your Java applications before execution—See "Java Code, Binaries, and Resources Storage" and "Preparing Java Class Methods for Execution", starting on page 2-6.
- Running within a database session—See "Database Sessions Imposed on Java Applications" on page 2-3.

In addition, certain features, included within standard Java, change when you run your application within a database session. These are covered in the following sections:

- Execution Control
- User Interfaces on the Server
- Shortened Class Names
- Class.forName() in Oracle Database
- Managing Your Operating System Resources
- Threading in Oracle Database

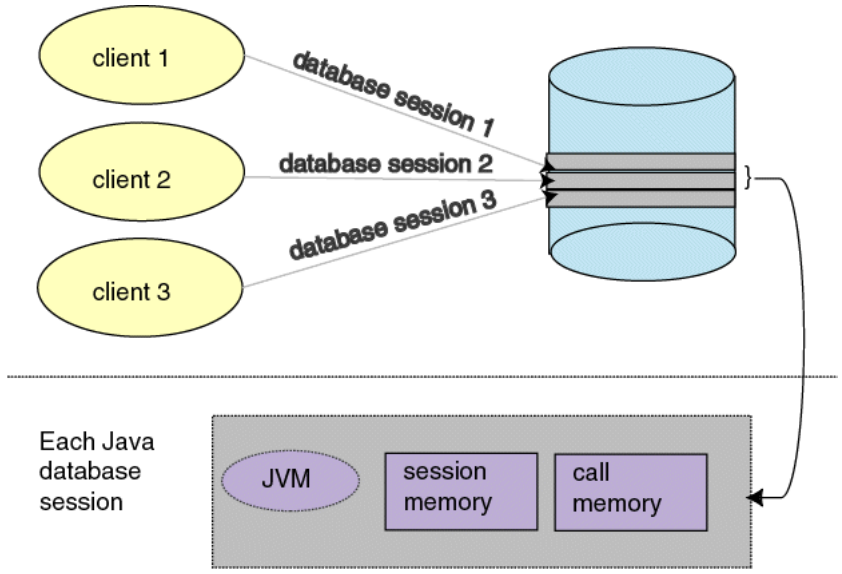
Once you are familiar with this chapter, see Chapter 3, "Invoking Java in the Database" for directions on how to set up your client, and examples for invoking different types of Java applications.

Database Sessions Imposed on Java Applications

In incorporating Java within Oracle Database, your Java application exists within the context of a database session. OracleJVM sessions are entirely analogous to traditional Oracle sessions. Each OracleJVM session maintains the client's Java state across calls within the session.

Figure 2–1 demonstrates how each Java client starts up a database session as the environment for executing Java within the database. Garbage collection, session memory, and call memory exist solely for each client within its session.

Figure 2–1 Java Environment Within Each Database Session



Within the context of a session, the client performs the following:

1. Connects to the database and opens a session.
2. Executes Java within the database. This is referred to as a call.
3. Continues to work within the session, performing as many calls as necessary.
4. Ends the session.

Within a single session, the client has its own Java environment, which is separate from every other client's environment. It appears to the client as if a separate, individual JVM was invoked for each session, although the implementation is vastly more efficient than this seems to imply. Within a session, the OracleJVM manages the scalability for you within the database. Every call executed from a single client is managed within its own session—separately from other clients. The OracleJVM maximizes sharing read-only data between clients and emphasizes a minimum amount of per-session incremental footprint to maximize performance for multiple clients.

The underlying server environment hides the details associated with session, network, state, and other shared resource management issues from Java server code. Static variables are all local to the client. No client can access another client's static variables, because the memory is not available across session boundaries.

Because each client executes its calls within its own session, each client's activities are separate from any other client. During a call, you can store objects in static fields of different classes, and you can expect this state to be available for your next call. The entire state of your Java program is private to you and exists for your entire session.

The OracleJVM manages the following within the session:

- all the objects referenced by Java static variables, all the objects referred to by these objects, and so on (their transitive closure)
- garbage collection for the single client
- session memory for static variables and across call memory needs
- call memory for variables that exist within a single call

Java Supported APIs

For the Oracle Database 10*g* release, we offer the following Java APIs—Java stored procedures, JNDI, and JDBC.

- JNDI—Store objects in a JNDI namespace.
- JDBC—You can access SQL data through JDBC. See Chapter 3, "Invoking Java in the Database", for examples of each Java API.
- Java stored procedures—The lifetime of a Java stored procedure session is identical to the SQL session in which it is embedded. This concept is familiar to PL/SQL users. Any state represented in Java transparently persists for the lifetime of the RDBMS session, simplifying the process of writing stored procedures, triggers, and methods for Oracle Abstract Data Types. Individual invocations of Java code within a session are known as calls. For example, a call may be initiated by a SQL call.

> **Note:** The concepts of call and session apply across all uses of Oracle Database.

Execution Control

In the Sun Microsystems J2SE environment, you develop Java applications with a `main()` method, which is called by the interpreter when the class is run. The `main()` method is invoked when you execute `java <classname>` on the command-line. This command starts the java interpreter and passes the desired

classname to be executed to the interpreter. The interpreter loads the class and starts the execution by invoking `main()`. However, Java applications within the database do not start their execution from a `main()` method.

After loading your Java application within the database (see "Loading Classes" on page 2-17), you can execute your Java code by invoking any static method within the loaded class. The class or methods must be published for you to execute them (see "Publishing" on page 2-24). Your only entry point is no longer always assumed to be `main()`. Instead, when you execute your Java application, you specify a method name within the loaded class as your entry point.

For example, in a normal Java environment, you would start up the Java object on the server by executing the following:

```
java myprogram
```

where `myprogram` is the name of a class that contains a `main()` method. In `myprogram`, `main()` immediately calls `mymethod` for processing incoming information.

In Oracle Database, you load the `myprogram.class` file into the database and publish `mymethod` as an entry-point. Then, the client or trigger can invoke `mymethod` explicitly.

Java Code, Binaries, and Resources Storage

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

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

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

Both of these concepts are different within the database. The following table describes how Oracle Database handles Java classes and locates dependent classes.

Table 2–2 Description for Java Code and Classes

Java Code and Classes	Description
Java code, binaries, and resources	In the OracleJVM environment, source, classes, and resources reside within Oracle Database. Because they reside in the database, they are known as Java schema objects, where a schema corresponds to a database user. There are three types of Java objects: source, class, and resource. There are no `.java`, `.class`, `.properties`, or `.ser` files on the server; instead, these files map to source, class, and resource Java schema objects.
Locating Java classes	Instead of a `CLASSPATH`, you use a resolver to specify one or more schemas to search for source, class, and resource Java schema objects.

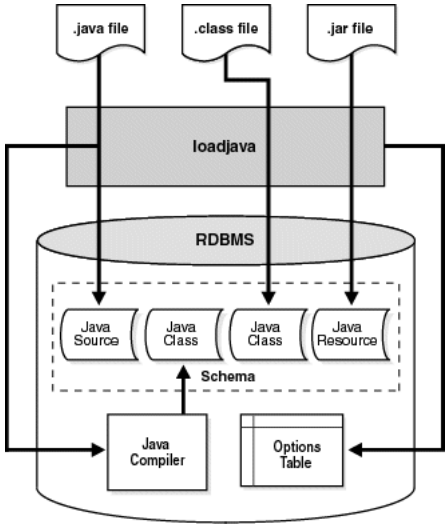
The call and session terms, used during our discussions, are not Java terms; but are server terms that apply to the OracleJVM platform. The Oracle Database memory manager preserves Java program state throughout your session (that is, between calls). The JVM uses the Oracle database to hold Java source,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

Java Classes Loaded in the Database

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

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

Figure 2–2 Loading Java into the Oracle Database



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

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

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

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

The Oracle RDBMS accepts Java names up to 4000 characters long. However, the names of Java schema objects cannot be longer than 31 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. See "Shortened Class Names" on page 2-26 for more information.

Preparing Java Class Methods for Execution

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

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

Compiling Java Classes

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

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

> **Note:** If you decide to compile through `loadjava`, you can specify compiler options. See "Specifying Compiler Options" on page 2-10 for more information.

Compiling Source Through javac

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

Compiling Source Through loadjava

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

1. The source file is loaded as a source schema object.
2. The source file is compiled.

3. Class schema objects are created for each class defined in the compiled `.java` file.
4. The compiled code is stored in the class schema objects.

Oracle Database logs all compilation errors both to `loadjava`'s log file and the `USER_ERRORS` view.

Compiling Source at Runtime

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

Oracle Database logs all compilation errors both to `loadjava`'s log file and the `USER_ERRORS` view.

Specifying Compiler Options

There are two ways to specify options to the compiler.

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

  You must create this table yourself if you wish to specify compiler options this way. See "Compiler Options Specified in a Database Table" on page 2-11 for instructions on how to create the `JAVA$OPTIONS` table.

The following sections describe your compiler options:

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

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

- `encoding = System.getProperty("file.encoding");`

- `online` = `true`: This option applies only to Java sources that contain SQLJ constructs.
- `debug` = `true`: This option is equivalent to `javac -g`.

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

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

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

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

The parameters for these methods are described in the following table:

Table 2–3 Definitions for Name and Option Parameters

Parameter	Description
name	The `name` parameter is a Java package name, a fully qualified class name, or the empty string. When the compiler searches the `JAVA$OPTIONS` table for the options to use for compiling a Java source schema object, it uses the row whose `name` most closely matches the schema object's fully qualified class name. A `name` whose value is the empty string matches any schema object name.
option	The `option` parameter is either `'online'`, `'encoding'` or `'debug'`.

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

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

Table 2–4 represents a hypothetical JAVA$OPTIONS database table. The pattern match rule is to match as much of the schema name against the table entry as possible. The schema name with a higher resolution for the pattern match is the entry that applies. Because the table has no entry for the encoding option, the compiler uses the default or the value specified on the command line. The online option shown in the table matches schema object names as follows:

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

Table 2–4 Example JAVA$OPTIONS Table

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

Automatic Recompilation

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

```
public class A
{
      B b;
```

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

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

The dependency system enables you to rely on Oracle Database to manage dependencies between classes, to recompile, and to resolve automatically. You must force compilation and resolution yourself only if you are developing and you want to find problems early. The `loadjava` utility also provides the facilities for forcing compilation and resolution if you do not want to allow the dependency management facilities to perform this for you.

Resolving Class Dependencies

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

All classes loaded within the database are referred to as class schema objects and are loaded within certain schemas. All JVM classes, such as `java.lang.*`, are loaded within `PUBLIC`. If your classes depend upon other classes you have defined, you will probably load them all within your own schema. For example, if your schema is `SCOTT`, the database resolver (the database replacement for `CLASSPATH`) searches

the SCOTT schema before PUBLIC. The listing of schemas to search is known as a resolver spec. Resolver specs are per-class, whereas in a classic Java virtual machine, CLASSPATH is global to all classes.

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

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

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

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

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

  ```
  loadjava -resolve
  ```

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

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

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

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

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

Allowing References to Non-Existent Classes

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

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

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

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

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

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

If you later intend to load the non-existent classes that were causing you to use such a resolver in the first place, you should not use a resolver containing the "`-`." Instead, include all referenced classes in the schema before resolving.

Note: An alternative mechanism for dealing with non-existent classes is with the `-gemissing` option of `loadjava`. This option causes `loadjava` to create and load definitions of classes that are referenced, but not defined. For more details, see "loadjava" on page 2-2.

ByteCode Verifier

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

Table 2–5 ORA Errors

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

The resolver also issues warnings, as defined below:

- Resolvers containing "-"

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

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

  The method is valid only if A extends B, or A implements the interface B. If A or B have been resolved using a "-" term, the resolver does not know that this

method is safe. It will replace the bytecodes of `myMethod` with bytecodes that throw an Exception if `myMethod` is ever called.

- Use of other resolvers

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

For more information on class resolution and loading your classes within the database, see Chapter 11, "Schema Object Tools".

Loading Classes

This section gives an overview of loading your classes into the database using the `loadjava` tool. You can also execute `loadjava` within your SQL. See Chapter 11, "Schema Object Tools" for complete information on `loadjava`.

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

Table 2–6 Description of Java Files

Java File Types	Description
`.java` source files	correspond to Java source schema objects
`.class` compiled Java files	correspond to Java class schema objects
`.properties` Java resource files or data files	correspond to Java resource schema objects

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

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

Table 2–7 loadjava Operations on Schema Objects

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

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

> **Note:** More options for loadjava are available. However, this section discusses only the major options. See Chapter 11, "Schema Object Tools" for complete information on loadjava and dropjava.

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

- Defining the Same Class Twice
- Designating Database Privileges and JVM Permissions

- Loading JAR or ZIP Files

After loading, you can access the `USER_OBJECTS` view in your database schema to verify that your classes and resources loaded properly. For more information, see "Checking Java Uploads" on page 2-23.

Defining the Same Class Twice

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

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

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

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

  - Drop `x.java`, load `z.java` (which defines `y`), then load the new `x.java` (which does not define `y`).
  - Load the new `x.java` (which does not define `y`), then load `z.java` (which defines `y`).

Designating Database Privileges and JVM Permissions

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

- `CREATE PROCEDURE` and `CREATE TABLE` privileges to load into your schema.
- `CREATE ANY PROCEDURE` and `CREATE ANY TABLE` privileges to load into another schema.
- `oracle.aurora.security.JServerPermission.loadLibraryInClass.<classname>`. See "Permission for Loading Classes" on page 9-25 for more information.

Loading JAR or ZIP Files

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

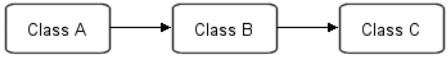
Note: Oracle Database does not reload a class if it has not changed since the last load. However, you can force a class to be reloaded through the `loadjava -force` option.

How to Grant Execute Rights

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

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

Figure 2–3 Execution Rights



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

For information on JVM security permissions, see Chapter 10, "Oracle Database Java Application Performance".

Controlling the Current User

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

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

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

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

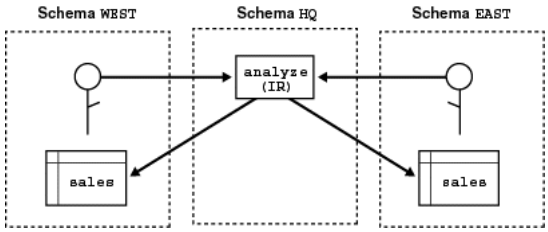
On the other hand, definer-rights procedures are bound to the schema in which they reside. They execute with the privileges of their definer, and their unqualified references to schema objects are resolved in the schema of the definer.

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

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

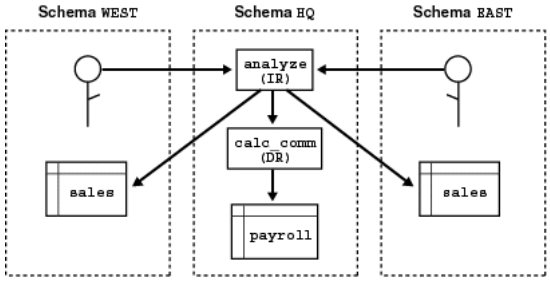
To solve the problem, the company installs an invoker-rights (IR) version of the procedure `analyze` at headquarters. Now, as Figure 2–4 shows, all regional sites can use the same procedure to query their own `sales` tables.

Figure 2–4 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 2–5 shows, the solution is to have procedure `analyze` call the definer-rights (DR) procedure `calcComm`, which, in turn, updates the `payroll` table.

Figure 2–5 Indirect Access



To override the default invoker-rights behavior, specify the `loadjava` option `-definer`, which is similar to the UNIX facility `setuid`, except that `-definer` applies to individual classes, not whole programs. Alternatively, you can execute the SQL DDL that changes the AUTHID of the current user.

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

Checking Java Uploads

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

Columns in USER_OBJECTS include those contained in Table 2–8 below.

Table 2–8 Key USER_OBJECT Columns

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

Object Name and Type

An OBJECT_NAME in USER_OBJECTS is the short name. The full name is stored as a short name if it exceeds 31 characters. See "Shortened Class Names" on page 2-26 for more information on full and short names.

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

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

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

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

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

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

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

Status

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

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

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

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

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

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

Publishing

Oracle Database enables clients and SQL to invoke Java methods that are loaded within the database, once published. You publish either the object itself or individual methods. If you write a Java stored procedure that you intend to invoke with a trigger, directly or indirectly in SQL DML or in PL/SQL, you must publish individual methods within the class. Specify how to access it through a call specification. Java programs consist of many methods in many classes; however,

only a few static methods are typically exposed with call specifications. See Chapter 6, "Publishing Java Classes With Call Specs" for more details.

User Interfaces on the Server

Oracle Database furnishes all core Java class libraries on the server, including those associated with presentation of the user interfaces `java.awt` and `java.applet`. It is, however, inappropriate for code executing in the server to attempt to materialize or display a user interface in the server. Users running applications in the OracleJVM should not be expected—nor allowed—to interact with or depend on the display and input hardware of the server on which Oracle Database is running.

To address compatibility on platforms that do not support a display, keyboard, or mouse, Java 1.4 outlines "Headless AWT" support. The Headless AWT API introduces a new public runtime exception class, `java.awt.HeadlessException`. The constructors of the `Applet` class, all heavy-weight components, and many of the methods in the `Toolkit` and `GraphicsEnvironment` classes, that rely on the native display devices are changed to throw the `HeadlessException` if the platform does not support a display. In Oracle Database, user interfaces are supported only on client applications. Accordingly, the OracleJVM is a "Headless Platform" and throws a `HeadlessException` if those methods are called.

Most AWT computation that does not involve accessing the underlying native display or input devices is allowed in Headless AWT. In fact, Headless AWT is quite powerful as it allows programmers access to fonts, imaging, printing, and color and ICC manipulation. For example, applications running in the OracleJVM can parse, manipulate, and write out images as long as they do not try to physically display it in the server. The Sun reference JVM implementation can be invoked in Headless mode (by supplying the property `-Djava.awt.headless=true`) and run with the same Headless AWT restrictions as the OracleJVM does. The OracleJVM fully complies with the Java Compatibility Kit (JCK) with respect to Headless AWT. See the following Web page for more information on Headless Support in J2SE 1.4:

`http://java.sun.com/j2se/1.4/docs/guide/awt/AWTChanges.html#headless`

The OracleJVM takes a similar approach for sound support to how it handles AWT. Applications in the OracleJVM are not allowed to access the underlying sound system for purposes of sound playback or recording. Instead, the system sound resources appear to be unavailable in a manner consistent with the sound API specification of the methods that are trying to access the resources. For example, methods in `javax.sound.midi.MidiSystem` that attempt to access the

underlying system sound resources throw the checked exception `MidiUnavailableException` to signal that the system is unavailable. However, similar to Oracle Database Headless AWT support, Oracle Database supports the APIs that allow sound file manipulation free of the native sound devices. The OracleJVM also fully complies with the JCK when it implements the sound API.

Shortened Class Names

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

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

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

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

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

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

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

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

Refer to Chapter 8, "Java Stored Procedures Application Example" for a detailed example of the use of this function and ways to determine which Java schema objects are present on the server.

Class.forName() in Oracle Database

The Java Language Specification provides the following description of `Class.forName()`:

> Given the fully-qualified name of a class, this method attempts to locate, load, and link the class. If it succeeds, a reference to the Class object for the class is returned. If it fails, a `ClassNotFoundException` is thrown.

Class lookup is always on behalf of a referencing class through a `ClassLoader`. The difference between the JDK implementation and the OracleJVM implementation is the method on which the class is found:

- The JDK uses one ClassLoader that searches the set of directory tree roots specified by the environment variable `CLASSPATH`.
- OracleJVM defines several resolvers, which define how to locate classes. Every class has a resolver associated with it, and each class can, potentially, have a different resolver. When you execute a method that calls `Class.forName()`, the resolver of the currently executing class (`this`) is used to locate the class. See "Resolving Class Dependencies" on page 2-13 for more information on resolvers.

You can receive unexpected results if you try to locate a class with an unexpected resolver. For example, if a class X in schema X requests a class Y in schema Y to look up class Z, you can experience an error if you expected class X's resolver to be used. Because class Y is performing the lookup, the resolver associated with class Y is used to locate class Z. In summary, if the class exists in another schema and you specified different resolvers for different classes—as would happen by default if they are in different schemas— you might not find the class.

You can solve this resolver problem as follows:

- Avoid any class name lookup by passing the Class object itself.
- Supply the `ClassLoader` in the `Class.forName` method.
- Supply the class and the schema it resides into `classForNameAndSchema` method.
- Supply the schema and class name to `ClassForName.lookupClass`.

- Serialize your objects with the schema name with the class name.

> **Note:** Another unexpected behavior can occur if system classes invoke `Class.forName()`. The desired class is found only if it resides in SYS or in PUBLIC. If your class does not exist in either SYS or PUBLIC, you can declare a PUBLIC synonym for the class.

Supply the ClassLoader in Class.forName

Oracle Database uses resolvers for locating classes within schemas. Every class has a specified resolver associated with it and each class can have a different resolver associated with it. Thus, the locating of classes is dependent on the definition of the associated resolver. The ClassLoader knows which resolver to use, based upon the class that is specified. When you supply a ClassLoader to `Class.forName()`, your class is looked up in the schemas defined within the resolver of the class. The syntax for this variant of `Class.forName` is as follows:

```
Class forName (String name, boolean initialize, ClassLoader loader);
```

The following examples show how to supply the class loader of either the current class instance or the calling class instance.

Example 2–1 Retrieve Resolver from Current Class

You can retrieve the class loader of any instance through the `Class.getClassLoader` method. The following example retrieves the class loader of the class represented by instance `x`.

```
Class c1 = Class.forName (x.whatClass(), true, x.getClass().getClassLoader());
```

Example 2–2 Retrieve Resolver from Calling Class

You can retrieve the class of the instance that invoked the executing method through the `oracle.aurora.vm.OracleRuntime.getCallerClass` method. Once you retrieve the class, invoke the `Class.getClassLoader` method on the returned class. The following example retrieves the class of the instance that invoked the `workForCaller` method. Then, its class loader is retrieved and supplied to the `Class.forName` method. Thus, the resolver used for looking up the class is the resolver of the calling class.

```
void workForCaller() {
    ClassLoader c1 =
             oracle.aurora.vm.OracleRuntime.getCallerClass().getClassLoader();
```

```
...
Class c = Class.forName (name, true, c1);
```

Supply Class and Schema Names to classForNameAndSchema

You can resolve the problem of where to find the class by either supplying the resolver, which knows the schemas to search, or by supplying the schema in which the class is loaded. If you know in which schema the class is loaded, you can use the `classForNameAndSchema` method. Oracle Database provides a method in the `DbmsJava` class, which takes in both the name of the class and the schema in which the class resides. This method locates the class within the designated schema.

Example 2–3 Providing Schema and Class Names

The following example shows how you can save the schema and class names in the `save` method. Both names are retrieved, and the class is located using the `DbmsJava.classForNameAndSchema` method.

```
import oracle.aurora.rdbms.ClassHandle;
import oracle.aurora.rdbms.Schema;
import oracle.aurora.rdbms.DbmsJava;

void save (Class c1) {
    ClassHandle handle = ClassHandle.lookup(c1);
    Schema schema = handle.schema();
    writeNmae (schema.getName());
    writeName (c1.getName());
}

Class restore() {
    String schemaName = readName();
    String className = readName();
    return DbmsJava.classForNameAndSchema (schemaName, className);
}
```

Supply Class and Schema Names to lookupClass

You can supply a single String, containing both the schema and class names, to the `oracle.aurora.util.ClassForName.lookupClass` method. When invoked, this method locates the class in the specified schema. The string must be in the following format:

```
"<schema>:<class>"
```

For example, to locate `com.package.myclass` in schema SCOTT, execute the following:

```
oracle.aurora.util.ClassForName.lookupClass("SCOTT:com.package.myclass");
```

Note: You must use uppercase characters for the schema name. In this case, the schema name is case-sensitive.

Supply Class and Schema Names when Serializing

When you de-serialize a class, part of the operation is to lookup a class based on a name. In order to ensure that the lookup is successful, the serialized object must contain both the class and schema names.

Oracle Database provides the following classes for serializing and de-serializing objects:

- `oracle.aurora.rdbms.DbmsObjectOutputStream`

This class extends `java.io.ObjectOutputStream` and adds schema names in the appropriate places.

- `oracle.aurora.rdbms.DbmsObjectInputStream`

This class extends `java.io.ObjectInputStream` and reads streams written by `DbmsObjectOutputStream`. You can use this class in any environment. If used within Oracle Database, the schema names are read out and used when performing the class lookup. If used on a client, the schema names are ignored.

Class.forName Example

The following example shows several methods for looking up a class.

- To use the resolver of this instance's class, invoke `lookupWithClassLoader`. This method supplies a class loader to the `Class.forName` method in the `from` variable. The class loader specified in the `from` variable defaults to this class.
- To use the resolver from a specific class, call `ForName` with the designated class name, followed by `lookupWithClassLoader`. The `ForName` method sets the `from` variable to the specified class. The `lookupWithClassLoader` method uses the class loader from the specified class.
- To use the resolver from the calling class, first invoke the `ForName` method without any parameters. It sets the `from` variable to the calling class. Then,

invoke the `lookupWithClassLoader` to locate the class using the resolver of the calling class.

- To lookup a class in a specified schema, invoke the `lookupWithSchema` method. This provides the class and schema name to the `classForNameAndSchema` method.

```
import oracle.aurora.vm.OracleRuntime;
import oracle.aurora.rdbms.Schema;
import oracle.aurora.rdbms.DbmsJava;

public class ForName {

  private Class from;
  /* Supply an explicit class to the constructor */
  public ForName(Class from) {
    this.from = from;
  }
  /* Use the class of the code containing the "new ForName()" */
  public ForName() {
    from = OracleRuntime.getCallerClass();
  }

  /* lookup relative to Class supplied to constructor */
  public Class lookupWithClassLoader(String name) throws ClassNotFoundException
{
    /* A ClassLoader uses the resolver associated with the class*/
    return Class.forName(name, true, from.getClassLoader());
  }

  /* In case the schema containing the class is known */
  static Class lookupWithSchema(String name, String schema) {
    Schema s = Schema.lookup(schema);
    return DbmsJava.classForNameAndSchema(name, s);
  }
}
```

Managing Your Operating System Resources

Operating system resources are a limited commodity on any computer. Because Java is targeted at providing a computing platform as well as a programming language, it contains platform-independent classes and frameworks for accessing platform-specific resources. The Java class methods access operating system resources through the JVM. Java has potential problems with this model, because

programmers rely on the garbage collector to manage all resources, when all that the garbage collector manages is Java objects, not the operating system resources that the Java object holds on to.

In addition, when you use shared servers, your operating system resources, which are contained within Java objects, can be invalidated if they are maintained across calls within a session. For further details, see "Operating System Resources Affected Across Calls".

The following sections discusses these potential problems:

- Overview of Operating System Resources
- Garbage Collection and Operating System Resources

Overview of Operating System Resources

In general, your operating system resources contain the following:

Operating System Resources	Description
memory	Oracle Database manages memory internally, allocating memory as you create new objects and freeing objects as you no longer need them. The language and class libraries do not support a direct means to allocate and free memory. "Automated Storage Management With Garbage Collection" on page 1-16 discusses garbage collection.
files and sockets	Java contains classes that represent file or socket resources. Instances of these classes hold on to your operating system's file or socket constructs, such as file handles.
threads	Threads are discouraged within the OracleJVM because of scalability issues. However, you can have a multi-threaded application within the database. "Threading in Oracle Database" on page 2-34 discusses in detail the OracleJVM threading model.

Operating System Resource Access

By default, a Java user does not have direct access to most operating system resources. A system administrator may give permission to a user to access these resources by modifying the JVM security restrictions. The JVM security enforced upon system resources conforms to Java 2 security. See "Java 2 Security" on page 9-3 for more information.

Operating System Resource Lifetime

You access operating system resources using the standard core Java classes and methods. Once you access a resource, the time that it remains active (usable) varies according to the type of resource. Memory is garbage collected as described in "Automated Storage Management With Garbage Collection" on page 1-16. Files, threads, and sockets persist across calls when you use a dedicated mode server. In shared server mode, files, threads, and sockets terminate when the call ends. For more information, see "Operating System Resources Affected Across Calls".

Garbage Collection and Operating System Resources

Imagine that memory is divided into two realms: Java object memory and operating system constructs. The Java object memory realm contains all objects and variables. Operating system constructs include resources that the operating system allocates to the object when it asks. These resources include files, sockets, and so on.

Basic programming rules dictate that you close all memory—both Java objects and operating system constructs. Java programmers incorrectly assume that all memory is freed by the garbage collector. The garbage collector was created to collect all unused Java object memory. However, it does not close any operating system constructs. All operating system constructs must be closed by the program before the Java object is collected.

For example, whenever an object opens a file, the operating system creates the file and gives the object a file handle. If the file is not closed, the operating system will hold the file handle construct open until the call ends or JVM exits. This can cause you to run out of these constructs earlier than necessary. There are a finite number of handles within each operating system. To guarantee that you do not run out of handles, close your resources before exiting the method. This includes closing the streams attached to your sockets. You should close the streams attached to the socket before closing the socket.

So why not expand the garbage collector to close all operating system constructs? For performance reasons, the garbage collector cannot examine each object to see if it contains a handle. Thus, the garbage collector collects Java objects and variables, but does not issue the appropriate operating system methods for freeing any handles.

Example 2–4 shows how you should close the operating system constructs.

Example 2–4 Closing Your Operating System Resources

```
public static void addFile(String[] newFile) {
```

```
File inFile = new File(newFile);
FileReader in = new FileReader(inFile);
int i;

while ((i = in.read()) != -1)
   out.write(i);
/*closing the file, which frees up the operating system file handle*/
in.close();
}
```

If you do not close the `in` file, eventually the `File` object will be garbage collected. However, even if the `File` object is garbage collected, the operating system still believes that the file is in use, because it was not closed.

> **Note:** You might want to use Java finalizers to close resources. However, finalizers are not guaranteed to run in a timely manner. Instead, finalizers are put on a queue to execute when the garbage collector has time. If you close your resources within your finalizer, it might not be freed up until the JVM exits. The best approach is to close your resources within the method.

Threading in Oracle Database

TheOracleJVM implements a non-preemptive threading model. With this model, the JVM runs all Java threads on a single operating system thread. It schedules them in a round-robin fashion and switches between them only when they block. Blocking occurs when you, for example, invoke the `Thread.yield()` method or wait on a network socket by invoking `mySocket.read()`.

Advantages of the Oracle Database Threading Model	Disadvantages
■ simple to program ■ efficient to implement in the Java virtual machine, because a thread switch does not require any system calls ■ safer, because the JVM can detect a deadlock that would hang a preemptive JVM and can then raise a runtime exception	■ does not exhibit any concurrency ■ lack of portability ■ performance considerations, because of the system calls required for locking when blocking the thread ■ memory scalability, because efficient multi-threaded memory allocation requires a larger pool of memory

Oracle chose this model because any Java application written on a single-processor system works identical to one written on a multi-processor system. Also, the lack of concurrency among Java threads is not an issue, because OracleJVM is embedded in the database, which provides a higher degree of concurrency than any conventional JVM.

There is no need to use threads within the application logic because the Oracle server preemptively schedules the session JVMs. If you must support hundreds or thousands of simultaneous transactions, start each one in its own JVM. This is exactly what happens when you create a session in the OracleJVM. The normal transactional capabilities of the Oracle database server accomplish coordination and data transfer between the JVMs. This is not a scalability issue, because in contrast to the 6 MB-8 MB memory footprint of the typical Java virtual machine, the Oracle server can create thousands of JVMs, with each one taking less than 40 KB.

Threading is managed within the OracleJVM by servicing a single thread until it completes or blocks. If the thread blocks, by yielding or waiting on a network socket, the JVM will service another thread. However, if the thread never blocks, it is serviced until completed.

The OracleJVM has added the following features for better performance and thread management:

- System calls are at a minimum. OracleJVM has exchanged some of the normal system calls with non-system solutions. For example, entering a monitor-synchronized block or method does not require a system call.
- Deadlocks are detected.
    * The OracleJVM monitors for deadlocks between threads. If a deadlock occurs, the OracleJVM terminates one of the threads and throws the `oracle.aurora.vm.DeadlockError` exception.
    * Single-threaded applications cannot suspend. If the application has only a single thread and you try to suspend it, the `oracle.aurora.vm.LimboError` exception is thrown.

Thread Life Cycle

In the single-threaded execution case, the call ends when one of the following events occurs:

1. The thread returns to its caller.
2. An exception is thrown and is not caught in Java code.

3. The `System.exit()`, `oracle.aurora.vm.OracleRuntime.exitCall()` method is invoked.

If the initial thread creates and starts other Java threads, the rules about when a call ends are slightly more complicated. In this case, the call ends in one of the following two ways:

1. The main thread returns to its caller, or an exception is thrown and not caught in this thread, *and* all other non-daemon threads complete execution. Non-daemon threads complete either by returning from their initial method or because an exception is thrown and not caught in the thread.
2. Any thread invokes the `System.exit()` or `oracle.aurora.vm.OracleRuntime.exitCall()` method.

In shared server mode, when a call ends because of a return or uncaught exceptions, the OracleJVM throws a `ThreadDeathException` in all daemon threads. The `ThreadDeathException` essentially forces threads to stop execution. For other considerations, see "Operating System Resources Affected Across Calls".

In both dedicated and shared server mode, when a call ends because of a call to `System.exit()` or `oracle.aurora.vm.OracleRuntime.exitCall()`, the OracleJVM ends the call abruptly and terminates all threads, but does not throw `ThreadDeathException`.

During the execution of a single call, a Java program can recursively cause more Java code to be executed. For example, your program can issue a SQL query using JDBC that in turn causes a trigger written in Java to be invoked. All the preceding remarks regarding call lifetime apply to the top-most call to Java code, not to the recursive call. For example, a call to `System.exit()` from within a recursive call will exit the entire top-most call to Java, not just the recursive call.

Special Considerations for Shared Servers

For sessions that use shared servers, the limitations across calls that applied in previous releases are still present. The reason is that a session that uses a shared server is not guaranteed to connect to the same process on a subsequent database call, and hence the session-specific memory and objects that need to live across calls are saved in the System Global Area. This means that process-specific resources, such as threads, open files and sockets must be cleaned up at the end of each call, and hence will not be available for the next call.

End-of-Call Migration

In shared server mode, Oracle Database preserves the state of your Java program between calls by migrating all objects that are reachable from static variables into session space at the end of the call. Session space exists within the client's session to store static variables and objects that exist between calls. OracleJVM performs this migration operation at the end of every call, without any intervention by you.

This migration operation is a memory and performance consideration; thus, you should be aware of what you designate to exist between calls, and keep the static variables and objects to a minimum. If you store objects in static variables needlessly, you impose an unnecessary burden on the memory manager to perform the migration and consume per-session resources. By limiting your static variables to only what is necessary, you help the memory manager and improve your server's performance.

To maximize the number of users who can execute your Java program at the same time, it is important to minimize the footprint of a session. In particular, to achieve maximum scalability, an inactive session should take up as little memory space as possible. A simple technique to minimize footprint is to release large data structures at the end of every call. You can lazily recreate many data structures when you need them again in another call. For this reason, the OracleJVM has a mechanism for calling a specified Java method when a session is about to become inactive, such as at end-of-call time.

This mechanism is the `EndOfCallRegistry` notification. It enables you to clear static variables at the end of the call and reinitialize the variables using a lazy initialization technique when the next call comes in. You should execute this only if you are concerned about the amount of storage you require the memory manager to store in between calls. It becomes a concern only for more complex stateful server applications you implement in Java.

The decision of whether to null-out data structures at end-of-call and then recreate them for each new call is a typical time and space trade-off. There is some extra time spent in recreating the structure, but you can save significant space by not holding on to the structure between calls. In addition, there is a time consideration, because objects—especially large objects—are more expensive to access after they have been migrated to session space. The penalty results from the differences in representation of session, as opposed to call-space based objects.

Examples of data structures that are candidates for this type of optimization include:

- Buffers or caches.

- Static fields, such as Arrays, that once initialized, can remain unchanged during the course of the program.
- Any dynamically built data structure that could have a space-efficient representation between calls and a more speed-efficient representation for the duration of a call. Because this can be tricky and complicate your code, making it hard to maintain, so you should consider doing this only after demonstrating that the space saved is worth the effort.

Oracle-Specific Support for End-of-Call Optimization

You can register the static variables that you want cleared at the end of the call when the buffer, field, or data structure is created. Within the Oracle-specified `oracle.aurora.memoryManager.EndOfCallRegistry` class, the `registerCallback` method takes in an object that implements a `Callback` object. The `registerCallback` object stores this object until the end of the call. When end-of-call occurs, OracleJVM invokes the `act` method within all registered `Callback` objects. The `act` method within the `Callback` object is implemented to clear the user-defined buffer, field, or data structure. Once cleared, the `Callback` is removed from the registry.

> **Note:** If the end of the call is also the end of the session, callbacks are not invoked, because the session space will be cleared anyway.

The way that you use the `EndOfCallRegistry` depends on whether you are dealing with objects held in static fields or instance fields.

- Static fields—You use `EndOfCallRegistry` to clear state associated with an entire class. In this case, the `Callback` object should be held in a private static field. Any code that requires access to the cached data that was dropped between calls must invoke a method that lazily creates—or recreates—the cached data. The example below does the following:
  1. Creates a `Callback` object within a static field, `thunk`.
  2. Registers this `Callback` object for end-of-call migration.
  3. Implements the `Callback.act` method to free up all static variables, including the Callback object itself.
  4. Provides a method, `createCachedField`, for lazily recreating the cache.

When the user creates the cache, the `Callback` object is automatically registered within the `getCachedField` method. At end-of-call, OracleJVM invokes the registered `Callback.act` method, which frees the static memory.

```
import oracle.aurora.memoryManager.Callback;
import oracle.aurora.memoryManager.EndOfCallRegistry;

class Example {
  static Object cachedField = null;
  private static Callback thunk = null;

  static void clearCachedField() {
      // clear out both the cached field, and the thunk so they don't
      // take up session space between calls
       cachedField = null;
       thunk = null;
  }

  private static Object getCachedField() {
   if (cachedField == null) {
      // save thunk in static field so it doesn't get reclaimed
      // by garbage collector
      thunk = new Callback () {
            public void act(Object obj) {
                  Example.clearCachedField();
            }
      };

      // register thunk to clear cachedField at end-of-call.
      EndOfCallRegistry.registerCallback(thunk);
      // finally, set cached field
      cachedField = createCachedField();
   }
   return cachedField;
  }

  private static Object createCachedField() {
  ....
  }
}
```

- Instance fields—Use `EndOfCallRegistry` to clear state in data structures held in instance fields. For example, when a state is associated with each instance of a class, each instance has a field that holds the cached state for the

instance and fills in the cached field as necessary. You can access the cached field with a method that ensures the state is cached.

1. Implements the instance as a `Callback` object.
2. Implements the `Callback.act` method to free up the instance's fields.
3. When the user requests a cache, the `Callback` object registers itself for end-of-call migration.
4. Provides a method, `createCachedField`, for lazily recreating the cache.

When the user creates the cache, the `Callback` object is automatically registered within the `getCachedField` method. At end-of-call, OracleJVM invokes the registered `Callback.act` method, which frees the cache.

This approach ensures that the lifetime of the `Callback` object is identical to the lifetime of the instance, because they are the same object.

```
import oracle.aurora.memoryManager.Callback;
import oracle.aurora.memoryManager.EndOfCallRegistry;

class Example2 implements Callback {
   private Object cachedField = null;

   public void act(Object obj) {
    // clear cached field
    cachedField = null;
    obj = null;
   }

  // our accessor method
  private static Object getCachedField() {
   if (cachedField == null) {
      // if cachedField is not filled in then we need to
      // register self, and fill it in.
      EndOfCallRegistry.registerCallback(self);
      cachedField = createCachedField();
   }
   return cachedField;
  }

  private Object createCachedField() {
  ....
  }
}
```

A weak table holds the registry of end-of-call callbacks. If either the `Callback` object or value are not reachable (see JLS section 12.6) from the Java program, both object and value will be dropped from the table. The use of a weak table to hold callbacks also means that registering a callback will not prevent the garbage collector from reclaiming that object. Therefore, you must hold on to the callback yourself if you need it—you cannot rely on the table holding it back.

You can find other ways in which end-of-call notification will be useful to your applications. The following sections give the details for methods within the `EndOfCallRegistry` class and the Callback interface:

EndOfCallRegistry.registerCallback method

The `registerCallback` method installs a `Callback` object within a registry. At the end of the call, OracleJVM invokes the `act` methods of all registered `Callback` objects.

You can register your `Callback` object by itself or with a `value` object. If you need additional information stored within an object to be passed into `act`, you can register this object within the value parameter.

```
public static void registerCallback(Callback thunk, Object value);
public static void registerCallback(Callback thunk);
```

Parameter	Description
thunk	The Callback object to be invoked at end-of-call migration.
value	If you need additional information stored within an object to be passed into `act`, you can register this object within the `value` parameter. In some cases, the `value` parameter is necessary to hold state the callback needs. However, most users do not need to specify a `value`.

EndOfCallRegistry.runCallbacks method

```
static void runCallbacks()
```

The JVM calls this method at end-of-call and calls `act` for every `Callback` object registered using `registerCallback`. You should never call this method in your code. It is called at end-of-call, before object migration and before the last finalization step.

Callback Interface

```
Interface oracle.aurora.memoryManager.Callback
```

Any object you want to register using `EndOfCallRegistry.registerCallback` implements the `Callback` interface. This interface can be useful in your application, where you require notification at end-of-call.

Callback.act method

```
public void act(Object value)
```

You can implement any activity that you require to occur at the end of the call. Normally, this method will contain procedures for clearing any memory that would be saved to session space.

Operating System Resources Affected Across Calls

In shared server mode, the OracleJVM closes any open operating system resources at the end of a database call, as shown in the following table:

Resource	Lifetime
files	The system closes all files left open when a database call ends.
threads	All threads are terminated when a call ends.
sockets	■ Sockets can exist across calls. ■ ServerSockets terminate when the call ends. See "Sockets" on page 2-44 more information.
objects that depend on operating system resources	Regardless of the usable lifetime of the object (for example, the defined lifetime for a thread object), the Java object can be valid for the duration of the session. This can occur, for example, if the Java object is stored in a static class variable, or a class variable references it directly or indirectly. If you attempt to use one of these Java objects after its usable lifetime is over, Oracle Database throws an exception. This is true for the following examples: ■ If an attempt is made to read from a `java.io.FileInputStream` that was closed at the end of a previous call, a `java.io.IOException` is thrown. ■ `java.lang.Thread.isAlive()` is false for any Thread object running in a previous call and still accessible in a subsequent call.

You should close resources that are local to a single call when the call ends. However, for static objects that hold on to operating system resources, you must be aware of how these resources are affected after the call ends.

Files

In shared server mode, the OracleJVM automatically closes any open operating system constructs—in Example 2–5, the file handle—when the call ends. This can affect any operating system resources within your Java object. If you have a file opened within a static variable, the file handle is closed at the end of the call for you. So, if you hold on to the `File` object across calls, the next usage of the file handle throws an exception.

In Example 2–5, class `Concat` enables multiple files to be written into a single file, `outFile`. On the first call, `outFile` is created. The first input file is opened, read, input into `outFile`, and the call ends. Because `outFile` is statically defined, it is moved into session space between call invocations. However, the file handle—that is, the `FileDescriptor`—is closed at the end of the call. The next time you call `addFile`, you will get an exception.

Example 2–5 Compromising Your Operating System Resources

```
public class Concat {
  static File outFile = new File("outme.txt");
  FileWriter out = new FileWriter(outFile);

public static void addFile(String[] newFile) {
 File inFile = new File(newFile);
 FileReader in = new FileReader(inFile);
 int i;

 while ((i = in.read()) != -1)
    out.write(i);
 in.close();
 }
}
```

There is a workaround: to make sure that your handles stay valid, close your files, buffers, and so on, at the end of every call; reopen the resource at the beginning of the next call. Another option is to use the database rather than using operating system resources. For example, try to use database tables rather than a file. Or do not store operating system resources within static objects expected to live across calls; use operating system resources only within objects local to the call.

Example 2–6 shows how you can perform concatenation, as in Example 2–5, without compromising your operating system resources. The `addFile` method opens the `outme.txt` file within each call, making sure that anything written into

the file is appended to the end. At the end of each call, the file is closed. Two things occur:

1. The `File` object no longer exists outside of a call.
2. The operating system resource, the `outme.txt` file, is reopened for each call. If you had made the `File` object a static variable, the closing of `outme.txt` within each call would ensure that the operating system resource is not compromised.

Example 2–6 Correctly Managing Your Operating System Resources

```
public class Concat {

public static void addFile(String[] newFile) {
 /*open the output file each call; make sure the input*/
 /*file is written out to the end by making it "append=true"*/
 FileWriter out = new FileWriter("outme.txt", TRUE);
 File inFile = new File(newFile);
 FileReader in = new FileReader(inFile);
 int i;

 while ((i = in.read()) != -1)
    out.write(i);
 in.close();
 /*close the output file between calls*/
 out.close();
 }
}
```

Sockets

Sockets are used in setting up a connection between a client and a server. For each database connection, sockets are used at either end of the connection. Your application does not set up the connection; the connection is set up by the underlying networking protocol: Oracle Net's TTC or IIOP. See "Configuring OracleJVM" on page 4-2 for information on how to configure your connection.

You might also wish to set up another connection—for example, connecting to a specified URL from within one of the classes stored within the database. To do so, instantiate sockets for servicing the client and server sides of the connection.

- The `java.net.Socket()` constructor creates a client socket.
- The `java.net.ServerSocket()` constructor creates a server socket.

A socket exists at each end of the connection. The server-side of the connection that listens for incoming calls is serviced by a `ServerSocket`. The client-side of the connection that sends requests is serviced through a `Socket`. You can use sockets as defined within the JVM with the following restriction: a `ServerSocket` instance within a shared server cannot exist across calls.

Socket Type	Description
`Socket`	Because the client-side of the connection is outbound, the `Socket` instance can be serviced across calls within a shared server.
`ServerSocket`	The server-side of the connection is a listener. The `ServerSocket` is closed at the end of a call within a shared server; the shared servers move on to another client at the end of every call. You will receive an I/O exception stating that the socket was closed if you try to use the `ServerSocket` outside of the call it was created in.

Threads

In shared server mode, when a call ends because of a return or uncaught exceptions, the OracleJVM throws a `ThreadDeathException` in all daemon threads. The `ThreadDeathException` essentially forces threads to stop execution. Code that depends on threads living across calls does not behave as expected in shared server mode. For example, the value of a static variable that tracks initialization of a thread may become incorrect in subsequent calls because all threads are killed at the end of a database call.

As a specific example, the RMI Server that Sun Microsystems supplies does function in shared server mode; however, it is useful only within the context of a single call. This is because the RMI Server forks daemon threads, which in shared server mode are killed off at the end of call (that is, when all non-daemon threads return). If the RMI server session is reentered in a subsequent call, these daemon threads aren't restarted and the RMI server won't function properly.

3

Invoking Java in the Database

This chapter gives you an overview and examples of how to invoke Java within the database.

Overview

In Oracle Database, you utilize Java in one of the following ways:

- Invoking Java Methods—Invoke Java methods in classes that are loaded within the database, such as Java stored procedures.
- Utilizing JDBC for Querying the Database—You can query the database from a Java client through utilizing JDBC.

We recommend that you approach Java development in Oracle Database incrementally, building on what you learn at each step.

1. You should master the process of writing simple Java stored procedures, as explained in "Preparing Java Class Methods for Execution" on page 2-9. This includes writing the Java class, deciding on a resolver, loading the class into the database, and publishing the class.
2. You should understand how to access and manipulate SQL data from Java. Most Java server programs, and certainly Java programs executing on Oracle Database, interact with database-resident data. The standard API for accomplishing this is JDBC.

Java is a simple, general purpose language for writing stored procedures. JDBC allow Java to access SQL data. They support SQL operations and concepts, variable bindings between Java and SQL types, and classes that map Java classes to SQL types. You can write portable Java code that can execute on a client or a server without change. With JDBC, the dividing line between client and server is usually obvious—SQL operations happen in the server, and application program logic resides in the client.

As you write more complex Java programs, you can gain performance and scalability by controlling the location where the program logic executes. You can minimize network traffic and maximize locality of reference to SQL data. JDBC furnishes ways to accomplish these goals. However, as you tend to leverage the object model in your Java application, a more significant portion of time is spent in Java execution, as opposed to SQL data access and manipulation. It becomes more important to understand and specify where Java objects reside and execute in an Internet application.

Invoking Java Methods

The way your client calls a Java method depends on the type of Java application. The following sections discuss each of the Java APIs available for creating a Java class that can be loaded into the database and accessed by your client:

Utilizing Java Stored Procedures

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

To invoke a Java stored procedure, you must publish it through a call specification. The following example shows how to create, resolve, load, and publish a simple Java stored procedure that echoes "Hello world".

1. Write the Java class.

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

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

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

   ```
   javac Hello.java
   ```

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

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

   - as an individual file in a directory, where the directory is specified in the CLASSPATH

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

3. Decide on the resolver for your class.

   In this case, you load `Hello.class` in the server, where it is stored in the database as a Java schema object. When you execute the `world()` method of the `Hello.class` on the server, it finds the necessary supporting classes, such as `String`, using a resolver—in this case, the default resolver. The default resolver looks for classes in the current schema first and then in PUBLIC. All core class libraries, including the `java.lang` package, are found in PUBLIC. You may need to specify different resolvers, and you can force resolution to occur when you use `loadjava`, to determine if there are any problems earlier, rather than at runtime. Refer to "Resolving Class Dependencies" on page 2-13 or Chapter 11, "Schema Object Tools" for more details on resolvers and `loadjava`.

4. Load the class on the Oracle Database server using `loadjava`. You must specify the user name and password.

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

5. Publish the stored procedure through a call specification.

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

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

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

6. Invoke the stored procedure.

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

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

```
SQL>
```

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

For more information on Java stored procedures, using Java in triggers, call specifications, rights models, and inter-language calls, see Chapter 5, "Developing Java Stored Procedures".

Utilizing Java Native Interface (JNI) Support

The Java Native Interface (JNI) is a standard programming interface for writing Java native methods and embedding the Java virtual machine into native applications. The primary goal of JNI is to provide binary compatibility of Java applications that use platform-specific native libraries.

Oracle does not support the use of JNI in Oracle Database Java applications. If you use JNI, your application is not 100% pure Java, and the native methods require porting between platforms. Native methods have the potential for crashing the server, violating security, and corrupting data.

Utilizing JDBC for Querying the Database

You can use JDBC protocols for querying the database from a Java client. This establishes a session with a given user name and password to the database and executes SQL queries against the database.

Protocol	Description
JDBC	Use this protocol for more complex or dynamic SQL queries. JDBC requires you to establish the session, construct the query, and so on.

JDBC

JDBC is an industry-standard API developed by Sun Microsystems that allows you to embed SQL statements as Java method arguments. JDBC is based on the X/Open

SQL Call Level Interface and complies with the SQL92 Entry Level standard. Each vendor, such as Oracle, creates its JDBC implementation by implementing the interfaces of the Sun Microsystems `java.sql` package. Oracle offers three JDBC drivers that implement these standard interfaces:

1. The JDBC Thin driver, a 100% pure Java solution you can use for either client-side applications or applets and requires no Oracle client installation.
2. The JDBC OCI drivers, which you use for client-side applications and requires an Oracle client installation.
3. The server-side JDBC driver embedded in the Oracle Database server.

For the developer, using JDBC is a step-by-step process of creating a statement object of some type for your desired SQL operation, assigning any local variables that you want to bind to the SQL operation, and then executing the operation. This process is sufficient for many applications but becomes cumbersome for any complicated statements. Dynamic SQL operations, where the operations are not known until runtime, require JDBC. In typical applications, however, this represents a minority of the SQL operations.

An Example

The following is an example of a simple operation in JDBC code.

JDBC:

```
// (Presume you already have a JDBC Connection object conn)
// Define Java variables
String name;
int id=37115;
float salary=20000;

// Set up JDBC prepared statement.
PreparedStatement pstmt = conn.prepareStatement
   ("select ename from emp where empno=? and sal>?");
pstmt.setInt(1, id);
pstmt.setFloat(2, salary);

// Execute query; retrieve name and assign it to Java variable.
ResultSet rs = pstmt.executeQuery();
while (rs.next()) {
   name=rs.getString(1);
   System.out.println("Name is: " + name);
}
```

```
// Close result set and statement objects.
rs.close()
pstmt.close();
```

1. Define the Java variables `name`, `id`, and `salary`.
2. Define a prepared statement (this presumes you have already established a connection to the database so that you can use the `prepareStatement()` method of the connection object).

   You can use a prepared statement whenever values within the SQL statement must be dynamically set. You can use the same prepared statement repeatedly with different variable values. The question marks in the prepared statement are placeholders for Java variables and are given values in the `pstmt.setInt()` and `pstmt.setFloat()` lines of code. The first "`?`" is set to the `int` variable `id` (with a value of 37115). The second "`?`" is set to the `float` variable `salary` (with a value of 20000).
3. Execute the query and return the data into a JDBC result set object. (You can use result sets to gather query data.)
4. Retrieve the data of interest (the name) from the result set and print it. A result set usually contains multiple rows of data, although this example has only one row.

Debugging Server Applications

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

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

How To Tell You Are Executing in the Server

You might want to write Java code that executes in a certain way in the server and another way on the client. In general, Oracle does not recommend this. In fact, JDBC goes to some trouble to enable you to write portable code that avoids this problem, even though the drivers used in the server and client are different.

If you must determine whether your code is executing in the server, use the `System.getProperty` method, as follows:

```
System.getProperty ("oracle.jserver.version")
```

The `getProperty` method returns the following:

- If executing in the server, it returns a `String` that represents the Oracle Database release.
- If executing on the client, it returns null.

Redirecting Output on the Server

`System.out` and `System.err` print to the current trace files. To redirect output to the SQL*Plus text buffer, use this 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 prints at the end of the call.

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

Support for Calling Java Stored Procedures Directly

Oracle Database 10*g* introduces new convenience features for calling Java stored procedures and functions.

In previous releases, calling Java stored procedures and functions from a database client required JDBC calls to associated PL/SQL wrappers. Each wrapper had to be

manually published with a SQL signature and a Java implementation. This had the following disadvantages:

- A separate step was required for publishing the SQL signatures for Java methods.
- The signatures permitted only Java types with SQL equivalents.
- Exceptions issued in Java were not properly returned.
- Only a single method invocation could be performed for each database round trip.

To remedy these deficiencies, a simple API has been implemented for direct invocation of static Java stored procedures and functions. The new functionality is useful for Web services, but is more generally useful as well.

Classes for the simple API are located in the package `oracle.jpub.reflect`, so you must import this into the client-side code.

Here is the Java interface for the API:

```
public class Client
{
   public static String getSignature(Class[]);
   public static Object invoke(Connection, String, String,
                               String, Object[]);
   public static Object invoke(Connection, String, String,
                               Class[], Object[]);
}
```

As an example, consider a call to the following method in the server:

```
public String oracle.sqlj.checker.JdbcVersion.to_string();
```

You can now accomplish this as follows:

```
Connection conn = ...;
String serverSqljVersion = (String)
           Client.invoke(conn, "oracle.sqlj.checker.JdbcVersion",
           "to_string", new Class[]{}, new Object[]{});
```

The `Class[]` array is for the method parameter types and the `Object[]` array is for the parameter values. In this case, because `to_string` has no parameters, the arrays are empty.

Note the following:

- Any serializable type (such as `int[]` and `String[]`, for example) can be passed as an argument.
- As an optimization, parameter values can be represented in a string:

```
String sig = oracle.jpub.reflect.Client.getSignature(new Class[]{});
...
Client.invoke(conn, "oracle.sqlj.checker.JdbcVersion", "to_string",
             sig, new Object[]{});
```

(This is offered as a general note; in this example it is a moot point because `to_string` has no parameters.)

- The semantics of this API are different than semantics for invoking stored procedures or functions through a PL/SQL wrapper, in the following ways:
  - Arguments cannot be `OUT` or `IN OUT`. Returned values must all be part of the function result.
  - Exceptions are properly returned.
  - The method invocation uses invoker's rights. (There is no tuning to obtain definer's rights.)

4

Java Installation and Configuration

This chapter describes what you need to know to install and configure OracleJVM within your database. To configure Java memory, see the "Java Memory Usage" section in Chapter 10, "Oracle Database Java Application Performance".

- Initializing a Java-Enabled Database
- Configuring OracleJVM
- Using The DBMS_JAVA Package
- Enabling the Java Client

Initializing a Java-Enabled Database

If you install Oracle Database with the OracleJVM option, the database is Java-enabled. That is, it is ready to run Java stored procedures and JDBC.

Oracle Database Template Configuration and Install

Configure the OracleJVM option within the database template. This is the recommended method for Java installation.

The Database Configuration Assistant allows you to create database templates for defining what each database instance installation will contain. Choose the OracleJVM option to have the Java platform installed within your database. See the Database Configuration Assistant documentation for more information on template creation.

Modifying an Existing Oracle Database to Include OracleJVM

If you have already installed your Oracle Database without OracleJVM, you can add Java to your database through the modify mode of the Oracle Database 10*g* Configuration Assistant. The modify mode enables you to choose the features, such as OracleJVM, that you would like installed on top of an existing Oracle Database.

Configuring OracleJVM

When you install OracleJVM as part of your normal Oracle Database installation, you will encounter configuration requirements for OracleJVM within the Oracle Database 10*g* Configuration Assistant and the Oracle Net Assistant.

The main configuration for Java classes within Oracle Database includes configuring Java memory requirements and the type of database processes.

- Java memory requirements—You must have at least 20 MB of `JAVA_POOL_SIZE` and 50 MB of `SHARED_POOL_SIZE`. See "Java Memory Usage" on page 10-18 for information on configuring these parameters.
- Database processes—You must decide whether to use dedicated server processes or shared server processes for your database server.

Using The DBMS_JAVA Package

Installing OracleJVM creates the PL/SQL package `DBMS_JAVA`. Some entry points of `DBMS_JAVA` are for your use; others are only for internal use. The corresponding Java class `DbmsJava` provides methods for accessing RDBMS functionality from Java.

See "DBMS_JAVA Package" on page A-1 for complete information.

Enabling the Java Client

To run Java between the client and server, your client system must perform the following:

1. Install J2SE on the Client.
2. Set up Environment Variables.
3. Test Install with Samples.

1. Install J2SE on the Client

The client requires JDK 1.2.1 or later. To confirm what version of the JDK you are using, perform the following:

```
$ which java
/usr/local/j2se1.4.1/bin/java
$ which javac
/usr/local/j2se1.4.1/bin/javac
$ java -version
java version "1.4.1"
```

2. Set up Environment Variables

After installing the JDK on your client, you must add the directory path to the following environment variables:

> **Note:** For NT users, the syntax for the environment variables is %ORACLE_HOME%, %JAVA_HOME%, %PATH%, and %LIB%.

- `$JAVA_HOME`—must be set to the top directory of the installed JDK base
- `$PATH`—requires `$JAVA_HOME/bin`

- $LD_LIBRARY_PATH for Solaris or %LIB% for Windows NT—must include $JAVA_HOME/lib

JAR Files Necessary for Java 2 Clients

For a Java 2 client to communicate with the Java 2 server, you must make sure that one of the following JVM JAR files are in the CLASSPATH:

- For JDK 1.2, include $JAVA_HOME/lib/dt.jar
- For JRE 1.2, include $JAVA_HOME/lib/rt.jar

For any interaction with JDBC, include the following ZIP file:

```
$ORACLE_HOME/jdbc/lib/classes12.zip
```

For any client that uses SSL, include the following JAR files:

```
$ORACLE_HOME/jlib/jssl-1_2.jar
$ORACLE_HOME/jlib/javax-ssl-1_2.jar
```

For any client that uses Java Transaction API (JTA) functionality, include the following JAR file:

```
$ORACLE_HOME/jlib/jta.jar
```

For any client that uses JNDI functionality, include the following JAR file:

```
$ORACLE_HOME/jlib/jndi.jar
```

If you are using the Accelerator for native compilation, include $JAVA_HOME/lib/tools.jar

Server Application Development on the Client

If you develop and compile your server applications on the client and you want to use the same JAR files that are loaded on the server, include $ORACLE_HOME/lib/aurora.zip in the CLASSPATH. This is not required for running Java clients.

3. Test Install with Samples

We provide a set of samples in the $ORACLE_HOME/javavm/demo directory. These samples compile and run for a database installed with the OracleJVM option. Execute these samples as a test of your installation.

```
$ORACLE_HOME/javavm/demo/examples/jsproc/helloworld
```

If these samples do not compile or run, your environment is incorrect. Similarly, if these samples compile and run, but your code does not, then a problem exists within your build environment or code.

> **Note:** It is important that you run these examples using the supplied Makefiles (or batch files on NT) when verifying your installation.

Verify that the samples work before using more complex build environments, such as Visual Cafe, JDeveloper, or VisualAge.

5

Developing Java Stored Procedures

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

This chapter contains the following information:

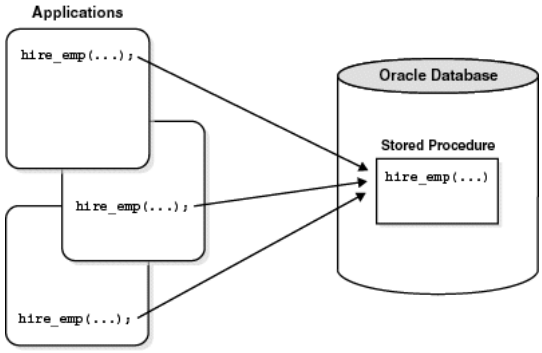
Stored Procedures and Run-Time Contexts

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 5–1 shows a stored procedure being called by various applications.

Figure 5–1 Calling a Stored Procedure



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

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

The next three sections describe these contexts.

Functions and Procedures

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

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

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

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

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

Database Triggers

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

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

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

Object-Relational Methods

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

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

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

Advantages of Stored Procedures

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

Performance

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

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

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

Productivity and Ease of Use

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

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

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

Scalability

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

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

Maintainability

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

Interoperability

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

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

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

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

Replication

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

Security

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

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

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

For a full discussion of OracleJVM security, see Chapter 9, "Security For Oracle Database Java Applications".

Java Stored Procedure Configuration

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

- Dedicated server mode—You must configure the database and clients in dedicated server mode using Oracle Net Services connections.
- shared server mode—You must configure the server for shared server mode with the DISPATCHERS parameter, as the *Oracle Net Services Administrator's Guide* explains.

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

Java Stored Procedures Steps

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

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

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

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

This section demonstrates how to develop a simple Java stored procedure. For more examples of a Java stored procedures application, see Chapter 8, "Java Stored Procedures Application Example".

Step 1: Create or Reuse the Java Classes

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

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

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

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

```
javac Oscar.java
```

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

Step 2: Load and Resolve the Java Classes

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

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

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

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

For more information, see Chapter 2, "Java Applications on Oracle Database".

Step 3: Publish the Java Classes

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

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

```
SQL> connect scott/tiger

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

For more information, see Chapter 6, "Publishing Java Classes With Call Specs".

Step 4: Call the Stored Procedures

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

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

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

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

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

SQL> PRINT theQuote;

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

For more information, see Chapter 7, "Calling Stored Procedures".

Step 5: If Necessary, Debug the Stored Procedures

Your Java stored procedures execute remotely on a server, which typically resides on a separate machine. However, the JDK debugger (`jdb`) cannot debug remote Java programs. For more information, see "Debugging Server Applications" on page 3-7.

Another Example

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

1. Write the Java class.

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

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

2. Compile the class on your client system. Using the Sun Microsystems J2SE, for example, invoke the Java compiler, `javac`, as follows:

   ```
   javac Hello.java
   ```

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

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

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

3. Decide on the resolver for your class.

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

occur when you use `loadjava`, to determine if there are any problems earlier, rather than at runtime. Refer to Chapter 2, "Java Applications on Oracle Database" for more details on resolvers and `loadjava`.

4. Load the class on the Oracle Application Server using `loadjava`. You must specify the user name and password.

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

5. Publish the stored procedure through a call specification.

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

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

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

6. Invoke the stored procedure.

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

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

SQL>
```

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

6

Publishing Java Classes With Call Specs

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

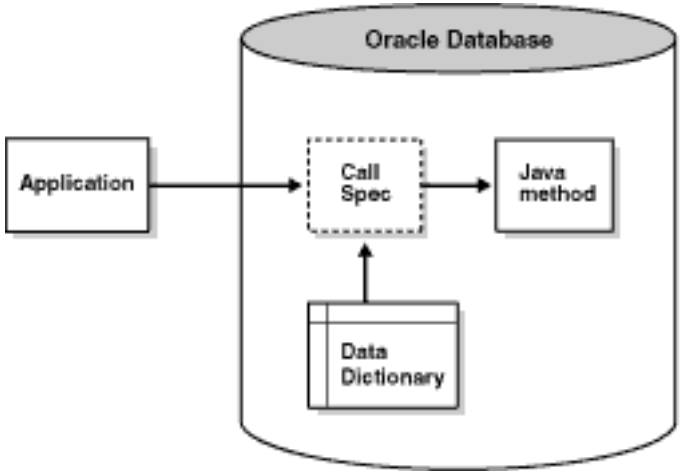
Understanding Call Specs

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

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

As Figure 6–1 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 6–1 Calling a Java Method



As an alternative, use the native Java interface to directly invoke Java in the database from a Java client. See "Native Java Interface" for more information.

Defining Call Specs: Basic Requirements

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

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

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

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

Setting Parameter Modes

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

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

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

Mapping Datatypes

In a call spec, corresponding SQL and Java parameters (and function results) must have compatible datatypes. Table 6–1 contains all the legal datatype mappings. Oracle converts between the SQL types and Java classes automatically.

Table 6–1 Legal Datatype Mappings

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

Table 6–1 Legal Datatype Mappings (Cont.)

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

Notes:

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

- The mappings to `oracle.sql` classes are optimal because they preserve data formats and require no character-set conversions (apart from the usual network conversions). Those classes are especially useful in applications that "shovel" data between SQL and Java.

Using the Server-Side Internal JDBC Driver

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

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

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

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

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

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

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

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

```
String sql = "SELECT COUNT(*) FROM " + tabName;
int rows = 0;
Statement stmt = conn.createStatement();
ResultSet rset = stmt.executeQuery(sql);
while (rset.next()) {rows = rset.getInt(1);}
```

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

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

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

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

Important Points

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

- The server-side internal JDBC driver runs within a default session and default transaction context. So, you are already "connected" to the database, and all your SQL operations are part of the default transaction. Note that this transaction is a local transaction and not part of a global transaction, such as implemented by JTA or JTS.
- Statements and result sets persist across calls, and their finalizers do not release database cursors. So, to avoid running out of cursors, close all statements and result sets when you are done with them. Alternatively, you can ask your DBA to raise the limit set by the Oracle initialization parameter `OPEN_CURSORS`.
- The server-side internal JDBC driver does not support auto-commits. So, your application must explicitly commit or roll back database changes.
- You cannot connect to a remote database using the server-side internal JDBC driver. You can "connect" only to the server running your Java program. For

server-to-server connections, use the server-side JDBC Thin driver. (For client/server connections, use the client-side JDBC Thin or JDBC OCI driver.)

- You cannot close the physical connection to the database established by the server-side internal JDBC driver. However, if you call method `close()` on the default connection, all connection instances (which, in fact, reference the same object) are cleaned up and closed. To get a new connection object, you must call method `getConnection()` again.

For more information, see the *Oracle Database JDBC 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 *Oracle Database SQL Reference.*

The NAME-clause string uniquely identifies the Java method. The Java full names and the call spec parameters, which are mapped by position, must correspond one to one. (This rule does not apply to method main. See Example 2 on page 6-10.) If the Java method takes no arguments, code an empty parameter list for it but *not* for the function or procedure.

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

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

Example 1

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

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

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

Class GenericDrop has one method named dropIt, which drops any kind of schema object. For example, if you pass the arguments 'table' and 'emp' to

dropIt, the method drops database table emp from your schema. Let's write a call spec for this method.

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

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

Example 2

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

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

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

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

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

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

Example 3

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

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

public class RowCounter {
  public static int rowCount (String tabName) throws SQLException {
    Connection conn =
      DriverManager.getConnection("jdbc:default:connection:");
    String sql = "SELECT COUNT(*) FROM " + tabName;
    int rows = 0;
    try {
      Statement stmt = conn.createStatement();
      ResultSet rset = stmt.executeQuery(sql);
      while (rset.next()) {rows = rset.getInt(1);}
      rset.close();
      stmt.close();
    } catch (SQLException e) {System.err.println(e.getMessage());}
    return rows;
  }
}
```

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

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

Example 4

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

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

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

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

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

Writing Packaged Call Specs

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

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

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

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

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

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

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

An Example

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

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

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

  public static void dropDept (int deptID) throws SQLException {
    Connection conn =
```

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

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

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

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

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

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

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

  PROCEDURE change_loc (dept_id NUMBER, new_loc VARCHAR2)
```

```
  AS LANGUAGE JAVA
  NAME 'DeptManager.changeLoc(int, java.lang.String)';
END dept_mgmt;
```

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

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

Writing Object Type Call Specs

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

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

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

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

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

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

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

Declaring Attributes

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

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

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

Declaring Methods

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

```
instance_expression.method()
```

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

```
object_type_name.method()
```

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

Map and Order Methods

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

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

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

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

Constructor Methods

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

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

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

Examples

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

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

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

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

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

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

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

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

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

Using Class oracle.sql.STRUCT

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

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

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

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

```
CREATE OR REPLACE FUNCTION wages (e Employee) RETURN NUMBER AS
  LANGUAGE JAVA
```

```
  NAME 'Paymaster.wages(oracle.sql.STRUCT) return BigDecimal';
```

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

Implementing the SQLData Interface

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

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

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

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

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

  // Implement SQLData interface.
```

```
  private String sql_type;

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

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

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

You must revise the call spec for method `wages`, as follows, because its parameter has changed from `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`. In an object type spec, all methods must be declared after the attributes. The object type body is unnecessary because the spec declares only attributes and call specs.

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

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

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

public class Paymaster implements SQLData {
  // Implement the attributes and operations for this type.
  private BigDecimal empno;
  private String ename;
  private String job;
```

```
private BigDecimal mgr;
private Date hiredate;
private BigDecimal sal;
private BigDecimal comm;
private Ref dept;

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

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

// Implement SQLData interface.

String sql_type;

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

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

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

7

Calling Stored Procedures

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

Calling Java from the Top Level

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

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

where param stands for the following syntax:

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

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

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

Redirecting Output

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

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

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

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

Output is printed when the stored procedure exits.

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

Example 1

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

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

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

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

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

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

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

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

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

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

Call completed.

SQL> SELECT ename, sal FROM emp;

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

7 rows selected.
```

Example 2

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

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

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

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

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

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

PL/SQL procedure successfully completed.
```

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

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

Call completed.

SQL> PRINT f

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

Calling Java from Database Triggers

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

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

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

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

Example 1

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

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

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

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

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

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

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

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

```
CREATE OR REPLACE TRIGGER sal_trig
AFTER UPDATE OF sal ON emp
FOR EACH ROW
```

```
WHEN (new.sal > 1.2 * old.sal)
CALL log_sal(:new.empno, :old.sal, :new.sal);
```

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

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

SQL> SELECT * FROM sal_audit;

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

6 rows selected.
```

Example 2

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

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

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

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

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

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

```
public static void addEmp (
  int empNo, String empName, String deptName)
```

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

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

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

Then, you create the `INSTEAD OF` trigger:

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

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

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

SQL> SELECT * FROM sales;

     EMPNO ENAME
---------- ----------
      8001 Chand
      8002 Van Horn
```

```
      8003 Waters

SQL> SELECT * FROM mktg;

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

SQL> SELECT * FROM emps;

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

Calling Java from SQL DML

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

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

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

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

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

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

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

Restrictions

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

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

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

Calling Java from PL/SQL

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

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

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

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

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

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

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

In the next example, you call the function `row_count` (defined in Example 3 on page 6-11) from a standalone PL/SQL stored procedure:

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

In the final example, you call the `raise_sal` method of object type `Employee` (defined in "Implementing Object Type Methods" on page 6-22) from an anonymous PL/SQL block:

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

Calling PL/SQL from Java

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

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

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

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

```
cstmt.executeUpdate();
float acctBal = cstmt.getFloat(1);
```

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

How OracleJVM Handles Exceptions

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

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

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

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

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

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

8

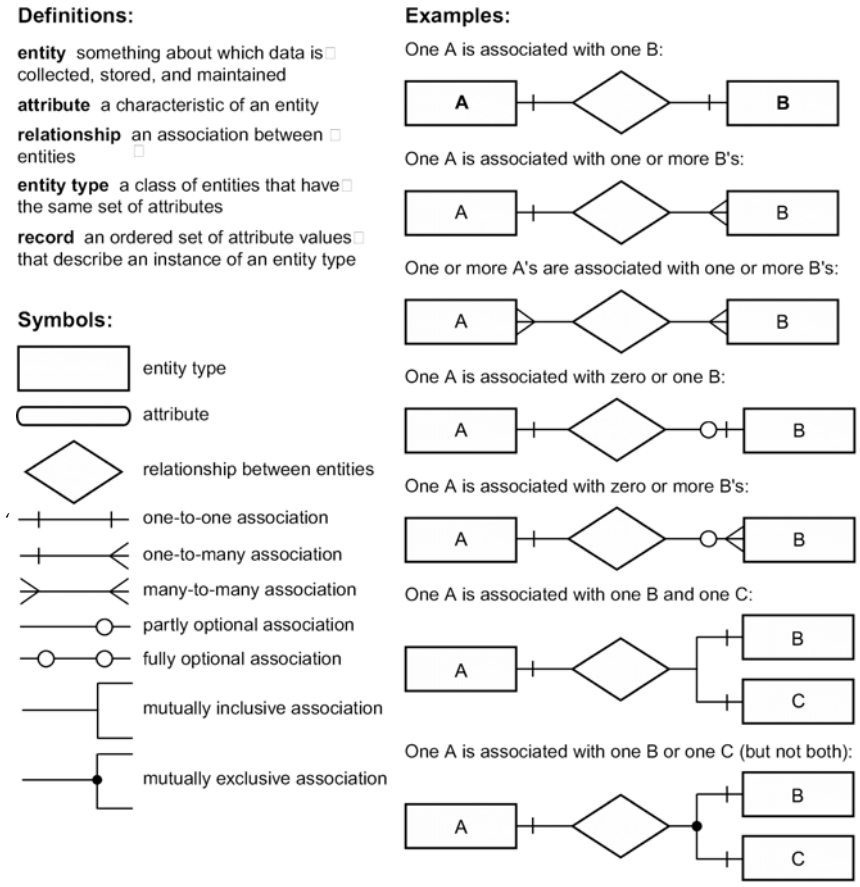
Java Stored Procedures Application Example

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

Drawing the Entity-Relationship Diagram

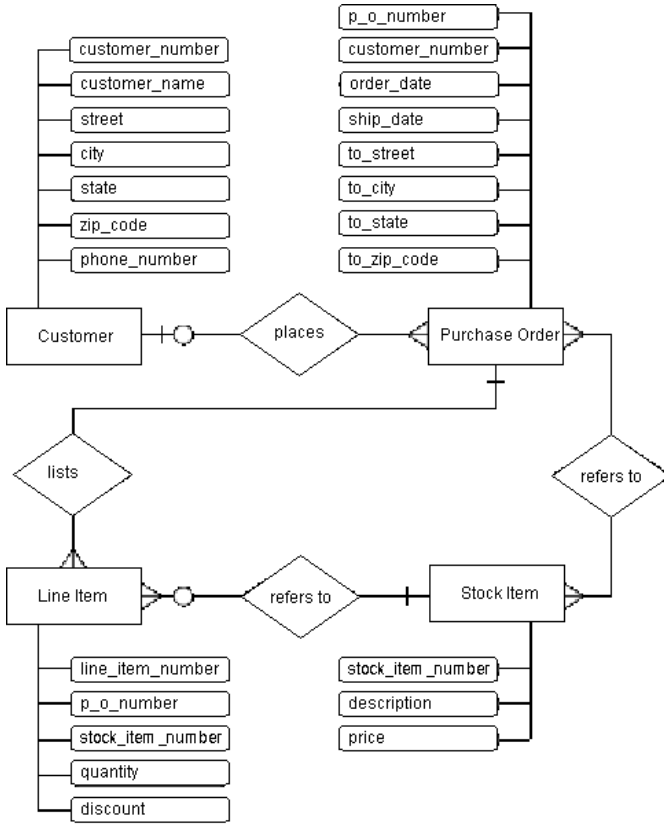
The objective is to develop a simple system for managing customer purchase orders. First, you must identify the business entities involved and their relationships. To do that, you draw an entity-relationship (E-R) diagram by following the rules and examples given in Figure 8–1.

Figure 8–1 Rules for Drawing an E-R Diagram



As Figure 8–2 illustrates, the basic entities in this example are customers, purchase orders, line items, and stock items.

Figure 8–2 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:

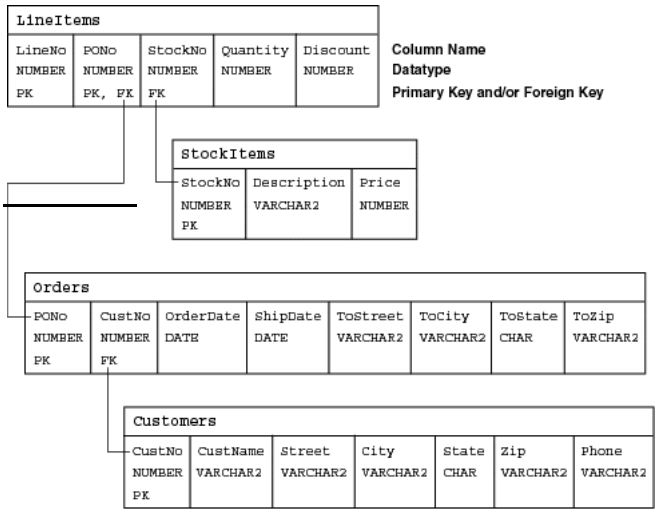
- `Customers`
- `Orders`
- `LineItems`
- `StockItems`

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

Figure 8–3 on page 8-5 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 8–3 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 8–2 showed that line items have a relationship with purchase orders and stock items. The table `LineItems` manages these relationships using foreign keys. For example, the foreign key (FK) column `StockNo` in the table `LineItems` references the primary key (PK) column `StockNo` in the table `StockItems`, which you define as:

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

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

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

Writing the Java Classes

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

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

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

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

  public static void enterOrder (int orderNo, int custNo,
    String orderDate, String shipDate, String toStreet,
    String toCity, String toState, String toZipCode)
    throws SQLException {
      String sql = "INSERT INTO Orders VALUES (?,?,?,?,?,?,?,?)";
```

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

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

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

```
    printResults(rset);
    rset.close();
    pstmt.close();
  } catch (SQLException e) {System.err.println(e.getMessage());}
}

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

public static void checkStockItem (int stockNo)
  throws SQLException {
    String sql = "SELECT O.PONo, O.CustNo, L.StockNo, " +
      "L.LineNo, L.Quantity, L.Discount " +
      "FROM Orders O, LineItems L " +
      "WHERE O.PONo = L.PONo AND L.StockNo = ?";
    try {
      Connection conn =
        DriverManager.getConnection("jdbc:default:connection:");
```

```
        PreparedStatement pstmt = conn.prepareStatement(sql);
        pstmt.setInt(1, stockNo);
        ResultSet rset = pstmt.executeQuery();
        printResults(rset);
        rset.close();
        pstmt.close();
      } catch (SQLException e) {System.err.println(e.getMessage());}
  }

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

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

Loading the Java Classes

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

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

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

Publishing the Java Classes

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

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

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

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

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

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

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

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

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

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

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

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

Calling the Java Stored Procedures

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

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

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

Then, you register your customers:

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

Next, you enter purchase orders placed by various customers:

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

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

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

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

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

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

Call completed.
```

9

Security For Oracle Database Java Applications

Security is a large arena that includes network security for the connection, access and execution control of operating system resources or of JVM-defined and user-defined classes, and bytecode verification of imported JAR files from an external source. The following sections describe the various security support available for Java applications within Oracle Database.

Network Connection Security

The two major aspects to network security are authentication and data confidentiality. The type of authentication and data confidentiality is dependent on how you connect to the database—through Oracle Net or JDBC connection.

Connection Security	Description
Oracle Net	The database can require both authentication and authorization before allowing a user to connect to it. Oracle Net database connection security can require one or more of the following: ■ Use a username and password for client verification. Each incoming connection into the database has to provide the correct username/password configured within Oracle Net. For more information, see the *Oracle Net Services Administrator's Guide.* ■ Use Advanced Networking Option for encryption, kerberos, or secureId. See the *Oracle Advanced Security Administrator's Guide*. ■ Use SSL for certificate authentication. See the *Oracle Advanced Security Administrator's Guide.*
JDBC	The JDBC connection security that is required is similar to the constraints required on an Oracle Net database connection. In addition to the books listed in the Oracle Net database connection section, see the *Oracle Database JDBC Developer's Guide and Reference.*

Database Contents and OracleJVM Security

Once you are connected to the database, you still must have the correct Java 2 Permissions and database privileges to access the resources stored within the database. These resources include the following:

- database resources, such as tables and PL/SQL packages
- operating system resources, such as files and sockets
- OracleJVM classes
- user-loaded classes

These resources can be protected by the following two methods:

Resource Security	Description
Database Resource Security	Authorization for database resources requires that database privileges (not the same as the Java 2 security permissions) are granted to resources. For example, database resources include tables, classes, and PL/SQL packages. For more information, see the *Oracle Database Application Developer's Guide - Fundamentals*.
	All user-defined classes are secured against users from other schemas. You can grant execution permission to other users/schemas through an option on the `loadjava` command. For more information on setting execution rights when loading classes, see the `-grant` option discussed in "Loading Classes" on page 2-17 or Chapter 11, "Schema Object Tools" for complete information on loadjava.
JVM Security	OracleJVM uses Java 2 security, which uses Permission objects to protect operating system resources. Java 2 security is automatically installed upon startup and protects all operating system resources and OracleJVM classes from all users, except `JAVA_ADMIN`. `JAVA_ADMIN` can grant permission to other users to access these classes.
	See "Java 2 Security" on page 9-3 for how to manage and modify Java 2 Permissions and policies.

Java 2 Security

Each user or schema must be assigned the proper permissions to access operating system resources. For example, this includes sockets, files, and system properties.

Java 2 security was created to provide a flexible, configurable security for Java applications. With Java 2 security, you can define exactly what permissions on each loaded object that a schema or role will have. In release 8.1.5, the security provided you the choice of two secure roles:

- `JAVAUSERPRIV`—few Permissions, including examining properties
- `JAVASYSPRIV`—major Permissions, including updating OracleJVM protected packages

> **Note:** Both roles still exist within this release for backward compatibility; however, Oracle recommends that you specify each Permission explicitly, rather than utilize these roles.

Because OracleJVM security is based on Java 2 security, you assign Permissions on a class by class basis. Permissions contains two string attributes:

- target (name) attribute
- action attribute

These permissions are assigned through database management tools. Each permission is encapsulated in a Permission object and is stored within a Permission table. The methods for managing all permissions are the subject for most of this chapter.

Java security was created for the non-database world. When you apply the Java 2 security model within the database, certain differences manifest themselves. For example, Java 2 security defines that all applets are implicitly untrusted, and all classes within the CLASSPATH are trusted. In Oracle Database, all classes are loaded within a secure database; thus, no classes are trusted.

The following table briefly describes the differences between the Sun Microsystems Java 2 security and the Oracle Database security implementation. This table assumes that you already understand the Sun Microsystems Java 2 security model. For more information, we recommend the following books:

- *Inside Java 2 Platform Security* by Li Gong
- *Java Security* by Scott Oaks

Java 2 Security Standard	Oracle Database Security Implementation
Java classes located within the CLASSPATH are trusted.	All Java classes are loaded within the database. Classes are trusted on a class by class basis according to the Permission granted.
You can specify the policy through the `-usepolicy` flag on the `java` command line.	You must specify the policy within the `PolicyTable`.
You can write your own `SecurityManager` or use the Launcher.	You can write your own `SecurityManager`; Oracle recommends that you use only the Oracle Database `SecurityManager` or that you extend the Oracle Database `SecurityManager`. If you want to modify the behavior, you should not define a `SecurityManager`; instead, you should extend `oracle.aurora.rdbms.SecurityManagerImpl` and override specific methods.

Java 2 Security Standard	Oracle Database Security Implementation
`SecurityManager` is not initialized for you. You must initialize the `SecurityManager`.	The OracleJVM always initializes `SecurityManager` at startup.
Permissions are determined by the location where the application or applet is loaded (the URL) or keycode (signed code).	Permissions are determined by the schema in which the class is loaded. Oracle Database does not support signed code.
The security policy is defined in a file.	The PolicyTable definition is contained within a secure database table.
You can update the security policy file through a text editor (if you have the correct Permissions) or through a tool.	You can update the PolicyTable through DBMS_JAVA procedures. After initialization, only JAVA_ADMIN has permission to modify the PolicyTable. JAVA_ADMIN must grant you the right to modify the PolicyTable for you to grant Permissions to others.
Permissions are assigned to a protection domain, which classes can belong to.	All classes within the same schema are within the same protection domain.
You can use the `CodeSource` class for identifying code. ■ The `equals` method returns true if the URL and certificates are equal. ■ The `implies` method returns true if the first CodeSource is a generic representation that includes the specific `CodeSource` object.	You can use the `CodeSource` class for identifying schema. ■ The `equals` method returns true if the schemas are the same. ■ The `implies` method returns true if the schemas are the same.
Supports positive Permissions only (grant).	Supports both positive (grant) and limitation (restrict) Permissions.

Setting Permissions

As with Java 2 security, Oracle Database supports the security classes. Normally, you set the Permissions for the code base either through a tool or by editing the security policy file. In Oracle Database, you set the Permissions dynamically through DBMS_JAVA procedures. These procedures modify a policy table, which is a new table within the database that exclusively manages Java 2 security Permissions.

Two views have been created for you to view the policy table: USER_JAVA_POLICY and DBA_JAVA_POLICY. Both views contain information about granted and limitation Permissions. The DBA_JAVA_POLICY view can see all rows within the policy table; the USER_JAVA_POLICY table can see only Permissions relevant to the current user. The following is a description of the rows within each view:

Table Column	Description
Kind	GRANT or RESTRICT. Shows whether this Permission is a positive (GRANT) or a limitation (RESTRICT) Permission.
Grantee	The name of the user, schema, or role to which the Permission object is assigned.
Permission_schema	The schema in which the Permission object is loaded.
Permission_type	The Permission class type, which is designated by a string containing the full class name, such as, `java.io.FilePermission`.
Permission_name	The target attribute (name) of the Permission object. You use this name when defining the Permission. When defining the target for a Permission of type PolicyTablePermission, the name can become quite complicated. See "Acquiring Administrative Permission to Update Policy Table" on page 9-12 for more information.
Permission_action	The action attribute for this Permission. Many Permissions expect a null value if no action is appropriate for the Permission.
Status	ACTIVE or INACTIVE. After creating a row for a Permission, you can disable or re-enable it. This column shows the status of whether the Permission is enabled (ACTIVE) or disabled (INACTIVE).
Key	Sequence number you use to identify this row. This number should be supplied when disabling, enabling, or deleting the Permission.

There are two ways to set your Permissions:

- Fine-Grain Definition for Each Permission—You grant each Permission individually for specific users or roles. If you do not grant a Permission for access, the schema will be denied access.
- General Permission Definition Assigned to Roles—If you do not want to grant specific Permissions for each user, you can grant roles, which grants a collection of Permissions to the user. Oracle Database supplies the roles: JAVAUSERPRIV or JAVASYSPRIV.

Note: For absolute certainty about your security, implement the fine-grain definition. The general definition is easier; but you might not get the exact security you require.

Fine-Grain Definition for Each Permission

To set individual Permissions within the policy table, you must provide the following information:

Parameter	Description
Grantee	The name of the user, schema, or role to which you want the grant to apply. PUBLIC specifies that the row applies to all users.
Permission type	The Permission class on which you are granting Permission. For example, if you were defining access to a file, the Permission type would be `FilePermission`. This parameter requires a fully-qualified name of a class that extends `java.lang.security.Permission`. If the class is not within SYS, the name should be prefixed by <*schema*>:. For example, `mySchema:myPackage.MyPermission` is a valid name for a user-generated Permission.
Permission name	The meaning of the target attribute is defined by the Permission class. Examine the appropriate Permission class for the relevant name.
Permission action	The type of action that you can specify varies according to the Permission type. For example, `FilePermission` can have the action of read or write.
Key	Number returned from grant or limit to use on enable, disable, or delete methods.

You can either grant Java 2 Permissions or create your own. The Java 2 Permissions are listed in Table 9–1. If you would like to create your own Permissions, see "Creating Permissions" on page 9-14.

Table 9–1 Permission Types

- `java.util.PropertyPermission`
- `java.io.SerializablePermission`
- `java.io.FilePermission`
- `java.net.NetPermission`

Table 9–1 Permission Types

■ `java.net.SocketPermission`
■ `java.lang.RuntimePermission`
■ `java.lang.reflect.ReflectPermission`
■ `java.security.SecurityPermission`
■ `oracle.aurora.rdbms.security.PolicyTablePermission`
■ `oracle.aurora.security.JServerPermission`

You can grant permissions using either SQL or Java, as shown below. However, each returns a row key identifier that identifies the row within the permission table. In the Java version of `DBMS_JAVA`, each method returns the row key identifier, either as a returned parameter or as an OUT variable in the parameter list. In the PL/SQL `DBMS_JAVA` package, the row key is returned only in the procedure that defines the `key OUT` parameter. This key is used to enable and disable specific Permissions. See "Enabling or Disabling Permissions" on page 9-18 for more information.

If, after executing the grant, a row already exists for the exact Permission, no update occurs, but the key for that row is returned. If the row was disabled, executing the grant enables the existing row.

> **Note:** If granting `FilePermission`, you must provide the physical name of the directory or file, such as `/private/oracle`. You cannot provide either an environment variable, such as $ORACLE_HOME, or a symbolic link. Also, to denote all files within a directory, provide the '*' symbol, as follows: `'/private/oracle/*'`. To denote all directories and files within a directory, provide the '-' symbol, as follows: `'/private/oracle/-'`.

Granting Permissions using the DBMS_JAVA package:

```
procedure grant_permission( grantee varchar2, permission_type varchar2,
                            permission_name varchar2,
                            permission_action varchar2 )

procedure grant_permission( grantee varchar2, permission_type varchar2,
                            permission_name varchar2,
                            permission_action varchar2, key OUT number)
```

Granting Permissions using Java:

```
long oracle.aurora.rdbms.security.PolicyTableManager.grant(
                              java.lang.String grantee,
                              java.lang.String permission_type,
                              java.lang.String permission_name,
                              java.lang.String permission_action);

void oracle.aurora.rdbms.security.PolicyTableManager.grant(
                              java.lang.String grantee,
                              java.lang.String permission_type,
                              java.lang.String permission_name,
                              java.lang.String permission_action,
                              long[] key);
```

Limiting Permissions using the DBMS_JAVA package:

```
procedure restrict_permission( grantee varchar2, permission_type varchar2,
                              permission_name varchar2,
                              permission_action varchar2)

procedure restrict_permission( grantee varchar2, permission_type varchar2,
                              permission_name varchar2,
                              permission_action varchar2, key OUT number)
```

Limiting Permissions using Java:

```
long oracle.aurora.rdbms.security.PolicyTableManager.restrict(
                              java.lang.String grantee,
                              java.lang.String permission_type,
                              java.lang.String permission_name,
                              java.lang.String permission_action);


void oracle.aurora.rdbms.security.PolicyTableManager.restrict(
                              java.lang.String grantee,
                              java.lang.String permission_type,
                              java.lang.String permission_name,
                              java.lang.String permission_action,
                              long[] key);
```

Example 9–1 Granting Permissions

Assuming that you have appropriate Permissions to modify the policy table, you use the `grant_permission` method within the DBMS_JAVA package to modify

the PolicyTable to allow the user access to the indicated file. In this example, the user, Larry, has `PolicyTable` modification `Permission`. Within a SQL package, Larry grants permission to read and write a file to the user Dave.

```
connect larry/larry

REM Grant DAVE permission to read and write the Test1 file.
call dbms_java.grant_permission('DAVE',
                        'java.io.FilePermission', '/test/Test1',
                        'read,write');

REM commit the changes to the PolicyTable
commit;
```

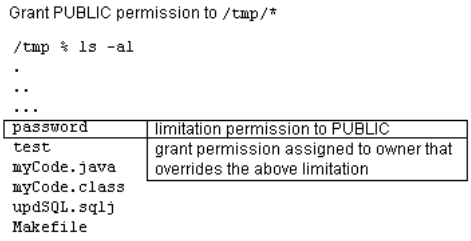
Example 9–2 Limiting Permissions

You use the `restrict` method for specifying a limitation or exception to general rules. A general rule is a rule where, in most cases, the Permission is true. However, there may be exceptions to this rule. For these exceptions, you specify a limitation Permission.

That is, if you have defined a general rule that no one can read or write for an entire directory, you can define a limitation on an aspect of this rule through the `restrict` method. For example, if you want to allow access to all files within the `/tmp` directory—except for your `password` file that exists in that directory—you would grant permission for read and write to all files within `/tmp` and limit read and write access to the `password` file.

If you want to specify an exception to the limitation, you would create an explicit grant Permission to override the limitation Permission. In the scenario mentioned above, if you want the file owner to still be able to modify the `password` file, you can grant a more explicit Permission to allow access to one user, which will override

the limitation. OracleJVM security combines all rules to understand who really has access to the `password` file. This is demonstrated in the following diagram:



The explicit rule is as follows:

> If the limitation Permission implies the request, then for a grant to be effective, the limitation Permission must also imply the grant.

The following is the code that implements this example:

1. Grant everyone (PUBLIC) read and write permission to all files in `/tmp`.
2. Limit everyone (PUBLIC) from reading or writing only the `password` file in `/tmp`.
3. Grant only Larry (owner) explicit permission to read and write the `password` file.

```
connect larry/larry

REM Grant permission to all users (PUBLIC) to be able to read and write
REM all files in /tmp.
call dbms_java.grant_permission('PUBLIC',
                        'java.io.FilePermission',
                        '/tmp/*',
                        'read,write');

REM Limit permission to all users (PUBLIC) from reading or writing the
REM password file in /tmp.
call dbms_java.restrict_permission('PUBLIC',
                        'java.io.FilePermission',
                        '/tmp/password',
                        'read,write');

REM By providing a more specific rule that overrides the limitation,
REM Larry can read and write /tmp/password.
call dbms_java.grant_permission('LARRY',
```

```
                        'java.io.FilePermission',
                        '/tmp/password',
                        'read,write');
commit;
```

Acquiring Administrative Permission to Update Policy Table

All Permissions are rows within the policy table. As it is a table within the database and thus a resource, permission is needed to modify it. Specifically, the `PolicyTablePermission` object is required to modify the table. After the first initialization for OracleJVM, only a single role—JAVA_ADMIN—is granted the `PolicyTablePermission` to modify the policy table. The JAVA_ADMIN role is immediately assigned to DBA; thus, if you are assigned to the DBA group, you will automatically take on all JAVA_ADMIN Permissions.

For you to be able to add Permissions as rows to this table, JAVA_ADMIN must grant your schema update rights for the `PolicyTablePermission`. This Permission defines that your schema can add rows to the table. Each `PolicyTablePermission` is for a specific Permission type. For example, for you to add a Permission that controls access to a file, you must have a `PolicyTablePermission` that allows you to grant or limit a Permission on a `FilePermission`. Once this occurs, you have administrative Permission for `FilePermission`.

The administrator could grant and limit the `PolicyTablePermissions` in the same manner as other Permissions, but the syntax is complicated. For ease of use, use one of the following methods within the DBMS_JAVA package to grant administrative Permissions.

Granting policy table administrative Permissions using DBMS_JAVA:

```
procedure grant_policy_permission( grantee varchar2, permission_schema varchar2,
                           permission_type varchar2,
                           permission_name varchar2)

procedure grant_policy_permission( grantee varchar2, permission_schema varchar2,
                           permission_type varchar2,
                           permission_name varchar2,
                           key OUT number)
```

Granting policy table administrative permission using Java:

```
long oracle.aurora.rdbms.security.PolicyTableManager.grantPolicyPermission(
                              java.lang.String grantee,
                              java.lang.String permission_type,
                              java.lang.String permission_name);


void oracle.aurora.rdbms.security.PolicyTableManager.grantPolicyPermission(
                              java.lang.String grantee,
                              java.lang.String permission_type,
                              java.lang.String permission_name,
                              long[] key);
```

Parameter	Description
Grantee	The name of the user, schema, or role to which you want the grant to apply. PUBLIC specifies that the row applies to all users.
Permission_schema	The <*schema*> where the Permission class is loaded.
Permission_type	The Permission class on which you are granting Permission. For example, if you were defining access to a file, the Permission type would be `FilePermission`. This parameter requires a fully-qualified name of a class that extends `java.lang.security.Permission`. If the class is not within SYS, the name should be prefixed by <*schema*>:. For example, `mySchema:myPackage.MyPermission` is a valid name for a user generated Permission.
Permission_name	The meaning of the target attribute is defined by the Permission class. Examine the appropriate Permission class for the relevant name.
Row_number	Number returned from grant or limitation to use on enable, disable, or delete methods.

Note: When looking at the policy table, the name within the PolicyTablePermission rows contains both the Permission type and the Permission name, which are separated by a "#". For example, to grant a user administrative rights for reading a file, the name in the row contains `java.io.FilePermission#read`. The "#" separates the Permission class from the Permission name.

Example 9–3 Granting PolicyTable Permission

The following example shows JAVA_ADMIN (as SYS) giving Larry permission to update the PolicyTable for `FilePermission`. Once this Permission is granted, Larry can grant permissions to other users for reading, writing, and deleting files.

```
REM Connect as SYS, which is assigned JAVA_ADMIN role, to give Larry permission
REM to modify the PolicyTable
connect SYS/SYS as SYSDBA

REM SYS grants Larry the right to administer permissions for
REM FilePermission
call dbms_java.grant_policy_permission('LARRY', 'SYS',
                                       'java.io.FilePermission', '*');
```

Creating Permissions

Create your own Permission type by performing the following steps:

1. Create and load the user Permission.
2. Grant administrative and action Permissions to specified users.
3. Implement security checks using the Permission.

1. Create and load the user Permission Create your own Permission by extending the Java 2 Permission class. Any user-created Permission must extend `Permission`. The following example creates `MyPermission`, which extends `BasicPermission`, which in turn extends `Permission`.

```
package test.larry;
import java.security.Permission;
import java.security.BasicPermission;

public class MyPermission extends BasicPermission {

  public MyPermission(String name) {
    super(name);
  }

  public boolean implies(Permission p) {
    boolean result = super.implies(p);
    return result;
  }
}
```

2. Grant administrative and action Permissions to specified users When you create a Permission, you are designated as owner of that Permission. The owner is implicitly granted administrative Permission. This means that the owner can be an administrator for this Permission and can execute `grant_policy_permission`. Administrative Permission permits the user to update the policy table for the user-defined Permission.

For example, if LARRY creates a Permission, `MyPermission`, only LARRY can invoke **`grant_policy_permission`** for himself or another user. This method updates the `PolicyTable` on who can grant rights to `MyPermission`. The following code demonstrates this:

```
REM Since Larry is the user that owns MyPermission, Larry connects to
REW the database to assign permissions for MyPermission.
connect larry/larry

REM As the owner of MyPermission, Larry grants himself the right to
REM administer permissions for test.larry.MyPermission within the JVM
REM security PolicyTable. Only the owner of the user-defined permission
REM can grant administrative rights.
call dbms_java.grant_policy_permission('LARRY', 'LARRY',
                     'test.larry.MyPermission', '*');

REM commit the changes to the PolicyTable
commit;
```

Once you have granted administrative rights, you can grant action Permissions for the user-created Permission. For example, the following SQL grants permission for LARRY to execute anything within `MyPermission` and for DAVE to execute only actions that start with "act.".

```
REM Since Larry is the user that creates MyPermission, Larry connects to
REW the database to assign permissions for MyPermission.
connect larry/larry

REM Once able to modify the PolicyTable for MyPermission, Larry grants himself
REM full permission for MyPermission. Notice that the Permission is prepended
REM with its owner schema.
call dbms_java.grant_permission( 'LARRY',
                      'LARRY:test.larry.MyPermission', '*', null);

REM Larry grants Dave permission to do any actions that start with 'act.*'.
call dbms_java.grant_permission
        ('DAVE', 'LARRY:test.larry.MyPermission', 'act.*', null);
```

```
REM commit the changes to the PolicyTable
commit;
```

3. Implement security checks using the Permission Once you have created, loaded, and assigned Permissions for `MyPermission`, you must implement the call to `SecurityManager` to have the Permission checked. There are four methods in the following example: `sensitive`, `act`, `print`, and `hello`. Because of the Permissions granted in the SQL example in step 2, the following users can execute methods within the example class:

- LARRY can execute any of the methods.
- DAVE is given permission to execute only the `act` method.
- Anyone can execute the `print` and `hello` methods. The `print` method does not check any Permissions, so anyone can execute the `print` method. The `hello` method executes `AccessController.doPrivileged`, which means that the method executes with LARRY's Permissions. This is referred to as definer's rights.

```
package test.larry;
import java.security.AccessController;
import java.security.Permission;
import java.security.PrivilegedAction;

import java.sql.Connection;
import java.sql.SQLException;

/**
 * MyActions is a class with a variety of public methods that
 * have some security risks associated with them. We will rely
 * on the Java security mechanisms to ensure that they are
 * performed only by code that is authorized to do so.
 */

public class Larry {

  private static String secret = "Larry's secret";
  MyPermission sensitivePermission = new MyPermission("sensitive");

  /**
   * This is a security sensitive operation. That is it can
   * compromise our security if it is executed by a "bad guy".
   * Only larry has permission to execute sensitive.
   */
  public void sensitive() {
```

```
    checkPermission(sensitivePermission);
    print();
  }

  /**
   * Will print a message from Larry.  We need to be
   * careful about who is allowed to do this
   * because messages from Larry may have extra impact.
   * Both larry and dave have permission to execute act.
   */
public void act(String message) {
    MyPermission p = new MyPermission("act." + message);
    checkPermission(p);
    System.out.println("Larry says: " + message);
  }

  /**
   * Print our secret key
   * No permission check is made; anyone can execute print.
   */
  private void print() {
    System.out.println(secret);
  }

  /**
   * Print "Hello"
   * This method invokes doPrivileged, which makes the method run
   * under definer's rights. So, this method runs under Larry's
   * rights, so anyone can execute hello. Only Larry can execute hello
   */
  public void hello() {
    AccessController.doPrivileged(new PrivilegedAction() {
      public Object run() { act("hello"); return null; }
    });
  }

  /**
   * If a security manager is installed ask it to check permission
   * otherwise use the AccessController directly
   */
  void checkPermission(Permission permission) {
    SecurityManager sm = System.getSecurityManager();
    sm.checkPermission(permission);
  }
}
```

Enabling or Disabling Permissions

Once you have created a row that defines a Permission, you can disable it so that it is no longer applied. However, if you decide you want the row action again, you can enable the row. You can delete the row from the table if you believe that it will never be used again. To delete, you must first disable the row. If you do not disable the row, the deletion will not occur.

To disable rows, you can use either the `disable_permission` or the `revoke` method.

- The `revoke_permission` method takes in parameters similar to the `grant` and `restrict` methods. It searches the entire policy table for all rows that match the supplied parameters.
- The `disable_permission` method disables only a single row within the policy table. To do this, it takes in the policy table key. This key is also necessary to enable or delete a Permission. To retrieve the Permission key number, perform one of the following:
    * Save the key when it is returned on the grant or limit calls. If you do not foresee a need to ever enable or disable the Permission, you can use the grant and limit calls that do not return the Permission number.
    * View DBA_JAVA_POLICY or USER_JAVA_POLICY for the appropriate Permission key number.

Disabling Permissions using DBMS_JAVA:

```
procedure revoke_permission(permission_schema varchar2,
                            permission_type varchar2,
                            permission_name varchar2,
                            permission_action varchar2)


procedure disable_permission(key number)
```

Disabling Permissions using Java:

```
void revoke(String schema, String type, String name, String action);

void oracle.aurora.rdbms.security.PolicyTableManager.disable(long number);
```

Enabling Permissions using DBMS_JAVA:

```
procedure enable_permission(key number)
```

Enabling Permissions using Java:

```
void oracle.aurora.rdbms.security.PolicyTableManager.enable(long number);
```

Deleting Permissions using DBMS_JAVA:

```
procedure delete_permission(key number)
```

Deleting Permissions using Java:

```
void oracle.aurora.rdbms.security.PolicyTableManager.delete(long number);
```

Permission Types

Table 9–2 lists the installed Permission types. Whenever you want to grant or limit a Permission, you must provide the Permission type within the DBMS_JAVA method. The Permission types with which you control access are the following:

- Oracle-provided Permission types listed in Table 9–2
- user created Permission types that extend `java.security.Permission`

Table 9–2 Permission Types

■ `java.util.PropertyPermission`
■ `java.io.SerializablePermission`
■ `java.io.FilePermission`
■ `java.net.NetPermission`
■ `java.net.SocketPermission`
■ `java.lang.RuntimePermission`
■ `java.lang.reflect.ReflectPermission`
■ `java.security.SecurityPermission`
■ `oracle.aurora.rdbms.security.PolicyTablePermission`
■ `oracle.aurora.security.JServerPermission`

All the Java Permission types are documented in the Sun Microsystems Java 2 documentation.

> **Note:** SYS is granted permission to load libraries that come with Oracle. However, OracleJVM does not support other users loading libraries, because loading C within the database is insecure. Therefore, you are not allowed to grant permission for `loadLibrary.*` of `RuntimePermission`.

The Oracle-specific Permissions, `PolicyTablePermission` and `JServerPermission`, are described below:

oracle.aurora.rdbms.security.PolicyTablePermission This Permission controls who can update the policy table. Once granted the right to update the policy table for a certain `Permission` type, the user can control other user's access to some resource.

After OracleJVM initialization, only the JAVA_ADMIN role can grant administrative rights for the policy table through `PolicyTablePermission`. Once it grants this right to other users, these users can in turn update the policy table with their own grant and limitation Permissions.

To grant policy table updates, use the DBMS_JAVA method: `grant_policy_permission`, as discussed in "Acquiring Administrative Permission to Update Policy Table" on page 9-12. Once you have updated the table, you can view either the DBA_JAVA_POLICY or USER_JAVA_POLICY views to see who has been granted Permissions.

oracle.aurora.security.JServerPermission Use this Permission to grant and limit access to OracleJVM resources. The `JServerPermission` extends from `BasicPermission`. The following table lists the names for which `JServerPermission` grants access:

Permission Name	Description
LoadClassInPackage.<*package_name*>	grants the ability to load a class within the specified package
Verifier	grants the ability to turn the bytecode verifier on or off
Debug	grants the ability for debuggers to connect to a session
JRIExtensions	grants the use of MEMSTAT
Memory.Call	grants rights to call certain methods in `oracle.aurora.vm.OracleRuntime` on call settings

Permission Name	Description
Memory.Stack	grants rights to call certain methods in `oracle.aurora.vm.OracleRuntime` on stack settings
Memory.SGAIntern	grants rights to call certain methods in `oracle.aurora.vm.OracleRuntime` on SGA settings
Memory.GC	grants rights to call certain methods in `oracle.aurora.vm.OracleRuntime` on garbage collector settings

Initial Permission Grants

When you first initialize OracleJVM, several roles are populated with certain Permission grants. The following tables show these roles and their initial Permissions:

1. The JAVA_ADMIN role is given access to modify the policy table for all Permissions. All DBAs, including SYS, are granted JAVA_ADMIN. Full administrative rights to update the policy table are granted for the following Permissions:

- `java.util.PropertyPermission`
- `java.io.SerializablePermission`
- `java.io.FilePermission`
- `java.net.NetPermission`
- `java.net.SocketPermission`
- `java.lang.RuntimePermission`
- `java.lang.reflect.ReflectPermission`
- `java.security.SecurityPermission`
- `oracle.aurora.rdbms.security.PolicyTablePermission`
- `oracle.aurora.security.JServerPermission`

2. In addition to the JAVA_ADMIN Permissions, SYS is also granted the following Permissions:

Note: Within the `RuntimePermission` grants, there seems to be unnecessary granting of more specific Permission for `loadlibrary.<package>`. The reason for this is to override the limitation given to PUBLIC for `loadLibrary.*`.

Table 9–3 SYS Initial Permissions

Permission Type	Permission Name	Action Granted
`oracle.aurora.rdbms.security.PolicyTablePermission`	*	Administrative rights to modify the policy table
`oracle.aurora.security.JServerPermission`	*	null
`java.net.NetPermission`	*	null
`java.security.SecurityPermission`	*	null
`java.util.PropertyPermission`	*	write
`java.lang.reflect.ReflectPermission`	*	null
`java.lang.RuntimePermission`	*	null
	`loadLibrary.xaNative`	null
	`loadLibrary.corejava`	null
	`loadLibrary.corejava_d`	null

3. All users are initially granted the following Permissions. For the `JServerPermission`, all users can load classes, except for the list of classes specified in the table. These exceptions are limitation Permissions. For more information on limitation Permissions, see Example 9–2.

Table 9–4 PUBLIC Default Permissions

Permission Type	Permission Name	Granted Action
`oracle.aurora.rdbms.security.PolicyTablePermission`	`java.lang.RuntimePermission.loadLibrary.*`	null
`java.util.PropertyPermission`	*	read
	`user.language`	write

Table 9–4 PUBLIC Default Permissions

Permission Type	Permission Name	Granted Action
`java.lang.RuntimePermission`	`_`	null
	`exitVM`	null
	`createSecurityManager`	null
	`modifyThread`	null
	`modifyThreadGroup`	null
`oracle.aurora.security.JServerPermission`	`loadClassInPackage.*` except for `loadClassInPackage.java.*`, `loadClassInPackage.oracle.aurora.*`, and `loadClassInPackage.jdbc.*`	null

Table 9–5 JAVAUSERPRIV Permissions

Permission Type	Permission Name	Action
`java.net.SocketPermission`	*	connect, resolve
`java.io.FilePermission`	<<ALL FILES>>	read
`java.lang.RuntimePermission`	`modifyThreadGroup`, `stopThread`, `getProtectionDomain`, `readFileDescriptor`, `accessClassInPackage.*`, and `defineClassInPackage.*`	null

Table 9–6 JAVASYSPRIV Permissions

Permission Type	Permission Name	Action
`java.io.SerializablePermission`	*	no applicable action
`java.io.FilePermission`	<<ALL FILES>>	read ,write, execute, delete
`java.net.SocketPermission`	*	accept, connect, listen, resolve

Table 9–6 JAVASYSPRIV Permissions (Cont.)

Permission Type	Permission Name	Action
`java.lang.RuntimePermission`	`createClassLoader`	null
	`getClassLoader`	null
	`setContextClassLoader`	null
	`setFactory`	null
	`setIO`	null
	`setFileDescriptor`	null
	`readFileDescriptor`	null
	`writeFileDescriptor`	null

Table 9–7 JAVADEBUGPRIV Permissions

Permission Type	Permission Name	Action
`oracle.aurora.security.JServerPermission`	`Debug`	null
`java.net.SocketPermission`	*	connect, resolve

General Permission Definition Assigned to Roles

In release 8.1.5, Oracle8*i* JVM security was controlled by granting the roles of JAVASYSPRIV, JAVAUSERPRIV, or JAVADEBUGPRIV to schemas. In the current version, these roles still exist as Permission groups. See the previous section, "Initial Permission Grants" on page 9-21 for the explicit Permissions set for each role. You can set up and define your own collection of Permissions. Once defined, you can grant any collection of Permissions to any user. That user will then have the same Permissions that exist within the role. In addition, if you need additional Permissions, you can add individual Permissions to either your specified user or role. Permissions defined within the policy table have a cumulative effect. See "Fine-Grain Definition for Each Permission" on page 9-7 for information on how to grant Permissions to a user or a role.

Note: The ability to write to properties, granted through the write action on `PropertyPermission`, is no longer granted to all users. Instead, you must have either JAVA_ADMIN grant this Permission to you or you can receive it by being granted the role of JAVASYSPRIV.

The following example gives Larry and Dave the following Permissions:

- Larry receives JAVASYSPRIV Permissions.
- Dave receives JAVADEBUGPRIV Permissions and the ability to read and write all files on the system.

```
REM Granting Larry the same permissions as exist within JAVASYSPRIV
grant javasyspriv to larry;

REM Granting Dave the ability to debug
grant javadebugpriv to dave;

commit;

REM I also want Dave to be able to read and write all files on the system
call dbms_java.grant_permission('DAVE', 'SYS:java.io.FilePermission',
                                '<<ALL FILES>>', 'read,write', null);
```

Debugging Permissions

A debug role, JAVADEBUGPRIV, was created to grant Permissions for running the debugger. The Permissions assigned to this role are listed in Table 9–7. To receive permission to invoke the debug agent, the caller must have been granted JAVADEBUGPRIV or the debug `JServerPermission` as follows:

```
REM Granting Dave the ability to debug
grant javadebugpriv to dave;

REM Larry grants himself permission to start the debug agent.
call dbms_java.grant_permission
       ('LARRY', 'oracle.aurora.security.JServerPermission', 'Debug', null);
```

Although a debugger provides extensive access to both code and data on the server, its use should be limited to development environments. Refer to the discussion in "Debugging Server Applications" on page 3-7 for information on using the debugging facilities in this release.

Permission for Loading Classes

To load classes, you must have the following Permission:

```
JServerPermission("LoadClassInPackage." + <class_name>)
```

The class name is the fully qualified name of the class that you are loading.

This excludes loading into system packages or replacing any system classes. Even if you are granted permission to load a system class, Oracle Database prevents you from performing the load. System classes are classes that are installed by Oracle Database with CREATE JAVA SYSTEM. The following error is thrown if you try to replace a system class:

```
ORA-01031 "Insufficient privileges"
```

The following shows the ability of each user after database installation, including Permissions and OracleJVM restrictions:

- SYS can load any class except for system classes.
- Any user can load classes in its own schema that do not start with the following patterns: `java.*`, `oracle.aurora.*`, `oracle.jdbc.*`. If the user wants to load such classes into another schema, it must be granted the `JServerPermission(LoadClassInPackage.<class>)` Permission.

The following example shows how to grant SCOTT Permission to load classes into the `oracle.aurora.*` package:

```
dbms_java.grant_permission('SCOTT', 'SYS:oracle.aurora.tools.*', null);
```

Database Authentication Mechanisms

- Password authentication
- Strong authentication (advanced security)
- Proxy authentication
- Single sign-on

10

Oracle Database Java Application Performance

You can increase your Java application performance through one of the following methods:

- Natively Compiled Code
- Java Memory Usage

Natively Compiled Code

The Java language was designed for a platform-independent, secure development model. To accomplish these goals, some execution performance was sacrificed. Translating Java bytecodes into machine instructions degrades performance. To regain some of the performance loss, you may choose to natively compile certain classes. For example, you may decide to natively compile code with CPU intensive classes.

Without native compilation, the Java code you load to the server is interpreted, and the underlying core classes upon which your code relies (`java.lang.*`) are natively compiled.

Native compilation provides a speed increase ranging from two to ten times the speed of the bytecode interpretation. The exact speed increase is dependent on several factors, including:

- use of numerics
- degree of polymorphic message sends
- use of direct field access, as opposed to accessor methods
- amount of Array accessing
- casts

Because Java bytecodes were designed to be compact, natively compiled code can be considerably larger than the original bytecode. However, because the native code is stored in a shared library, it is shared among all users of the database.

Most JVMs use Just-In-Time compilers that convert the Java bytecodes to native machine instructions when methods are invoked. The Accelerator uses an Ahead-Of-Time approach to recompiling the Java classes.

Native Compiler	Description
Just-In-Time	Provides the JVM the ability to translate the Java instructions just before needed by the JDK. The benefits depends on how accurately the native compiler anticipates code branches and the next instruction. If incorrect, no performance gain is realized.

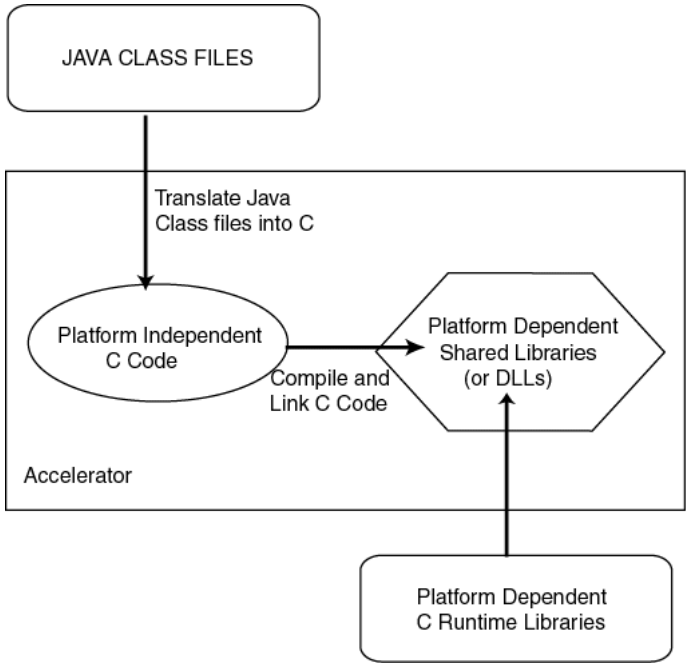
Native Compiler	Description
Ahead-Of-Time	The Accelerator natively compiles all Java code within a JAR file into native shared libraries, which are organized by Java package, before execution time. At runtime, Accelerator checks if a Java package has been natively compiled; and if so, uses the machine code library instead of interpreting the deployed Java code.

This static compilation approach provides a large, consistent performance gain, regardless of the number of users or the code paths they traverse on the server. After compilation, the tool loads the statically compiled libraries into Oracle Database, which are then shared between users, processes, and sessions.

Accelerator Overview

Most Ahead-Of-Time native compilers compile directly into a platform-dependent language. For portability requirements, this was not feasible. Figure 10–1 illustrates how the Accelerator translates the Java classes into a version of C that is platform-independent. The C code is compiled and linked to supply the final platform-dependent, natively compiled shared libraries or DLLs.

Figure 10–1 Native Compilation Using Accelerator



Given a JAR file, the Accelerator performs the following:

1. Verifies the classes that are loaded in the database.
2. Retrieves the Java bytecodes for these classes from the database and stores them in a project directory where the Accelerator was invoked.
3. Translates the Java bytecodes to C code.
4. Compiles and links the C code using the C compiler for your platform.

   The Accelerator translates, compiles, and links the retrieved classes on the client. For this reason, you must natively compile on the intended platform environment to which this application will be deployed. The result is a single deployment JAR file for all classes within the project.
5. The resulting shared library is loaded into the `$ORACLE_HOME/javavm/admin` directory.

Note: The Accelerator natively compiled libraries can be used only within Oracle Database. Also, these libraries can only be used within the same version of Oracle Database in which it was produced. If you want your application to be natively compiled on subsequent releases, you must recompile these classes. That is, native recompilation of existing libraries will not be performed automatically by any upgrade process.

Oracle Database Core Java Class Libraries

All core Java class libraries and Oracle-provided Java code within Oracle Database is natively compiled for greater execution speed. Java classes exist as shared libraries in `$ORACLE_HOME/javavm/admin`, where each shared library corresponds to a Java package. For example, `orajox10java_lang.so` on Solaris and `orajox10java_lang.dll` on Windows NT hold `java.lang` classes. Specifics of packaging and naming can vary by platform. The OracleJVM uses natively compiled Java files internally and opens them, as necessary, at runtime.

Natively Compiling Java Application Class Libraries

The Accelerator can be used by Java application products that need performance increased and are deployed in Oracle Database. The Accelerator command-line tool, `ncomp`, natively compiles your code and loads it in Oracle Database. However, in order to use `ncomp`, you must first provide some initial setup.

Installation Requirements

You must install the following before invoking Accelerator:

1. Install a C compiler for the intended platform on the machine where you are running `ncomp`.
2. Verify that the correct compiler and linker commands are referenced within the `System*.properties` file located in the `$ORACLE_HOME/javavm/jahome` directory. Since the compiler and linker information is platform-specific, the configuration for these items is detailed in the README for your platform.
3. Add the appropriate J2SE JAR files, library, and binary information in the following environment variables:

Environment Variables	Addition Required
JAVA_HOME	Set to the location where your JDK is installed.
CLASSPATH	Include the $JAVA_HOME/lib/tools.jar and $JAVA_HOME/lib/dt.jar files in your CLASSPATH.
PATH	Add the JDK binary path: $JAVA_HOME/bin.
LD_LIBRARY_PATH	Add the JDK library path: $JAVA_HOME/lib.

4. Grant the user that executes ncomp the following role and security permissions:

> **Note:** DBA role contains both the JAVA_DEPLOY role and the FilePermission for all files under $ORACLE_HOME.

   a. JAVA_DEPLOY: The user must be assigned to the JAVA_DEPLOY role in order to be able to deploy the shared libraries on the server, which both the ncomp and deploync utilities perform. For example, the role is assigned to DAVE, as follows:

```
SQL> GRANT JAVA_DEPLOY TO DAVE;
```

   b. FilePermission: Accelerator stores the shared libraries with the natively compiled code on the server. In order for Accelerator to store these libraries, the user must be granted FilePermission for read and write access to directories and files under $ORACLE_HOME on the server. One method for granting FilePermission for all desired directories is to grant the user the JAVASYSPRIV role, as follows:

```
SQL> GRANT JAVASYSPRIV TO DAVE;
```

   See Chapter 9, "Security For Oracle Database Java Applications" for more information JAVASYSPRIV and granting FilePermission.

Executing Accelerator

The following sections show how to do basic native compilation using Accelerator.

> **Note:** Before you natively compile your Java server code, you must have already loaded and tested it within Oracle Database. Native compilation of untested code is not recommended.
>
> Keep in mind that debuggers, such as the debugger provided with JDeveloper, are useful only with interpreted Java code. You cannot debug a natively compiled library.

All the Java classes contained within a JAR file must already be loaded within the database. Execute the `ncomp` tool to instruct Accelerator to natively compile all these classes. The following code natively compiles all classes within the `pubProject.JAR` file:

```
ncomp -user scott/tiger pubProject.JAR
```

> **Note:** Because native compilation must compile and link all your Java classes, this process may execute over the span of a few hours. The time involved in natively compiling your code depends on the number of classes to compile and the type of hardware on your machine.

If you change any of the classes within this JAR file, Accelerator recompiles the shared library for the package that contains the changed classes. It will not recompile all shared libraries. However, if you want all classes within a JAR file to be recompiled—regardless of whether they were previously natively compiled—execute `ncomp` with the `-force` option, as follows:

```
ncomp -user scott/tiger -force pubProject.JAR
```

ncomp

Accelerator, implemented within the `ncomp` tool, natively compiles all classes within the specified JAR, ZIP, or list of classes. Accelerator natively compiles these classes and places them into shared libraries according to their package. Note that these classes must first be loaded into the database.

If the classes are designated within a JAR file and have already been loaded in the database, you can natively compile your Java classes by executing the following:

```
ncomp -user SCOTT/TIGER myClasses.jar
```

> **Note:** Because native compilation must compile and link all of your Java classes, this process may execute over the span of a few minutes or a few hours. The time involved depends on the number of classes to compile and the type of hardware on your machine.

There are options that allow you control over how the details of native compilation are handled.

Syntax

```
ncomp [ options ] <class_designation_file>
  -user | -u <username>/<password>[@<database_url>]
  [-load]
  [-projectDir | -d <project_directory>]
  [-force]
  [-lightweightDeployment]
  [-noDeploy]
  [-outputJarFile | -o <jar_filename>]
  [-thin]
  [-oci | -oci8]
  [-update]
  [-verbose]
```

> **Note:** These options are demonstrated within the scenarios described in "Native Compilation Usage Scenarios" on page 10-12.

Argument Summary

Table 10–1 summarizes the `ncomp` arguments. The *<class_designation_file>* can be a *<file>.jar*, *<file>.zip*, or *<file>.classes*.

Table 10–1 ncomp Argument Summary

Argument	Description and Values
`<file>.jar`	The full path name and file name of a JAR file that contains the classes that are to be natively compiled. If you are executing in the directory where the JAR file exists and you do not specify the `-projectDir` option, you may give only the name of the JAR file.
`<file>.zip`	The full path name and file name of a ZIP file that contains the classes that are to be natively compiled. If you are executing in the directory where the ZIP file exists and you do not specify the `-projectDir` option, you may give only the name of the ZIP file.
`<file>.classes`	The full path name and file name of a classes file, which contains the list of classes to be natively compiled. If you are executing in the directory where the classes file exists and you do not specify the `-projectDir` option, you may give only the name of the classes file. See "Natively Compiling Specific Classes" on page 10-14 for a description of a classes file.
`-user | -u <username>/<password> [@<database>]`	Specifies a user, password, and database connect string; the files will be loaded into this database instance. The argument has the form *`<username>`*`/`*`<password>`*`[@`*`<database>`*`]`. If you specify the database URL on this option, you must specify it with OCI syntax. To provide a JDBC Thin database URL, use the -thin option. See "user" on page 10-10 for more information.
`-force`	The native compilation is performed on all classes. Previously compiled classes are not passed over.
`-lightweightDeployment`	Provides an option for deploying shared libraries and native compilation information separately. This is useful if you need to preserve resources when deploying. See "lightweightDeployment" on page 10-11 for more information.
`-load`	Executes `loadjava` on the specified class designation file. You cannot use this option in combination with a *`<file>`*`.classes` file.

Table 10–1 ncomp Argument Summary

Argument	Description and Values
`-outputJarFile <jar_filename>`	All natively compiled classes output into a deployment JAR file. This option specifies the name of the deployment JAR file and its destination directory. If omitted, the `ncomp` tool names the output deployment JAR file the same name as the input <*file*> with "`_depl.jar`" appended as the suffix. If directory is not supplied, it stores the output JAR file into the project directory (denoted by `-projectDir`).
`-noDeploy`	Specifies that the native compilation results only in the output deployment JAR file, which is not deployed to the server. The resulting deployment JAR can be deployed to any server using the deploync tool.
`-thin`	The database URL that is provided on the `-user` option uses a JDBC Thin URL address for the database URL syntax.
`-oci | -oci8`	The database URL that is provided on the -user option uses an OCI URL address for the database URL syntax. However, if neither `-oci` or `-thin` are specified, the default assumes that you used an OCI database URL.
`-projectDir | -d <absolute_path>`	Specifies the full path for the project directory. If not specified, Accelerator uses the directory from which `ncomp` is invoked as the project directory. This directory must exist; the tool will not create this directory for you. If it does not exist, the current directory is used.
`-update`	If you add more classes to a <*class_designation_file*> that has already been natively compiled, this flag informs Accelerator to update the deployment JAR file with the new classes. Thus, Accelerator compiles the new classes and adds them to the appropriate shared libraries. The deployment JAR file is updated.
`-verbose`	Output native compilation text with detail.

Argument Details

user

`{-user | -u} <user>/<password>[@<database>]`

The permissible forms of `@<database>` depend on whether you specify `-oci` or `-thin`; `-oci` is the default.

- `-oci`: `@<database>` is optional; if you do not specify, then `ncomp` uses the user's default database. If specified, then `<database>` can be a TNS name or a Oracle Net Services name-value list.

- `-thin`: `@<database>` is required. The format is `<host>:<lport>:<SID>`.
  - `<host>` is the name of the machine running the database.
  - `<lport>` is the listener port that has been configured to listen for Oracle Net Services connections; in a default installation, it is 5521.
  - `<SID>` is the database instance identifier; in a default installation, it is `ORCL`.

lightweightDeployment

Accelerator places compilation information and the compiled shared libraries in one JAR file, copies the shared libraries to `$ORACLE_HOME/javavm/admin` directory on the server, and deploys the compilation information to the server. If you want to place the shared libraries on the server yourself, you can do so through the `lightweightDeployment` option. The `lightweightDeployment` option enables you to do your deployment in two stages:

1. Natively compile your JAR file with `-noDeploy` and `-lightweightDeployment` options. This creates an deployment JAR file with only `ncomp` information, such as transitive closure information. The shared libraries are not saved within the deployment JAR file. Thus, the deployment JAR file is much smaller.
2. Deploy as follows:
   a. Copy all output shared libraries from the `lib` directory of the native compilation project directory to the server's `$ORACLE_HOME/javavm/admin` directory.

      Note: You need to have `FilePermission` to write to this directory. `FilePermission` is included in the DBA or JAVASYSPRIV roles.

   b. Deploy the lightweight deployment JAR file to the server using `deploync`.

Errors

Any errors that occur during native compilation are printed to the screen. Any errors that occur during deployment of your shared libraries to the server or during runtime can be viewed with the `statusnc` tool or by referring to the `JACCELERATOR$DLL_ERRORS` table.

If an error is caught while natively compiling the designated classes, Accelerator denotes these errors, abandons work on the current package, and continues its compilation task on the next package. The native compilation continues for the rest of the packages. The package with the class that contained the error will not be natively compiled at all.

After fixing the problem with the class, you can choose to do one of the following:

- recompile the shared library
- reload the Java class into the database

If you choose not to recompile the classes, but to load the correct Java class into the database instead, then the corrected class and all classes that are included in the resolution validation for that class—whether located within the same shared library or a different shared library—will be executed in interpreted mode. That is, the JVM will not run these classes natively. All the other natively compiled classes will continue to execute in native format. When you execute the `statusnc` command on the reloaded class or any of its referred classes, they will have a `NEED_NCOMPING` status message.

Possible errors for a Java class:

1. The Java class does not exist in the database. If you do not load the Java class into Oracle Database, Accelerator does not include the class in the shared library. The class is simply skipped.
2. The Java class is invalid; that is, one of its references may not be found.
3. Any Java class that is unresolved, Accelerator will try to resolve it before natively compiling. However, if the class cannot be resolved, it is ignored by Accelerator.

Possible errors for deployment of native compilation JAR file:

- The native compilation of your JAR file executes correctly, but the deployment fails. In this case, do not recompile the JAR file, but deploy the output natively compiled JAR file with the deploync command.

Native Compilation Usage Scenarios

The following scenarios demonstrate how you can use each of the options for the `ncomp` tool can be used:

- Natively Compiling on Test Platform—Java Classes Already Loaded in the Database

Natively Compiling on Test Platform—Java Classes Already Loaded in the Database

If all classes are loaded into the database and you have completed your testing of the application, you can request Accelerator to natively compile the tested classes. Accelerator takes in a JAR, ZIP, or list of classes to determine the packages and classes to be included in the native compilation. The Accelerator then retrieves all of the designated classes from the server and natively compiles them into shared libraries—each library containing a single package of classes.

Assuming that the classes have already been loaded within the server, you execute the following command to natively compile all classes listed within a class designation file, such as the `pubProject.jar` file, as follows:

```
ncomp -user SCOTT/TIGER pubProject.jar
```

If you change any of the classes within the class designation file and ask for recompilation, Accelerator recompiles only the packages that contain the changed classes. It will not recompile all packages.

Natively Compiling Java Classes Not Loaded in the Database

Once you have tested the designated classes, you may wish to natively compile them on a host other than the test machine. Once you transfer the designated class file to this platform, the classes in this file must be loaded into the database before native compilation can occur. The following loads the classes through `loadjava` and then executes native compilation for the class designation file—`pubProject.jar`:

```
ncomp -user SCOTT/TIGER@dbhost:5521:orcl -thin -load pubProject.jar
```

Clean Compile and Generate Output for Future Deployment

If you want all classes within a class designation file to be recompiled—regardless of whether they were previously natively compiled—execute `ncomp` with the `-force` option. You might want to use the `-force` option to ensure that all classes

are compiled, resulting in a deployment JAR file that can be deployed to other Oracle Database instances. You can specify the native compilation deployment JAR file with the `-outputJarFile` option. The following forces a recompilation of all Java classes within the class designation file—`pubProject.jar`—and creates a deployment JAR file with the name of `pubworks.jar`:

```
ncomp -user SCOTT/TIGER -force -outputJarFile pubworks.jar pubProject.jar
```

The deployment JAR file contains the shared libraries for your classes, and installation classes specified to these shared libraries. It does not contain the original Java classes. To deploy the natively compiled deployment JAR file to any Oracle Database (of the appropriate platform type), you must do the following:

1. Load the original Java classes into the destination server. In the previous example, the `pubProject.jar` file would be loaded into the database using the `loadjava` tool.
2. Deploy the natively compiled deployment JAR file with the Accelerator deploync tool, which is described in deploync on page 10-15.

Controlling Native Compilation Build Environment

By default, the Accelerator uses the directory where `ncomp` is executed as its build environment. The Accelerator downloads several class files into this directory and then uses this directory for the compilation and linking process.

If you do not want to have Accelerator put any of its files into the current directory, create a working directory, and specify this working directory as the project directory with the `-projectDir` option. The following directs Accelerator to use `/tmp/jaccel/pubComped` as the build directory. This directory must exist before specifying it within the `-projectDir` option. Accelerator will not create this directory for you.

```
ncomp -user SCOTT/TIGER -projectDir /tmp/jaccel/pubComped pubProject.jar
```

Natively Compiling Specific Classes

You can specify one or more classes that are to be natively compiled, within a text-based `<file>.classes` file. Use the following Java syntax to specify packages and/or individual classes within this file:

- To specify classes within one or more packages, as follows:

```
import com.myDomain.myPackage.*;
import com.myDomain.myPackage.mySubPackage.*;
```

> **Note:** Java has no formal notion of a sub-package. You must specify each package independently.

- To specify an individual class, as follows:

```
import com.myDomain.myPackage.myClass;
```

Once explicitly listed, specify the name and location of this class designation file on the command line. Given the following pubworks.classes file:

```
import com.myDomain.myPackage.*;
import com.myDomain.hisPackage.hisSubPackage.*;
import com.myDomain.herPackage.herClass;
import com.myDomain.petPackage.petClass;
```

The following directs Accelerator to compile all classes designated within this file: all classes in `myPackage`, `hisSubPackage` and the individual classes, `herClass` and `myClass`. These classes must have already been loaded into the database:

```
ncomp -user SCOTT/TIGER /tmp/jaccel/pubComped/pubworks.classes
```

Natively Compiling Packages That Are Fully or Partially Modified

If you change any of the classes within this JAR file, Accelerator will only recompile shared libraries that contain the changed classes. It will not recompile all shared libraries designated in the JAR file. However, if you want all classes within a JAR file to be recompiled—regardless of whether they were previously natively compiled—you execute `ncomp` with the `-force` option, as follows:

```
ncomp -user scott/tiger -force pubProject.JAR
```

deploync

You can deploy any deployment JAR file with the `deploync` command. This includes the default output JAR file, `<file>_depl.jar` or the JAR created when you used the `ncomp -outputJarFile` option. The operating system and Oracle Database version must be the same as the platform where it was natively compiled.

> **Note:** The list of shared libraries deployed into Oracle Database are listed within the `JACCELERATOR$DLLS` table.

Syntax

```
deploync [options] <deployment>.jar
  -user | -u <username>/<password>[@<database_url>]
  [-projectDir | -d <project_directory>]
  [-thin]
  [-oci | -oci8]
```

Argument Summary

Table 10–2 summarizes the `deploync` arguments.

Table 10–2 deploync Argument Summary

Argument	Description and Values
`<deployment>.jar`	The full path name and file name of a deployment JAR file. This JAR file is created when you specify the `-outputJarFile` option on the `ncomp` tool. Note that `deploync` does not verify that this is a native compilation deployment JAR.
`-user | -u <username>/<password> [@<database>]`	Specifies a user, password, and database connect string; the files will be loaded into this database instance. The argument has the form `<username>/<password>[@<database>]`. If you specify the database URL on this option, you must specify it with OCI syntax. To provide a JDBC Thin database URL, use the `-thin` option.
`-projectDir | -d <absolute_path>`	Specifies the full path for the project directory. If not specified, Accelerator uses the directory from which `ncomp` is invoked as the project directory.
`-thin`	The database URL that is provided on the `-user` option uses a JDBC Thin URL address for the database URL syntax.
`-oci | -oci8`	The database URL that is provided on the `-user` option uses an OCI URL address for the database URL syntax. However, if neither `-oci` or `-thin` are specified, the default assumes that you used an OCI database URL.

Example

Deploy the natively compiled deployment JAR file `pub.jar` to the `dbhost` database as follows:

```
deploync -user SCOTT/TIGER@dbhost:5521:orcl -thin /tmp/jaccel/PubComped/pub.jar
```

statusnc

After the native compilation is completed, you can check the status for your Java classes through the `statusnc` command. This tool will print out—either to the screen or to a designated file—the status of each class. In addition, the `statusnc` tool always saves the output within the JACCELERATOR$STATUS table. The values can be the following:

Class Native Compilation Status	Description
`ALREADY_NCOMPED`	The class is currently natively compiled.
`NEED_NCOMPING`	A class within the shared library was reloaded after native compilation. Thus, you should recompile this shared library.
`INVALID`	A class loaded in the database is invalid. Accelerator tried to validate it and failed. The class will be excluded from the natively compiled shared library.

Note: The `JACCELERATOR$STATUS` table contains only the output from the last execution of the `statusnc` command. When executed, the `statusnc` command cleans out this table before writing the new records into it.

Syntax

```
statusnc [ options ] <class_designation_file>
   -user <user>/<password>[@database]
   [-output | -o <filename>]
   [-projectDir | -d <directory>]
   [-thin]
   [-oci | -oci8]
```

Argument Summary

Table 10–3 summarizes the `statusnc` arguments. The *<class_designation_file>* can be a *<file>.jar*, *<file>.zip*, or *<file>.classes*.

Table 10–3 statusnc Argument Summary

Argument	Description
`<file>.jar`	The full path name and file name of a JAR file that was natively compiled.
`<file>.zip`	The full path name and file name of a ZIP file that was natively compiled.
`<file>.classes`	The full path name and file name of a classes file, which contains the list of classes that was natively compiled. See "Natively Compiling Specific Classes" on page 10-14 for a description of a classes file.
`-user | -u <username>/<password> [@<database>]`	Specifies a user, password, and database connect string where the files are loaded. The argument has the form *<username>*`/`*<password>*`[@`*<database>*`]`. If you specify the database URL on this option, you must specify it with OCI syntax. To provide a JDBC Thin database URL, use the `-thin` option.
`-output <filename>`	Designates that the `statusnc` should output to the specified text file rather than to the screen.
`-projectDir | -d <absolute_path>`	Specifies the full path for the project directory. If not specified, Accelerator uses the directory from which `ncomp` is invoked as the project directory.
`-thin`	The database URL that is provided on the `-user` option uses a JDBC Thin URL address for the database URL syntax.
`-oci | -oci8`	The database URL that is provided on the `-user` option uses an OCI URL address for the database URL syntax. However, if neither `-oci` or `-thin` are specified, the default assumes that you used an OCI database URL.

Example

```
statusnc -user SCOTT/TIGER -output pubStatus.txt /tmp/jaccel/PubComped/pub.jar
```

Java Memory Usage

The typical and custom database installation process furnishes a database that has been configured for reasonable Java usage during development. However, runtime use of Java should be determined by the usage of system resources for a given deployed application. Resources you use during development can vary widely, depending on your activity. The following sections describe how you can configure

memory, how to tell how much SGA memory you are using, and what errors denote a Java memory issue:

Configuring Memory Initialization Parameters

You can modify the following database initialization parameters to tune your memory usage to reflect more accurately your application needs:

- **SHARED_POOL_SIZE**—Shared pool memory is used by the class loader within the JVM. The class loader uses an average of about 8 KB for each loaded class. Shared pool memory is used when loading and resolving classes into the database. It is also used when compiling source in the database or when using Java resource objects in the database.

  The memory specified in `SHARED_POOL_SIZE` is consumed transiently when you use `loadjava`. The database initialization process (executing `initjvm.sql` against a clean database, as opposed to the installed seed database) requires `SHARED_POOL_SIZE` to be set to 50 MB as it loads the Java binaries for approximately 8,000 classes and resolves them. The `SHARED_POOL_SIZE` resource is also consumed when you create call specifications and as the system tracks dynamically loaded Java classes at runtime.

- **JAVA_POOL_SIZE**—The OracleJVM memory manager allocates all other Java state during runtime execution from the amount of memory allocated using `JAVA_POOL_SIZE`. This memory includes the shared in-memory representation of Java method and class definitions, as well as the Java objects migrated to session space at end-of-call. In the first case, you will be sharing the memory cost with all Java users. In the second case, in a shared server, you must adjust `JAVA_POOL_SIZE` allocation based on the actual amount of state held in static variables for each session. See "Java Pool Memory" on page 10-21 for more information on `JAVA_POOL_SIZE`.

- **JAVA_SOFT_SESSIONSPACE_LIMIT**—This parameter allows you to specify a soft limit on Java memory usage in a session, which will warn you if you must increase your Java memory limits. Every time memory is allocated, the total memory allocated is checked against this limit.

When a user's session-duration Java state exceeds this size, OracleJVM generates a warning that is written into the trace files. While this warning is simply an informational message and has no impact on your application, you should understand and manage the memory requirements of your deployed classes, especially as they relate to usage of session space.

- **`JAVA_MAX_SESSIONSPACE_SIZE`**—If a user-invokable Java program executing in the server can be used in a way that is not self-limiting in its memory usage, this setting may be useful to place a hard limit on the amount of session space made available to it. The default is 4 GB. This limit is purposely set extremely high to be normally invisible.

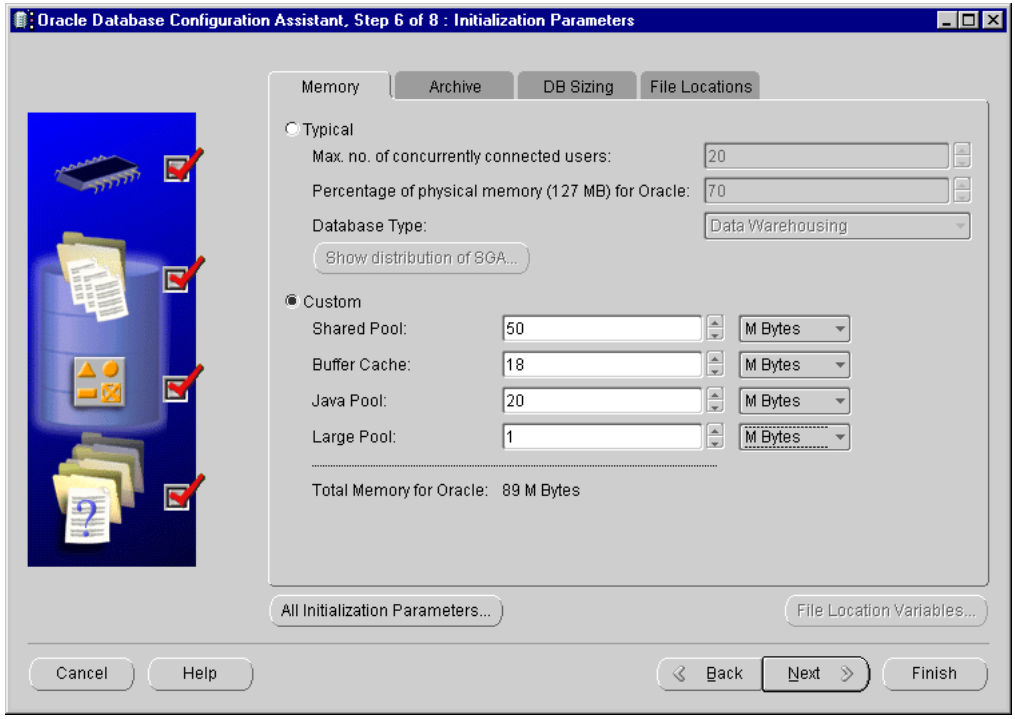
  When a user's session-duration Java state attempts to exceeds this size, your application can receive an out-of-memory failure.

Oracle Database's unique memory management facilities and sharing of read-only artifacts (such as bytecodes) enables `HelloWorld` to execute with a per-session incremental memory requirement of only 35 KB. More stateful server applications have a per-session incremental memory requirement of approximately 200 KB. Such applications must retain a significant amount of state in static variables across multiple calls. Refer to the discussion in the "End-of-Call Migration" section on page 2-37 for more information on understanding and controlling migration of static variables at end-of-call.

Initializing Pool Sizes within Database Templates

You can set the defaults for JAVA_POOL_SIZE and SHARED_POOL_SIZE in the database installation template. The Database Configuration Assistant (DBCA) allows you to modify these values within the Memory section, as shown below in Figure 10–2.

Figure 10–2 Configuring OracleJVM Memory Parameters



Java Pool Memory

Java pool memory is used in server memory for all session-specific Java code and data within the JVM. Java pool memory is used in different ways, depending on what mode the Oracle Database server is running in.

Java pool memory used within a dedicated server

The following is what constitutes the Java pool memory used within a dedicated server:

- The shared part of each Java class used per session

  This includes read-only memory, such as code vectors, and methods. In total, this can average about 4 KB-8 KB for each class.

- None of the per-session Java state of each session.

For a dedicated server, this is stored in UGA within the PGA—not within the SGA.

Under dedicated servers, the total required Java pool memory depends on the applications running and may range between 10 and 50 MB.

Java pool memory used within a shared server

The following is what constitutes the Java pool memory used within a shared server:

- The shared part of each Java class that is used per session

  This includes read-only memory, such as vectors, and methods. In total, this can average about 4 KB-8 KB for each class.

- Some of the UGA used for per-session state of each session is allocated from the Java pool memory within the SGA

  Because Java pool memory size is fixed, you must estimate the total requirement for your applications and multiply by the number of concurrent sessions the applications want to create to calculate the total amount of necessary Java pool memory. Each UGA grows and shrinks as necessary; however, all UGAs combined must be able to fit within the entire fixed Java pool space.

Under shared servers, this figure could be large. Java-intensive, multi-user benchmarks could require more than 100 MB.

Note: If you are compiling code on the server, rather than compiling on the client and loading to the server, you might need a bigger JAVA_POOL_SIZE than the default 20 MB.

Displaying Used Amounts of Java Pool Memory

You can find out how much of Java pool memory is being used by viewing the V$SGASTAT table. Its rows include pool, name, and bytes. Specifically, the last two rows show the amount of Java pool memory used and how much is free. The total of these two items equals the number of bytes that you configured in the database initialization file.

```
SVRMGR> select * from v$sgastat;

POOL        NAME                         BYTES
----------- -------------------------- ----------
```

```
                fixed_sga                    69424
                db_block_buffers           2048000
                log_buffer                  524288
shared pool     free memory                22887532
shared pool     miscellaneous                559420
shared pool     character set object          64080
shared pool     State objects                 98504
shared pool     message pool freequeue       231152
shared pool     PL/SQL DIANA                2275264
shared pool     db_files                      72496
shared pool     session heap                  59492
shared pool     joxlod: init P                 7108
shared pool     PLS non-lib hp                 2096
shared pool     joxlod: in ehe              4367524
shared pool     VIRTUAL CIRCUITS             162576
shared pool     joxlod: in phe              2726452
shared pool     long op statistics array      44000
shared pool     table definiti                  160
shared pool     KGK heap                       4372
shared pool     table columns                148336
shared pool     db_block_hash_buckets         48792
shared pool     dictionary cache            1948756
shared pool     fixed allocation callback       320
shared pool     SYSTEM PARAMETERS             63392
shared pool     joxlod: init s                 7020
shared pool     KQLS heap                   1570992
shared pool     library cache               6201988
shared pool     trigger inform                32876
shared pool     sql area                    7015432
shared pool     sessions                     211200
shared pool     KGFF heap                      1320
shared pool     joxs heap init                 4248
shared pool     PL/SQL MPCODE                405388
shared pool     event statistics per sess    339200
shared pool     db_block_buffers             136000
java pool       free memory                30261248
java pool       memory in use              19742720
37 rows selected.
```

Correcting Out of Memory Errors

If you run out of memory while loading classes, it can fail silently, leaving invalid classes in the database. Later, if you try to invoke or resolve any invalid classes, you will see `ClassNotFoundException` or `NoClassDefFoundException`

exceptions being thrown at runtime. You would get the same exceptions if you were to load corrupted class files. You should perform the following:

- Verify that the class was actually included in the set you are loading to the server. Many people have accidently forgotten to load just one class out of hundreds and spend considerable time chasing this down.
- Use the `loadjava -force` option to force the new class being loaded to replace the class already resident in the server.
- Use the `loadjava -resolve` option to attempt resolution of a class during the load process. This allows you to catch missing classes at load time, not run time.
- Double check the status of a newly loaded class by connecting to the database in the schema containing the class, and execute the following:

  ```
  select * from user_objects where object_name = dbms_java.shortname('');
  ```

  The STATUS field should be "VALID". If `loadjava` complains about memory problems or failures such as "connection lost", increase SHARED_POOL_SIZE and JAVA_POOL_SIZE, and try again.

11

Schema Object Tools

This chapter describes the schema object tools that you use in the Oracle Database Java environment. You run these tools from a UNIX shell or the Windows NT DOS prompt.

> **Note:** All names supplied within these tools are case sensitive. Thus, the schema, user name, and password will not be changed to upper case.

The following sections describe the schema object tools:

Schema Object Tool Overview

Unlike a conventional JVM, which compiles and loads Java files, the OracleJVM compiles and loads schema objects. The three kinds of Java schema objects are as follows:

- *Java class schema objects*, which correspond to Java class files.
- *Java source schema objects*, which correspond to Java source files.
- *Java resource schema objects*, which correspond to Java resource files.

To make a class file runnable by the OracleJVM, you use the `loadjava` tool to create a Java class schema object from the class file or the source file and load it into a schema. To make a resource file accessible to the OracleJVM, you use `loadjava` to create and load a Java resource schema object from the resource file.

The `dropjava` tool does the reverse of the `loadjava` tool; it deletes schema objects that correspond to Java files. You should always use `dropjava` to delete a Java schema object that was created with `loadjava`; dropping by means of SQL DDL commands will not update auxiliary data maintained by `loadjava` and `dropjava`.

What and When to Load

You must load resource files with `loadjava`. If you create `.class` files outside the database with a conventional compiler, then you must load them with `loadjava`. The alternative to loading class files is to load source files and let the Oracle Database system compile and manage the resulting class schema objects. In Oracle Database 10*g*, the most productive approach is to compile and debug most of your code outside the database, and then load the `.class` files. For a particular Java class, you can load either its `.class` file or its `.java` file, but not both.

The `loadjava` tool accepts JAR files that contain either source and resource files or class and resource files. You can load a class's source or its class file but not both. When you pass `loadjava` a JAR file or a ZIP file, `loadjava` opens the archive and loads its members individually; there are no JAR or ZIP schema objects. A file whose content has not changed since the last time it was loaded is not reloaded; therefore, there is little performance penalty for loading JARs. Loading JAR files is the simplest and most foolproof way to use `loadjava`.

It is illegal for two schema objects in the same schema to define the same class. For example, suppose `a.java` defines class `x` and you want to move the definition of `x` to `b.java`. If `a.java` has already been loaded, then `loadjava` will reject an attempt to load `b.java` (which also defines `x`). Instead, do either of the following:

- Drop `a.java`, load `b.java` (which defines `x`), then load the new `a.java` (which does not define `x`).
- Load the new `a.java` (which does not define `x`), then load `b.java` (which defines `x`).

Resolution

All Java classes contain references to other classes. A conventional JVM searches for classes in the directories, ZIP files, and JARs named in the CLASSPATH. The OracleJVM, by contrast, searches schemas for class schema objects. Each Oracle Database class has a *resolver spec*, which is the Oracle Database counterpart to the CLASSPATH. For a hypothetical class, `alpha`, its resolver spec is a list of schemas to search for classes that `alpha` uses. Notice that resolver specs are per-class, whereas in a classic JVM, CLASSPATH is global to all classes.

In addition to a resolver spec, each class schema object has a list of interclass reference bindings. Each reference list item contains a reference to another class and one of the following:

- the name of the class schema object to invoke when class uses the reference
- a code indicating whether the reference is unsatisfied; in other words, whether the referent schema object is known

An Oracle Database facility known as the *resolver* maintains reference lists. For each interclass reference in a class, the resolver searches the schemas specified by the class's resolver spec for a valid class schema object that satisfies the reference. If all references are resolved, the resolver marks the class *valid*. A class that has never been resolved, or has been resolved unsuccessfully, is marked *invalid*. A class that depends on a schema object that becomes invalid is also marked invalid at the time the first class is marked invalid; in other words, invalidation cascades upward from a class to the classes that use it and the classes that use them, and so on. When resolving a class that depends on an invalid class, the resolver first tries to resolve the referenced class, because it may be marked invalid only because it has never been resolved. The resolver does not re-resolve classes that are marked valid.

A class developer can direct `loadjava` to resolve classes or can defer resolution until run time. The resolver runs automatically when a class tries to load a class that is marked invalid. It is best to resolve before run time to learn of missing classes early; unsuccessful resolution at run time produces a "class not found" exception. Furthermore, run-time resolution can fail for the following reasons:

- lack of database resources if the tree of classes is very large

- deadlocks due to circular dependencies

The `loadjava` tool has two resolution modes:

1. Load-and-resolve (`-resolve` option): Loads all classes you specify on the command line, marks them invalid, and then resolves them. Use this mode when initially loading classes that refer to each other, and in general when reloading isolated classes as well. By loading all classes and then resolving them, this mode avoids the error message that occurs if a class refers to a class that will be loaded later in the execution of the command.
2. Load-then-resolve (no `-resolve` option): Resolves each class at runtime.

> **Note:** As with a Java compiler, `loadjava` resolves references to classes but not to resources; be sure to correctly load the resource files your classes need.

If you can, defer resolution until all classes have been loaded; this technique avoids the situation in which the resolver marks a class invalid because a class it uses has not yet been loaded.

Digest Table

The schema object digest table is an optimization that is usually invisible to developers. The digest table enables `loadjava` to skip files that have not changed since they were last loaded. This feature improves the performance of makefiles and scripts that invoke `loadjava` for collections of files, only some of which need to be reloaded. A reloaded archive file might also contain some files that have changed since they were last loaded and some that have not.

The `loadjava` tool detects unchanged files by maintaining a digest table in each schema. The digest table relates a file name to a *digest*, which is a shorthand representation of the file's content (a hash). Comparing digests computed for the same file at different times is a fast way to detect a change in the file's content—much faster than comparing every byte in the file. For each file it processes, `loadjava` computes a digest of the file's content and then looks up the file name in the digest table. If the digest table contains an entry for the file name that has the identical digest, then `loadjava` does not load the file, because a corresponding schema object exists and is up to date. If you invoke `loadjava` with the `-verbose` option, then it will show you the results of its digest table lookups.

Normally, the digest table is invisible to developers, because `loadjava` and `dropjava` keep the table synchronized with schema object additions, changes, and deletions. For this reason, always use `dropjava` to delete a schema object that was created with `loadjava`, even if you know how to drop a schema object using DDL. If the digest table becomes corrupted (`loadjava` does not update a schema object whose file has changed), use `loadjava`'s `-force` option to bypass the digest table lookup or delete all rows from the table, which is named `JAVA$CLASS$MD5$TABLE`.

Compilation

Loading a source file creates or updates a Java source schema object and invalidates the class schema object(s) previously derived from the source. If the class schema objects do not exist, `loadjava` creates them. The `loadjava` tool invalidates the old class schema objects because they were not compiled from the newly loaded source. Compilation of a newly loaded source, called for instance A, is automatically triggered by any of the following conditions:

- The resolver, working on class B, finds that it refers to class A, but class A is invalid.
- The compiler, compiling source B, finds that it refers to class A, but A is invalid.
- The class loader, trying to load class A for execution, finds that it is invalid.

To force compilation when you load a source file, use `loadjava -resolve`.

The compiler writes error messages to the predefined `USER_ERRORS` view; `loadjava` retrieves and displays the messages produced by its compiler invocations.

The compiler recognizes some options. There are two ways to specify options to the compiler. If you run `loadjava` with the `-resolve` option (which may trigger compilation), you can specify compiler options on the command line.

You can additionally specify persistent compiler options in a per-schema database table known as `JAVA$OPTIONS`, which you create as described shortly. You can use the `JAVA$OPTIONS` table for default compiler options, which you can override selectively with a `loadjava` command-line option.

Note: A command-line option both overrides and clears the matching entry in the `JAVA$OPTIONS` table.

A JAVA$OPTIONS row contains the names of source schema objects to which an option setting applies; you can use multiple rows to set the options differently for different source schema objects. The compiler looks up options in the JAVA$OPTIONS table when it has been invoked without a command line—that is, by the class loader—or when the command line does not specify an option. When compiling a source schema object for which there is neither a JAVA$OPTIONS entry nor a command-line value for an option, the compiler assumes a default value as follows:

- encoding = System.getProperty("file.encoding");
- debug = true: This option is equivalent to javac -g.

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

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

The name parameter is a Java package name, or a fully qualified class name, or the empty string. When the compiler searches the JAVA$OPTIONS table for the options to use for compiling a Java source schema object, it uses the row whose name most closely matches the schema object's fully qualified class name. A name whose value is the empty string matches any schema object name.

The option parameter is either 'online' or 'encoding'.

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

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

Table 11–1 represents a hypothetical JAVA$OPTIONS database table. Because the table has no entry for the encoding option, the compiler will use the default or the value specified on the command line. The online options shown in the table match schema object names as follows:

- The name a.b.c.d matches class and package names beginning with a.b.c.d; they will be compiled with online = true.

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

Table 11–1 Example JAVA$OPTIONS Table

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

loadjava

The `loadjava` tool creates schema objects from files and loads them into a schema. Schema objects can be created from Java source, class, and data files.

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

- `CREATE PROCEDURE` and `CREATE TABLE` privileges to load into your schema.
- `CREATE ANY PROCEDURE` and `CREATE ANY TABLE` privileges to load into another schema.
- `oracle.aurora.security.JServerPermission.loadLibraryInClass.<classname>`. See "Database Contents and OracleJVM Security" on page 9-2 for more information.

You can execute the `loadjava` tool either through the command line (as described below) or through the `loadjava` method contained within the `DBMS_JAVA` class. To execute within your Java application, do the following:

```
call dbms_java.loadjava('... options...');
```

where the options are the same as specified below. Separate each option with a blank. Do not separate the options with a comma. The only exception for this is the `-resolver` option, which contains blanks. For `-resolver`, specify all other options in the first input parameter, and the `-resolver` options in the second parameter. This is demonstrated below:

```
call dbms_java.loadjava('..options...', 'resolver_options');
```

Do not specify the following options, because they relate to the database connection for the `loadjava` command-line tool: `-thin, -oci, -user, -password`. The output is directed to stderr. Set `serveroutput` on, and call `dbms_java.set_output` as appropriate.

> **Note:** The `loadjava` tool is located in the `bin` subdirectory under $ORACLE_HOME.

Just before the `loadjava` tool exits, it checks whether the execution was successful. All failures are summarized preceded by the following header:

```
The following operations failed
```

Some conditions, such as losing the connection to the database, cause `loadjava` to terminate prematurely. There errors are printed with the following syntax:

```
exiting: <error_reason>
```

Syntax

```
loadjava {-user | -u} <user>/<password>[@<database>] [options]
<file>.java | <file>.class | <file>.jar | <file>.zip |
<file>.sqlj | <resourcefile> ...
  [-action]
  [-andresolve]
  [-casesensitivepub]
  [-cleargrants]
  [-debug]
  [-d | -definer]
  [-dirprefix <prefix>]
  [-e | -encoding <encoding_scheme>]
  [-fileout <file>]
  [-f | -force]
  [-genmissing]
  [-genmissingjar <jar_file>]
  [-g | -grant <user> [, <user>]...]
```

```
[-help]
[-jarasresource]
[-noaction]
[-nocasesensitivepub]
[-nocleargrants]
[-nodefiner]
[-nogrant]
[-norecursivejars]
[-noschema]
[-noserverside]
[-nosynonym]
[-nousage]
[-noverify]
[-o | -oci | oci8]
[-optionfile <file>]
[-optiontable <table_name>]
[-publish <package>]
[-pubmain <number>]
[-recursivejars]
[-r | -resolve]
[-R | -resolver "resolver_spec"]
[-resolveonly]
[-S | -schema <schema>]
[-stdout]
[-stoponerror]
[-s | -synonym]
[-tableschema <schema>]
[-t | -thin]
[-time]
[-unresolvedok]
[-v | -verbose]
```

Argument Summary

Table 11–2 summarizes the `loadjava` arguments. If you execute `loadjava` multiple times specifying the same files and different options, the options specified in the most recent invocation hold. There are two exceptions:

1. If `loadjava` does not load a file because it matches a digest table entry, most options on the command line have no effect on the schema object. The exceptions are `-grant` and `-resolve`, which are always obeyed. Use the `-force` option to direct `loadjava` to skip the digest table lookup.
2. The `-grant` option is cumulative; every user specified in every `loadjava` invocation for a given class in a given schema has the EXECUTE privilege.

Table 11–2 loadjava Argument Summary

Argument	Description
<filenames>	You can specify any number and combination of .java, .class, .ser, .jar .zip, and resource file name arguments, in any order.
-action	Perform all actions. This is the default behavior. This option can be used to override a -noaction option, which may be specified in an option file.
-andresolve	To be used in place of -resolve. This option causes files to be compiled or resolved at the time that they are loaded—rather than in a separate pass (as -resolve does). Resolving at the time of loading the class will not invalidate dependent classes. This option should be used only to replace classes that were previously loaded. If you changed only the code for existing methods within the class, you should use this option instead of the -resolve option.
-casesensitivepub	Publishing will create case sensitive names. Unless the names are already all upper case, it will usually require quoting the names in PL/SQL.
-cleargrants	The -grant option causes loadjava to grant execute privileges to classes, sources, and resources. However, it does not cause it to revoke any privileges. If -cleargrants is specified, loadjava will revoke any existing grants of execute privilege before it grants execute privilege to the users and roles specified by the -grant operand. For example, if the intent is to have execute privilege granted to SCOTT and only SCOTT, then the proper options are the following: -grant SCOTT -cleargrants.
-debug	Turns on SQL logging.
-definer	By default, class schema objects run with the privileges of their invoker. This option confers definer (the developer who invokes loadjava) privileges upon classes instead. (This option is conceptually similar to the UNIX setuid facility.)
-dirprefix <prefix>	For any files or JAR entries that start with <prefix>, this prefix will be deleted from the name before the name of the schema object is determined. For classes and sources, the name of the schema object is determined by their contents, so this option will only have an effect for resources.

Table 11–2 loadjava Argument Summary (Cont.)

Argument	Description
`-encoding`	Identifies the source file encoding for the compiler, overriding the matching value, if any, in the `JAVA$OPTIONS` table. Values are the same as for the `javac -encoding` option. If you do not specify an encoding on the command line or in a `JAVA$OPTIONS` table, the encoding is assumed to be "`System.getProperty("file.encoding");`". The `-encoding` option is relevant only when loading a source file.
`-fileout <file>`	Prints all message to the designated file.
`-force`	Forces files to be loaded, even if they match digest table entries.
`-genmissing`	Determines what classes and methods are referred to by the classes that `loadjava` is asked to process. Any classes not found in the database or file arguments are called "missing" classes. This option generates dummy definitions for missing classes containing all referred to methods. It then loads the generated classes into the database. This processing happens before the class resolution. Because detecting references from source is more difficult than detecting references from class files, and because source is not generally used for distributing libraries, `loadjava` will not attempt to do this processing for source files. The schema in which the missing classes are loaded will be the one specified by the `-user` command-line option, even when referring classes are created in some other schema. The created classes will be flagged as such so that tools can recognize them. In particular, this is needed, so that the verifier can recognize generated classes.
`-genmissingjar <jar_file>`	This option performs the same actions as `-genmissing`. In addition, it creates a JAR file, named `<jar_file>`, that contains the definitions of any generated classes.

Table 11–2 loadjava Argument Summary (Cont.)

Argument	Description
-grant	Grants the EXECUTE privilege on loaded classes to the listed users. (To call the methods of a class, users must have the EXECUTE privilege.) Any number and combination of user names can be specified, separated by commas but not spaces (-grant Bob,Betty not -grant Bob, Betty). Note: -grant is a "cumulative" option; users are added to the list of those with the EXECUTE privilege. To remove privileges, use the -cleargrants option. To grant the EXECUTE privilege on an object in someone else's schema requires that the original CREATE PROCEDURE privilege was granted with WITH GRANT options. **Note**: You must uppercase the schema name.
-help	Prints the usage message on how to use the loadjava tool and its options.
-jarasresource	Instead of unpacking the JAR file and loading each class within it, loads the whole JAR file into the schema as a resource.
-noaction	Take no action on the files. Actions include creating the schema objects, granting execute permissions, and so on. The normal use is within an option file to suppress creation of specific classes in a JAR. When used on the command-line (unless overridden in the option file), it will cause loadjava to ignore all files. Except that JAR files will still be examined to determine if they contain a META-INF/loadjava-options entry. If so, then the option file is processed. An -action option contained in the option file will override a -noaction option specified on the command-line.
-nocasesensitivepub	All lower case characters are converted to upper case. Transitions from lower to upper case characters will cause an underscore (_) to be inserted. For example, the method name IsXCharView becomes IS_XCHAR_VIEW. This command only modifies the -publish option.
-nocleargrants	Causes loadjava to omit revoking of execute privileges. This option can be used to override a -cleargrants option.
-nodefiner	Make the loaded classes (or classes derived from loaded sources) invoker's rights classes. This is the default behavior. This option can be used to override a -definer option.
-nogrant	Do not grant any execute privileges to the loaded classes. This is the default behavior. This option is used to override a -grant option.

Table 11–2 loadjava Argument Summary (Cont.)

Argument	Description
-norecursivejars	Treat JARs contained in other JARs as resources. This is the default behavior. This option can be used to override a -recursivejars option.
-noschema	Place the loaded classes, sources, and resources into the schema associated with the user specified in a -user option. This is the default behavior. It can be used to override a -schema option.
-nosynonym	Do not create a public synonym for the classes. This is the default behavior. This overrides a -synonym option.
-noserverside	Changes the behavior of dbms_java.loadjava to use a JDBC driver to access objects.Normally, server-side loadjava has a performance enhancement that it will modify the object directly—without using a JDBC driver to access the schemas. However, if you want the server-side to use a JDBC driver, use this option.
-nousage	Suppresses the usage message that is given if either no option is specified or if the -help option is specified.
-noverify	Causes the classes to be loaded without bytecode verification. You must be granted oracle.aurora.security.JServerPermission ("Verifier") to execute this option. To be effective, this option must be used in conjunction with -resolve.
-oci | -oci8	Directs loadjava to communicate with the database using the OCI JDBC driver. -oci and -thin are mutually exclusive; if neither is specified, -oci is used by default. Choosing -oci implies the syntax of the -user value. You do not need to provide the URL.
-optionfile <file>	A file can be provided with loadjava options. See **optionfile** discussion below for full information.

Table 11–2 loadjava Argument Summary (Cont.)

Argument	Description
`-optiontable <tablename>`	This option works like `-optionfile` except that the source for the patterns and options is a SQL table rather than a file. It is intended to allow people to specify the properties of classes persistently. No mechanism is provided for loading the table. The table name must contain three character columns named PATTERN, OPTION, and VALUE. The value of PATTERN is interpreted in the same way as a pattern in an option file. The other two columns specify a command-line option (including the dash) and for options that take an operand, the value of the operand. For options that do not take an operand, the VALUE column should be null. The rows are processed just like lines of an option file would be. To determine the options for a given schema object, the rows are examined (shortest pattern first) and for any that match the option is appended to the list of options. If two rows have the same pattern and contradictory options, such as `-synonym` and `-nosynonym`, it is unspecified which will prevail. If two rows have the same pattern and option columns, it is unspecified which VALUE will prevail.
`-publish <package>`	The `<package>` is created (or replaced) by `loadjava`. Wrappers for the eligible methods will be defined in this package. Through the use of option files, a single invocation of `loadjava` can be instructed to create more than one package. Each package will undergo the same name transformations as the methods. See the **publish** section below for more information.
`-pubmain <number>`	A special case applied to methods with a single argument, which is of type `java.lang.String`. Multiple variants of the SQL procedure or function will be created, each of which takes a different number of arguments of type `VARCHAR`. In particular, variants are created taking all numbers of arguments up to and including `<number>`. The default value is three. This option applies to main, as well as any method that has exactly one argument of type `java.lang.String`.
`-recursivejars`	Normally, if `loadjava` encounters an entry in a JAR with a `.jar` extension, it will load the entry as a resource. If this option is specified, then `loadjava` will process contained JARs as if they were top-level JARs. That is, it will read their entries and load classes, sources, and resources.
`-resolve`	Compiles (if necessary) and resolves external references in classes after all classes on the command line have been loaded. `I`f you do not specify `-resolve`, `loadjava` loads files but does not compile or resolve them.

Table 11–2 loadjava Argument Summary (Cont.)

Argument	Description
-resolver	Specifies an explicit resolver spec, which is bound to the newly loaded classes. If -resolver is not specified, the default resolver spec, which includes current user's schema and PUBLIC, is used. See "resolver" on page 11-20 for details.
-resolveonly	Causes loadjava to skip the initial creation step. It will still perform grants, resolves, create synonyms, and so on.
-schema	Designates the schema where schema objects are created. If not specified, the -user schema is used. To create a schema object in a schema that is not your own, you must have the CREATE PROCEDURE or CREATE ANY PROCEDURE privilege. You must have CREATE TABLE or CREATE ANY TABLE privilege. Finally, you must have the JServerPermission loadLibraryInClass for the class.
-stdout	Causes the output to be directed to stdout, rather than to stderr.
-stoponerror	Normally, if an error occurs while loadjava is processing files, it will issue a message and continue to process other classes. This option stops when an error occurs. In addition, it reports all errors that apply to Java objects and are contained in the USER_ERROR table of the schema in which classes are being loaded. Except that is does not report ORA-29524 errors. These are errors that are generated when a class cannot be resolved because a referred to class could not be resolved. Thus, these errors are a secondary effect of whatever caused a referred to class to be unresolved. This usually makes it easy to pinpoint the underlying cause of the failure.
-synonym	Creates a PUBLIC synonym for loaded 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 -synonym is specified for source files, classes compiled from the source files are treated as if they had been loaded with -synonym.
-tableschema <schema>	Creates the loadjava internal tables within this specified schema, rather than in the Java file destination schema.
-thin	Directs loadjava to communicate with the database using the thin JDBC driver. -oci and -thin are mutually exclusive; if neither is specified, then -oci is used by default. Choosing -thin implies the syntax of the -user value. You do need to specify the appropriate URL through the -user option.
-time	Prints a timestamp on every message.

Table 11–2 loadjava Argument Summary (Cont.)

Argument	Description
-unresolvedok	When combined with -resolve, will ignore unresolved errors.
-user	Specifies a user, password, and database connect string; the files will be loaded into this database instance. The argument has the form <username>/<password>[@<database>].
-verbose	Directs loadjava to print detailed status messages while running. Use -verbose to learn when loadjava does not load a file because it matches a digest table entry.

Argument Details

This section describes the details of loadjava arguments whose behavior is more complex than the summary descriptions contained in Table 11–2.

File Names

You can specify as many .class, .java, .jar, .zip, and resource files as you like, in any order. If you specify a JAR or ZIP file, then loadjava processes the files in the JAR or ZIP; there is no JAR or ZIP schema object. If a JAR or ZIP contains a JAR or ZIP, loadjava does not process them.

The best way to load files is to put them in a JAR or ZIP and then load the archive. Loading archives avoids the resource schema object naming complications described later in this section. If you have a JAR or ZIP that works with the JDK, then you can be sure that loading it with loadjava will also work, without having to learn anything about resource schema object naming.

Schema object names are slightly different from file names, and loadjava names different types of schema objects differently. Because class files are self-identifying (they contain their names), loadjava's mapping of class file names to schema object names is invisible to developers. Source file name mapping is also invisible to developers; loadjava gives the schema object the fully qualified name of the first class defined in the file. JAR and ZIP files also contain the names of their files; however, resource files are not self identifying. loadjava generates Java resource schema object names from the *literal* names you supply as arguments (or the literal names in a JAR or ZIP file). Because running classes use resource schema objects, it is important that you specify resource file names correctly on the command line, and the correct specification is not always intuitive. The surefire way to load individual resource files correctly is:

Run `loadjava` from the top of the package tree and specify resource file names relative to that directory. (The "top of the package tree" is the directory you would name in a Java CLASSPATH list.)

If you do not want to follow this rule, observe the details of resource file naming that follow. When you load a resource file, `loadjava` generates the resource schema object name from the resource file name *as literally specified on the command line*. Suppose, for example you type:

```
% cd /home/scott/javastuff
% loadjava options alpha/beta/x.properties
% loadjava options /home/scott/javastuff/alpha/beta/x.properties
```

Although you have specified the same file with a relative and an absolute path name, `loadjava` creates *two* schema objects, one called `alpha/beta/x.properties`, the other `ROOT/home/scott/javastuff/alpha/beta/x.properties`. (`loadjava` inserts `ROOT` because schema object names cannot begin with the "`/`" character; however, that is an implementation detail that is unimportant to developers.) The important point is that a resource schema object's name is generated from the file name *as entered*.

Classes can refer to resource files relatively (for example, `b.properties`) or absolutely (for example, `/a/b.properties`). To ensure that `loadjava` and the class loader use the same name for a schema object, follow this rule when loading resource files:

Enter the name on the command line that the class passes to `getResource()` or `getResourceAsString()`.

Instead of remembering whether classes use relative or absolute resource names and changing directories so that you can enter the correct name on the command line, you can load resource files in a JAR as follows:

```
% cd /home/scott/javastuff
% jar -cf alpharesources.jar alpha/*.properties
% loadjava options alpharesources.jar
```

Or, to simplify further, put both the class and resource files in a JAR, which makes the following invocations equivalent:

```
% loadjava options alpha.jar
% loadjava options /home/scott/javastuff/alpha.jar
```

The two `loadjava` commands in this example make the point that you can use any path name to load the contents of a JAR file. Even if you did execute the redundant commands shown above, `loadjava` would realize from the digest table that it did not need to load the files twice. That means that reloading JAR files is not as time-consuming as it might seem, even when few files have changed between `loadjava` invocations.

definer

`{-definer | -d}`

The `-definer` option is identical to definer's rights in stored procedures and is conceptually similar to the UNIX `setuid` facility; however, whereas `setuid` applies to a complete program, you can apply `-definer` class by class. Moreover, different definers may have different privileges. Because an application may consist of many classes, you must apply `-definer` with care to achieve the results desired, namely classes that run with the privileges they need,0 but no more. For more information on definer's rights, see "Controlling the Current User" on page 2-21.

noverify

`[-noverify]`

Causes the classes to be loaded without bytecode verification. You must be granted `oracle.aurora.security.JServerPermission(Verifier)` to execute this option. In addition, this option must be used in conjunction with `-r`.

The verifier ensures that incorrectly formed Java binaries cannot be loaded for execution in the server. If you know that the JAR or classes you are loading are valid, use of this option will speed up the `loadjava` process. Some Oracle Database-specific optimizations for interpreted performance are put in place during the verification process. Thus, interpreted performance of your application may be adversely affected by using this option.

optionfile

`[-optionfile <file>]`

A file can be provided with `loadjava` options. This `<file>` is read and processed by `loadjava` before any other `loadjava` options are processed. This `<file>` may contain one or more lines, each of which contains a pattern and a sequence of options. Each line must be terminated by a newline character (`\n`). For each file (or JAR entry) that is processed by `loadjava`, the long name of the schema object that is going to be created (typically, the name of the class with a dot "." replaced by a slash "/") is checked against the patterns. Patterns can end in a wildcard (*) to indicate an arbitrary sequence of characters; otherwise, they must match the name exactly. Options to be applied to matching Java schema objects are supplied on the rest of the line. Options are appended to the command-line options, they do not

replace them. In case more than one line matches a name, the matching rows are sorted by length of pattern, with the shortest first, and the options from each row are appended. In general, `loadjava` options are not cumulative. Rather, later options override earlier ones. This means that an option specified on a line with a longer pattern will override a line with a shorter pattern.

This file is parsed by a `java.io.StreamTokenizer`.

Java comments (both `/* */` and `//`) are allowed. A line comment begins with a #. Empty lines are ignored. The quote character is a double quote ("). That is, options containing spaces (common in `-resolver` options, for example) should be surrounded by double quotes. Certain options, such as `-user` or `-verbose`, affect the overall processing of `loadjava` and not the actions performed for individual Java schema objects. Such options are ignored if they appear in an option file.

As an aid in packaging applications, `loadjava` looks for an entry named `META-INF/loadjava-options` in each JAR it processes. If it finds such an entry, it treats it as an options file that is applied for all other entries in the option file. However, `loadjava` does some processing on entries in the order in which they occur in the JAR.

In case it has partially processed entities before it processes the `META-INF/loadjava-options`, the `loadjava` tool will attempt to patch up the schema object to conform to the applicable options. For example, by altering classes that were created with invoker's rights when they should have been created with definer's rights. The fix for `-noaction` will be to drop the created schema object. This will yield the correct effect except that if a schema object existed before loadjava started, it will have been dropped.

publish

```
[-publish <package>]
[-pubmain <number>]
```

The publishing options cause `loadjava` to create PL/SQL wrappers for methods contained in the processed classes. Typically, a user wants to publish wrappers for only a few classes in a JAR. These options are most useful when specified in an option file.

To be eligible for publication, the method must satisfy the following:

1. The method must be a member of a public class.
2. The method must itself be declared public and static.
3. The method signature must be "mappable", which is defined in the following rules:

- Java arithmetic types (`byte`, `int`, `long`, `float`, `double`) as arguments and return types are mapped to NUMBER.
- `char` as an argument and return type is mapped to VARCHAR.
- `java.lang.String` as an argument and return type is mapped to VARCHAR.
- If the only argument of the method has type `java.lang.String`, special rules apply, as listed in the -`pubstring` option description.
- If the return type is void, then a procedure is created.
- If the return type is arithmetic, `char`, or `java.lang.String`, then a function is created and its return type is as specified in an earlier rule.

Methods that take arguments or return types that are not covered by the above rules are not eligible. No provision is made for OUT, IN-OUT SQL arguments, OBJECT types, or for many other SQL features.

resolve

```
{-resolve | -r}
```

Use `-resolve` to force `loadjava` to compile (if necessary) and resolve a class that has previously been loaded. It is not necessary to specify `-force`, because resolution is performed after, and independently of, loading.

resolver

```
{-resolver | -R} "resolver spec"
```

This option associates an explicit resolver spec with the class schema objects that `loadjava` creates or replaces.

A resolver spec consists of one or more items, each of which consists of a *name spec* and a *schema spec* expressed in the following syntax:

```
"((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 fully qualified Java class name, or a package name whose final element is the wildcard character "`*`", or (unlike an imported package name) simply the wildcard character "`*`"; however, the elements of a name spec must be separated by "`/`" characters, not periods. For example, the name spec `a/b/*` matches all classes whose names begin with `a.b`. The special name `*` matches all class names.
- A schema spec can be a schema name or the wildcard character "`-`". The wildcard does not identify a schema but directs the resolve operation to not

mark a class invalid because a reference to a matching name cannot be resolved. (Without a "-" wildcard in a resolver spec, an unresolved reference in the class makes the class invalid and produces an error message.) Use a "-" wildcard when you must test a class that refers to a class you cannot or do not want to load; for example, GUI classes that a class refers to but does not call because when run in the server there is no GUI.

The resolution operation interprets a resolver spec item as follows:

When looking for a schema object whose name matches the name spec, look in the schema named by the partner schema spec.

The resolution operation searches schemas in the order in which the resolver spec lists them. For example,

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

means the following:

Search for any reference first in SCOTT and then in PUBLIC. If a reference is not resolved, then mark the referring class invalid and display an error message; in other words, call attention to missing classes.

The following example:

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

means the following:

Search for any reference first in SCOTT and then in PUBLIC. If the reference is not found, and is to a class in the package my.gui then mark the referring class valid, and do not display an error; in other words, ignore missing classes in this package. If the reference is not found and is not to a class in my.gui, then mark the referring class invalid and produce an error message.

user

```
{-user | -u} <user>/<password>[@<database>]
```

By default, `loadjava` loads into the login schema specified by the `-user` option. Use the `-schema` option to specify a different schema to load into. This does not involve a login into that schema, but does require that you have sufficient permissions to alter it.

The permissible forms of `@<database>` depend on whether you specify `-oci` or `-thin`; `-oci` is the default.

- `-oci`: `@<database>` is optional; if you do not specify, `loadjava` uses the user's default database. If specified, `<database>` can be a TNS name or a Oracle Net Services name-value list.
- `-thin`: `@<database>` is required. The format is `<host>:<lport>:<SID>`.
  - `<host>` is the name of the machine running the database.
  - `<lport>` is the listener port that has been configured to listen for Oracle Net Services connections; in a default installation, it is 5521.
  - `<SID>` is the database instance identifier; in a default installation it is `ORCL`.

Here are examples of `loadjava` commands:

- Connect to the default database with the default OCI driver, load the files in a JAR into the TEST schema, then resolve them.

```
loadjava -u joe/shmoe -resolve -schema TEST ServerObjects.jar
```

- Connect with the thin driver, load a class and a resource file, and resolve each class:

```
loadjava -thin -u SCOTT/TIGER@dbhost:5521:orcl \
  -resolve alpha.class beta.props
```

- Add Betty and Bob to the users who can execute `alpha.class`:

```
loadjava -thin -schema test -u SCOTT/TIGER@localhost:5521:orcl \
- -grant BETTY,BOB alpha.class
```

dropjava

The `dropjava` tool is the converse of `loadjava`. It transforms command-line file names and JAR or ZIP file contents to schema object names, then drops the schema objects and deletes their corresponding digest table rows. You can enter `.java`, `.class`, `.ser`, `.zip`, `.jar`, and resource file names on the command line in any order.

Alternatively, you can specify a schema object name (full name, not short name) directly to `dropjava`. A command-line argument that does not end in `.jar`, `.zip`, `.class`, `.java` is presumed to be a schema object name. If you specify a schema object name that applies to multiple schema objects (such as a source schema object `Foo` and a class schema object `Foo`), all will be removed.

Dropping a class invalidates classes that depend on it, recursively cascading upwards. Dropping a source drops classes derived from it.

Note: You must remove Java schema objects in the same way that you first loaded them. If you translate on a client and load classes and resources directly, run `dropjava` on the same classes and resources.

You can execute the `dropjava` tool either through the command line (as described below) or through the `dropjava` method contained within the `DBMS_JAVA` class. To execute within your Java application, do the following:

```
call dbms_java.dropjava('... options...');
```

where the options are the same as specified below. Separate each option with a blank. Do not separate the options with a comma. The only exception for this is the `-user` option. The connection is always made to the current session, so you cannot specify another user name through the `-user` option.

For `-resolver`, you should specify all other options first, a comma, then the `-resolver` option with its definition. Do not specify the following options, because they relate to the database connection for the `loadjava` command-line tool: `-thin, -oci, -user, -password`. The output is directed to stderr. Set `serveroutput` on and call `dbms_java.set_output` as appropriate.

Syntax

```
dropjava [options] {<file>.java | <file>.class | file.sqlj |
<file>.jar | <file.zip> | <resourcefile>} ...
  -u | -user <user>/<password>[@<database>]
  [-genmissingjar <JARfile>]
  [-jarasresource]
  [-noserverside]
  [-o | -oci | -oci8]
  [-optionfile <file>]
  [-optiontable <table_name>]
  [-S | -schema <schema>]
  [ -stdout ]
  [-s | -synonym]
  [-t | -thin]
  [-time]
  [-v | -verbose]
```

Argument Summary

Table 11–3 summarizes the `dropjava` arguments.

Table 11–3 dropjava Argument Summary

Argument	Description
`-user`	Specifies a user, password, and optional database connect string; the files will be dropped from this database instance.
`<filenames>`	You can specify any number and combination of `.java`, `.class`, `.sqlj`, `.ser`, `.jar`, `.zip`, and resource file names, in any order.
`-genmissingjar <file>`	`dropjava` treats the operand of this option as a file to be processed.
`-jarasresource`	Drops the whole JAR file, which was previously loaded as a resource.
`-noserverside`	Changes the behavior of the server-side `dropjava` tool to use a JDBC driver to access shemas.Normally, server-side `dropjava` has a performance enhancement that it will modify the schema directly—without using a JDBC driver to access the schemas. However, if you want the server-side to use a JDBC driver, use this option.
`-oci | -oci8`	Directs `dropjava` to connect with the database using the OCI JDBC driver. `-oci` and `-thin` are mutually exclusive; if neither is specified, then `-oci` is used by default. Choosing `-oci` implies the form of the `-user` value.
`-optionfile <file>`	This has the same usage as for `loadjava`.
`-optiontable <table>`	This has the same usage as for `loadjava`.
`-schema`	Designates the schema from which schema objects are dropped. If not specified, the logon schema is used. To drop a schema object from a schema that is not your own, you need the DROP ANY PROCEDURE and UPDATE ANY TABLE privileges.
`-stdout`	Causes the output to be directed to `stdout`, rather than to `stderr`.
`-synonym`	Drops a PUBLIC synonym that was created with `loadjava`.
`-thin`	Directs `dropjava` to communicate with the database using the thin JDBC driver. `-oci` and `-thin` are mutually exclusive; if neither is specified, then `-oci` is used by default. Choosing `-thin` implies the form of the `-user` value.
`-time`	Prints a timestamp on every message.
`-verbose`	Directs `dropjava` to emit detailed status messages while running.

Argument Details

File Names

dropjava interprets most file names as loadjava does:

- .class files: dropjava finds the class name in the file and drops the corresponding schema object.
- .java and .sqlj files: dropjava finds the first class name in the file and drops the corresponding schema object.
- .jar and .zip files: dropjava processes the archived file names as if they had been entered on the command line.

If a file name has another extension or no extension, then dropjava interprets the file name as a schema object name and drops all source, class, and resource objects that match the name. For example, the hypothetical file name alpha drops whichever of the following exists: the source schema object named alpha, the class schema object named alpha, and the resource schema object named alpha. If the file name begins with the "/" character, then dropjava inserts ROOT to the schema object name.

If dropjava encounters a file name that does not match a schema object, it displays a message and processes the remaining file names.

user

{-user | -u} <user>/<password>[@<database>]

The permissible forms of @<database> depend on whether you specify -oci or -thin; -oci is the default.

- -oci: @<database> is optional; if you do not specify, then dropjava uses the user's default database. If specified, then <database> can be a TNS name or a Oracle Net Services name-value list.
- -thin: @<database> is required. The format is <host>:<lport>:<SID>.
  - <host> is the name of the machine running the database.
  - <lport> is the listener port that has been configured to listen for Oracle Net Services connections; in a default installation, it is 5521.
  - <SID> is the database instance identifier; in a default installation, it is ORCL.

Here are some dropjava examples.

- Drop all schema objects in schema `TEST` in the default database that were loaded from `ServerObjects.jar`:

  ```
  dropjava -u SCOTT/TIGER -schema TEST ServerObjects.jar
  ```

- Connect with the thin driver, then drop a class and a resource file from the user's schema:

  ```
  dropjava -thin -u SCOTT/TIGER@dbhost:5521:orcl alpha.class beta.props
  ```

Dropping Resources

Care must be taken if you are removing a resource that was loaded directly into the server. This includes profiles if you translated on the client without using the `-ser2class` option. When dropping source or class schema objects, or resource schema objects that were generated by the server-side SQLJ translator, the schema objects will be found according to the package specification in the applicable `.sqlj` source file. However, the fully qualified schema object name of a resource that was generated on the client and loaded directly into the server depends on path information in the `.jar` file or on the command line at the time you loaded it. If you use a `.jar` file to load resources and use the same `.jar` file to remove resources, there will be no problem. If, however, you use the command line to load resources, then you must be careful to specify the same path information when you run `dropjava` to remove the resources.

ojvmjava

The `ojvmjava` tool is an interactive interface to a database instance's session namespace. You specify database connection arguments when you start `ojvmjava`. It then presents you with a prompt to indicate that it is ready for commands.

The shell can launch an executable, that is, a class with a static `main()` method. Executables must have been loaded with `loadjava`.

Syntax

```
ojvmjava {-user <user>[/<password>@database] [options]
  [@<filename>]
  [-batch]
  [-c | -command <command> <args>]
  [-debug]
  [-d | -database <conn_string>]
  [-fileout <filename>]
  [-o | -oci | -oci8]
```

```
[-oschema <schema>]
[-t | -thin]
[-version | -v]
```

Argument Summary

Table 11–4 summarizes the `ojvmjava` command-line arguments.

Table 11–4 ojvmjava Argument Summary

Option	Description
`-user | -u`	Specifies user's name for connecting to the database. This name is case insensitive; the name will always be converted to upper case. If you provide the database information, the default syntax used is OCI. You can also specify the default database by the following option: `-user <user>/<password>@`
`-password | -p`	Specifies user's password for connecting to the database. This name case insensitive; the name will always be converted to upper case.
`@<filename>`	Specifies a script file that contains `ojvmjava` commands to be executed. See "Scripting ojvmjava Commands in the @<filename> Option" on page 11-28 for structure of the indicated file.
`-batch`	Disables all messages printed to the screen. No help messages or prompts will be printed. Only responses to entered commands are printed.
`-command`	Executes the desired command. If you do not want to run `ojvmjava` in interpretive mode, but only want to execute a single command, execute `ojvmjava` with the `-command` option followed by a string that contains the command and the arguments. Once the command executes, `ojvmjava` exits. The following executes the "`ls -lR`" command on the designated host: `ojvmjava -user/TIGER -command "java foo"`
`-debug`	Prints debugging information.
`-d | -database <conn_string>`	Provide a database connection string.
`-fileout <file>`	Redirect output to the provided file.
`-o | -oci | -oci8`	Use the JDBC OCI driver. The OCI driver is the default. This flag specifies the syntax used in either the @database or `-database` option.
`-oschema <schema>`	Use this schema for class lookup.

Table 11–4 ojvmjava Argument Summary (Cont.)

Option	Description
`-t | -thin`	Specifies that the database syntax used is for the JDBC Thin driver. The database connection string must be of the form <host>:<port>:<SID> or an Oracle Net Services Name-Value list.
`-verbose`	Print the connection information.
`-version`	Shows the version.

Example Here is a `ojvmjava` example.

Open a shell on the session namespace of the database `orcl` on listener port 2481 on host `dbserver`.

```
ojvmjava -thin -user SCOTT/TIGER@dbserver:2481:orcl
```

The `ojvmjava` commands span several different types of functionality, which are grouped as follows:

- ojvmjava Options—Describes the options for the `ojvmjava` command-line tool
- Shell Commands—Describes the commands that are used for manipulating and viewing contexts and objects in the namespace.

ojvmjava Options

- ojvmjava Tool Output Redirection
- Scripting ojvmjava Commands in the @<filename> Option

ojvmjava Tool Output Redirection You can specify that any output generated by the `ojvmjava` tool is put into a file by appending the "`&><filename>`" at the end of the command options. The following pipes all output to the `listDir` file:

```
ls -lR &>/tmp/listDir
```

Scripting ojvmjava Commands in the @<filename> Option This option designates a script file that contains one or more `ojvmjava` commands. The script file specified is located on the client. The `ojvmjava` tool reads in the file and then executes all commands on the designated server. Also, because the script file is executed on the server, any interaction with the operating system in the script file—such as redirecting output to a file or executing another script—will occur on the server. If you direct `ojvmjava` to execute another script file, this file must exist within `$ORACLE_HOME` directory on the server.

Type in the ojvmjava command followed by any options and any expected input arguments.

The script file contains any ojvmjava command followed by options and input parameters. The input parameters can be passed in on the ojvmjava command-line. The ojvmjava command processes all known ojvmjava options and then passes on any other options and arguments to the script file.

To access arguments within the commands in the script file, place &1...&n to denote the arguments. If all input parameters are passed into a single command, you can supply a the string "&*" to denote that all input parameters are to be passed to this command.

The following shows the contents of the script file, execShell:

```
chmod +x SCOTT nancy /alpha/beta/gamma
chown SCOTT /alpha/beta/gamma
java testhello &*
```

Because only two input arguments are expected, you can implement the java command input parameters as follows:

```
java testhello &1 &2
```

Note: You can also supply arguments to the -command option in the same manner. The following shows an example:

```
ojvmjava ... -command "cd &1" contexts
```

After processing all other options, the ojvmjava tool passes "contexts" in as the argument to the "cd" command.

To execute this file, do the following:

```
ojvmjava -user SCOTT -password TIGER -thin -database dbserver:2481:orcl \
        @execShell alpha beta
```

The ojvmjava processes all options that it knows about and passes along any other input parameters to be used by the commands that exist within the script file. In this example, the parameters, alpha and beta, are passed to the java command in the script file. Thus, the actual command executed is as follows:

```
java testhello alpha beta
```

You can add any comments in your script file with the hash symbol (#). The "#" symbol makes anything to the end of the line a comment, which is ignored by `ojvmjava`. For example:

```
#this whole line is ignored by ojvmjava
```

Shell Commands

The following shell commands behave similarly to their UNIX counterparts:

Shell Commands		
echo	exit Command	help Command
java Command	version Command	whoami

Each of these shell commands contains the following common options:

Table 11–5 ojvmjava Command Common Options

Option	Description
`-describe | -d`	Summarizes the tool's operation.
`-help | -h`	Summarizes the tool's syntax.
`-version`	Shows the version.

echo

Prints to stdout exactly what is indicated. This is used mostly in script files.

The syntax is as follows:

```
echo [<echo_string>] [<args>]
```

where *<echo_string>* is a string that contains the text you want written to the screen during the shell script invocation and *<args>* are input arguments from the user. For example, the following prints out a notification:

```
echo "Adding an owner to the schema" &1
```

If the input argument is "SCOTT", the output would be "Adding an owner to the schema SCOTT"

exit Command

The `exit` command terminates `ojvmjava`.

Syntax

```
exit
```

Here is an example:

Leave the shell:

```
$ exit
%
```

help Command

The `help` command summarizes the syntax of the shell commands. You can also use the `help` command to summarize the options for a particular command.

Syntax

```
help [<command>]
```

java Command

The `java` command is analogous to the JDK `java` command; it invokes a class's static `main()` method. The class must have been loaded with `loadjava`. (There is no point to publishing a class that will be invoked with the `java` command.) The `java` command provides a convenient way to test Java code that runs in the database. In particular, the command catches exceptions and redirects the class's standard output and standard error to the shell, which displays them as with any other command output. (The usual destination of standard out and standard error for Java classes executed in the database is one or more database server process trace files, which are inconvenient and may require DBA privileges to read.)

Syntax

```
java [-schema <schema>] <class> [arg1 ... argn]
```

Argument Summary

Table 11–6 summarizes the `java` arguments.

Table 11–6 java Argument Summary

Option	Description
`class`	Names the Java class schema object that is to be executed.
`-schema`	Names the schema containing the class to be executed; the default is the invoker's schema. The schema name is case sensitive.
`arg1 ... argn`	Arguments to the class's `main()` method.

Here is a `java` command example.

Say hello and display arguments:

```
package hello;
public class World {
    public World() {
        super();
    }
    public static void main(String[] argv) {
        System.out.println("Hello from the Oracle Database");
        if (argv.length != 0)
            System.out.println("You supplied " + argv.length + " arguments: ");
            for (int i = 0; i < argv.length; i++)
                System.out.println(" arg[" + i + "] : " + argv[i]);
    }
}
```

Compile, load, publish, and run the executable as follows, substituting your user ID, host, and port information as appropriate:

```
% javac hello/World.java
% loadjava -r -user SCOTT/TIGER@localhost:2481:orcl hello/World.class
% ojvmjava -user SCOTT -password TIGER -database localhost:2481:orcl
$ java testhello alpha beta
Hello from the Oracle Database
You supplied 2 arguments:
arg[0] : alpha
arg[1] : beta
```

version Command

The `version` command shows the version of the `ojvmjava` tool. You can also show the version of a specified command.

Syntax

```
version [options] [<command>]
```

Here is an example of the `version` command.

Display the shell's version:

```
$ version
1.0
```

whoami

Prints out the current user that logged into this session.

12

Database Web Services

This chapter provides and overview of database Web services and discusses how to call out to an existing Web service. For information on how to create a Web service, see the *Oracle9iAS Web Services Developer's Guide* and `http://otn.oracle.com/webservices`. For more information on database Web services, see `http://otn.oracle.com/tech/webservices/database.html`.

This chapter covers the following topics:

Database Web Services

Web services technology enables application-to-application interaction over the Web – regardless of platform, language, or data formats. The key ingredients, including XML, SOAP, WSDL, and UDDI, have been adopted across the entire software industry. Web services technology usually refers to services implemented and deployed in middle-tier application servers. However, in heterogeneous and disconnected environments, there is an increasing need to access stored procedures as well as data and metadata, through Web services interfaces. Database Web services technology is a database approach to Web services.

It works in two directions:

- accessing database resources as a Web service
- consuming external Web services from the database itself

Turning the Oracle database into a Web service provider leverages investment in Java stored procedures, PL/SQL packages, pre-defined SQL queries and DML. Conversely, consuming external Web services from the database itself, together with integration with the SQL engine, enables Enterprise Information Integration.

This chapter focuses on the advantages of opening up the Oracle Database, through PL/SQL packages and Java classes deployed within the database, to the world of Web services, using the Oracle Application Server and the Oracle database.

Refer to the *Oracle9iAS Web Services Developer's Guide* for information on:

- Testing and securing Web services.
- Using PL/SQL-specific legacy types and REF CURSORS.
- Writing static or dynamic Java clients to call Web Services.

This chapter also provides a general road map for database Web services, mapping out Web services support for additional database capabilities, such as SQL queries, DML statements, and Java Stored Procedures—through synchronous invocation.

See the *Oracle9iAS Web Services Developer's Guide* on how to create Web services and invoke them from clients. This chapter covers how to call out to Web services from within the database and how a Web service can call in to the database.

Using the Database as Service Provider for Web Services

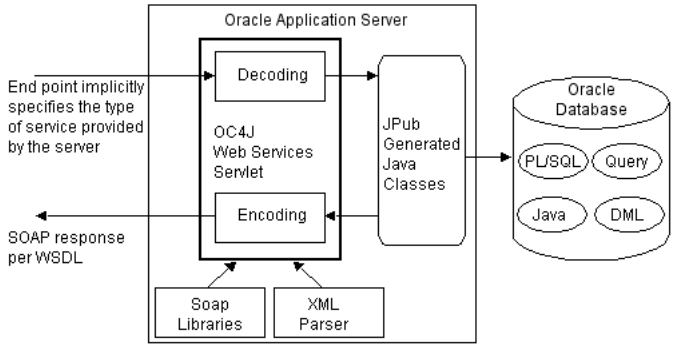
Web Services uses industry-standard mechanisms to provide easy access to remote content and applications, *regardless of the provider's platform, location, implementation, or data format*. Client applications can query and retrieve data from Oracle databases

and invoke stored procedures using standard web service protocols. There is no dependency on Oracle-specific database connectivity protocols. This approach is highly beneficial in heterogeneous, distributed, and non-connected environments.

You can call into the database from a Web Service, using the database as a service provider. This allows you to leverage existing or new SQL, PL/SQL, Java stored procedures, or Java classes within an Oracle database. You can access and manipulate database tables from a Web service client.

Use JPublisher to generate Java wrappers that correspond to database operations, then deploy the wrappers as Web services in Oracle AS. Figure 12–1 demonstrates how you use JPublisher to publish PL/SQL packages, SQL objects, collections, and packages as Java classes. Once published, these objects can be accessed by any Web service through an OC4J Web services servlet.

Figure 12–1 Web Services Calling In to the Database



How to Use

For directions on how to use JPublisher to publish your PL/SQL or SQL objects, collections or packages as Java classes, see the "What JPublisher Can Publish" section in the "Introduction to JPublisher" chapter of the *Oracle Database JPublisher User's Guide.*

See the *Oracle9iAS Web Services Developer's Guide* for more information on creating and using Web services.

Features of the Database as a Web Service Provider

Using the database as a Web Service provider offers the following features:

- Enhanced PL/SQL Web Services – Improves PL/SQL Web Services by extending Web Services support for additional PL/SQL types including CLOB, BLOB, XMLType, REfCursor, PL/SQL records and tables. This enables you to use most of your existing PL/SQL packages as Web Services.
- Java-in-the-database Web Services – Exposes existing Java classes deployed in the database as Web Services. Java classes implementing data-related services can be migrated between the middle tier and the database. Java portability results in database independence.
- SQL Query Web Services – Leverages warehousing or business intelligence queries, data monitoring queries, and any predefined SQL statements as web services.
- DML Web Services – Offers secure, persistent, transactional and scalable logging, auditing and tracking operations implemented via SQL DML, as web services. DML web services are implemented as atomic or group/batch insert, update, and delete operations.

JPublisher Support for Web Services Call-Ins to the Database

The following JPublisher features support Web services call-ins to code running in Oracle Database. Refer to the JPublisher *Oracle Database JPublisher User's Guide* for complete information.

- Generation of Java interfaces
- JPublisher styles and style files
- REF CURSOR returning and result set mapping
- Options to filter what JPublisher publishes
- Support for calling Java classes in the database without PL/SQL call specs
- Support for publishing SQL queries or DML statements
- Support for unique method names

Using the Database as Service Consumer for Web Services

You can extend a relational database's storage, indexing, and searching capabilities to include semistructured and nonstructured data (including Web Services) in addition to enabling federated data. By calling Web Services, the database can track, aggregate, refresh, and query dynamic data produced on-demand, such as stock prices, currency exchange rates, and weather information.

An example of using the database as a service consumer would be to call external Web Services from a predefined database job in order to obtain inventory information from multiple suppliers, then update your local inventory database. Another example is that of a Web Crawler: a database job can be scheduled to collate product and price information from a number of sources.

How to Use

The Web services client code is written in SQL, PL/SQL, or Java to run inside the database, which then invokes the external Web service. Figure 12–2 demonstrates how you can call out to a Web service from a Java client within the database by using one of the following methods:

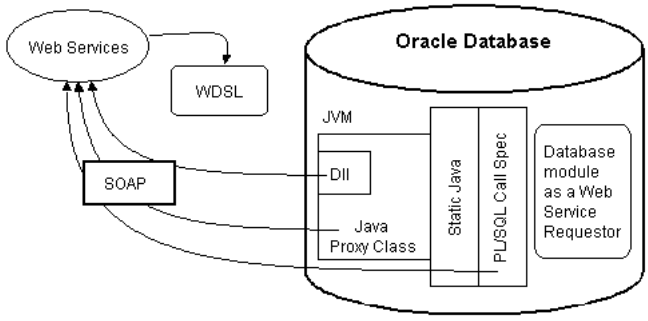
- SQL and PL/SQL call specs – Invoke a Web service through a user-defined function call (generated through JPublisher) either directly within a SQL statement or view or through a variable.
- Pure Java static proxy class – Use JPublisher to pre-generate a client proxy class. which uses JAX-RPC). This method simplifies the Web service invocation as the location of the service is already known without needing to look up the service in the UDDI registry. The client proxy class does all of the work to construct the SOAP request, including marshalling and unmarshalling parameters.
- Pure Java using DII (dynamic invocation interface) over JAX-RPC – Dynamic invocation provides the ability to construct the SOAP request and access the service without the client proxy.

Which method to use depends on if you want to execute from SQL or PL/SQL or from Java classes.

To call out to any Web service through PL/SQL, use the `UTL_DBWS` PL/SQL package. This package essentially uses the same APIs as the DII classes. See the PL/SQL Packages and Types Reference for a full description of this package.

You can use a Web Services Data Source to process the results from any Web service request as if it was a real database table.

Figure 12–2 Calling Web Services From Within the Database



Web Service Data Sources (Virtual Table Support)

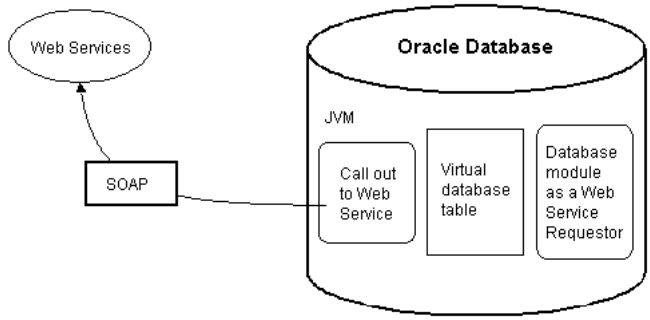
To access data (returned from single or multiple Web service invocations) through a database table, create a *virtual table* through a Web service data source. This table function allows you to query a set of returned rows as though it were a table.

The client invokes a Web service and the results are stored in a virtual table in the database. You can pass result sets from function to function, allowing you to set up a sequence of transformation without a table holding intermediate results. To reduce memory usage, you can return the result set rows a few at a time within a function.

By using Web services with the table function, you can manipulate a range of input values (from single or multiple Web services) as a real table. In the following example, the inner `SELECT` creates rows whose columns are used as arguments for invoking the `CALL_WS` Web service call-out. The table expression could be used in other SQL queries, for constructing views, and so on.

```
SELECT <some-columns>
FROM
  TABLE(WS_TABFUN(CURSOR(SELECT s FROM <some_table>))),
WHERE...
```

Figure 12–3 Storing Results from Request in a Virtual Table



Features of the Database as a Web Service Consumer

Using the database as a Web Service consumer offers the following features:

- Consuming Web Services form Java-in-the-database – Provides an easy-to-use interface for calling-out web services, thereby insulating developers from low-level SOAP programming. Java classes running in the database can simply and directly invoke external web services by using the previously loaded Java proxy or dynamic invocation.
- Consuming Web Services from SQL and PL/SQL – Allows any SQL-enabled tool or application to transparently and easily consume dynamic data from external web services. After Exposing Web Services methods as Java Stored PRocedures, A PL/SQL wrapper on top of a Java stored procedures hides all Java and SOAP programming details from the SQL client.
- Web Services Data Source – Enables Application and Data integration by turning external web service into an SQL data source, making external Web services appear as regular SQL tables. This table function represents the output of calling external web services and can be used in an SQL query.

Installation Requirements

Before generating any stubs or call specs, you must install the `UTL_DBWS` package in the database. This package includes both the Java and PL/SQL implementation necessary for facilitating the Web services functionality within the database.

Use the following script to install the Web Services Client Library:

```
$ORACLE_HOME/sqlj/lib/inctldbws.sql
```

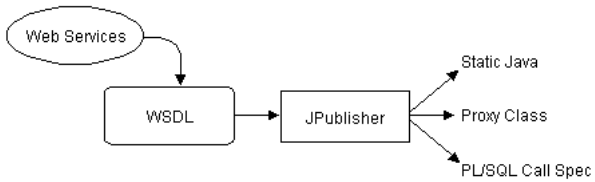
To remove the Web Services Client Library at a later time, use the following script:

```
$ORACLE_HOME/sqlj/lib/rmctldbws.sql
```

JPublisher Generation Overview

JPublisher can accept receive the WSDL from the Web Service and create the static java, proxy class, or PL/SQL call spec, as shown in Figure 12–4.

Figure 12–4 Creating Web Services Callout Stubs



This support is created through the following JPublisher key options:

- `-proxywsdl=`*url*

  Use this option to generate JAX-RPC static client proxies, given the WSDL document at the specified URL. This option also generates additional wrapper classes to expose instance methods as static methods, and generates PL/SQL wrappers. In all, it executes the following steps:

  1. Generates JAX-RPC client proxy classes.
  2. Generates wrapper classes to publish instance methods as static methods.
  3. Generates PL/SQL wrappers (call specs) for classes that must be accessible from PL/SQL.
  4. Loads generated code into the database.

  Note: The `-proxywsdl` option uses the `-proxyclasses` option behind the scenes for steps 2 and 3, and takes the `-proxyopts` setting as input.

  Once generated, your database client can access the Web service through PL/SQL using the call specs or through the JAX-RPC client proxy classes. The

PL/SQL wrappers use the static methods: your client would not normally access any Web service through the static method directly.

- `-httpproxy=`*proxy_url*

  Where WSDL is accessed through a firewall, use this option to specify a proxy URL to use in resolving the URL of the WSDL document.

- `-proxyclasses=`*class_list*

  For Web services, this option is used behind the scenes by the -proxywsdl option and is set automatically, as appropriate. In addition, you can use this option directly, for general purposes, any time you want to create PL/SQL wrappers for Java classes with static methods, and optionally to produce wrapper classes to expose instance methods as static methods.

  The `-proxyclasses` option takes the `-proxyopts` setting as input.

- `-proxyopts=`*wrapper_specifications*

  This option specifies JPublisher behavior in generating wrapper classes and PL/SQL wrappers—usually, but not necessarily, for Web services. For typical usage of the -`proxywsdl` option, the `-proxyopts` default setting is sufficient. In situations where you use the `-proxyclasses` option directly, you might want to use special `-proxyopts` settings.

See the "Additional Features" chapter in the *Oracle Database JPublisher User's Guide* for more information on how to use JPublisher.

Adjusting the Mapping of SQL Types

Although Oracle Application Server does not currently support LOB types, XMLTYPE, REF CURSORS, and OUT/IN OUT arguments (they will be addressed in future releases), you can use an alternative approach to expose PL/SQL methods and SQL types as Web services.

You can change JPublisher's default behavior to generate code that uses a user-provided subclass. For example, if you have a PL/SQL method that returns a REF CURSOR, JPublisher automatically maps the return type to `java.sql.ResultSet`. However, this `ResultSet` type cannot be published as a Web service. To solve this, simply create a new method that can return the result set in a Web service-supported format, such as:

```
public String [] readRefCursorArray(String arg1, Integer arg2)
  {java.sql.ResultSet rs = getRefCursor(arg1,arg2);
    ... create a String[] from rs and return it... }
```

Then create an interface that contains the exact methods to publish. You can use JPublisher to easily accomplish this mapping by using the following:

```
jpub -sql=MYAPP:MyAppBase:MyApp#MyAppInterf...
```

where:

- `MyApp` contains the method to return the result set.
- `MyAppInterf` is the interface that contains the method to publish.

After translating the code for your application, archive all the class files into a single JAR file and use the Web Services Assembler to create a deployable Web service EAR file. Refer to *Oracle Database JPublisher User's Guide* for more information.

Using the Native Java Interface

Oracle Database 10*g* introduces the *native Java interface*—new features for calls to server-side Java code. It is a simplified application integration: client-side (and middle-tier) Java applications can directly invoke Java in database without the need for defining a PL/SQL wrapper. Uses server-side Java class reflection capability.

In previous releases, calling Java stored procedures and functions from a database client required JDBC calls to associated PL/SQL wrappers. Each wrapper had to be manually published with a SQL signature and a Java implementation. This had the following disadvantages:

- The signatures permitted only Java types that had direct SQL equivalents.
- Exceptions issued in Java were not properly returned.

The JPublisher -java option provides functionality to avoid these disadvantages. To remedy the deficiencies of JDBC calls to associated PL/SQL wrappers, the `-java` option makes convenient use of an API for direct invocation of static Java methods. This functionality is also useful for Web services.

The functionality of the `-java` option mirrors that of the -sql option, creating a client-side Java stub class to access a server-side Java class, as opposed to creating a client-side Java class to access a server-side SQL object or PL/SQL package. The client-side stub class uses JPublisher code that mirrors the server-side class and includes the following features:

- Methods corresponding to the public static methods of the server class
- Two constructors: one that takes a JDBC connection and one that takes the JPublisher default connection context instance

At runtime, the stub class is instantiated with a JDBC connection. Calls to its methods result in calls to the corresponding methods of the server-side class. Any Java types used in these published methods must be primitive or serializable.

As an example, assume you want to call the following method in the server:

```
public String oracle.sqlj.checker.JdbcVersion.to_string();
```

Use the following `-java` setting:

```
-java=oracle.sqlj.checker.JdbcVersion
```

A

DBMS_JAVA Package

This chapter provides a description of the DBMS_JAVA package. Use these entry points to provide methods for accessing RDBMS functionality from Java.

FUNCTION longname (shortname VARCHAR2) RETURN VARCHAR2
Return the full name from a Java schema object. Because Java classes and methods can have names exceeding the maximum SQL identifier length, OracleJVM uses abbreviated names internally for SQL access. This function simply returns the original Java name for any (potentially) truncated name. An example of this function is to print the fully qualified name of classes that are invalid:

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

FUNCTION shortname (longname VARCHAR2) RETURN VARCHAR2
You can specify a full name to the database by using the `shortname()` routine of the `DBMS_JAVA` package, which takes a full name as input and returns the corresponding short name. This is useful when verifying that your classes loaded by querying the `USER_OBJECTS` view.

Refer to "Shortened Class Names" on page 2-26 for examples of these functions.

FUNCTION get_compiler_option(what VARCHAR2, optionName VARCHAR2)
PROCEDURE set_compiler_option(what VARCHAR2, optionName VARCHAR2,
value VARCHAR2)
PROCEDURE reset_compiler_option(what VARCHAR2, optionName VARCHAR2)
These three entry points control the options of the Oracle Database Java compiler that Oracle Database delivers. See "Compiling Java Classes" on page 2-9 for an example of these options.

FUNCTION resolver (name VARCHAR2, owner VARCHAR2, type VARCHAR2) RETURN VARCHAR2
This functions returns the resolver specification for a given object `name` in schema `owner` where object is of type `type`. The caller must have EXECUTE privilege and have access to the given object to use this call.

The *name* parameter is the shortened name for the object. Refer to `dbms_java.shortname()` for details.

The value of *type* is one of `SOURCE` or `CLASS`.

If there is an error then a null is returned. If the underlying object has changed then a `ObjectTypeChangedException` may be signaled.

To execute this function:

```
select dbms_java.resolver('tst', 'SCOTT', 'CLASS') from dual;
```

which would return:

```
DBMS_JAVA.RESOLVER('TST','SCOTT','CLASS')
-----------------------------------------
((* SCOTT)(* PUBLIC))
```

FUNCTION derivedFrom (name VARCHAR2, owner VARCHAR2, type VARCHAR2) RETURN VARCHAR2
This function returns the source name for object `name` in schema `owner` where object is of type `type`. The caller must have EXECUTE privilege and have access to the given object to use this call.

The *name* parameter (as well as the returned source) is the shortened name for the object. Refer to `dbms_java.shortname()` for details.

The value of *type* is of `SOURCE` or `CLASS`.

If there is an error then a null is returned. If the underlying object has changed then a `ObjectTypeChangedException` may be signaled.

The returned value will be null if the object was not compiled in the `ojvm`.

To execute this function:

```
select dbms_java.derivedFrom('tst', 'SCOTT', 'CLASS') from dual;
```

which would return:

```
DBMS_JAVA.DERIVEDFROM('TST','SCOTT','CLASS')
-----------------------------------------
tst
```

FUNCTION fixed_in_instance (name VARCHAR2, owner VARCHAR2, type VARCHAR2) RETURN NUMBER
This function returns the permanently kept status for object `name` in schema `owner` where the object is of type `type`. The caller must have EXECUTE privilege and have access to the given object to use this call.

The *name* parameter is the shortened name for the object. Refer to `dbms_java.shortname()` for details.

The value of *type* is of `RESOURCE`, `SOURCE`, `CLASS`, or `SHARED_DATA`.

The *return number* is either **0** (not kept) or **1** (kept).

To execute this function:

```
select dbms_java.fixed_in_instance('tst', 'SCOTT', 'CLASS') from dual;
```

which would return:

```
DBMS_JAVA.FIXED_IN_INSTANCE('TST','SCOTT','CLASS')
--------------------------------------------
0
```

or

```
select dbms_java.fixed_in_instance('java/lang/String', 'SYS', 'CLASS') from
dual;
```

which would return:

```
DBMS_JAVA.FIXED_IN_INSTANCE('JAVA/LANG/STRING','SYS','CLASS')
-------------------------------------------------------------
1
```

PROCEDURE set_output (buffersize NUMBER)
This procedure redirects the output of Java stored procedures and triggers to the `DBMS_OUTPUT` package. See "Redirecting Output on the Server" on page 3-8 for an example.

PROCEDURE start_debugging(host varchar2, port number, timeout number)
PROCEDURE stop_debugging
PROCEDURE restart_debugging(timeout number)
These entry points start and stop the debug agent when debugging. See "Debugging Server Applications" on page 3-7 for a description and example of these options.

```
procedure export_source(name varchar2, schema varchar2, blob BLOB)
procedure export_source(name varchar2, blob BLOB)
procedure export_source(name varchar2, CLOB clob)

procedure export_class(name varchar2, schema varchar2, blob BLOB)
procedure export_class(name varchar2, blob BLOB)

procedure export_resource(name varchar2, schema varchar2, blob BLOB)
procedure export_resource(name varchar2, blob BLOB)
procedure export_resource(name varchar2, schema varchar2, clob CLOB)
procedure export_resource(name varchar2, clob CLOB)
```

These entry points export a Java source, class, or resource schema object into an Oracle large object (LOB).

In all cases, *name* is the name of the Java schema object to be exported, *schema* is the name of the schema owning the object (if not supplied, then the current schema is used), and *blob* | *clob* is the large object that receives the specified Java schema object.

You cannot export a class into a CLOB, only into a BLOB. In addition, the internal representation of the source uses the UTF8 format, so that format is used to store the source in the BLOB as well.

```
PROCEDURE loadjava(options varchar2)
PROCEDURE loadjava(options varchar2, resolver varchar2)
PROCEDURE dropjava(options varchar2)
```

These procedures allow you to load and drop classes within the database using a call, rather than through the `loadjava` or `dropjava` command-line tools. To execute within your Java application, do the following:

```
call dbms_java.loadjava('... options...');
call dbms_java.dropjava('... options...');
```

The options are identical to those specified for the `loadjava` and `dropjava` command-line tools. Each option should be separated by a blank. Do not separate the options with a comma. The only exception to this is the `loadjava -resolver` option, which contains blanks. For `-resolver`, specify all other options first, separate these options by a comma, and then specify the `-resolver` options, as follows:

```
call dbms_java.loadjava('... options...', 'resolver_options');
```

Do not specify the following options, because they relate to the database connection for the `loadjava` command-line tool: `-thin, -oci, -user, -password`. The output is directed to `System.err`. The output typically goes to a trace file, but can be redirected.

For more information on the available options, see Chapter 11, "Schema Object Tools" for complete information on `loadjava`.

```
PROCEDURE grant_permission( grantee varchar2,
                    permission_type varchar2,
                    permission_name varchar2,
                    permission_action varchar2 )
PROCEDURE restrict_permission( grantee varchar2,
                    permission_type varchar2,
                    permission_name varchar2,
                    permission_action varchar2)
PROCEDURE grant_policy_permission( grantee varchar2,
                    permission_schema varchar2,
                    permission_type varchar2,
                    permission_name varchar2)
PROCEDURE revoke_permission(permission_schema varchar2,
                    permission_type varchar2,
                    permission_name varchar2,
                    permission_action varchar2)
PROCEDURE disable_permission(key number)
PROCEDURE enable_permission(key number)
PROCEDURE delete_permission(key number)
```

These entry points control the JVM permissions. See "Setting Permissions" on page 9-5 for a description and example of these options.

```
procedure set_preference(user varchar2,type varchar2, abspath varchar2, key
varchar2, value varchar2)
```

This procedure inserts or updates a row in the `SYS:java$prefs$` table as follows:

```
call dbms_java.set_preference('SCOTT','U','/my/package/method/three',
'windowsize','22:32');
```

The following table identifies the valid values for each parameter in this procedure.

Parameter	Description
user	The schema name to which to attach the preference. If the login schema is not `SYS`, then **user** must be the current login schema, or the insert will fail.
type	Select the type of preference: ■ **U** = user preference ■ **S** = System preference
abspath	The absolute path for the preference.
key	The key value to be used for the lookup or the value name
value	The value of the preference key.

Glossary

API

API stands for Application Programming Interface. As applied to Java, an API is a well-defined set of classes and methods that furnish a specific set of functionality to the Java programmer. JDBC are APIs for accessing SQL data.

Bytecodes

The set of single-byte, machine-independent instructions to which Java source code is compiled using the Java compiler.

Call Memory

The memory that the memory manager uses to allocate new objects.

CLASSPATH

The environment variable (or command line argument) that the JDK or JRE uses to specify the set of directory tree roots in which Java source, classes, and resources are located.

Context Switch

In a uniprocessor system, the current thread is interrupted by a higher priority thread or by some external event, and the system switches to a different thread. The choice of which thread to dispatch is usually made on a priority basis or based on how long a thread has been waiting.

Cooperative Multitasking

The programmer places calls to the `Thread.yield()` method in locations in the code where it is appropriate to suspend execution so that other threads can run.

This is quite error-prone because it is often difficult to assess the concurrent behavior of a program as it is being written.

Core Class Libraries

Generally, the Java packages delivered with the Sun Microsystems JDK, `java`.*. We also use this term to denote some `sun`.* packages.

Deadlock

The conflict state where two or more synchronized Java objects depend on locking each other, but cannot, because they themselves are locked by the dependent object. For example, object A tries to lock object B while object B is trying to lock object A. This situation is difficult to debug, because a preemptive Java virtual machine can neither detect nor prevent deadlock. Without deadlock detection, a deadlocked program simply hangs.

Dispatch

The system saves the state of the currently executing thread, restores the state of the thread to be executed, and branches to the stored program counter for the new thread, effectively continuing the new thread as if it had not been interrupted.

Driver

As used with JDBC, a layer of code that determines the low-level libraries employed to access SQL data and/or communicate across a network. The three JDBC drivers supported in OracleJVM are: Thin, OCI, and KPRB.

End-of-Call

Within your session, you may invoke Java many times. Each time you do this, end-of-call occurs at the point at which Java code execution completes. The memory manager migrates static variables to session space at end-of-call.

Garbage Collection

The popular name for the automatic storage reclamation facility provided by the Java virtual machine.

IDE

Integrated Development Environment. A Java IDE runs on a client workstation, providing a graphical user interface for access to the Java class library and development tools.

Java Schema Object

The term that Oracle Database uses to denote either Java source, binary, or resources when stored in the database. These three Java schema objects correspond to files under the JDK— `.java`, `.class`, or other files (such as `.properties` files) used in the JDK CLASSPATH.

JCK

Java Compatibility Kit. The set of Java classes that test a Java virtual machine and Java compiler's compliance with the Java standard. JCK releases correspond to the Sun Microsystems JDK releases, although in the case of Oracle Database, only the Java classes and not the virtual machine, are identical to the Sun Microsystems JDK.

JDBC

Java Database Connectivity. The standard Java classes that provide vendor-independent access to databases.

JDBC Driver

The vendor-specific layer of JDBC that provides access to a particular database. Oracle provides three JDBC drivers—Thin, OCI, and KPRB.

JDK

Java Development Kit. The Java virtual machine, together with the set of Java classes and tools that Sun Microsystems furnishes to support Java application and applet development. The JDK includes a Java compiler; the JRE does not.

JLS

Java Language Specification. This specification defines the syntax and semantics of the Java language.

JRE

Java Runtime Environment. The set of Java classes supporting a Java application or applet at runtime. The JRE classes are a subset of the JDK classes.

Lazy Initialization

A technique for initializing data, typically used in accessor methods. The technique checks to see if a field has been initialized (is non-null) before returning the initialized object to it. The overhead associated with the check is often small, especially in comparison to initializing a data structure that may never be accessed. You can employ this technique in conjunction with end-of-call processing to minimize session space overhead.

Object Graph

An object is said to reference the objects held in its fields. This collection of objects forms an object graph. The memory manager actually migrates the object graphs held in static variables; that is, it migrates not only the objects held in static fields, but the objects that those objects reference, and so on.

OracleJVM

Oracle's scalable Java server platform, composed of the Java virtual machine running within the Oracle Database server, the Java runtime environment and Oracle extensions.

Preemptive Multitasking

The operating system preempts, or takes control away from a thread, under certain conditions, such as when another thread of higher priority is ready to run, or when an external interrupt occurs, or when the current thread waits on an I/O operation, such as a socket accept or a file read. Some Java virtual machines implement a type of round-robin preemption by preempting the current thread on certain virtual machine instructions, such as backward branches, method calls, or other changes in control flow. For a Java virtual machine that maps Java threads to actual operating system threads, the preemption takes place in the operating system kernel, outside the control of the virtual machine. Although this yields decent parallelism, it complicates garbage collection and other virtual machine activities.

Process

An address space and one or more threads.

Session Memory

The memory that the memory manager uses to hold objects that survive past the end-of-call—those objects reachable from Java static variables within your session.

Strong Typing

In Java, the requirement that the class of each field and variable, and the return type of each method be explicitly declared.

Symmetric Multiprocessing (SMP)

The hardware has multiple processors, and the operating system maps threads to different processors, depending on their load and availability. This assumes that the Java virtual machine maps OS threads to Java threads. This mechanism provides true concurrency among the threads, but can lead to subtle programming errors and deadlock conflicts on synchronized objects.

System

Often used in discussion as the combination of the hardware, the operating system, and the Java virtual machine.

Thread

An execution context consisting of a set of registers, a program counter, and a stack.

Virtual Machine

A program that emulates the functionality of a traditional processor. A Java virtual machine must conform to the requirements of the Java Virtual Machine Specification.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W