Oracle® Database

Java Developer's Guide

10*g* Release 2 (10.2)

B14187-01

August 2006

This book describes how to develop, load, and run Java applications in Oracle Database.

ORACLE®

Oracle Database Java Developer's Guide, 10*g* Release 2 (10.2)

B14187-01

Primary Author: Venkatasubramaniam Iyer, Sheryl Maring, Rick Sapir, Michael Wiesenberg

Contributing Author: Brian Wright, Timothy Smith

Contributor: 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, David Unietis, Robert H Lee, Xuhua Li

Contents

2 Java Applications on Oracle Database

3 Calling Java Methods in Oracle Database

4 Java Installation and Configuration

5 Developing Java Stored Procedures

6 Publishing Java Classes With Call Specifications

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 Objects and Oracle JVM Utilities

12 Database Web Services

A DBMS_JAVA Package

Index

List of Figures

List of Tables

Preface

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

Java has emerged as the object-oriented programming language of choice. It provides platform independence and automated storage management techniques. It enables you to create applications and applets. Oracle Database provides support for developing and deploying Java applications.

This preface contains the following topics:

Intended Audience

The Oracle Database Java Developer's Guide is intended for both Java and non-Java developers. For PL/SQL developers who are not familiar with Java programming, this manual provides a brief overview of Java and object-oriented concepts. For both Java and PL/SQL developers, this manual discusses the following:

- How Java and Database concepts merge
- How to develop, load, and run Java stored procedures
- The Oracle Java virtual machine (JVM)
- Database concepts for managing Java objects in the database
- Oracle Database and Java security policies

To use this document, you need knowledge of Oracle Database, SQL, and PL/SQL. Prior knowledge of Java and object-oriented programming can be helpful.

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. Accessibility 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 more information, visit the Oracle Accessibility Program Web site at

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

Accessibility of Code Examples in Documentation

Screen readers 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, some screen readers 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.

TTY Access to Oracle Support Services

Oracle provides dedicated Text Telephone (TTY) access to Oracle Support Services within the United States of America 24 hours a day, seven days a week. For TTY support, call 800.446.2398.

Structure

This document contains the following chapters:

Chapter 1, "Introduction to Java in Oracle Database"

Gives an overview of how to develop, load, and run Java applications in Oracle Database.

Chapter 2, "Java Applications on Oracle Database"

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

Chapter 3, "Calling Java Methods in Oracle Database"

Gives and overview and examples of how to call Java stored procedures in the database.

Chapter 4, "Java Installation and Configuration"

Describes the procedure for installing and configuring the Oracle JVM.

Chapter 5, "Developing Java Stored Procedures"

Describes developing of Java stored procedures, which access the database.

Chapter 6, "Publishing Java Classes With Call Specifications"

Describes how to publish methods with call specifications, 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 application with stored procedures.

Chapter 10, "Security for Oracle Database Java Applications"

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

Chapter 9, "Oracle Database Java Application Performance"

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

Chapter 11, "Schema Objects and Oracle JVM Utilities"

Describes the schema objects used in the Oracle Database Java environment and the Oracle JVM utilities.

Chapter 12, "Database Web Services"

Describes database Web services and Web services call-outs.

Appendix A, "DBMS_JAVA Package"

Describes the `DBMS_JAVA` package.

Related Documents

For more information, refer to the following Oracle resources:

- *Oracle Database JDBC Developer's Guide and Reference*
- *Oracle Database JPublisher User's Guide*
- *Oracle Database SQLJ Developer's Guide and Reference*
- *Oracle Database Net Services Administrator's Guide*
- *Oracle Database Advanced Security Administrator's Guide*
- *Oracle Database Application Developer's Guide - Fundamentals*

Conventions

The following conventions are also used in this manual:

Convention	Meaning
. . .	Vertical ellipsis points in an example mean that information not directly related to the example has been omitted.
. . .	Horizontal ellipsis points in statements or commands mean that parts of the statement or command not directly related to the example have been omitted
boldface text	Boldface type in text indicates a term defined in the text, the glossary, or in both locations.
< >	Angle brackets enclose user-supplied names.
[ ]	Brackets enclose optional clauses from which you can choose one or none.

1

Introduction to Java in Oracle Database

Oracle Database provides support for developing, storing, and deploying Java applications. This chapter introduces the Java language to Oracle PL/SQL developers, who are accustomed to developing server-side applications that are integrated with SQL data. You can develop server-side Java applications that take advantage of the scalability and performance of Oracle Database.

This chapter contains the following sections:

Overview of Java

Java has emerged as the object-oriented programming language of choice. Some of the important concepts of Java include:

- Java virtual machine (JVM), which provides the fundamental basis for platform independence
- Automated storage management techniques, such as garbage collection
- Language syntax that is similar to that of the C language

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

This section covers the following topics:

Java and Object-Oriented Programming Terminology

The following terms are common in Java application development in the Oracle Database environment:

- Classes
- Objects
- Interfaces
- Encapsulation
- Inheritance
- Polymorphism

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

  Attributes are variables that are present in each object, or instance, of a particular class. Attributes within an instance are known as fields. Instance fields are analogous to the fields of a relational table row. The class defines the variables and the type of each variable. You can declare variables in Java as `static` and `public`, `private`, `protected`, or default access.

  Variables of a class that are declared as `static` are global and common to all instances of that class. Variables that are not declared as `static` are created within each instance of the class.

  The `public`, `private`, `protected`, and default access modifiers define the scope of the variable in the application. The Java Language Specification (JLS) defines the rules of visibility of data for all variables. These rules define under what circumstances you can access the data in these variables.

  In the example illustrated in Figure 1–1, the employee identifier is defined as `private`, indicating that other objects cannot access this attribute directly. In the example, objects can access the `id` attribute by calling the `getId()` method.

- Methods

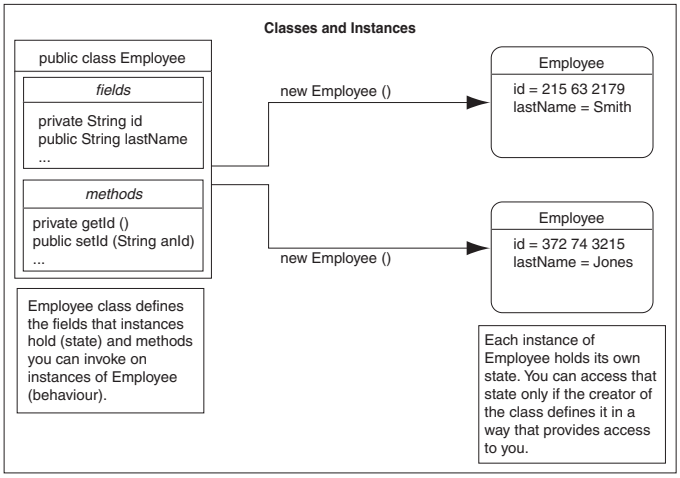
  Methods are blocks of code that perform specific tasks or actions. You can call the methods on an instance of the class. You can also call methods that are inherited by the class. Methods provide a way for instances to interact with other instances and the application. Similar to attributes, methods can be declared as `public`, `private`, `protected`, or default access.

Objects

A class needs to be instantiated before you can use the instance variables or attributes and methods. An object is an instance of a class and is analogous to a relational table row. The instance contains the attributes, which are known as its data or state, and the methods defined in the class.

Figure 1–1 shows an example of an `Employee` class defined with two attributes, `id`, which is the employee identifier, and `lastName`, which is the last name of the employee, and the `getId()` and `setId(String anId)` methods. The `id` attribute and the `getId()` method are `private`, and the `lastName` attribute and the `setId(String anId)` method are `public`.

Figure 1–1 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 to that particular employee. The example in Figure 1–1 shows two instances of the `Employee` class, one for the employee Smith and one for Jones. Each instance contains information relevant to the individual employee.

Interfaces

Java supports only single inheritance, that is, each class can inherit attributes and methods of only one class. If you need to inherit properties from more than one source, then Java provides the concept of interfaces, which is equivalent to multiple inheritance. Interfaces are similar to classes. However, they define only the signature of the methods and not their implementations. The methods that are declared in the interface are implemented in the classes. Multiple inheritance occurs when a class implements multiple interfaces.

Encapsulation

Encapsulation describes the ability of an object to hide its data and methods from the rest of the world and is one of the fundamental principles of object-oriented programming. In Java, a class encapsulates the attributes, which hold the state of an object, and the methods, which define the actions of the object. Encapsulation enables you to write reusable programs. It also enables you to restrict access only to those features of an object that are declared `public`. All other attributes and methods are `private` and can be used for internal object processing.

In the example illustrated in Figure 1–1, the `id` attribute is `private` and access to it is restricted to the object that defines it. Other objects can access this attribute using the `getId()` method. Using encapsulation, you can deny access to the `id` attribute either by declaring the `getId()` method as `private` or by not defining the `getId()` method.

Inheritance

Inheritance is an important feature of object-oriented programming languages. It enables classes to acquire properties of other classes. The class that inherits the properties is called a child class or subclass and the class from which the properties are inherited is called a parent class or superclass. This feature also helps in reusing an already defined code.

In the example illustrated in Figure 1–1, you can create a `FullTimeEmployee` class that inherits the properties of the `Employee` class. The properties inherited depend on the access modifiers declared for each attribute and method of the superclass.

Polymorphism

Polymorphism is the ability for different objects to respond differently to the same message. In object-oriented programming languages, you can define one or more methods with the same name. These methods can perform different actions and return different values.

In the example in Figure 1–1, assume 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:

- Full-time employees are eligible for a bonus.
- Non-exempt employees get overtime pay.

In procedural languages, you would write a `switch` statement, with the different possible cases defined, as follows:

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

If you add a new type of employee, then you must update the `switch` statement. In addition, if you modify the data structure, then you need to modify all `switch` statements that use it. In an object-oriented language, such as Java, you can implement a method, `compensationToDate()`, for each subclass of the `Employee` class, if it contains information beyond what is already defined in the `Employee` class. For example, you could implement the `compensationToDate()` method for a non-exempt employee, as follows:

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

For a full-time employee, the `compensationToDate()` method can be implemented as follows:

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

This common usage of the method name enables you to call methods of different classes and obtain the required results, without specifying the type of the employee. You do not have to write specific methods to handle full-time employees and part-time employees.

In addition, you could create a `Contractor` class that does not inherit properties from `Employee` and implements 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 or part-time employees or contractors, and add up the

values returned from calling the `compensationToDate()` method on each. You can safely make changes to the individual `compensationToDate()` methods or the classes with the knowledge that callers of the methods will work correctly.

Key Features of the Java Language

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

- Simplicity

  Java is simpler than most other languages that are used in creating 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, and it is also simple to write programs that move seamlessly across systems.

  See Also: "The JVM" on page 1-6

- Automatic storage management

  The JVM automatically performs all memory allocation and deallocation while the program is running. Java programmers cannot explicitly allocate memory for new objects or free memory for objects that are no longer referenced. Instead, they depend on the JVM to perform these operations. The process of freeing memory is known as garbage collection.

- Strong typing

  Before you use a variable, you must declare the type of the variable. Strong typing in Java 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 and not by value. As a result, the object identity is preserved. Java does not provide low level, direct access to pointers, thereby eliminating any possibility of 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 places them within a hierarchical structure that mirrors the domain namespace of the Internet. 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. The namespace approach of Java is flexible enough for Oracle to incorporate the concept of a schema for resolving class names in full compliance with the JLS.

- Security

The design of Java bytecodes and the JVM allow for built-in mechanisms to verify the security of Java binary code. Oracle Database is installed with an instance of Security Manager that, when combined with Oracle Database security, determines who can call any Java methods.

- Standards for connectivity to relational databases

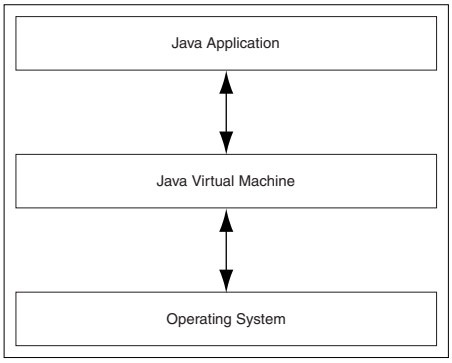
  Java Database Connectivity (JDBC) and SQLJ enable Java code to access and manipulate data in relational databases. Oracle provides drivers that allow vendor-independent, portable Java code to access the relational database.

The JVM

As with other high-level computer languages, the Java source compiles to low-level machine instructions. In Java, these instructions are known as bytecodes, because each instruction has a uniform size of one byte. Most other languages, such as C, compile to machine-specific instructions, such as instructions specific to an Intel or HP processor.

When compiled, the Java code gets converted to a standard, platform-independent set of bytecodes, which interacts with a JVM. The JVM is a separate program that is optimized for the specific platform on which you run your Java code. Figure 1–2 illustrates how Java can maintain platform independence. Each platform has a JVM installed that is specific to the operating system. The Java bytecodes get interpreted through the JVM into appropriate platform dependent actions.

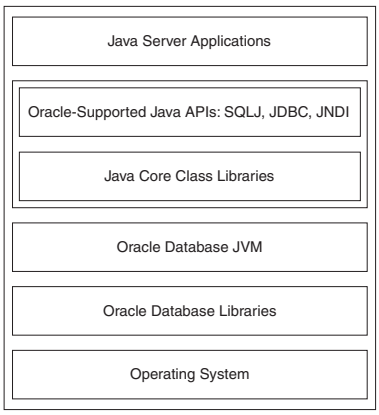
Figure 1–2 Java Component Structure



When you develop a Java application, 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. Basic language support is provided by the `java.lang` package, I/O support is provided by the `java.io` package, and network access is provided by the `java.net` package. Together, the JVM and core class libraries provide a platform on which Java programmers can develop applications, which will run successfully on any operating system that supports Java. This concept is what drives the "write once, run anywhere" idea of Java.

Figure 1–3 illustrates how Oracle Java applications reside on top of the Java core class libraries, which reside 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 interacting with the operating system.

Figure 1–3 Oracle Database Java Component Structure



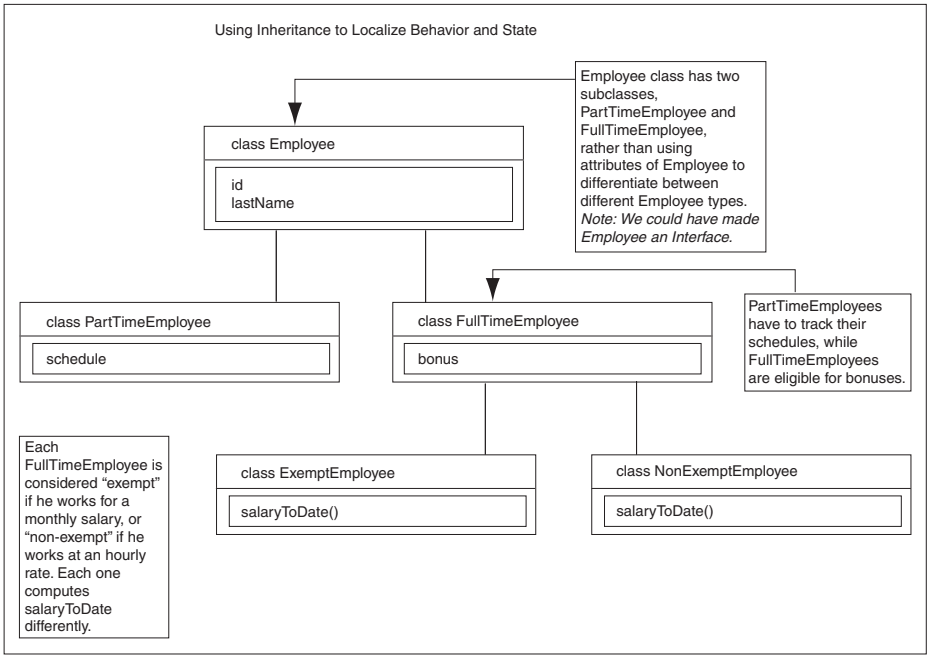
Sun Microsystems provides publicly available specifications for both the Java language and the JVM. The JLS defines the syntax and semantics, and the JVM specification defines the necessary low-level actions for the system that runs the application. In addition, Sun Microsystems provides a compatibility 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 Oracle JVM 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.

Java 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 call any method on an instance of Class B that was defined in either Class A or Class B. In the example, the `FullTimeEmployee` instance can call methods defined only in the `FullTimeEmployee` class and methods defined in 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 classes that define behavior and state where it makes sense in the hierarchy, yet make use of preexisting functionality in class libraries.

Using Java in Oracle Database

The only reason that you can write and load Java applications within the database is because it is a safe language with a lot of security features. 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. However, Java, because of its design, is a robust language to be used 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 the following challenges:

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

Java and RDBMS: A Robust Combination

Oracle Database provides Java applications with a dynamic data-processing engine that supports complex queries and different views of the same data. All client requests

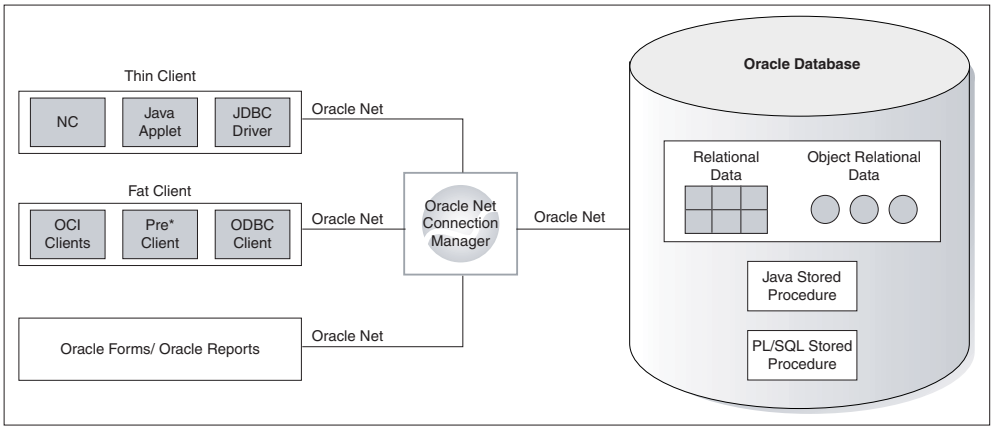
are assembled as data queries for immediate processing, and query results are generated on the fly.

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

Java and Oracle Database 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 1–5 shows a traditional two-tier, client/server configuration in which clients call Java stored procedures the same way they call PL/SQL stored procedures. The figure also shows how the Oracle Net Services Connection Manager can combine many network connections into a single database connection. This enables Oracle Database to support a large number of concurrent users.

Figure 1–5 Two-Tier Client/Server Configuration



Multithreading

Multithreading is one of the key scalability features of the Java language. 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 Oracle JVM uses the facilities of the database server to concurrently schedule the running of Java application for thousands of users. Although Oracle Database supports Java language-level threads required by the JLS and JCK, scalability will not increase by using threads within the scope of the database. By using the embedded scalability of the database, the need for writing multithreaded Java servers is eliminated.

You should use the facilities of the Oracle Database for scheduling users by writing single-threaded Java applications. The database can schedule processes between each application, and thus, you achieve scalability without having to manage threads. You can still write multithreaded Java applications, but multiple Java threads will not increase the performance of the server.

One complication multithreading creates is the interaction of threads and automated storage management or garbage collection. The garbage collector running in a generic JVM has no knowledge of which Java language threads are running or how the underlying operating system schedules them. The difference between a non-Oracle Database model and an Oracle JVM model is as follows:

- Non-Oracle Database model

  A single user maps to a single Java thread and a single garbage collector manages all garbage from all users. Different techniques typically deal with allocation and collection of objects of varying lifetimes and sizes. The result in a heavily multithreaded application is, at best, dependent upon operating system support for native threads, which can be unreliable and limited in scalability. High levels of scalability for such implementations have not been convincingly demonstrated.

- Oracle JVM model

  Even when thousands of users connect to the server and run the same Java code, each user experiences it as if he is running his own Java code on his own JVM. The responsibility of the Oracle JVM is to make use of operating system processes and threads and the scalable approach of the Oracle Database. As a result of this approach, the garbage collector of the Oracle JVM is more reliable and efficient because it never collects garbage from more than one user at any time.

See Also: "Threading in Oracle Database" on page 2-26

Automated Storage Management With Garbage Collection

Garbage collection is a major function of the automated storage management feature of Java, 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. However, garbage collection contributes to the overhead of program execution speed and footprint.

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

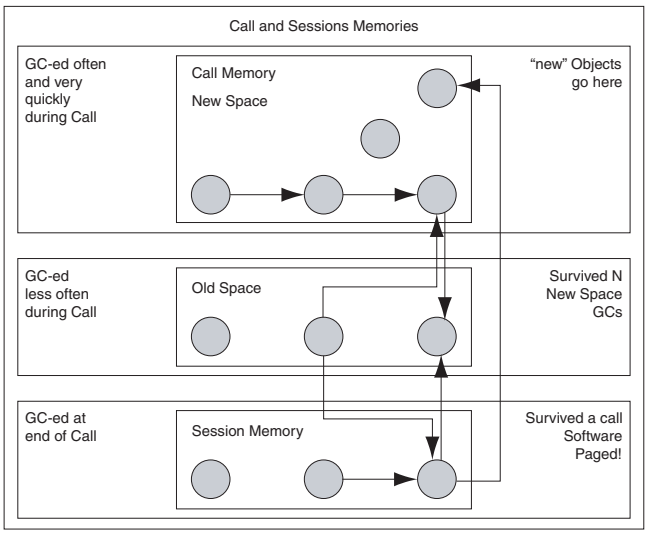
- The Oracle JVM uses the Oracle Database scheduling facilities, which can manage multiple users efficiently.
- Garbage collection is performed consistently for multiple users, because garbage collection is focused on a single user within a single session. The Oracle JVM has an advantage, because the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

The two types of memory space are call space and session space.

Memory space	Description
Call space	It is a fast and cheap memory. It primarily exists for the length of a call. Call memory space is divided into new and old segments. All new objects are created within new memory. Objects that have survived several **scavenges** are moved into old memory.

Memory space	Description
Session space	It is an expensive, performance-wise memory. It primarily exists for the length of a session. All `static` variables and any objects that exist beyond the lifetime of a call exist here.

Figure 1–6 Garbage Collection



Garbage collection algorithms within the Oracle JVM adhere to the following rules:

1. New objects are created within new call space.
2. Scavenging occurs at a set interval. Some programmers create objects frequently for only a short duration. These types of objects are created and garbage-collected quickly within the new call space. This is known as **scavenging**.
3. Any objects that have survived several iterations of scavenging are considered to be objects that can exist for a while. These objects are moved out of new call space into old call space. During the move, they are also compacted. Old call space is scavenged or garbage collected less often and, therefore, provides better performance.
4. At the end of the call, any objects that are to exist beyond the call are moved into session space.

Figure 1–6 illustrates the steps listed in the preceding text. This approach applies sophisticated allocation and collection schemes tuned to the types and lifetimes of objects. For example, new objects are allocated in fast and cheap call memory, designed for quick allocation and access. Objects held in Java `static` variables are migrated to the more precious and expensive session space.

Footprint

The footprint of a running Java program is affected by many factors:

- Size of the program

  The size of the program depends on the number of classes and methods and how much code they contain.

- Complexity of the program

The complexity of the program depends on the number of core class libraries that the Oracle JVM uses as the program runs, as opposed to the program itself.

- Amount of space the JVM uses

  The amount of space the JVM uses depends on the number of objects the JVM allocates, how large these objects are, and how many objects must be retained across calls.

- Ability of the garbage collector and memory manager to deal with the demands of the program running

  This is often nondeterministic. 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 multiple clients concurrently is a minimum per-user session footprint. The Oracle JVM 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 Oracle JVM uses the following 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, that is, objects that live across calls within a session

Performance

The performance of the Oracle JVM is enhanced by implementing a native compiler. The platform-independent Java bytecodes run on top of a JVM, and the JVM interacts with the specific hardware platform. Any time you add levels within software, the performance is degraded. Because Java requires going through an intermediary to interpret the bytecodes, a degree of inefficiency exists for Java applications compared to applications developed using 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.

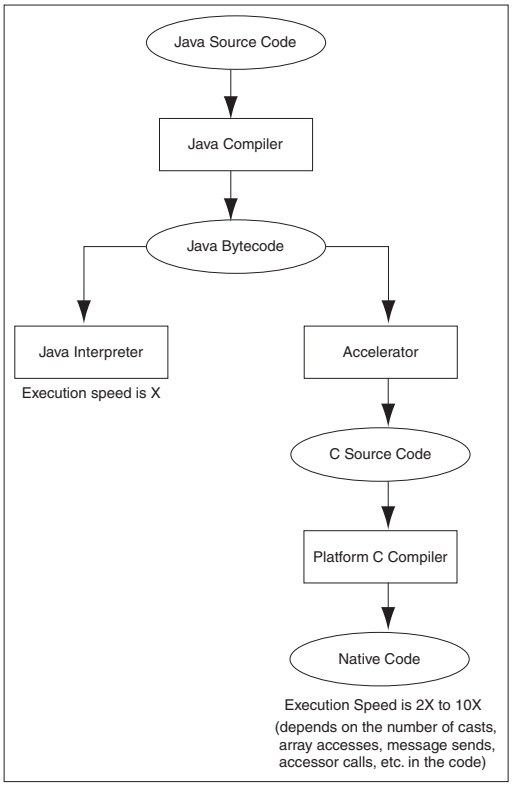
The following table describes two methods for native compilation:

Compiler	Description
Just-In-Time (JIT) Compilation	JIT compilers quickly compile Java bytecodes to platform-specific, or native, machine code during run time. These compilers do not produce an executable to be run on the platform. Instead, they provide platform-dependent code from Java bytecodes that is run directly after it is translated. JIT compilers should be used for Java code that is run frequently and at speeds closer to that of code developed in other languages, such as C.
Ahead-of-Time Compilation	This compilation translates Java bytecodes to platform-independent C code before run time. 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, such as JDBC code, in natively compiled form. It is applicable across all the platforms Oracle supports. 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 1–7 shows, natively compiled code runs up to ten times faster than interpreted code. As a result, the more native code your program uses, the faster it runs.

Figure 1–7 Interpreter versus Accelerator



See Also: "Natively Compiled Code" on page 9-1

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. The class loader locates the classes in `CLASSPATH` and loads them only when they are used while the program is running. 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 when the program is run for the first time. A simple program can cause the Oracle JVM to load many core classes to support its needs. A programmer cannot easily predict or determine the number of classes loaded.	The Oracle JVM loads classes dynamically, just as with any other JVM. 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, and they will simply use the same preloaded 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 runs 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 run time. 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 enables you to specify the schemas in which the classes reside. This separation of resolution from class loading ensures that you always know what programs users run. **See Also:** Chapter 11, "Schema Objects and Oracle JVM Utilities"

The Oracle JVM

The Oracle JVM is a standard, Java-compatible environment that runs any pure Java application. It is compatible with the JLS and the JVM specification laid down by Sun Microsystems. 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 run time.

Java in Oracle Database introduces the following terms:

- Session

  A session in the Oracle Database Java environment is identical to the standard Oracle Database usage. A session is typically, although not necessarily, bounded by the time a single user connects to the server. As a user who calls a Java code, you must establish a session in the server.

- Call

  When a user causes a Java code to run within a session, it is termed as a call. A call can be started in the following different ways:

  – A SQL client program runs a Java stored procedure.

  – A trigger runs a Java stored procedure.

  – A PL/SQL program calls a Java code.

  In all the cases defined, a call begins, some combination of Java, SQL, or PL/SQL code is run to completion, and the call ends.

  Note: The concept of session and call apply across all uses of Oracle Database.

Unlike other Java environments, the Oracle JVM is embedded within Oracle Database and introduces a number of new concepts. This section discusses some important differences between the Oracle JVM and typical client JVMs based on:

- Process Area
- The main() Method
- The GUI
- The IDE

Process Area

In a standard Java environment, you run a Java application through the interpreter by issuing the following command on the command line, where *classname* is the name of the class that you want the JVM to interpret first:

```
java classname
```

This command causes the application to run within a process on your operating system. However, with Oracle JVM, you must load the application into the database, publish the interface, and then run the application within a database session.

> **See Also:** Chapter 2, "Java Applications on Oracle Database" for information on loading, publishing, and running Java applications

The main() Method

Client-based Java applications declare a single, top-level method, `public static void main(String args[])`. This method 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 process of management of sessions, networks, and other shared resources from hosted Java programs.

The GUI

A server cannot provide GUIs, but it can provide the logic that drives them. The Oracle JVM supports only the headless mode of the Java Abstract Window Toolkit (AWT). All Java AWT classes are available within the server environment and your programs can use the Java AWT functionality, as long as they do not attempt to materialize a GUI on the server.

> **See Also:** "User Interfaces on the Server" on page 2-19

The IDE

The Oracle JVM is oriented to Java application deployment, and not development. You can write and test applications on any preferred integrated development environment (IDE), such as Oracle JDeveloper, and then deploy them within the database for the clients to access and run them.

> **See Also:** "Development Tools" on page 1-21

The binary compatibility of Java enables you to work on any IDE and then upload the 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 Oracle JVM

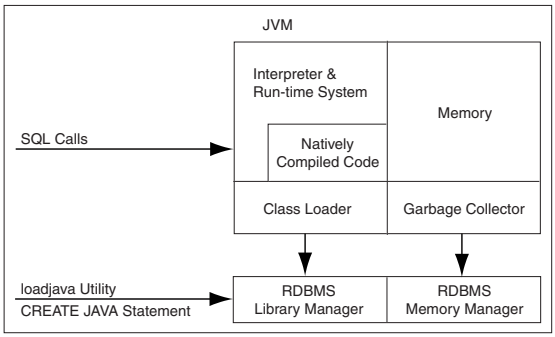
This section briefly describes the main components of the Oracle JVM and some of the facilities they provide.

The Oracle JVM is a complete, Java2-compliant environment for running Java applications. It runs in the same process space and address space as the database

kernel by sharing its memory heaps and directly accessing its relational data. This design optimizes memory use and increases throughput.

The Oracle JVM provides a run-time environment for Java objects. It fully supports Java data structures, method dispatch, exception handling, and language-level threads. It also supports all the core Java class libraries, including `java.lang`, `java.io`, `java.net`, `java.math`, and `java.util`. Figure 1–8 shows the main components of the Oracle JVM.

Figure 1–8 Main Components of the Oracle JVM



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

In addition, the Oracle JVM is tightly integrated with the scalable, shared memory architecture of the database. Java programs use call, session, and object lifetimes efficiently without user intervention. As a result, the Oracle JVM and middle-tier Java business objects can be scaled, even when they have session-long state.

The following sections provide an overview of some of the components of the Oracle JVM and the JDBC driver and the SQLJ translator:

- Library Manager
- Compiler
- Interpreter
- Class Loader
- Verifier
- Server-Side JDBC Internal Driver
- Server-Side SQLJ Translator

> **See Also:** "Automated Storage Management With Garbage Collection" on page 1-10 and "Performance" on page 1-12

Library Manager

To store Java classes in the Oracle Database, you use the `loadjava` command-line utility, which uses the SQL `CREATE JAVA` statements to do its work. When called by the `CREATE JAVA {SOURCE | CLASS | RESOURCE}` statement, the library manager loads Java source, class, or resource files into the database. These Java schema objects are not accessed directly, and only the Oracle JVM uses them.

Compiler

The Oracle JVM includes a standard Java compiler. When the `CREATE JAVA SOURCE` statement is run, 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.

See Also: "Natively Compiled Code" on page 9-1

Interpreter

To run Java programs, the Oracle JVM includes a standard Java2 bytecode interpreter. The interpreter and the associated Java run-time system run standard Java class files. The run-time system supports native methods and call-in and call-out from the host environment.

Note: You can also compile your Java code to improve performance. The Oracle JVM uses natively compiled versions of the core Java class libraries, SQLJ translator, and JDBC drivers.

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 and generates the data structures needed to run 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. In addition, if the source files are available, then the class loader calls the Java compiler automatically when Java class files must be recompiled.

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 and defines a set of Java classes providing vendor-independent access to relational data. Specified by Sun Microsystems and modeled after ODBC and the X/Open SQL Call Level Interface (CLI), the JDBC classes provide 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 Oracle Database, providing fast access to Oracle data from Java stored procedures. The server-side JDBC internal driver complies fully with the standard JDBC specification. Tightly integrated with the database, the JDBC driver supports Oracle-specific data types, globalization character sets, and stored procedures. In addition, the client-side and server-side JDBC APIs are the same, which makes it easy to partition applications.

Server-Side SQLJ Translator

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

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

Java Application Strategy of Oracle

Oracle provides enterprise application developers 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 JVM integrated with the Oracle Database. All these products are fully compatible with Java standards. This section covers the following topics:

- Java in Database Application Development
- Java Programming Environment
- Java Stored Procedures
- PL/SQL Integration and Oracle RDBMS Functionality
- Development Tools

Java in Database Application Development

The most important features of Java in database application development are:

- Providing flexible partitioning of Java2 Platform, Standard Edition (J2SE) applications for symmetric data access at the JDBC and SQLJ level.
- Bridging SQL and the Java2 Platform, Enterprise Edition (J2EE) world by:
  - Calling out Web components, such as JSP and servlet
  - Calling out Enterprise JavaBean (EJB) components
  - Bridging SQL and Web Services
    - Calling out Web Services
  - Using Oracle JVM as ERP Integration Hub
  - Invalidating cache

Java Programming Environment

In addition to the Oracle JVM, the Java programming environment provides:

- Java stored procedures as the Java equivalent and companion for PL/SQL. Java stored procedures are tightly integrated with PL/SQL. You can call Java stored procedures from PL/SQL packages and PL/SQL procedures from Java stored procedures.

- The JDBC and SQLJ programming interfaces for accessing SQL data.
- Tools and scripts that assist in developing, loading, and managing classes.

The following table helps you decide when to use which Java API:

Type of functionality you need	Java API to use
To have a Java procedure called from SQL, such as a trigger.	Java stored procedures
To call a static, simple SQL statement from a known table with known column names from a Java object.	SQLJ
To call dynamic, complex SQL statements from a Java object.	JDBC

Java Stored Procedures

Java stored procedures are Java programs written and deployed on a server and run from the server, exactly like 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, such as OCI and PRO*, or JDBC or SQLJ.

See Also: Chapter 5, "Developing Java Stored Procedures"

In addition, you can use Java to develop powerful, server-side programs, which can be independent of PL/SQL. Oracle Database provides a complete implementation of the standard Java programming language and a fully-compliant JVM.

In Oracle Database, the lifetime of a Java stored procedure session is identical to the database session in which it is embedded. Any state represented in Java transparently persists for the lifetime of the database session, simplifying the process of writing stored procedures, triggers, and methods for Oracle abstract data types.

PL/SQL Integration and Oracle RDBMS Functionality

You can call existing PL/SQL programs from Java and Java programs from PL/SQL. This solution protects and leverages your PL/SQL and Java code and opens up the advantages and opportunities of Java-based Internet computing.

Oracle Database offers two different Java APIs for accessing SQL data, JDBC and SQLJ. Both these APIs are available on the client and the server. As a result, you can deploy your applications on the client and server, without modifying the code.

The following topics introduce the Java APIs and the JPublisher tool provided by Oracle Database:

- JDBC Drivers
- SQLJ
- JPublisher

JDBC Drivers

JDBC is a database access protocol that enables you to connect to a database and run SQL statements and queries to the database. The core Java class libraries provide only one JDBC API, `java.sql`. However, JDBC is designed to enable vendors to supply drivers that offer the necessary specialization for a particular database. Oracle provides the following distinct JDBC drivers:

Driver	Description
JDBC Thin driver	You can use the JDBC Thin driver to write pure Java applications and applets that access Oracle SQL data. The JDBC Thin driver is especially well-suited for Web-based applications and applets, because you can dynamically download it from a Web page, similar to any other Java applet.
JDBC OCI driver	The JDBC OCI driver accesses Oracle-specific native code, that is, non-Java code, and libraries on the client or middle tier, providing 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 the Java code runs on the server. It allows Java applications running in the Oracle JVM on the server to access locally defined data, that is, data on the same system and in the same process, with JDBC. It provides a performance boost, because of its ability to use the underlying Oracle RDBMS libraries directly, without the overhead of an intervening network connection between the 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.

See Also:

- "Utilizing SQLJ and JDBC for Querying the Database" on page 3-3
- *Oracle Database JDBC Developer's Guide and Reference*

SQLJ

Oracle has worked with other vendors, including IBM, Tandem, Sybase, and Sun Microsystems, to develop a standard way to embed SQL statements in Java programs called SQLJ. This work has resulted in a new standard, ANSI x.3.135.10-1998, for a simpler and more highly productive programming API than JDBC. A user writes applications to this higher-level API and then uses a preprocessor to translate the program to standard Java source with JDBC calls. At run time, the program can communicate with multi-vendor databases using standard JDBC drivers.

SQLJ provides a simple, but powerful, way to develop both client-side and middle-tier applications that access databases from Java. You can use SQLJ in stored procedures, triggers, and methods within the Oracle Database 10*g* environment. In addition, you can combine SQLJ programs with JDBC.

The SQLJ translator is a Java program that translates embedded SQL in Java source code to pure JDBC-based Java code. Because Oracle Database 10*g* provides a complete Java environment, you cannot compile SQLJ programs on a client that will run on the server. Instead, you can compile them directly on the server. The adherence of Oracle Database to the Internet standards enables you to choose the development style as per your requirements.

See Also:

- "Utilizing SQLJ and JDBC for Querying the Database" on page 3-3
- *Oracle Database SQLJ Developer's Guide and Reference*

JPublisher

JPublisher provides a simple and convenient tool to create Java programs that access existing Oracle relational database tables.

See Also: *Oracle Database JPublisher User's Guide*

Development Tools

The introduction of Java in Oracle Database enables you to use several Java IDEs. The adherence of Oracle Database to the Java standards and specifications and the open Internet standards and protocols ensures that your Java programs work successfully, when you deploy them on Oracle Database. Oracle provides many tools or utilities that are written in Java making development and deployment of Java server applications easier. Oracle JDeveloper, a Java IDE provided by Oracle, has many features designed specifically to make deployment of Java stored procedures and EJBs easier. You can download JDeveloper from:
http://www.oracle.com/technology/software/products/jdev/index.html

Desupport of J2EE Technologies in the Oracle Database

With the introduction of Oracle Application Server Containers for J2EE (OC4J), a new, light-weight, easy-to-use, fast, and certified J2EE container, Oracle began the desupport of the J2EE and Common Object Request Broker Architecture (CORBA) stacks from the database, starting with Oracle9*i* Database release 2. However, the database-embedded Oracle JVM is still present and will continue to be enhanced to provide J2SE features, Java stored procedures, and JDBC and SQLJ 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:
  - EJB container
  - JavaServer Pages (JSP) container
  - Oracle9*i* Servlet Engine (OSE)
- The embedded 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.

Memory Model for Dedicated Mode Sessions

In Oracle Database 10*g*, the Oracle JVM 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 (PGA) 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 (SGA). 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.

What's New in this Release?

The following sections describe the additions to this release:

- Upgrading to J2SE 1.4.2
- Auditing SQL Statements Related to Java Schema Objects
- The PGA_AGGREGATE_TARGET parameter

Upgrading to J2SE 1.4.2

In this release, the system classes are upgraded from J2SE 1.4.1 to J2SE 1.4.2. Sun Microsystems publishes the list of incompatibilities between J2SE 1.4.2 and previous versions at the following Web site:

`http://java.sun.com/j2se/1.4.2/compatibility.html`

Auditing SQL Statements Related to Java Schema Objects

Oracle Database 10*g* release 2 (10.2) provides support for auditing DDL statements related to Java schema objects. That is, you can audit the SQL statements used to create, alter, and drop the Java source, class, and resource schema objects.

See Also: "Auditing" on page 2-18 for further information

The PGA_AGGREGATE_TARGET parameter

The `PGA_AGGREGATE_TARGET` parameter is a database parameter that can be set in the `init.ora` file. When the PGA usage is greater than the desired value set using this parameter, the JVM performs additional memory management actions at the end of call to reduce PGA usage. Specifically, the live contents of the Java heap are copied and compacted to a new area and the existing Java heap is freed. The new area becomes the Java heap for subsequent calls. This step is intended to compact Java heap down to the minimal necessary size between database calls. However, this step does not run if there are live Java threads at the end of the call.

See Also: *Oracle Database Performance Tuning Guide*

Notes:

- The "Setting PGA_AGGREGATE_TARGET Initially" section in *Oracle Database Performance Tuning Guide* does not take into account the memory usage of Java applications.
- The memory usage of Java code in the server is not as predictable as that of SQL. Therefore, it is important to monitor and tune Automatic PGA Memory Management.

2

Java Applications on Oracle Database

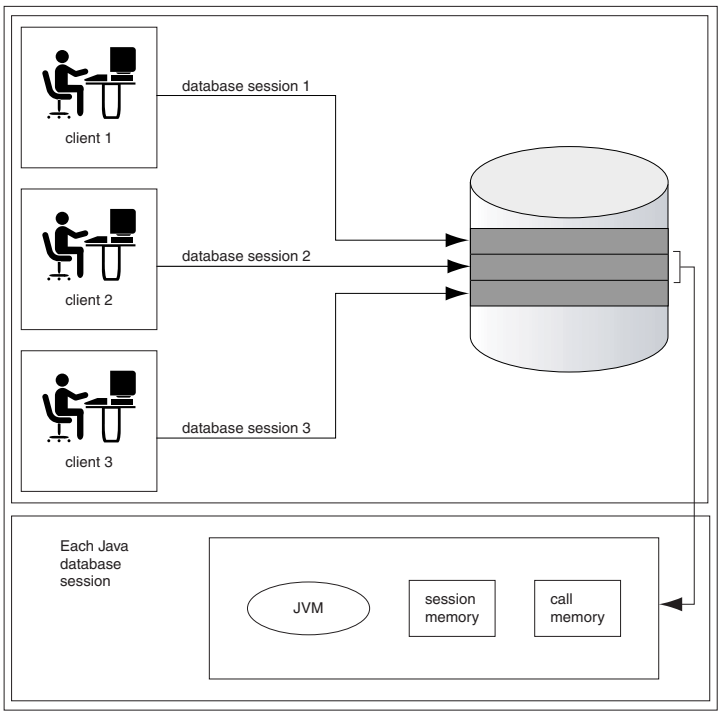
Oracle Database runs standard Java applications. However, the Java-integrated Oracle Database environment is different from a typical Java development environment. This chapter describes the basic differences for writing, installing, and deploying Java applications within Oracle Database. This chapter contains the following sections:

Database Sessions Imposed on Java Applications

In the Java-integrated Oracle Database, your Java applications exist within the context of a database session. Oracle Java virtual machine (JVM) sessions are entirely analogous to traditional Oracle sessions. Each Oracle JVM session maintains the state of the Java applications accessed by the client across calls within the session.

Figure 2–1 illustrates how each Java client starts a database session as the environment for running Java applications within the database. Each Java database session has a separate garbage collector, session memory, and call memory.

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. Runs Java within the database. This is referred to as a call.
3. Continues to work within the session, performing as many calls as required.
4. Ends the session.

Within a session, the client has its own Java environment. It appears to the client as if a separate, individual JVM was started for each session, although the implementation is more efficient than this seems to imply. Within a session, the Oracle JVM manages the scalability of applications. Every call from a single client is managed within its own session, and calls from each client is handled separately. The Oracle JVM 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 the Java code. Variables defined as `static` are local to the client. No client can access the `static` variables of other clients, because the memory is not available across session boundaries. Because each client runs the Java application calls within its own session, activities of each client are separate from any other client. During a call, you can store objects in `static` fields of different classes, which will be available in the next session. The entire state of your Java program is private and exists for your entire session.

The Oracle JVM manages the following within the session:

- All the objects referenced by `static` Java variables, all the objects referred to by these objects, and so on, till their transitive closure
- Garbage collection for the client that created the session
- Session memory for `static` variables and across call memory needs

- Call memory for variables that exist within a call

Execution Control

In the Java2 Platform, Standard Edition (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 called when you enter the following command on the command-line:

```
java classname
```

This command starts the Java interpreter and passes the desired class, that is, the class specified by *`classname`*, to the Java interpreter. The interpreter loads the class and starts running the application by calling `main()`. However, Java applications within the database do not start by a call to the `main()` method.

After loading your Java application within the database, you can run it by calling any `static` method within the loaded class. The class or methods must be published before you can run them. In Oracle Database, the entry point for Java applications is not assumed to be `main()`. Instead, when you run 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 the Java object on the server by running the following command:

```
java myprogram
```

where, *`myprogram`* is the name of a class that contains the `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 standard Java development environment, Java source code, binaries, and resources are stored as files in a file system, as follows:

- Source code files are saved as `.java` files.
- Compiled Java binary files are saved as `.class` files.
- Resources are any data files, such as `.properties` or `.ser` files, that are stored in the file system hierarchy and are loaded and used at run time.

In addition, when you run a Java application, you specify the `CLASSPATH`, which is a file or directory path in the file system that contains your `.class` files. Java also provides a way to group these files into a single archive form, a ZIP or Java Archive (JAR) file.

Both these concepts are different in an Oracle Database environment. Table 2–1 describes how Oracle Database handles Java classes and locates dependent classes:

Table 2–1 Description of Java Code and Classes

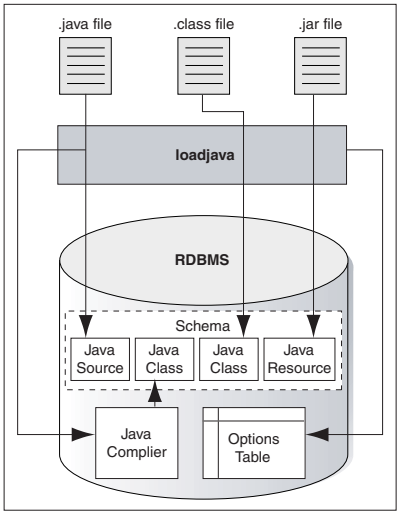
Java Code and Classes	Description
Storing Java code, binaries, and resources	In Oracle Database, source code, classes, and resources reside within the database and are known as Java schema objects, where a schema corresponds to a database user. There are three types of Java schema objects: source, class,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

Java Classes Loaded in the Database

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

Figure 2–2 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 Oracle Database



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

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

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

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

Oracle Database accepts Java names up to 4000 characters long. However, the names of Java schema objects cannot be longer than 31 characters. Therefore, if a schema object name is longer than 31 characters, then the system generates a short name, or alias, for the schema object. Otherwise, the fully qualified name, also called full name, is used.

You can specify the full name in any context that requires it. When needed, name mapping is handled by Oracle Database.

See Also: "Shortened Class Names" on page 2-20

Preparing Java Class Methods for Execution

To ensure that your Java methods run, you must do the following:

1. Decide when the Java source code 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, then you should specify a separate resolver with the load command.
4. Publish your class or method.

This sections covers the following topics:

- Compiling Java Classes
- Resolving Class Dependencies
- Loading Classes
- Granting Execute Rights
- Controlling the Current User
- Checking Java Uploads
- Publishing
- Auditing

Compiling Java Classes

Compilation of the Java source code can be done in one of the following ways:

- You can compile the source explicitly on a client system 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, which is managed by the `loadjava` utility.
- You can force the compilation to occur dynamically at run time.

Note: If you decide to compile through `loadjava`, then you can specify the compiler options. Refer to "Specifying Compiler Options" on page 2-6 for more information.

This section includes the following topics:

- Compiling Source Through javac
- Compiling Source Through loadjava
- Compiling Source at Run Time
- Specifying Compiler Options

- Recompiling Automatically

Compiling Source Through javac

You can compile Java source code 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 the Java code on your own system, rather than debugging it on the database.

Compiling Source Through loadjava

When you specify the `-resolve` option with `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 writes all compilation errors to the log file of the `loadjava` utility as well as the `USER_ERRORS` view.

Compiling Source at Run Time

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 run time. The source file is loaded into a source schema object.

Oracle Database writes all compilation errors to the log file of the `loadjava` utility as well as the `USER_ERRORS` view.

Specifying Compiler Options

You can specify the compiler options in the following ways:

- Specify compiler options on the command line with `loadjava`. You can also specify the encoding option with `loadjava`.
- Specify persistent compiler options in the `JAVA$OPTIONS` table. The `JAVA$OPTIONS` table exists for each schema. Every time you compile, the compiler uses these options. However, any compiler options specified with the `loadjava` command override the options defined in this table. You must create this table yourself if you wish to specify compiler options in this manner.

See Also: "Compiler Options Specified in a Database Table"

Default Compiler Options

When compiling a source schema object for which neither a `JAVA$OPTIONS` entry exists nor a command-line value for any option is specified, 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 `encoding` compiler option specified with `loadjava` identifies the encoding of the `.java` file. This option overrides any matching value in the `JAVA$OPTIONS` table. The values are identical to:

```
javac -encoding
```

This option is relevant only when loading a source file.

Compiler Options Specified in a Database Table

Each `JAVA$OPTIONS` entry 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 using the following procedures and functions, 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 2–2 Definitions for the Name and Option Parameters

Parameter	Description
name	This is a Java package name, a fully qualified class name, or an 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 that has a value for `name` that most closely matches the fully qualified class name of a schema object. A `name` whose value is the empty string matches any schema object name.
option	The `option` parameter is either `online`, `encoding`, or `debug`.

Initially, a schema does not have a `JAVA$OPTIONS` table. To create a `JAVA$OPTIONS` table, use the `java.set_compiler_option` procedure from the `DBMS_JAVA` package 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 2–3 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`. 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 2–3 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 the full qualified 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 this rule.
Empty string	`online`	`true`	▪ `a.c` No pattern match with any defined name. Defaults to the empty string rule. ▪ `x.y` No pattern match with any defined name. Defaults to the empty string rule.

Recompiling Automatically

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

```
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, then 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 and recompile, if necessary. Note that classes can be recompiled only if the source file 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 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 JVM searches for `.class`, `.zip`, and `.jar` files within the directories specified in `CLASSPATH`. In contrast, the Oracle JVM searches database schemas for class objects. In Oracle Database, because you load all Java classes into the database, you may 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 predefined Java application programming interfaces (APIs), such as `java.lang.*`, are loaded within the `PUBLIC` schema. If your classes depend on other classes you have defined, then you will probably load them all within your own schema. For example, if your schema is `SCOTT`, the database resolver searches the `SCOTT` schema before searching the `PUBLIC` schema. The listing of schemas to search is known as a **resolver specification**. Resolver specifications are defined for each class. This is in contrast to a classic JVM, where `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, then the class is listed as invalid. Unsuccessful resolution at run time produces a `ClassNotFound` exception. Also, run-time 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. Ensure that you correctly load the resource files that your classes require.

For each interclass reference in a class, the resolver searches the schemas specified by the resolver specification for a valid class schema object that satisfies the reference. If all references are resolved, then the resolver marks the class valid. A class that has never been resolved, or has been resolved unsuccessfully, is marked invalid. A class that 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 specification that searches the definer's schema first and then searches the PUBLIC schema. 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 specification.

Classes can be resolved in the following ways:

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

  ```
  loadjava -resolve
  ```

- Loading using your own resolver specification definition:

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

  In the preceding example, the resolver specification definition includes the SCOTT schema, OTHER schema, and PUBLIC.

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

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

The first parameter within the resolver specification is for the class schema object, and the second parameter defines the schema within which to search for these class schema objects.

Allowing References to Nonexistent Classes

You can specify a special option within a resolver specification that allows an unresolved reference to a nonexistent class. Sometimes, internal classes are never used within a product. In a normal Java environment, this is not a problem, because as long as the methods are not called, the JVM ignores them. However, the Oracle Database resolver tries to resolve all classes referenced within the JAR file, including the unused classes. If the reference cannot be validated, then the classes within the JAR file are marked as invalid.

To ignore references, you can specify the wildcard, minus sign (-), within the resolver specification. 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 specification schema list.

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

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.

In addition, you can define that all classes not found are to be ignored. However, this is dangerous, because a class that has a dependent class will be marked as valid, even if the dependent class does not exist. However, the class can never run without the dependent class. In this case, you will receive an exception at run time.

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

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

If you later intend to load the nonexistent classes that required you to use such a resolver, then you should not use a resolver containing the minus sign (-) wildcard. Instead, include all referenced classes in the schema before resolving.

> **Note:** An alternative mechanism for dealing with nonexistent classes is using the `-gemissing` option of `loadjava`. This option causes `loadjava` to create and load definitions of classes that are referenced, but not defined.

Bytecode Verifier

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

Table 2–4 ORA Errors

Error Code	Description
ORA-29545	If the resolver determines that the class is malformed, then the resolver does not mark it valid. When the resolver rejects a class, it issues an ORA-29545 error. The `loadjava` utility 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 run time. In these cases, the resolver issues an ORA-29552 warning that `loadjava` reports. The `loadjava` utility issues this warning when the Java Language Specification (JLS) requires an `IncompatibleClassChangeError` to be thrown. Oracle JVM relies on the resolver to detect these situations, supporting the proper run-time behavior that the JLS requires.

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

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

The method is valid only if `A` extends `B` or `A` implements the interface `B`. If `A` or `B` have been resolved using the minus sign (-) wildcard, then the resolver does not know that this method is safe. In this case, the resolver replaces the bytecodes of `myMethod` with bytecodes that throw an exception if `myMethod` is called.

A resolver without the minus sign (`-`) wildcard 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 may consider using the alternative resolver is if

you must load an existing JAR file containing classes that reference other nonsystem classes, which are not included in the JAR file.

> **See Also:** Chapter 11, "Schema Objects and Oracle JVM Utilities" for more information on class resolution and loading your classes within the database.

Loading Classes

This section gives an overview of loading your classes into the database using the `loadjava` utility. You can also run `loadjava` from within SQL commands.

> **See Also:** Chapter 11, "Schema Objects and Oracle JVM Utilities"

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

Table 2–5 Description of Java Files

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

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

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

Table 2–6 loadjava Operations on Schema Objects

Schema Object	loadjava Operations on Objects
`.java` source files	1. Creates a Java source schema object in the definer's schema unless another schema is specified. 2. Loads the contents of the source file into a schema object. 3. Creates a class schema object for all classes defined in the source file. 4. If `-resolve` is requested, compiles the source schema object and resolves the class and its dependencies. It then stores the compiled class into a class schema object.
`.sqlj` source files	1. Creates a source schema object in the definer's schema unless another schema is specified. 2. Loads contents of the source file into the schema object. 3. Creates a class schema object for all classes and resources defined in the source file. 4. If `-resolve` is requested, translates and compiles the source schema object and stores the compiled class into a class schema object. It then stores the profile into a `.ser` resource schema object and customizes it.

Table 2–6 (Cont.) loadjava Operations on Schema Objects

Schema Object	loadjava Operations on Objects
`.class` compiled Java files	1. Creates a class schema object in the definer's schema unless another schema is specified. 2. Loads the class file into the schema object. 3. Resolves and verifies the class and its dependencies if `-resolve` is specified.
`.properties` Java resource files	1. Creates a resource schema object in the definer's schema unless another schema is specified. 2. Loads a resource file into a schema object.
`.ser` SQLJ profile	1. Creates a resource schema object in the definer's schema unless another schema is specified. 2. Loads the `.ser` resource file into a schema object and customizes it.

The `dropjava` utility performs the reverse of `loadjava`. It deletes schema objects that correspond to Java files. Always use `dropjava` to delete a Java schema object created with `loadjava`. Dropping with SQL data definition language (DDL) commands will not update the auxiliary data maintained by `loadjava` and `dropjava`. You can also run dropjava from within SQL commands.

After loading the classes and resources, you can access the `USER_OBJECTS` view in your database schema to verify whether your classes and resources have been loaded properly.

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 want to update the class, then you must load the same type of file that you originally loaded. If you want to update the other type, then you must drop the first before loading the second. For example, if you loaded `x.java` as the source for class `y`, then to load `x.class`, you must first drop `x.java`.

- You cannot define the same class within two different schema objects in 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, then `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`, and then load the new `x.java`, which does not define `y`.
  - Load the new `x.java`, which does not define `y`, and 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 Also: "Permission for Loading Classes" on page 10-17

Loading JAR or ZIP Files

The `loadjava` utility accepts `.class`, `.java`, `.properties`, `.sqlj`, `.ser`, `.jar`, or `.zip` files. The JAR or ZIP files can contain source, class, and data files. When you pass a JAR or ZIP file to `loadjava`, it opens the archive and loads the members of the archive 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, then 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`.

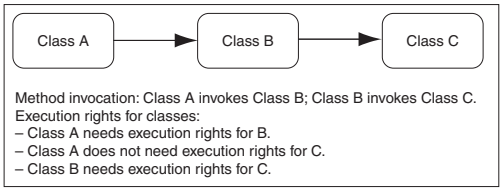
Note: Oracle Database does not reload a class if it has not changed since the last load. However, you can force a class to be reloaded using the `-force` option.

Granting Execute Rights

If you load all classes within your own schema and do not reference any class outside your schema, then you already have rights to run the classes. You have the privileges necessary for your objects to call other objects loaded in the same schema. That is, the ability for class `A` to call class `B`. Class `A` must be given the right to call 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 cannot be run by other users, because of security concerns. You can provide other users the right to run your class through the `loadjava -grant` option. You can grant rights to run your classes to a certain user or schema, but you cannot grant rights to a role, which includes the superuser `DBA` role. The setting of rights to run classes is the same as used to grant or revoke privileges in SQL DDL statements.

Figure 2–3 Rights to Run Classes



See Also: Chapter 9, "Oracle Database Java Application Performance" for information on JVM security permissions

Controlling the Current User

When running Java or PL/SQL code, 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 running SQL, PL/SQL, or Java Database Connectivity (JDBC). The current user controls the interpretation of SQL and determines privileges. For example, if a table is referenced by a simple name, then it is assumed that the table belongs to 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. Therefore, you can 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 runs under the user that deployed the Java class.

By default, Java stored procedures run without changing the current user, that is, with the privileges of their invoker, and 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, and not the definer.

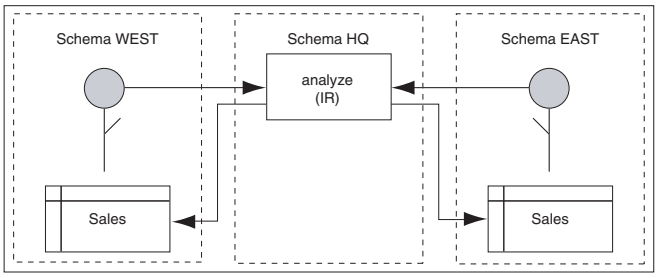
On the other hand, definer-rights procedures are bound to the schema in which they reside. They run with the privileges of their definer, and their unqualified references to schema objects are resolved in the schema of the definer.

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 this, the procedure must also reside at each regional site. This causes a maintenance problem.

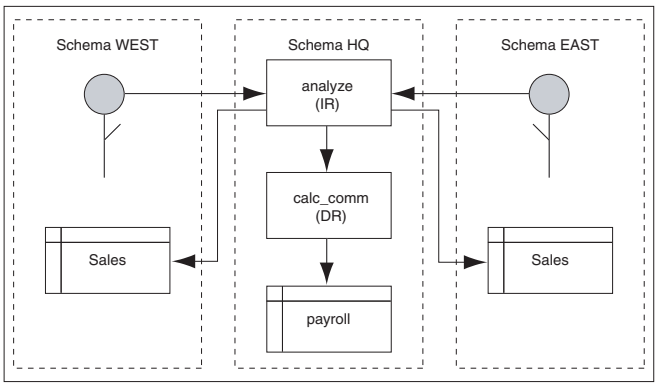
To solve the problem, the company installs an invoker-rights version of the procedure `analyze` at headquarters. Now, as Figure 2–4 shows, all regional sites can use the same procedure to query their own `sales` tables.

Figure 2–4 Invoker-Rights Solution



Occasionally, you may want to override the default invoker-rights behavior. Suppose headquarters wants the `analyze` procedure to calculate sales commissions and update a central `payroll` table. This 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 shown in Figure 2–5, the solution is to have the `analyze` procedure call the definer-rights 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`. This option is similar to the `setuid` UNIX facility, except that `-definer` applies to individual classes, not whole programs. Alternatively, you can run the SQL DDL statement that changes the `AUTHID` of the current user.

Different definers can have different privileges, and applications can consist of many classes. Therefore, use the `-definer` option carefully ensuring that classes have only the required privileges.

Checking Java Uploads

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

Table 2–7 lists the key columns in `USER_OBJECTS` and their description.

Table 2–7 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. The values can be either `VALID` or `INVALID`. It is always `VALID` for `JAVA RESOURCE`.

Object Name and Type

An `OBJECT_NAME` in `USER_OBJECTS` is the alias. The fully qualified name is stored as an alias if it exceeds 31 characters.

> **See Also:** "Shortened Class Names" on page 2-20 for information on fully qualified names and aliases.

If the server uses an alias for a schema object, then you can use the `LONGNAME()` function of the `DBMS_JAVA` package to receive it from a query as a fully qualified name, without having to know the alias or the conversion rules.

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

This statement displays the fully qualified name of the Java source schema objects. Where no alias is used, no conversion occurs.

> **Note:** SQL and PL/SQL are *not* case-sensitive.

You can use the SHORTNAME() function of the DBMS_JAVA package to use a fully qualified name as a query criterion, without having to know whether it was converted to an alias in the database.

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

This statement displays the OBJECT_TYPE of the schema object with the specified fully qualified name. This presumes that the fully qualified 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 Java source schema object is VALID if it compiled successfully, and a Java class schema object is VALID if it was resolved successfully. A Java 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';
```

The preceding statement finds any OBJECT_NAME entries that end with the characters Alerter.

Publishing

Oracle Database enables clients and SQL to call Java methods that are loaded in the database after they are published. You publish either the object itself or individual methods. If you write a Java stored procedure that you intend to call with a trigger, directly or indirectly in SQL data manipulation language (DML) or in PL/SQL, then

you must publish individual methods in the class. Using a call specification, specify how to access the method. Java programs consist of many methods in many classes. However, only a few `static` methods are typically exposed with call specifications.

> **See Also:** Chapter 6, "Publishing Java Classes With Call Specifications"

Auditing

In releases prior to Oracle Database 10*g* release 2 (10.2), Java classes in the database cannot be audited directly. However, you can audit the PL/SQL wrapper. Typically, all Java stored procedures are started from some wrappers. Therefore, all Java stored procedures can be audited, though not directly.

In Oracle Database 10*g* release 2 (10.2), you can audit DDL statements for creating, altering, or dropping Java source, class, and resource schema objects, as with any other DDL statement. Oracle Database 10*g* release 2 (10.2) provides auditing options for auditing Java activities easily and directly. You can also audit any modification of Java sources, classes, and resources.

You can audit database activities related to Java schema objects at two different levels, statement level and object level. At the statement level you can audit all activities related to a special pattern of statements. Table 2–8 lists the statement auditing options and the corresponding SQL statements related to Java schema objects.

Table 2–8 Statement Auditing Options Related to Java Schema Objects

Statement Option	SQL Statements
`CREATE JAVA SOURCE`	`CREATE JAVA SOURCE` `CREATE OR REPLACE JAVA SOURCE`
`ALTER JAVA SOURCE`	`ALTER JAVA SOURCE`
`DROP JAVA SOURCE`	`DROP JAVA SOURCE`
`CREATE JAVA CLASS`	`CREATE JAVA CLASS` `CREATE OR REPLACE JAVA CLASS`
`ALTER JAVA CLASS`	`ALTER JAVA CLASS`
`DROP JAVA CLASS`	`DROP JAVA CLASS`
`CREATE JAVA RESOURCE`	`CREATE JAVA RESOURCE` `CREATE OR REPLACE JAVA RESOURCE`
`ALTER JAVA RESOURCE`	`ALTER JAVA RESOURCE`
`DROP JAVA RESOURCE`	`DROP JAVA RESOURCE`

For example, if you want to audit the `ALTER JAVA SOURCE` DDL statement, then enter the following statement at the SQL prompt:

```
AUDIT ALTER JAVA SOURCE
```

Object level auditing provides finer granularity. It enables you to identify specific problems by zooming into specific objects. Table 2–9 lists the object auditing options for each Java schema object. The entry X in a cell indicates that the corresponding SQL command can be audited for that Java schema object. The entry NA indicates that the corresponding SQL command is not applicable for that Java schema object.

Table 2–9 Object Auditing Options Related to Java Schema Options

Object Option	Java Source	Java Resource	Java Class
ALTER	X	NA	X
EXECUTE	NA	NA	X
AUDIT	X	X	X
GRANT	X	X	X

See Also:

- *Oracle Database Security Guide*
- *Oracle Database SQL Reference*

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. However, it is inappropriate for code running on the server to attempt to materialize or display a user interface on the server. Users running applications in the Oracle JVM environment should not be expected nor allowed to interact with or depend on the display and input hardware of the server where Oracle Database is running.

To address compatibility issues on platforms that do not support display, keyboard, or mouse, Java 1.4 outlines Headless Abstract Window Toolkit (AWT) support. The Headless AWT API introduces a new `public` run-time 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, which rely on the native display devices, are changed to throw `HeadlessException` if the platform does not support a display. In Oracle Database, user interfaces are supported only on client applications. Accordingly, the Oracle JVM is a Headless Platform and throws `HeadlessException` if these 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 provides programmers access to fonts, imaging, printing, and color and ICC manipulation. For example, applications running in the Oracle JVM can parse, manipulate, and write out images as long as they do not try to physically display it on the server. The Sun Microsystems reference JVM implementation can be started in the Headless mode, by supplying the `-Djava.awt.headless=true` property, and run with the same Headless AWT restrictions as the Oracle JVM does. The Oracle JVM fully complies with the Java Compatibility Kit (JCK) with respect to Headless AWT.

See Also:

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

The Oracle JVM takes a similar approach for sound support. Applications in the Oracle JVM 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 `MidiUnavailableException` checked exception to

signal that the system is unavailable. However, similar to the Headless AWT support, Oracle Database supports the APIs that allow sound file manipulation, free of the native sound devices. The Oracle JVM 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.

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 fully qualified name is longer than 31 characters or contains illegal or nonconvertible characters, then Oracle Database converts it to a short name, or alias, to use as the name of the schema object. Oracle Database keeps track of both the names and how to convert between them. If the fully qualified name is 31 characters or less and has no illegal or inconvertible characters, then it 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 the `LONGNAME()` function 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 fully qualified name of the Java schema object, which is specified using its alias. The following is an example of a statement used 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';
```

You can also specify a full name to the database by using the `SHORTNAME()` function of the `DBMS_JAVA` package. The function takes a full name as input and returns the corresponding short name. This function is useful for verifying whether the classes are loaded successfully, by querying the `USER_OBJECTS` view.

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

See Also: Chapter 8, "Java Stored Procedures Application Example"

Class.forName() in Oracle Database

The JLS 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, then a reference to the `Class` object for the class is returned. If it fails, then an instance of `ClassNotFoundException` is thrown.

Class lookup is always on behalf of a referencing class and is done through an instance of `ClassLoader`. The difference between the Java Development Kit (JDK) implementation and the Oracle JVM implementation is the method in which the class is found:

- The JDK uses one instance of `ClassLoader` that searches the set of directory tree roots specified by the `CLASSPATH` environment variable.

- Oracle JVM defines several resolvers that specify how to locate classes. Every class has a resolver associated with it, and each class can, potentially, have a different resolver. When you run a method that calls `Class.forName()`, the resolver of the currently running class, which is `this`, is used to locate the class.

  See Also: "Resolving Class Dependencies" on page 2-9

You can receive unexpected results if you try to locate a class with an incorrect resolver. For example, if a class `X` in schema `X` requests a class `Y` in schema `Y` to look up class `Z`, you will experience an error if you expected the resolver of class `X` 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 may 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` instance in the `Class.forName()` method.
- Supply the class and the schema it resides in to the `classForNameAndSchema()` method.
- Supply the schema and class name to `ClassForName.lookupClass()`.
- Serialize your objects with the schema name and 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`, then you can declare a `PUBLIC` synonym for the class.

This section covers the following topics:

- Supply ClassLoader in Class.forName()
- Supply Class and Schema Names to classForNameAndSchema()
- Supply Class and Schema Names to lookupClass()
- Supply Class and Schema Names when Serializing
- Class.forName Example

Supply 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. As a result, the locating of classes is dependent on the definition of the associated resolver. The `ClassLoader` instance knows which resolver to use, based on the class that is specified. When you supply a `ClassLoader` instance to `Class.forName()`, your class is looked up in the schemas defined in the resolver of the class. The syntax of 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 by using 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 called the running method by using the `oracle.aurora.vm.OracleRuntime.getCallerClass()` method. After you retrieve the class, call the `Class.getClassLoader()` method on the returned class. The following example retrieves the class of the instance that called the `workForCaller()` method. Then, its class loader is retrieved and supplied to the `Class.forName()` method. As a result, 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 supplying the resolver, which can identify the schemas to be searched. Alternatively, you can supply the schema in which the class is loaded. If you know in which schema the class is loaded, then you can use the `classForNameAndSchema()` method, which is in the `DbmsJava` class provided by Oracle Database. This method takes both the name of the class and the schema in which the class resides and 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 using 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();
  writeName (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 `String` value containing both the schema and class names to the `oracle.aurora.util.ClassForName.lookupClass()` method. When called, 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 the `SCOTT` schema, use the following:

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

> **Note:** 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 deserialize a class, part of the operation is to lookup a class based on a name. 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 deserializing 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, then the schema names are read out and used when performing the class lookup. If used on a client, then the schema names are ignored.

Class.forName Example

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

```
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);
  }
}
```

The preceding example uses the following methods for locating a class:

- To use the resolver of the class of an instance, call `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 call the `ForName()` method without any parameters. It sets the `from` variable to the calling class. Then, call the `lookupWithClassLoader()` method to locate the class using the resolver of the calling class.
- To lookup a class in a specified schema, call the `lookupWithSchema()` method. This provides the class and schema name to the `classForNameAndSchema()` method.

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 and not the operating system resources that the Java objects hold 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.

See Also: "Operating System Resources Affected Across Calls" on page 2-34

The following sections discuss 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 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. **See Also:** "Automated Storage Management With Garbage Collection" on page 1-10.
files and sockets	Java contains classes that represent file or socket resources. Instances of these classes hold on to the file or socket constructs, such as file handles, of the operating system.
threads	Threads are discouraged within the Oracle JVM because of scalability issues. However, you can have a multithreaded application within the database. **See Also:** "Threading in Oracle Database" on page 2-26.

Operating System Resource Access

By default, a Java user does not have direct access to most operating system resources. A system administrator can give permissions to a user to access these resources by modifying the JVM security restrictions. The JVM security enforced upon system resources conforms to Java2 security.

> **See Also:** "Java2 Security" on page 10-2

Operating System Resource Lifetime

You can access operating system resources using the standard core Java classes and methods. Once you access a resource, the time that it remains active varies according to the type of resource. Memory is garbage collected. 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.

> **See Also:** "Operating System Resources Affected Across Calls" on page 2-34.

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 memory is freed by the garbage collector. The garbage collector was created to collect all unused Java object memory. However, it does not close operating system constructs. All operating system constructs must be closed by the program before the Java object is garbage 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, then the operating system holds

the file handle construct open until the call ends or JVM exits. This may 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 before closing the socket.

For performance reasons, the garbage collector cannot examine each object to see if it contains a handle. As a result, 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 to 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 `inFile`, then eventually the `File` object will be garbage collected. Even after the `File` object is garbage collected, the operating system treats the file as if it were in use, because it was not closed.

Note: You may 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 run when the garbage collector has time. If you close your resources within your finalizer, then it might not be freed until the JVM exits. The best approach is to close your resources within the method.

Threading in Oracle Database

The Oracle JVM implements a nonpreemptive 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, call the `Thread.yield()` method or wait on a network socket by calling `mySocket.read()`.

The following table lists the advantages and disadvantages of the Oracle Database threading model:

Advantages	Disadvantages
■ Simple to program ■ Efficient to implement in the JVM, 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 then raise a run-time 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 an application written on a multiprocessor system. Also, the lack of concurrency among Java threads is not an issue, because the Oracle JVM 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 need to support hundreds or thousands of simultaneous transactions, then start each one in its own JVM. This is exactly what happens when you create a session in the Oracle JVM. The normal transactional capabilities of Oracle Database accomplish coordination and data transfer between the JVMs. This is not a scalability issue, because in contrast to the 6 MB to 8 MB memory footprint of a typical JVM, Oracle Database can create thousands of JVMs, with each one taking less than 40 KB of memory.

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

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

- System calls are at a minimum. Oracle JVM has exchanged some of the normal system calls with nonsystem solutions. For example, entering a monitor-synchronized block or method does not require a system call.
- Deadlocks are detected.
  - The Oracle JVM monitors for deadlocks between threads. If a deadlock occurs, then the Oracle JVM 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, then the `oracle.aurora.vm.LimboError` exception is thrown.

Thread Life Cycle

In a single-threaded application, a call ends when one of the following events occurs:

- The thread returns to its caller.
- An exception is thrown and is not caught in Java code.
- The `System.exit()` or `oracle.aurora.vm.OracleRuntime.exitCall()` method is called.

If the initial thread creates and starts other Java threads, then the call ends in one of the following ways:

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

In the shared server mode, when a call ends because of a return or uncaught exceptions, the Oracle JVM throws an instance of `ThreadDeathException` in all daemon threads. `ThreadDeathException` essentially forces threads to stop running.

> **See Also:** "Operating System Resources Affected Across Calls" on page 2-34.

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

During a call, a Java program can recursively cause more Java code to be run. For example, your program can issue a SQL query using JDBC or SQLJ that, in turn, calls a trigger written in Java. 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.

System.exit(), OracleRuntime.exitSession(), and OracleRuntime.exitCall()

The `System.exit()` method terminates the JVM, preserving no Java state. It does not cause the database session to terminate or the client to disconnect. However, the database session may, and often does, terminate itself immediately afterward. `OracleRuntime.exitSession()` also terminates the JVM, preserving no Java state. However, it also terminates the database session and disconnects the client.

The behavior of `OracleRuntime.exitCall()` varies depending on `OracleRuntime.threadTerminationPolicy()`. This method returns a `boolean` value. If it is `true`, then any active thread should be terminated, rather than left quiescent, at the end of a database call. In a shared server process, `threadTerminationPolicy()` is always `true`. In a shadow (dedicated) process, the default value is `false`. You can change the value by calling `OracleRuntime.setThreadTerminationPolicy()`.

In addition, there is another method, `OracleRuntime.callExitPolicy()`. This method determines when a call is exited if none of the `OracleRuntime.exitSession()`, `OracleRuntime.exitCall()`, or `System.exit()` methods are ever called. The call exit policy can be set to one of the following, using `OracleRuntime.setCallExitPolicy()`:

- `OracleRuntime.EXIT_CALL_WHEN_MAIN_THREAD_TERMINATES`

  If set to this value, then as soon as the main thread returns or an uncaught exception occurs on the main thread, all remaining threads, both daemon and non-daemon, are either killed or left quiescent until the next call depending on the value of `threadTerminationPolicy()`.

- `OracleRuntime.EXIT_CALL_WHEN_ALL_NON_DAEMON_THREADS_TERMINATE`

  This is the default value. If this value is set and if none of the various exit methods are called, then the call ends when only daemon threads are left running. At this

point, either the daemon threads are killed, if `threadTerminationPolicy()` is `true`, or they are left quiescent until the next call, which is the case by default for shadow processes.

- `OracleRuntime.EXIT_CALL_WHEN_ALL_THREADS_TERMINATE`

  If set to this value, then the call does not end until all threads have returned or ended due to an uncaught exception. At this point, the call ends regardless of the value of `threadTerminationPolicy()`.

In Oracle9*i* Database, the JVM behaves as if the `callExitPolicy()` were `OracleRuntime.EXIT_CALL_WHEN_ALL_NON_DAEMON_THREADS_TERMINATE` and the `threadTerminationPolicy()` were `true` for both shared and dedicated server processes. This means kill the daemon threads at this point. Also, if `exitCall()` were executed, then all threads are killed before the call is ended, in both shared and dedicated server processes.

In all releases, both `System.exit()` and `OracleRuntime.exitSesssion()` terminate the JVM abruptly, without running `finally` blocks on any thread. Also, `OracleRuntime.exitCall()` attempts to end each thread by throwing a `ThreadDeath` exception on each thread, which causes any `finally` blocks on active thread stacks to be run.

Shared Servers Considerations

For sessions that use shared servers, certain limitations exist across calls. 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 SGA. This means that process-specific resources, such as threads, open files, and sockets, must be cleaned up at the end of each call, and therefore, will not be available for the next call.

This section covers the following topics:

End-of-Call Migration

In the shared server mode, Oracle Database preserves the state of your Java program between calls by migrating all objects that are reachable from `static` variables to session space at the end of the call. Session space exists within the session of the client to store `static` variables and objects that exist between calls. Oracle JVM automatically performs this migration operation at the end of every call.

This migration operation is a memory and performance consideration. Hence, 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, then 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 the performance of your server.

To maximize the number of users who can run 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 re-create many data structures when you need them again in another call. For this reason, the Oracle JVM has a mechanism for calling a specified Java method when a session is about to become inactive, such as at the end of a call.

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 run 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 complex stateful server applications that you implement in Java.

The decision of whether to null-out data structures at the end of the call and then re-create them for each new call is a typical time and space trade-off. There is some extra time spent in re-creating 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 objects based on call-space.

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

- Buffers or caches.
- Static fields, such as arrays, which once initialized can remain unchanged during the course of the program.
- Any dynamically built data structure that can have a space-efficient representation between calls and a more speed-efficient representation for the duration of a call. This can be tricky and may complicate your code, making it hard to maintain. Therefore, 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.aurora.memoryManager.EndOfCallRegistry` class, the `registerCallback()` method takes an object that implements a `Callback` object. The `registerCallback()` method stores this object until the end of the call. At the end of the call, the Oracle JVM calls 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` object is removed from the registry.

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

A weak table holds the registry of end-of-call callbacks. If either the `Callback` object or value are not reachable from the Java program, then both the object and the 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, and you cannot rely on the table holding it back.

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

Static fields

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 call a method that lazily creates, or re-creates, the cached data.

Consider the following example:

```
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()
  {
    ...
  }
}
```

The preceding example 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 re-creating the cache.

When the user creates the cache, the `Callback` object is automatically registered within the `getCachedField()` method. At end-of-call, the Oracle JVM calls the registered `Callback.act()` method, which frees the static memory.

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.

Consider the following example:

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

class Example2 implements Callback
{
  private Object cachedField = null;

  public voidact (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()
  {
    ...
  }
}
```

The preceding example does the following:

1. Implements the instance as a `Callback` object.
2. Implements the `Callback.act()` method to free up the instance fields.

3. When the user requests a cache, the `Callback` object registers itself for the end-of-call migration.
4. Provides a method, `createCachedField()`, for lazily re-creating the cache.

When the user creates the cache, the `Callback` object is automatically registered within the `getCachedField()` method. At end-of-call, the Oracle JVM calls 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.

The EndOfCallRegistry.registerCallback() Method

The `registerCallback()` method installs a `Callback` object within a registry. At the end of the call, the Oracle JVM calls the `act()` method of all registered `Callback` objects.

You can register your `Callback` object by itself or with an `Object` instance. If you need additional information stored within an object to be passed into `act()`, then you can register this object with the `value` parameter, which is an instance of `Object`.

The following are the valid signatures of the `registerCallback()` method:

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

public static void registerCallback(Callback thunk);
```

The following table lists the parameters of `registerCallback` and their description:

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

The EndOfCallRegistry.runCallbacks() Method

The signature of the `runCallbacks()` method is as follows:

```
static void runCallbacks()
```

The JVM calls this method at end-of-call and calls `act()` for every `Callback` object registered using `registerCallback()`. It is called at end-of-call, before object migration and before the last finalization step.

Note: Do not call this method in your code.

The Callback Interface

The interface is declared as follows:

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

Any object you want to register using `EndOfCallRegistry.registerCallback()` must implement the `Callback`

interface. This interface can be useful in your application, where you require notification at end-of-call.

The Callback.act() method

The signature of the `act()` method is as follows:

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

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

Operating System Resources Affected Across Calls

In the shared server mode, the Oracle JVM 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	■ Client sockets can exist across calls. ■ Server sockets terminate when the call ends.
Objects that depend on operating system resources	Regardless of the usable lifetime of the 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, then Oracle Database will throw 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, then a `java.io.IOException` is raised. ■ `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 the shared server mode, the Oracle JVM automatically closes open operating system constructs 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, then the file handle is closed at the end of the call for you. Therefore, if you hold on to the `File` object across calls, then the next usage of the file handle throws an exception.

In Example 2–5, the `Concat` class 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, written to `outFile`, and the call ends. Because `outFile` is defined as a `static` variable, it is moved into session space between call invocations. However, the file handle 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 are workarounds. To ensure that your handles stay valid, close your files, buffers, and so on, at the end of every call, and 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. Alternatively, do not store operating system resources within `static` objects that are expected to live across calls. Instead, 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, ensuring that anything written into the file is appended to the end. At the end of each call, the file is closed. Two things occur:

- The `File` object no longer exists outside a call.
- The operating system resource, the `outme.txt` file, is reopened for each call. If you had made the `File` object a `static` variable, then 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, TTC or IIOP of Oracle Net.

> **See Also:** "Configuring Oracle JVM" on page 4-2 for information on how to configure your connection.

You may also want 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 using the following:

- 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` instance. The client side of the connection that sends requests is serviced through a `Socket` instance. You can use sockets as defined within the JVM with the restriction that a `ServerSocket` instance within a shared server cannot exist across calls.

The following table lists the socket types and their description:

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` instance 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` instance outside of the call it was created in.

Threads

In the shared server mode, when a call ends because of a return or uncaught exceptions, the Oracle JVM throws `ThreadDeathException` in all daemon threads. `ThreadDeathException` essentially forces threads to stop running. Code that depends on threads living across calls does not behave as expected in the 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, which Sun Microsystems supplies, functions in the 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 the shared server mode are killed at the end of call, that is, when all non-daemon threads return. If the RMI server session is reentered in a subsequent call, then these daemon threads are not restarted and the RMI server fails to function properly.

3

Calling Java Methods in Oracle Database

This chapter provides an overview and examples of calling Java methods that reside in Oracle Database. It contains the following sections:

Invoking Java Methods

The type of the Java application determines how the client calls a Java method. The following sections discuss each of the Java application programming interfaces (APIs) available for creating a Java class, which can be loaded into the database and accessed by a client:

Utilizing Java Stored Procedures

You can run Java stored procedures in the same way as PL/SQL stored procedures. Normally, in Oracle Database, a call to a Java stored procedure is a result of database manipulation, because the call is usually the result of a trigger or SQL data manipulation language (DML) call.

To call 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 returns a string:

1. Define a class, `Hello`, as follows:

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

Save the file as a `Hello.java` file.

2. Compile the class on your client system using the standard Java compiler, as follows:

```
javac Hello.java
```

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

You need to determine the location at which this Java code must run. If you run `Hello.class` on your client system, then it searches the `CLASSPATH` for all the supporting core classes that `Hello.class` needs for running. This search should result in locating the dependent classes in one of the following:

- As individual files in one or more directories, where the directories are specified in the `CLASSPATH`
- Within `.jar` or `.zip` files, where the directories containing these files are specified in the `CLASSPATH`

3. Decide on the resolver for the `Hello` class.

In this case, load `Hello.class` on the server, where it is stored in the database as a Java schema object. When you call the `world()` method, the Oracle Java virtual machine (JVM) locates the necessary supporting classes, such as `String`, using a resolver. In this case, the Oracle JVM uses the default resolver. The default resolver looks for these classes, first in the current schema, and then in `PUBLIC`. All core class libraries, including the `java.lang` package, are found in `PUBLIC`. You may need to specify different resolvers. You can trace problems earlier, rather than at run time, by forcing resolution to occur when you use `loadjava`.

> **See Also:** "Resolving Class Dependencies" on page 2-9 and Chapter 11, "Schema Objects and Oracle JVM Utilities"

4. Load the class on the server using `loadjava`. You must specify the user name and password. Run the `loadjava` command as follows:

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

5. Publish the stored procedure through a call specification.

To call a Java `static` method with a SQL call, you need to publish the method with a call specification. A call specification defines the arguments that the method takes and the SQL types that it returns.

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

```
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. Call the stored procedure, as follows:

```
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. 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 that are called with triggers or SQL and PL/SQL calls. Furthermore, JDeveloper can automate the task of writing call specifications.

See Also: Chapter 5, "Developing Java Stored Procedures"

Utilizing JNI Support

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

Oracle Database does not support the use of JNI in Java applications. If you use JNI, then your application is not 100 percent pure Java and the native methods require porting between platforms. Native methods can cause server failure, violate security, and corrupt data.

Utilizing SQLJ and JDBC for Querying the Database

You can use SQLJ and Java Database Connectivity (JDBC) protocols to query the database from a Java client. Both protocols establish a session with a given user name and password on the database and run SQL queries against the database. The following table lists the protocols and their description:

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.
SQLJ	Use this protocol for static, easy SQL queries. SQLJ typically runs against a known table with known column names.

This section covers the following topics:

- JDBC
- SQLJ
- Example Comparing JDBC and SQLJ
- Complete SQLJ Example
- SQLJ Strong Typing Paradigm
- Translating a SQLJ Program
- Running a SQLJ Program in the Server
- Converting a Client Application to Run on the Server

- Interacting with PL/SQL

JDBC

JDBC is an industry-standard API developed by Sun Microsystems that lets you embed SQL statements as Java method arguments. JDBC is based on the X/Open SQL Call Level Interface (CLI) and complies with the SQL92 Entry Level standard. Each vendor, such as Oracle, creates its JDBC implementation by implementing the interfaces of the standard `java.sql` package. Oracle provides the following JDBC drivers that implement these standard interfaces:

- The JDBC Thin driver, a 100 percent pure Java solution that you can use for either client-side applications or applets and requires no Oracle client installation.
- The JDBC Oracle Call Interface (OCI) driver, which you use for client-side applications and requires an Oracle client installation.
- The server-side JDBC driver embedded in Oracle Database.

Using JDBC is a step-by-step process of performing the following tasks:

1. Creating a statement object of some type for your desired SQL operation
2. Assigning any local variables that you want to bind to the SQL operation
3. Carrying out 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 run time, require JDBC. However, in typical applications, this represents a minority of the SQL operations.

SQLJ

SQLJ offers an industry-standard way to embed any static SQL operation directly into the Java source code in one simple step, without requiring the multiple steps of JDBC. Oracle SQLJ complies with the X3H2-98-320 American National Standards Institute (ANSI) standard.

SQLJ consists of a translator, which is a precompiler that supports standard SQLJ programming syntax, and a run-time component. After creating your SQLJ source code in a `.sqlj` file, you process it with the translator. The translator translates the SQLJ source code to standard Java source code, with SQL operations converted to calls to the SQLJ run time. In the Oracle Database SQLJ implementation, the translator calls a Java compiler to compile the Java source code. When your SQLJ application runs, the SQLJ run time calls JDBC to communicate with the database.

SQLJ also enables you to catch errors in your SQL statements before run time. JDBC code, being pure Java, is compiled directly. The compiler cannot detect SQL errors. On the other hand, when you translate SQLJ code, the translator analyzes the embedded SQL statements semantically and syntactically, catching SQL errors during development, instead of allowing an end user to catch them when running the application.

Example Comparing JDBC and SQLJ

The following is an example of a JDBC code and a SQLJ code that perform a simple operation:

JDBC:

```
// Assume 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();
```

Assume that you have established a JDBC connection, `conn`. Next, you need to do the following:

1. Define the Java variables, `name`, `id`, and `salary`.
2. Create a `PreparedStatement` instance.

   You can use a prepared statement whenever values in 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. In the preceding example, these variables are assigned values using the `pstmt.setInt()` and `pstmt.setFloat()` methods. The first ? refers to the `int` variable `id` and is set to a value of `37115`. The second ? refers to the `float` variable `salary` and is set to a value of `20000`.
3. Run the query and return the data into a `ResultSet` object.
4. Retrieve the data of interest from the `ResultSet` object and print it. In this case, the `ename` column. A result set usually contains multiple rows of data, although this example has only one row.

SQLJ:

```
String name;
int id=37115;
float salary=20000;
#sql {SELECT ename INTO :name FROM emp WHERE empno=:id AND sal>:salary};
System.out.println("Name is: " + name);
```

In addition to allowing SQL statements to be directly embedded in Java code, SQLJ supports Java host expressions, also known as bind expressions, to be used directly in the SQL statements. In the simplest case, a host expression is a simple variable, as in this example. However, more complex expressions are allowed as well. Each host expression is preceded by colon (:). This example uses Java host expressions, `name`, `id`, and `salary`. In SQLJ, because of its host expression support, you do not need a result set or equivalent when you are returning only a single row of data.

Note: All SQLJ statements, including declarations, start with the `#sql` token.

Complete SQLJ Example

This section presents a complete example of a simple SQLJ program:

```
import java.sql.*;
import sqlj.runtime.ref.DefaultContext;
import oracle.sqlj.runtime.Oracle;

#sql ITERATOR MyIter (String ename, int empno, float sal);

public class MyExample
{
  public static void main (String args[]) throws SQLException
  {
    Oracle.connect("jdbc:oracle:thin:@oow11:5521:sol2", "scott", "tiger");
    #sql { INSERT INTO emp (ename, empno, sal) VALUES ('SALMAN', 32, 20000) };
    MyIter iter;
    #sql iter={ SELECT ename, empno, sal FROM emp };
    while (iter.next())
    {
      System.out.println(iter.ename()+" "+iter.empno()+" "+iter.sal());
    }
  }
}
```

In the preceding example, you do the following:

1. Declare your iterators.

   SQLJ uses a strongly-typed version of JDBC result sets, known as iterators. An iterator has a specific number of columns of specific data types. You must define your iterator types before using them, as in this example.

   ```
   #sql ITERATOR MyIter (String ename, int empno, float sal);
   ```

   This declaration results in SQLJ creating an iterator class, `MyIter`. Iterators of type `MyIter` can store results whose first column maps to a Java `String`, second column maps to a Java `int`, and third column maps to a Java `float`. This definition also names the three columns as `ename`, `empno`, and `sal`, to match the column names of the referenced table in the database. `MyIter` is a named iterator.

2. Connect to the database.

   ```
   Oracle.connect("jdbc:oracle:thin:@oow11:5521:sol2","scott", "tiger");
   ```

   SQLJ provides the `Oracle` class and its `connect()` method accomplishes the following important tasks:

   a. Registers the Oracle JDBC drivers that SQLJ uses to access the database, in this case, the JDBC Thin driver.

   b. Opens a database connection for the specified schema, in this case, user `scott` with password `tiger`, at the specified URL. In this case, the URL points to host `oow11`, port `5521`, and SID `so12`.

   c. Establishes this connection as the default connection for the SQLJ statements. Although each JDBC statement must explicitly specify a connection object, a

SQLJ statement can either implicitly use a default connection or optionally specify a different connection.

3. Process a SQL statement. The following is accomplished:

   a. Insert a row into the `emp` table:

   ```
   #sql {INSERT INTO emp (ename, empno, sal) VALUES ('SALMAN', 32, 20000)};
   ```

   b. Instantiate and populate the iterator:

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

4. Access the data that was populated within the iterator.

   ```
   while (iter.next())
   {
     System.out.println(iter.ename()+" "+iter.empno()+" "+iter.sal());
   }
   ```

   The `next()` method is common to all iterators and plays the same role as the `next()` method of a JDBC result set, returning `true` and moving to the next row of data, if any rows remain. You can access the data in each row by calling iterator accessor methods whose names match the column names. This is a characteristic of all named iterators. In this example, you access the data using the methods `ename()`, `empno()`, and `sal()`.

SQLJ Strong Typing Paradigm

SQLJ uses strong typing, such as iterators, instead of result sets. This enables the SQL instructions to be checked against the database during translation. For example, SQLJ can connect to a database and check your iterators against the database tables that will be queried. The translator will verify that they match, enabling you to catch SQL errors during translation that would otherwise not be caught until a user runs your application. Furthermore, if changes are subsequently made to the schema, then you can determine if these changes affect the application by rerunning the translator.

Translating a SQLJ Program

Integrated development environments (IDEs), such as Oracle JDeveloper, can translate, compile, and customize your SQLJ program as you build it. Oracle JDeveloper is a Microsoft Windows-based visual development environment for Java programming. If you are not using an IDE, then use the front-end SQLJ utility, `sqlj`. You can run it as follows:

```
%sqlj MyExample.sqlj
```

The SQLJ translator checks the syntax and semantics of your SQL operations. You can enable online checking to check your operations against the database. If you choose to do this, then you must specify an example database schema in your translator option settings. It is not necessary for the schema to have data identical to the one that the program will eventually run against. However, the tables must have columns with corresponding names and data types. Use the user option to enable online checking and specify the user name, password, and URL of your schema, as in the following example:

```
%sqlj -user=scott/tiger@jdbc:oracle:thin:@oow11:5521:sol2 MyExample.sqlj
```

Running a SQLJ Program in the Server

Many SQLJ applications run on a client. However, SQLJ offers an advantage in programming stored procedures, which are usually SQL-intensive, to run on the server.

There is almost no difference between writing a client-side SQLJ program and a server-side SQLJ program. The SQLJ run-time packages are automatically available on the server. However, you need to consider the following:

- There are no explicit database connections for code running on the server. There is only a single implicit connection. You do not need the usual connection code. If you are porting an existing client-side application, then you do not have to remove the connection code, because it will be ignored.
- The JDBC server-side internal driver does not support auto-commit functionality. Use SQLJ syntax for manual commits and rollbacks of your transactions.
- On the server, the default output device is a trace file, not the user screen. This is, normally, an issue only for development, because you would not write to `System.out` in a deployed server application.

To run a SQLJ program on the server, presuming you developed the code on a client, you have two options:

- Translate your SQLJ source code on the client and load the individual components, such as the Java classes and resources, on the server. In this case, it is easy to bundle them into a `.jar` file first and then load them on the server.
- Load your SQLJ source code on the server for the embedded translator to translate.

In either case, use the `loadjava` utility to load the file or files to the server.

Converting a Client Application to Run on the Server

To convert an existing SQLJ client-side application to run on the server, after the application has already been translated on the client, perform the following steps:

1. Create a `.jar` file for your application components.
2. Use the `loadjava` utility to load the `.jar` file on the server.
3. Create a SQL wrapper in the server for your application. For example, to run the preceding `MyExample` application on the server, run the following statement:

```
CREATE OR REPLACE PROCEDURE sqlj_myexample AS LANGUAGE JAVA NAME
`MyExample.main(java.lang.String[])';
```

You can then run `sqlj_myexample`, similar to any other stored procedure.

Interacting with PL/SQL

All the Oracle JDBC drivers communicate seamlessly with Oracle SQL and PL/SQL, and it is important to note that SQLJ interoperates with PL/SQL. You can start using SQLJ without having to rewrite any PL/SQL stored procedures. Oracle SQLJ includes syntax for calling PL/SQL stored procedures and also lets you embed anonymous PL/SQL blocks in SQLJ statements.

Debugging Server Applications

Oracle Database furnishes a debugging capability that is useful for developers who use the `jdb` debugger. The interface that is provided is the Java Debug Wire Protocol (JDWP), which is supported by Java Development Kit (JDK) 1.4 and later versions.

Some of the new features that the JDWP protocol supports are:

- Listening for connections
- Changing the values of variables while debugging
- Evaluating arbitrary Java expressions, including method evaluation

Oracle JDeveloper provides a user-friendly integration with these debugging features. Other independent IDE vendors will be able to integrate their own debuggers with Oracle Database.

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* and *Oracle Database Application Developer's Guide - Fundamentals*

How To Tell You Are Running on the Server

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

If you want to determine whether your code is running on the server, then use the `System.getProperty()` method, as follows:

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

The getProperty() method returns the following:

- A `String` that represents the Oracle Database release, if running on the server
- `null`, if running on the client

Redirecting Output on the Server

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

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

The minimum and default buffer size is 2,000 bytes and 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);
```

The output is printed at the end of the call.

Support for Calling Java Stored Procedures Directly

Oracle Database 10*g* introduces new features for calling Java stored procedures and functions. In releases prior to Oracle Database 10*g*, calling Java stored procedures and functions from a database client required JDBC calls to the 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 to directly call `static` Java stored procedures and functions. This new functionality is useful for general applications, but is particularly useful for Web services.

Classes for the simple API are located in the `oracle.jpub.reflect` package. Import this package into the client-side code.

The following 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 call the method, 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[]`, can be passed as an argument.
- As an optimization, parameter values can be represented as `String`:

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

- The semantics of this API are different from the semantics for calling 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 the definer's rights.

Using the Native Java Interface

Oracle Database 10*g* introduces the native Java interface, a new feature for calls to server-side Java code. It is a simplified application integration. Client-side and middle-tier Java applications can directly call Java in the database without defining a PL/SQL wrapper. The native Java interface uses the server-side Java class reflection capability.

In previous releases, calling Java stored procedures and functions from a database client required Java Database Connectivity (JDBC) calls to the 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 overcome these disadvantages. To remedy the deficiencies of JDBC calls to associated PL/SQL wrappers, the `-java` option uses an API for direct invocation of `static` Java methods. This functionality is also useful for Web services.

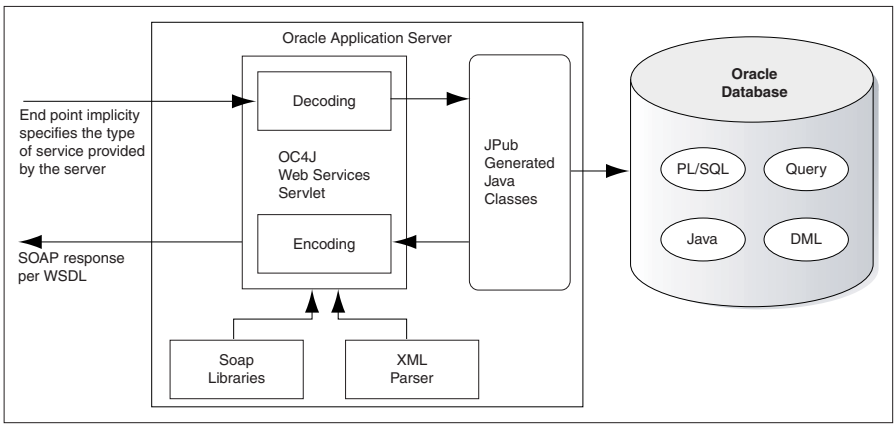
The functionality of the `-java` option mirrors that of the `-sql` option, creating a client-side Java stub class to access a server-side Java class, as opposed to creating a client-side Java class to access a server-side SQL object or PL/SQL package. The client-side stub class uses JPublisher code that 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 run time, the stub class is instantiated with a JDBC connection. Calls to the methods of the stub class 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.

Figure 3–1 demonstrates a client-side stub API for direct invocation of `static` server-side Java methods. JPublisher transparently takes care of stub generation.

Figure 3–1 Native Java Interface



You can use the `-java` option to publish a server-side Java class, as follows:

```
-java=className
```

Consider the `oracle.sqlj.checker.JdbcVersion` server-side Java class, with the following APIs:

```
public class oracle.sqlj.checker.JdbcVersion
{
 ...
 public java.lang.String toString();
 public static java.lang.String to_string();
 ...
}
```

As an example, assume that you want to call the following method on the server:

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

Use the following command to publish `JdbcVersion` for client-side invocation, using JPublisher:

```
% jpub -sql=scott/tiger -java=oracle.sqlj.checker.JdbcVersion:JdbcVersion Client
```

This command generates the client-side Java class, `JdbcVersionClient`, which contains the following APIs:

```
public class JdbcVersionClient
{
 ...
 public java.lang.String toString(long _handle);
 public java.lang.String to_string();
 ...
}
```

All static methods are mapped to instance methods in the client-side code. A instance method in the server-side class, `toString()` for example, is mapped to a method with an extra handle. A handle represents an instance of `oracle.sqlj.checker.JdbcVersion` in the server. The handle is used to call the instance method on the server-side.

See Also: *Oracle Database JPublisher User's Guide*

4

Java Installation and Configuration

This chapter describes how to install and configure of the Oracle Java virtual machine (JVM). It also describes how to enable the Java client. This chapter covers the following topics:

Initializing a Java-Enabled Database

If you install Oracle Database with the Oracle JVM option, then the database is Java-enabled. That is, it is ready to run Java stored procedures, Java Database Connectivity (JDBC), and SQLJ.

This section contains the following topics:

Configuring with Oracle Database Template

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

The Database Configuration Assistant enables you to create database templates for defining what each database instance installation will contain. Choose the Oracle JVM option to have the Java platform installed within your database.

Modifying an Existing Oracle Database to Include Oracle JVM

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

Configuring Oracle JVM

Before you install Oracle JVM as part of your normal Oracle Database installation, you need to ensure that the configuration requirements for Oracle JVM are fulfilled. The main configuration for Java classes within Oracle Database includes configuring the:

- Java memory requirements

  You must have at least 20 MB of `JAVA_POOL_SIZE` and 50 MB of `SHARED_POOL_SIZE`.

  See Also: "Java Memory Usage" on page 9-12

- 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 Oracle JVM creates the `DBMS_JAVA` PL/SQL package. Some entry points of `DBMS_JAVA` are for external use. That is, these entry points are used by developers. Other entry points are only for internal use. The corresponding Java class, `DbmsJava`, provides methods for accessing database functionality from Java.

See Also: Appendix A, "DBMS_JAVA Package"

Enabling the Java Client

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

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

Install J2SE on the Client

The client requires Java Development Kit (JDK) 1.4.2 or later. To confirm the version of JDK you are using, run the following commands on the command line:

```
$ which java
/usr/local/j2se1.4.2/bin/java

$ which javac
/usr/local/j2se1.4.2/bin/javac

$ java -version
java version "1.4.2"
```

Set Up Environment Variables

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

- `$JAVA_HOME`

  This variable must be set to the top directory of the installed JDK base.

- `$PATH`

  This variable must include `$JAVA_HOME/bin`.

- `$LD_LIBRARY_PATH`

  This variable must include `$JAVA_HOME/lib`.

JAR Files Necessary for Java2 Clients

To ensure that the Java client successfully communicates with the server, include the following files in the `CLASSPATH`:

- For JDK 1.4.2, include `$JAVA_HOME/lib/dt.jar`
- For JRE 1.4.2, include `$JAVA_HOME/lib/rt.jar`
- For any interaction with JDBC, include `$ORACLE_HOME/jdbc/lib/classes12.zip`
- For any client that uses SSL, include `$ORACLE_HOME/jlib/jssl-1_2.jar` and `$ORACLE_HOME/jlib/javax-ssl-1_2.jar`
- For any client that uses the Java Transaction API (JTA) functionality, include `$ORACLE_HOME/jlib/jta.jar`
- For any client that uses the Java Naming and Directory Interface (JNDI) functionality, include `$ORACLE_HOME/jlib/jndi.jar`
- If you are using the accelerator for native compilation, include `$JAVA_HOME/lib/tools.jar`

JAR Files Included for Clients that use SQLJ

You must include the `$ORACLE_HOME/sqlj/lib/translator.zip` file for SQLJ.

In addition to this file, add the appropriate `runtime`*`X`*`.zip` file, as follows:

- For a Java client using the current release of JDBC, include `$ORACLE_HOME/sqlj/lib/runtime12.zip`
- For a Java2 Platform, Enterprise Edition (J2EE) client using the current release of JDBC, include `$ORACLE_HOME/sqlj/lib/runtime12ee.zip`
- For any JDK client using JDBC 8.1.7 or earlier version, include `$ORACLE_HOME/sqlj/lib/runtime.zip`

Server Application Development on the Client

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

Test Install with Samples

When you install Oracle Database with the Oracle JVM option, a set of samples is also installed and available in the `$ORACLE_HOME/javavm/demo` directory. These samples can be compiled and run as a test of your installation.

If these samples do not compile or run, then the environment may be incorrectly set. Similarly, if these samples compile and run, but a code written by you does not, then a problem exists within the build environment or code.

Note: When verifying your installation, it is important that you run these examples using the supplied makefiles.

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

5

Developing Java Stored Procedures

Oracle Java virtual machine (JVM) has all the features you need to build a new generation of enterprise-wide applications at a low cost. The most important feature is the support for stored procedures. Using stored procedures, you can implement business logic at the server level, thereby improving application performance, scalability, and security.

This chapter contains the following sections:

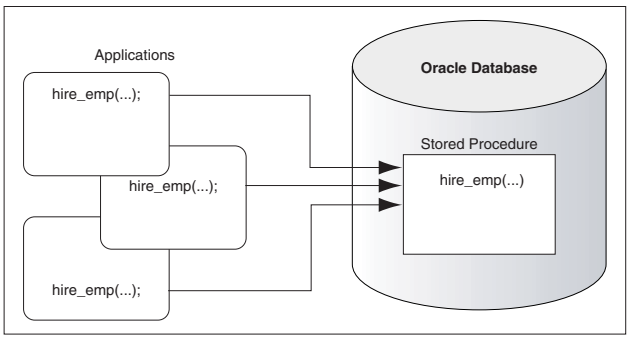
Stored Procedures and Run-Time Contexts

Stored procedures are Java methods published to SQL and stored in the database for general use. To publish Java methods, you write call specifications, 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 specification publishes the existence of a Java method. As a result, when you call the method through its call specification, 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, Oracle JVM can run any Java method as a stored procedure. The run-time contexts are:

- Functions and Procedures
- Database Triggers
- Object-Relational Methods

Functions and Procedures

Functions and procedures are named blocks that encapsulate a sequence of statements. They are 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. Therefore, you use procedure call specifications for `void` Java methods and function call specifications for value-returning methods.

Only top-level and package-level PL/SQL functions and procedures can be used as call specifications. 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 data manipulation language (DML) statements
- SQL `CALL` statements
- PL/SQL blocks, subprograms, and packages

Database Triggers

A database trigger is a stored procedure that is associated with a specific table or view. Oracle Database calls the trigger automatically whenever a given DML operation modifies the table or view.

A trigger has the following parts:

- A triggering event, which is generally a DML operation
- An optional trigger constraint
- A trigger action

When the event occurs, the trigger is called. A `CALL` statement in the trigger calls a Java method through the call specification of the method, to perform the action.

Database triggers are used to enforce complex business rules, derive column values automatically, prevent invalid transactions, log events transparently, audit transactions, and gather statistics.

Object-Relational Methods

A SQL object type is a user-defined composite data type that encapsulates a set of variables, called attributes, with a set of operations, called methods, which can be written in Java. The data structure formed by the set of attributes is `public`. However, as a good programming practice, you must ensure that your application does not manipulate these attributes directly and uses the set of methods provided.

You can create an abstract template for some real-world object as a SQL object type. The template specifies only those attributes and methods that 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 as required.

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 a `VARRAY`, a nested table, or both.

For example, the purchase order object type can contain a variable number of line items.

Advantages of Stored Procedures

Stored procedures offer several advantages. The following advantages are covered in this section:

- Performance
- Productivity and Ease of Use
- Scalability
- Maintainability
- Interoperability
- Replication
- Security

Performance

Stored procedures are compiled once and stored in an executable form. As a result, 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 the statements to be processed with a single call. This minimizes the use of slow networks, reduces network traffic, and improves round-trip response time. Also, 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 run faster. Likewise, stored functions called from SQL statements enhance performance by running 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 the productivity. Moreover, stored procedures let you extend the functionality of the database.

You can use the Java integrated development environment (IDE) of your choice to create stored procedures. Then, you can deploy these procedures on any tier of the network architecture. Moreover, they can be called by standard Java interfaces, such as Java Database Connectivity (JDBC), and by programmatic interfaces and development tools, such as SQLJ, Oracle Call Interface (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 Java facilities of the server while continuing to write applications for a preferred programmatic interface.

Scalability

Stored procedures increase scalability by isolating application processing on the server. In addition, automatic dependency tracking for stored procedures helps in developing 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

After a stored procedure is validated, you can use it with confidence in any number of applications. If its definition changes, then 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 computers.

Interoperability

Java in Oracle Database fully conforms to the Java Language Specification (JLS) 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. As a result, any procedure that is written in PL/SQL can also 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.

Oracle Database 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 stored procedures from one Oracle Database instance to another. This enables you to use stored procedures to implement a central set of business rules. Once you write the procedures, you can replicate and distribute them 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
- Bytecode verification of JAR files imported from an external source.

In Oracle Database, all classes are loaded into a secure database and, therefore, are untrusted. A user requires the appropriate permissions to access classes and operating system resources. Likewise, all stored procedures are secured against other users. You can grant the `EXECUTE` database privilege to users who need to access the stored procedures.

You can restrict access to Oracle data by allowing users to manipulate the data only through stored procedures that run 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.

> **See Also:** Chapter 10, "Security for Oracle Database Java Applications"

Java Stored Procedure Configuration

To configure the database to run Java stored procedures, you must decide on which of the following modes the database should run:

- 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.

Java, SQL, or PL/SQL clients, which run Java stored procedures on the server, connect to the database over an Oracle Net Services connection.

> **See Also:** *Oracle Database Net Services Administrator's Guide*

Java Stored Procedures Steps

You can run Java stored procedures in the same way as PL/SQL stored procedures. Normally, a call to a Java stored procedure is a result of database manipulation, because it is usually the result of a trigger or SQL DML call. To call 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 instance and publish them to SQL. Loading and publishing are separate tasks. Many Java classes, which are referenced only by other Java classes, are never published.

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

The following steps are involved in creating, loading, and calling Java stored procedures:

- Step 1: Create or Reuse the Java Classes
- Step 2: Load and Resolve the Java Classes
- Step 3: Publish the Java Classes

- Step 4: Call the Stored Procedures
- Step 5: Debug the Stored Procedures, if Necessary

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

See Also: Chapter 8, "Java Stored Procedures Application Example"

Step 1: Create or Reuse the Java Classes

Use a preferred Java IDE to create classes, or reuse existing classes that meet your requirements. Oracle Database supports many Java development tools and client-side programmatic interfaces. For example, the Oracle JVM accepts programs developed in popular Java IDEs, such as Oracle JDeveloper, Symantec Visual Cafe, 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.";
  }
}
```

Save the class as `Oscar.java`. Using a Java compiler, compile the `.java` file on your client system, as follows:

```
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 `loadjava` utility, you can load Java source, class, and resource files into an Oracle Database instance, 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, the `Oscar` class is loaded into the schema of the user you log in as, in this case, `scott`.

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

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

Step 3: Publish the Java Classes

For each Java method that can be called from SQL, you must write a call specification, which exposes the top-level entry point of the method to Oracle Database. Typically, only a few call specifications are needed. If preferred, you can generate these call specifications using Oracle JDeveloper.

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

```
SQL> connect scott/tiger

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

See Also: Chapter 6, "Publishing Java Classes With Call Specifications"

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 the stored procedures from the top level, for example, from SQL*Plus. Stored procedures can also be called 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.
```

See Also: Chapter 7, "Calling Stored Procedures"

Step 5: Debug the Stored Procedures, if Necessary

Debugging the stored procedures is mandatory. Your Java stored procedures run remotely on a server that resides on a separate computer. However, the JDK debugger, `jdb`, cannot debug remote Java programs. As a result, you need to debug your stored procedures on your client computer before you load them on to the database.

See Also: "Debugging Server Applications" on page 3-9

6

Publishing Java Classes With Call Specifications

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

This chapter contains the following sections:

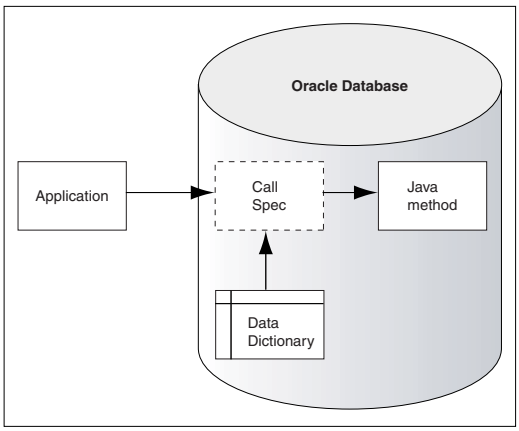
Understanding Call Specifications

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

You publish Java methods that return a value 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.

Figure 6–1 shows applications calling the Java method through its call specification, that is, by referencing the name of the call specification. The run-time system looks up the call specification definition in the Oracle data dictionary and runs the corresponding Java method.

Figure 6–1 Calling a Java Method



As an alternative, you can use the native Java interface to directly call Java methods in the database from a Java client.

See Also: "Using the Native Java Interface" on page 3-11

Defining Call Specifications

A call specification 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 specification as a:

- Standalone PL/SQL function or procedure
- Packaged PL/SQL function or procedure
- Member method of a SQL object type

A call specification exposes the top-level entry point of a Java method to Oracle Database. As a result, you can publish only `public static` methods. However, there is an exception. You can publish instance methods as member methods of a SQL object type.

Packaged call specifications perform as well as top-level call specifications. As a result, to ease maintenance, you may want to place call specifications in a package body. This will help you to modify call specifications without invalidating other schema objects. Also, you can overload the call specifications.

This section covers the following topics:

- Setting Parameter Modes
- Mapping Data Types
- Using the Server-Side Internal JDBC Driver

Setting Parameter Modes

In Java and other object-oriented languages, a method cannot assign values to objects passed as arguments. 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 specification. The corresponding Java parameter must be an array with only one element.

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

> **Note:** A function that declares `OUT` or `IN OUT` parameters cannot be called from SQL data manipulation language (DML) statements.

Mapping Data Types

In a call specification, the corresponding SQL and Java parameters and function results must have compatible data types. Table 6–1 lists the legal data type mappings. Oracle Database converts between the SQL types and Java classes automatically.

Table 6–1 Legal Data Type 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

Table 6–1 (Cont.) Legal Data Type Mappings

SQL Type	Java Class
RAW, LONG RAW	oracle.sql.RAW
	byte[]
ROWID	oracle.sql.CHAR
	oracle.sql.ROWID
	java.lang.String
BFILE	oracle.sql.BFILE
BLOB	oracle.sql.BLOB
	oracle.jdbc2.Blob
CLOB, NCLOB	oracle.sql.CLOB
	oracle.jdbc2.Clob
OBJECT	oracle.sql.STRUCT
Object types	java.sql.Struct
	java.sql.SqlData
	oracle.sql.ORAData
REF	oracle.sql.REF
Reference types	java.sql.Ref
	oracle.sql.ORAData
TABLE, VARRAY	oracle.sql.ARRAY
Nested table types and VARRAY types	java.sql.Array
	oracle.sql.ORAData
any of the preceding SQL types	oracle.sql.CustomDatum
	oracle.sql.Datum

You also need to consider the following:

- The UROWID type and the NUMBER subtypes, such as INTEGER and REAL, are not supported.
- A value larger than 32 KB cannot be retrieved from a LONG or LONG RAW column into a Java stored procedure.
- Java wrapper classes, such as java.lang.Byte and java.lang.Short, are useful for returning NULL from SQL.
- The following member must be defined when using the oracle.sql.CustomDatum class to declare parameters:

```
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. Similarly, the value returned by a method with the 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 do not require any character set conversions, apart from the usual network conversions. These classes are especially useful in applications that move data between SQL and Java.

Using the Server-Side Internal JDBC Driver

Java Database Connectivity (JDBC) enables you establish a connection to the database using the `DriverManager` class, which manages a set of JDBC drivers. You can use the `getConnection()` method after loading the JDBC drivers. When the `getConnection()` method finds the right driver, it returns a `Connection` object that represents a database session. All SQL statements are run within the context of that session.

However, the server-side internal JDBC driver runs within a default session and a default transaction context. As a result, 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 preregistered. To get a `Connection` object, run the following line of code:

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

Use the `Statement` class for SQL statements that do not take `IN` parameters and are run only once. When called on a `Connection` object, the `createStatement()` method returns a new `Statement` object, as follows:

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

Use the `PreparedStatement` class for SQL statements that take `IN` parameters or are run more than once. The SQL statement, which can contain one or more parameter placeholders, is precompiled. A question mark (?) serves as a placeholder. When called on a `Connection` object, the `prepareStatement()` method returns a new `PreparedStatement` object, which contains the precompiled SQL statement. For 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 meet the search condition. You can use the `next()` method to move to the next row, which then becomes the current row. You can use the `get`*`XXX`*`()` methods to retrieve column values from the current row. For example:

```
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 `IN OUT` 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 applications that access stored procedures, you need to consider the following:

- The server-side internal JDBC driver runs within a default session and default transaction context. 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 that implemented by Java Transaction API (JTA) or Java Transaction Service (JTS).
- Statements and result sets persist across calls and their finalizers do not release database cursors. To avoid running out of cursors, close all statements and result sets after you have finished using them. Alternatively, you can ask your DBA to raise the limit set by the initialization parameter, `OPEN_CURSORS`.
- The server-side internal JDBC driver does not support auto-commits. As a result, 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 Oracle Call Interface (OCI) driver.
- You cannot close the physical connection to the database established by the server-side internal JDBC driver. However, if you call the `close()` method on the default connection, all connection instances that reference the same object are cleaned up and closed. To get a new connection object, you must call `getConnection()` again.

See Also: *Oracle Database JDBC Developer's Guide and Reference*

Writing Top-Level Call Specifications

In SQL*Plus, you can define top-level call specifications 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` is represented by the following syntax:

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

The `AUTHID` clause determines whether a stored procedure runs with the privileges of its definer or invoker, which is 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 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 variables. Otherwise, results might vary across sessions.

The DETERMINISTIC option helps the optimizer avoid redundant function calls. If a stored function was called previously with the same arguments, then the optimizer can decide to use the previous result. The function result should not depend on the state of session variables or schema objects. Otherwise, results can vary across calls. Only DETERMINISTIC functions can be called from a function-based index or a materialized view that has query-rewrite enabled.

The string in the NAME clause uniquely identifies the Java method. The fully-qualified Java names and the call specification parameters, which are mapped by position, must correspond. However, this rule does not apply to the main() method. If the Java method does not take any arguments, then write an empty parameter list for it, but not for the function or procedure.

Write fully-qualified Java names using the dot notation. The following example shows that the fully-qualified names can be broken across lines at dot boundaries:

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

This section provides the following examples:

- Example 6–1, "Publishing a Simple JDBC Stored Procedure"
- Example 6–2, "Publishing the main() Method"
- Example 6–3, "Publishing a Method That Returns an Integer Value"
- Example 6–4, "Publishing a Method That Switches the Values of Its Arguments"

Example 6–1 Publishing a Simple JDBC Stored Procedure

Assume that the executable for the following Java class has been loaded into the 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());
    }
  }
}
```

The `GenericDrop` class has one method, `dropIt()`, which drops any kind of schema object. For example, if you pass the `table` and `emp` arguments to `dropIt()`, then the method drops the database table `emp` from your schema.

The call specification for the `dropIt()` method is as follows:

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

Note that you must fully qualify the reference to `String`. The `java.lang` package is automatically available to Java programs, but must be named explicitly in the call specifications.

Example 6–2 Publishing the main() Method

As a rule, Java names and call specification parameters must correspond. However, that rule does not apply to the `main()` method. Its `String[]` parameter can be mapped to multiple `CHAR` or `VARCHAR2` call specification parameters. Consider the `main()` method in the following class, 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 `main()`, write the following call specification:

```
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, and `NOT NULL`, on the call specification parameters. As a result, you cannot specify a maximum size for the `VARCHAR2` parameters. However, you must do so for `VARCHAR2` variables, as in:

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

Example 6–3 Publishing a Method That Returns an Integer Value

In the following example, the `rowCount()` method, which returns the number of rows in a given database table, is published:

```
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;
  }
}
```

NUMBER subtypes, such as INTEGER, REAL, and POSITIVE, are not allowed in a call specification. As a result, in the following call specification, the return type is NUMBER and not INTEGER:

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

Example 6–4 Publishing a Method That Switches the Values of Its Arguments

Consider the swap() method in the following Swapper class, 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 specification publishes the swap() method as a call specification, swap(). The call specification declares IN OUT formal parameters, because values must be passed in and out. All call specification 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[])';
```

> **Note:** A Java method and its call specification can have the same name.

Writing Packaged Call Specifications

A PL/SQL package is a schema object that groups logically related types, items, and subprograms. Usually, packages have two parts, a specification and a body. The specification is the interface to your applications and declares the types, constants, variables, exceptions, cursors, and subprograms available for use. The body defines the cursors and subprograms.

In SQL*Plus, you can define PL/SQL packages interactively, using the following 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 specification 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 is the body, which is the optional initialization part. It holds statements that initialize package variables. It is run only once, the first time you reference the package.

A call specification declared in a package specification cannot have the same signature, that is, the name and parameter list, as a subprogram in the package body. If you declare all the subprograms in a package specification as call specifications, then the package body is not required, unless you want to define a cursor or use the initialization part.

The `AUTHID` clause determines whether all the packaged subprograms run with the privileges of their definer, which is the default, or invoker. It also determines whether unqualified references to schema objects are resolved in the schema of the definer or invoker.

Example 6–5 provides an example of packaged call specification.

Example 6–5 Packaged Call Specification

Consider a Java class, `DeptManager`, which consists of methods for adding a new department, dropping a department, and changing the location of a department. Note that the `addDept()` method uses a database sequence to get the next department number. The three methods are logically related, and therefore, you may want to group their call specifications 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 the methods `addDept()`, `dropDept()`, and `changeLoc()`. First, you must create the package specification, 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 must create the package body by writing the call specifications for the Java methods, as follows:

```
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 `dept_mgmt` package, use the dot notation, as follows:

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

Writing Object Type Call Specifications

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 required 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 and a body. The specification is the interface to your applications and declares a data structure, which is a set of attributes, along with the operations or methods required to manipulate the data. The body implements the specification by defining PL/SQL subprogram bodies or call specifications.

If the specification declares only attributes or call specifications, then the body is not required. If you implement all your methods in Java, then you can place their call specifications in the specification part of the object type and omit the body part.

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

```
CREATE [OR REPLACE] TYPE type_name
  [AUTHID {CURRENT_USER | DEFINER}] {IS | AS} OBJECT (
  attribute_name data_type[, attribute_name data_type]...
  [{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 run with the current user privileges, which determines the invoker's or definer's rights.

This section covers the following topics:

- Declaring Attributes
- Declaring Methods

Declaring Attributes

In an object type specification, all attributes must be declared before any methods are. In addition, you must declare at least one attribute. The maximum number of attributes that can be declared is 1000. Methods are optional.

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

After declaring attributes, you can declare 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 called. MEMBER methods are called on instances, as follows:

```
instance_expression.method()
```

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

```
object_type_name.method()
```

If you want to call a Java method that is not static, then you must specify the keyword MEMBER in its call specification. Similarly, if you want to call a static Java method, then you must specify the keyword STATIC in its call specification.

This section contains the following topics:

- Map and Order Methods
- Constructor Methods
- Examples

Map and Order Methods

The values of a SQL scalar data type, such as `CHAR`, have a predefined order and, therefore, can be compared with other values. 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 function without any parameters and with one of the following return types: `DATE`, `NUMBER`, or `VARCHAR2`.

Alternatively, you can supply SQL with an `order` method, which compares two objects. An `order` method takes only two parameters: the built-in parameter, `SELF`, and another object of the same type. If `o1` and `o2` are objects, then 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 less than, equal to, or greater than the other parameter, respectively. 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, then you can compare objects in SQL and PL/SQL. However, if you declare neither method, then you can compare objects only in SQL and solely for equality or inequality.

Note: 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, 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 Database 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 data types. SQL never calls a constructor implicitly. As a result, 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 specification 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 specification for the object type `Department`. The body is not required, because the specification declares only attributes. The specification is as follows:

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

Then, you create the object type Employee. The deptno attribute stores a handle, called a REF, to objects of the 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 data type REF and the object type that the REF targets. The Employee type is created as follows:

```
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 the SQL object tables to hold objects of type Department and Employee. Create the depts object table, which will hold objects of the Department type. Populate the object table by selecting data from the dept relational table and passing it to a constructor, which is a system-defined function with the same name as the object type. Use the constructor to initialize and return an instance of that object type. The depts table is created as follows:

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

Create the emps object table, which will hold objects of type Employee. The last column in the emps object table, which corresponds to the last attribute of the Employee object type, holds references to objects of type Department. To fetch the references into this column, use the operator REF, which takes a table alias associated with a row in an object table as its argument. The emps table is created as follows:

```
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 the oracle.sql.REF class, which supports Oracle object references. This class, which is a subclass of oracle.sql.Datum, extends the standard JDBC interface, oracle.jdbc2.Ref.

Using Class oracle.sql.STRUCT

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

```
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 `wages()` method returns a value, you write a function call specification 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 specification, because it is not defined inside a package or object type.

Implementing the SQLData Interface

To make access to object attributes more natural, create a Java class that implements the `SQLData` interface. To do so, you must provide the `readSQL()` and `writeSQL()` methods as defined by the `SQLData` interface. The JDBC driver calls the `readSQL()` method to read a stream of database values and populate an instance of your Java class. In the following example, you revise `Paymaster` by adding a second method, `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 specification for `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`, write a procedure call specification 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 specification.

Implementing Object Type Methods

Assume you decide to drop the top-level call specifications `wages` and `raise_sal` and redeclare them as methods of the object type `Employee`. In an object type specification, all methods must be declared after the attributes. The body of the object type is not required, because the specification declares only attributes and call specifications. The `Employee` object type can be re-created as follows:

```
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 `Paymaster` accordingly. You need not pass an array to `raiseSal()`, because the SQL parameter `SELF` corresponds directly to the Java parameter `this`, even when `SELF` is declared as `IN OUT`, which is 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 describes the procedure for calling Java stored procedures in various contexts. It also describes how Oracle Java virtual machine (JVM) handles SQL exceptions.

This chapter contains the following sections:

Calling Java from the Top Level

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

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

where `param` is represented by the following syntax:

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

Host variables are variables that are declared in a host environment. They 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 subprogram without parameters must be called with an empty parameter list:

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

This section covers the following topics:

Redirecting Output

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

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

The minimum buffer size is 2,000 bytes, which is also the default size, and the maximum buffer 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);
```

The output is printed when the stored procedure exits.

Examples of Calling Java Stored Procedures From the Top Level

This section provides the following examples

- Example 7–1, "A Simple JDBC Stored Procedure"
- Example 7–2, "Fibonacci Sequence"

Example 7–1 A Simple JDBC Stored Procedure

In the following example, the `main()` method accepts the name of a database table, such as `emp`, and an optional `WHERE` clause specifying a condition, such as `sal > 1500`. If you omit the condition, then the method deletes all rows from the table, else it 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 `main()` method 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, this clause is not allowed in a call specification. As a result, you must overload two packaged 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 can call the `delete_rows` procedure, as follows:

```
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.
```

Note: You cannot overload top-level procedures.

Example 7–2 Fibonacci Sequence

Assume that the executable for the following Java class is stored in 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 `Fibonacci` class has a method, `fib()`, which returns the *n*th Fibonacci number. The Fibonacci sequence, 1, 1, 2, 3, 5, 8, 13, 21, . . ., is recursive. Each term in the sequence, after the second term, is the sum of the two terms that immediately precede it. Because `fib()` returns a value, you must publish it as a function, as follows:

```
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 and initialize the first one:

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

PL/SQL procedure successfully completed.
```

Now, you can call the `fib()` function. In a `CALL` statement, host variables must be prefixed with a colon. The function can be called, as follows:

```
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 that is associated with a specific table or view. Oracle Database runs the trigger automatically whenever a data manipulation language (DML) operation affects the table or view.

When a triggering event occurs, the trigger runs and either a PL/SQL block or a `CALL` statement performs the action. A statement trigger runs once, before or after the triggering event. A row trigger runs once for each row affected by the triggering event.

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

The following are examples of calling Java stored procedures from a database trigger:

- Example 7–3, "Calling Java Stored Procedure from Database Trigger - I"
- Example 7–4, "Calling Java Stored Procedure from Database Trigger - II"

Example 7–3 Calling Java Stored Procedure from Database Trigger - I

Assume 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 `DBTrigger` class has one method, `logSal()`, which inserts a row into the `sal_audit` table. Because `logSal()` is a `void` method, you must 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, create the `sal_audit` table, as follows:

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

Finally, create the database trigger, which fires when a salary increase exceeds 20 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 run the following `UPDATE` statement, it updates all rows in the `emp` table:

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

For each row that meets the condition set in the `WHEN` clause of the trigger, the trigger runs and the Java method inserts a row into the `sal_audit` table.

```
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 7–4 Calling Java Stored Procedure from Database Trigger - II

Assume you want to create a trigger that inserts rows into a database view, which is 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;
```

The `sales` and `mktg` database tables 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, the trigger will insert rows into the base tables.

First, add the following Java method to the `DBTrigger` class, which is defined in Example 7–3:

```
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 `addEmp()` method inserts a row into the `sales` or `mktg` table depending on the value of the `deptName` parameter. Write the call specification 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)';
```

Next, create the `INSTEAD OF` trigger, as follows:

```
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 run each of the following INSERT statements, the trigger runs 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, then 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 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 `Formatter` class has the `formatEmp()` method, which returns a formatted string containing a staffer's name and job status. Write the call specification 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';
```

Now, call the `format_emp` function 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

A Java method must adhere to the following rules, which are meant to control side effects:

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

If any SQL statement inside the method violates any of the preceding rules, then you get an error at run time.

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 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 `Adjuster` class has one method, which raises the salary of an employee by a given percentage. Because `raiseSalary()` is a `void` method, you must 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 `raise_salary` procedure 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 following example, you call the `row_count` function, which defined in Example 6–3 on page 6-8, 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 following example, you call the `raise_sal` method of the `Employee` object type, which is defined in "Implementing Object Type Methods" on page 6-18, 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

Java Database Connectivity (JDBC) and SQLJ enable you to call PL/SQL stored functions and procedures. For example, 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;
```

In a JDBC program, a call to the `balance` function can be written as follows:

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

In a SQLJ program, the call can be written as follows:

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

How Oracle JVM Handles Exceptions

Java exceptions are objects and have a naming and inheritance hierarchy. As a result, you can substitute a subexception, that is, a subclass of an exception class, for its superexception, that is, the superclass of an exception class.

All Java exception objects support the `toString()` method, 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 runs a SQL statement, any exception thrown is materialized to the procedure as a subclass of `java.sql.SQLException`. This class has the `getErrorCode()` and `getMessage()` methods, which return the Oracle error code and message, respectively.

If a stored procedure called from SQL or PL/SQL throws an exception and is not caught by Java, then the following error message appears:

```
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

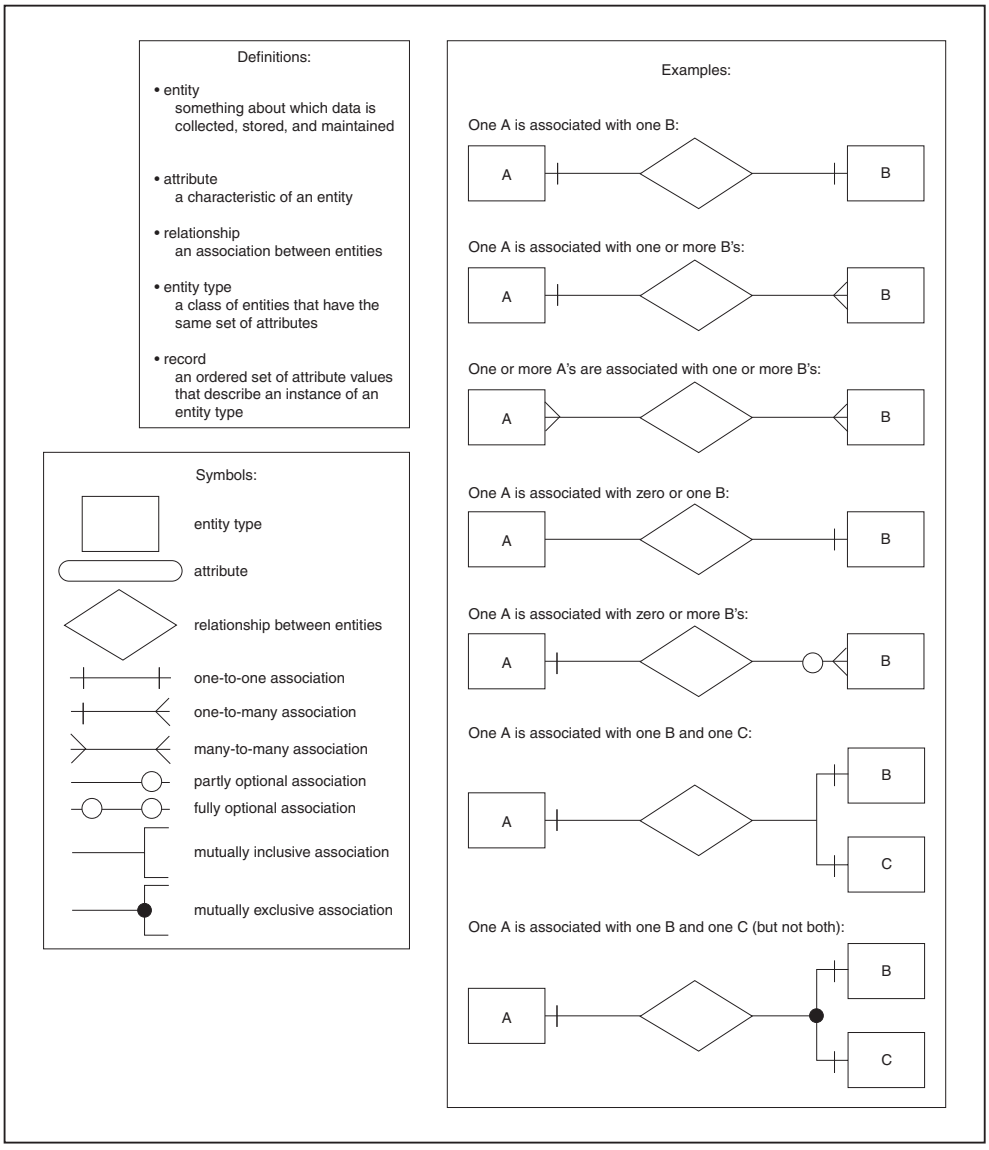
This chapter describes how to build a Java application with stored procedures. By following the steps mentioned in this chapter from the design phase to the actual implementation, you can write your own applications.

This chapter contains the followings sections:

Drawing the Entity-Relationship Diagram

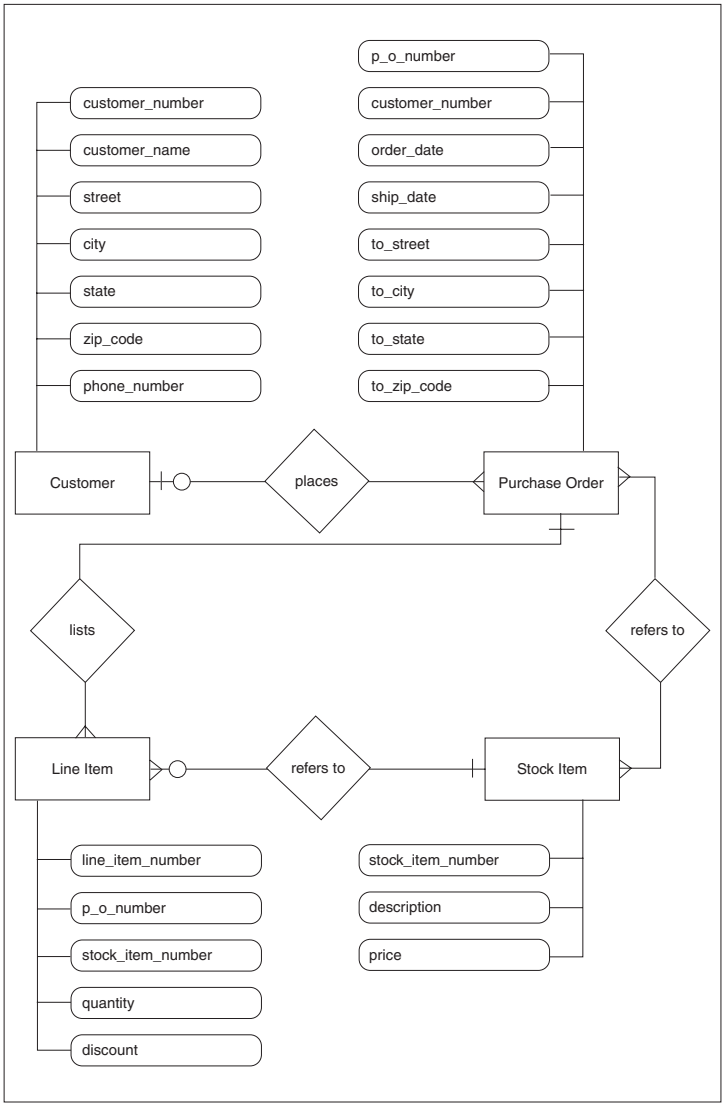
The objective is to develop a simple system for managing customer purchase orders. First, you must identify the business entities involved and their relationships. To do this, you must draw an entity-relationship (E-R) diagram by following the rules and examples given in Figure 8–1.

Figure 8–1 Rule 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



`Customer` has a one-to-many relationship with `Purchase Order` because a customer can place one or many orders, but a given purchase order can be placed by only one customer. The relationship is optional because zero customers may place a given order. For example, an order may be placed by someone previously not defined as a customer.

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

`Line Item` has a many-to-one relationship with `Stock Item` 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 may refer to a given stock item.

Planning the Database Schema

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

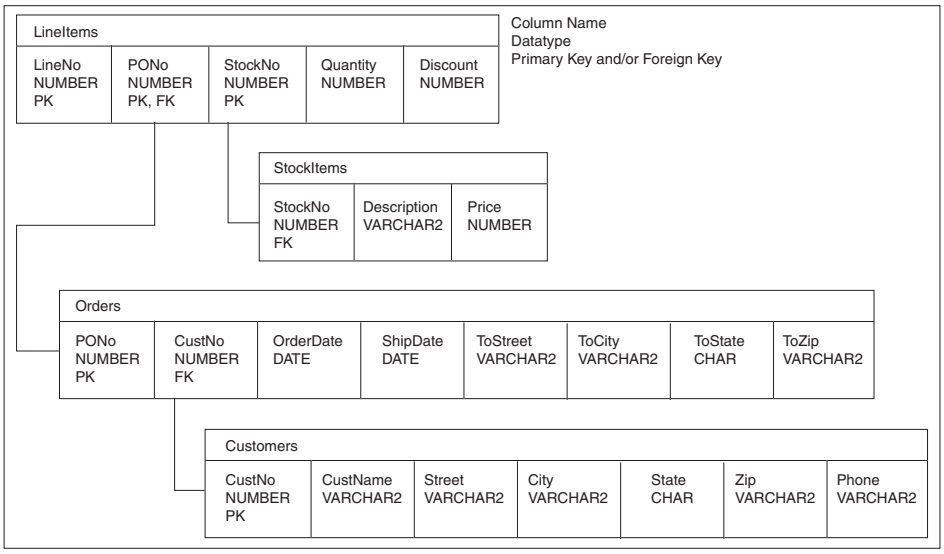
- `Customers`
- `Orders`
- `LineItems`
- `StockItems`

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

Figure 8–3 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, the `PONo` column in the `LineItems` table is a foreign key matching the primary key in the `Orders` table. Every purchase order number in the `LineItems.PONo` column must also appear in the `Orders.PONo` column.

Figure 8–3 Schema Plan for Purchase Order Application



Creating the Database Tables

After planning the database schema, create the database tables required by the schema plan. You begin by defining the `Customers` table, 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 `Customers` table stores information about customers. Essential information is defined as `NOT NULL`. For example, every customer must have a shipping address. However, the `Customers` table does not manage the relationship between a customer and his or her purchase order. As a result, this relationship must be managed by the `Orders` table, which you can define as follows:

```
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 `LineItems` table manages these relationships using foreign keys. For example, the `StockNo` foreign key column in `LineItems` references the `StockNo` primary key column in `StockItems`, which you can define as follows:

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

The `Orders` table manages the relationship between a customer and purchase order using the `CustNo` foreign key column, which references the `CustNo` primary key column in `Customers`. However, `Orders` does not manage the relationship between a purchase order and its line items. As a result, this relationship must be managed by `LineItems`, which you can define as follows:

```
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

After creating the database tables, you consider the operations required in a purchase order system and write the appropriate Java methods. In a simple system based on the tables defined in the preceding examples, you need methods for registering customers, stocking parts, entering orders, and so on. You can implement these methods in a 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

After writing the Java classes, use the `loadjava` command-line utility to upload your Java stored procedures into 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
```

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

Publishing the Java Classes

After loading the Java classes, publish your Java stored procedures in the Oracle data dictionary. To do this, you must write call specifications that map Java method names, parameter types, and return types to their SQL counterparts.

The methods in the `POManager` Java class are logically related. You can group their call specifications in a PL/SQL package. To do this, first, create the package specification, 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, create the package body by writing call specifications for the Java methods, as follows:

```
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

After publishing the Java classes, call your Java stored procedures from the top level and from database triggers, SQL data manipulation language (DML) statements, and PL/SQL blocks. Use the dot notation to reference these stored procedures in the `po_mgr` package.

From an anonymous PL/SQL block, you may 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;
```

Register your customers, as follows:

```
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;
```

Enter the purchase orders placed by various customers, as follows:

```
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;
```

In SQL*Plus, after redirecting output to the SQL*Plus text buffer, you can call the `totalOrders()` method, 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

Oracle Database Java Application Performance

You can enhance the performance of your Java application using the following:

Natively Compiled Code

The Java language was designed for a platform-independent and 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 can natively compile certain classes. For example, you can 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, such as `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 Java virtual machines (JVMs) use Just-In-Time (JIT) compilers that convert Java bytecodes to native machine instructions when methods are called. Accelerators use an Ahead-Of-Time approach for recompiling the Java classes. The following table briefly describes the native compilers:

Native Compiler	Description
Just-In-Time	Provides JVM the ability to translate Java instructions just before they are needed by the application. The benefits depend on how accurately the native compiler anticipates code branches and the next instruction. If incorrect, then no performance gain is realized.
Ahead-Of-Time	An accelerator natively compiles all Java code within a Java Archive (JAR) file to native shared libraries organized by a Java package, before run time. At run time, accelerator checks if a Java package has been natively compiled. If so, then the accelerator 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 traversed on the server. After compilation, the `loadjava` utility can be used to load the statically compiled libraries into Oracle Database, which are then shared between users, processes, and sessions.

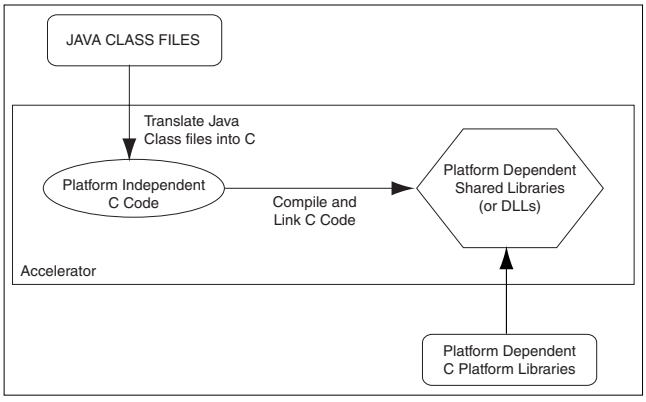
This sections covers the following topics:

- Accelerator Overview
- Oracle Database Core Java Class Libraries
- Natively Compiling Java Application Class Libraries
- Running the Accelerator
- The ncomp Tool
- Native Compilation Usage Scenarios
- The deploync Tool
- The statusnc Tool

Accelerator Overview

Most Ahead-Of-Time native compilers compile the bytecode directly into a platform-dependent language. For portability requirements, this is not feasible. Figure 9–1 illustrates how an 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 dynamic-link libraries (DLLs).

Figure 9–1 Native Compilation Using Accelerator



Given a JAR file, an 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 called.
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 the code on the intended platform environment in 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 libraries, which are natively compiled by the accelerator, can be used only within Oracle Database. In addition, these libraries can be used only within the same version of Oracle Database in which it was produced. If you want your application to be natively compiled on subsequent releases, then 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 are 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 Sun Solaris and `orajox10java_lang.dll` on Microsoft Windows hold `java.lang` classes. Specifics of packaging and naming can vary from platform to platform. The Oracle JVM uses natively compiled Java files internally and opens them, as necessary, at run time.

Natively Compiling Java Application Class Libraries

The command-line tool, `ncomp`, is an accelerator that natively compiles your code and loads it in Oracle Database. However, to use `ncomp`, you must first satisfy the installation requirements.

You must perform the following before calling the accelerator:

1. Install a C compiler for the intended platform on the computer 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. Because the compiler and linker information is platform-specific, the configuration for these items is detailed in the README for your platform.
3. Set the `JAVA_HOME` environment variable to the location where your Java Development Kit (JDK) is installed.
4. Grant the user that runs `ncomp` the following role and security permissions:

   a. `JAVA_DEPLOY`

The user must be assigned the JAVA_DEPLOY role to be able to deploy the shared libraries on the server, which can be performed using the ncomp and deploync utilities. For example, this role can be assigned to the user DAVE, as follows:

```
SQL> GRANT JAVA_DEPLOY TO DAVE;
```

b. FilePermission

The accelerator stores the shared libraries with the natively compiled code on the server. To enable the 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 of granting FilePermission for all desired directories is to grant the user the JAVASYSPRIV role, as follows:

```
SQL> GRANT JAVASYSPRIV TO DAVE;
```

You must run the following grant by a party that has the permission to run this grant, given that $ORACLE_HOME is equivalent to /private/oracle on the server:

```
call dbms_java.grant_permission(<user>, 'java.io.FilePermission',
'/private/oracle/-', 'read,write');
```

The following example demonstrates Larry giving Dave FilePermission for $ORACLE_HOME, where $ORACLE_HOME is located in /private/oracle. You cannot provide an environment variable to the grant_permission() method. Once completed, Dave can run ncomp.

```
connect larry/larry

REM Grant DAVE permission to read and write the ncomp file.
call dbms_java.grant_permission('DAVE', 'java.io.FilePermission',
'/private/oracle/-','read,write');

REM commit the changes to the PolicyTable
commit;
```

Note: DBA role contains both the JAVA_DEPLOY role and FilePermission for all files under $ORACLE_HOME.

Running the Accelerator

All the Java classes contained within a JAR file must already be loaded within the database. Run the ncomp tool to instruct the 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 run for 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 computer.

If you change any of the classes within this JAR file, then the 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 natively compiled previously, then run the ncomp with the -force option, as follows:

```
ncomp -user scott/tiger -force pubProject.JAR
```

The ncomp Tool

The accelerator, implemented within the ncomp tool, natively compiles all classes within the specified JAR or ZIP file or the list of classes. It 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, then you can natively compile your Java classes, as follows:

```
ncomp -user SCOTT/TIGER myClasses.jar
```

There are options that let you control the details of native compilation.

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]
```

Argument Summary

Table 9–1 summarizes the ncomp arguments. *class_designation_file* can be a *file*.jar, *file*.zip, or *file*.classes file.

Table 9–1 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 running ncomp in the directory where the JAR file exists and you do not specify the -projectDir option, then 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 running ncomp in the directory where the ZIP file exists and you do not specify the -projectDir option, then you may give only the name of the ZIP file.

Table 9–1 (Cont.) ncomp Argument Summary

Argument	Description and Values
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 running ncomp in the directory where the classes file exists and you do not specify the -projectDir option, then you may give only the name of the classes file.
-user | -u *username/password* [@database_url]	Specifies a user, password, and database connect string. The files will be loaded into this database instance. If you specify the database URL on this option, then you must specify it with Oracle Call Interface (OCI) syntax. To provide a JDBC Thin database URL, use the -thin option.
-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.
-load	Runs loadjava on the specified class designation file. You cannot use this option in combination with a *file*.classes file.
-outputJarFile *jar_filename*	The output of all natively compiled classes is stored in 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 with the same name as the input file with _depl.jar appended as the suffix. If directory is not supplied, then the tool stores the output JAR file into the project directory, which is 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 nor -thin are specified, then it is assumed that you used an OCI database URL.
-projectDir | -d *absolute_path*	Specifies the full path for the project directory. If not specified, then the accelerator uses the directory from which ncomp is called as the project directory. This directory must exist. The tool will not create this directory for you. If it does not exist, then the current directory is used.
-update	If you add more classes to *class_designation_file* that has already been natively compiled, then this flag informs the accelerator to update the deployment JAR file with the new classes. Thus, the 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

Table 9–1 summarizes all the arguments. This section explains the following arguments in detail.

- user

```
{-user | -u} user/password[ @database_url ]
```

The permissible forms of `@database_url` depend on whether you specify `-oci`, which is the default, or `-thin`:

– `-oci:@database_url` is optional. If you do not specify, then `ncomp` uses the user's default database. If specified, then `database_url` can be a TNS name or an Oracle Net Services name-value list.

– `-thin:@database_url` is required. The format is `host:lport:SID`.

  where:

  * `host` is the name of the computer 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`

  The accelerator places compilation information and the compiled shared libraries in a single JAR file, copies the shared libraries to `$ORACLE_HOME/javavm/admin` on the server, and deploys the compilation information to the server. If you want to place the shared libraries on the server yourself, then you can do so using the `lightweightDeployment` option. This option enables you to perform the deployment in two stages:

  1. Natively compile your JAR file with the `-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 in the deployment JAR file. As a result, the deployment JAR file is much smaller.

  2. Deploy as follows:

     Copy all output shared libraries from the `lib` directory of the native compilation project directory to `$ORACLE_HOME/javavm/admin` on the server.

     Note: You need to have `FilePermission` to write to this directory. `FilePermission` is included in the `DBA` and `JAVASYSPRIV` roles.

     Deploy the lightweight deployment JAR file to the server using `deploync`.

Errors

Any error that occurs during native compilation is printed to the screen. Errors that occur during deployment of shared libraries to the server or during run time 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, then the 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 to load the correct Java class into the database instead of recompiling the classes, then the corrected class and all classes included in the resolution validation for that class are processed in the interpreted mode. These classes are processed regardless of whether they are located within the same shared library or in a different shared library. This means that the JVM will not run these classes natively. All the other natively compiled classes will continue to run in the native format. When you run the `statusnc` command on the reloaded class or any of its referred classes, they will have a `NEED_NCOMPING` status message.

The possible errors for a Java class are:

- The Java class does not exist in the database. If you do not load the Java class into Oracle Database, then the accelerator does not include the class in the shared library. The class is skipped.
- The Java class is invalid. That is, one of its references may not be found.
- For any Java class that is unresolved, the accelerator will try to resolve it before natively compiling. However, if the class cannot be resolved, then it is ignored by the accelerator.

A possible error during deployment of natively compiled JAR file is that the native compilation of your JAR file runs correctly, but the deployment fails. In this case, do not recompile the JAR file, but deploy the natively compiled output JAR file with the `deploync` command.

Native Compilation Usage Scenarios

The following scenarios demonstrate how each of the `ncomp` tool options can be used:

- Natively Compiling on a Test Platform With Java Classes Already Loaded in the Database
- Natively Compiling Java Classes Not Loaded in the Database
- Clean Compile and Generate Output for Future Deployment
- Controlling Native Compilation Build Environment
- Natively Compiling Specific Classes
- Natively Compiling Packages That Are Fully or Partially Modified

Natively Compiling on a Test Platform With Java Classes Already Loaded in the Database

If all classes are loaded into the database and you have completed your testing of the application, then you can use the accelerator to natively compile the tested classes. The accelerator accepts a JAR or ZIP file or a file with a list of classes to determine the packages and classes to be included in the native compilation. It then retrieves all the designated classes from the server and natively compiles them into shared libraries. Each library contains a single package of classes.

Assuming that the classes have already been loaded on the server, you run 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, then the accelerator recompiles only the packages that contain the changed classes. It will not recompile all the 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 computer. After 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 command loads the classes through `loadjava` and then performs native compilation of classes in 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, then run `ncomp` with the `-force` option. You may 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 command forces a recompilation of all Java classes in the class designation file, `pubProject.jar`, and creates a deployment JAR file, `pubworks.jar`:

```
ncomp -user SCOTT/TIGER -force -outputJarFile pubworks.jar pubProject.jar
```

The deployment JAR file contains the shared libraries for your classes, and the 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 instance of the appropriate platform type, you must do the following:

1. Load the original Java classes into the destination server. For example, the class designation file, `pubProject.jar`, would be loaded into the database using the `loadjava` tool.
2. Deploy the natively compiled deployment JAR file using the `deploync` tool.

Controlling Native Compilation Build Environment

By default, the accelerator uses the directory where `ncomp` is run as its build environment. It downloads several class files into this directory and then uses this directory for the compilation and linking process.

If you do not want to the accelerator to store any of its files in the current directory, then create a working directory and specify it as the project directory with the `-projectDir` option. The following command directs the accelerator to use `/tmp/jaccel/pubComped` as the build directory.

```
ncomp -user SCOTT/TIGER -projectDir /tmp/jaccel/pubComped pubProject.jar
```

Note: The working directory must exist before specifying it with the `-projectDir` option. The accelerator will not create this directory for you.

Natively Compiling Specific Classes

You can specify one or more classes, which are to be natively compiled, in a `.classes` file. You can use the following syntax to specify packages or individual classes or both within this file:

- 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 subpackage. You must specify each package independently.

- 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.

Consider 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 command directs the accelerator to compile all the classes designated in the `pubworks.classes` file:

```
ncomp -user SCOTT/TIGER /tmp/jaccel/pubComped/pubworks.classes
```

All the classes in `myPackage` and `hisSubPackage` and the individual classes, `herClass` and `myClass`, are compiled.

Natively Compiling Packages That Are Fully or Partially Modified

If you change any of the classes within the deployment JAR file, then the 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 the classes in a JAR file to be recompiled, regardless of whether they were previously natively compiled, then you must run `ncomp` with the `-force` option, as follows:

```
ncomp -user scott/tiger -force pubProject.JAR
```

The deploync Tool

You can deploy any deployment JAR file with the `deploync` command. This includes the default output JAR file, *`file`*`_depl.jar`, or the JAR file created when you use 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 in 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 9–2 summarizes the `deploync` arguments.

Table 9–2 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 file.
`-user | -u` *username/password* `[@`*database_url*`]`	Specifies a user name, password, and database connect string. The files will be loaded into this database instance. If you specify the database URL on this option, then you must specify it with the 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, the accelerator uses the directory from which `ncomp` is called 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` nor `-thin` are specified, then the OCI database URL is assumed, by default.

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
```

The statusnc Tool

After the native compilation is completed, you can check the status of the Java classes using the `statusnc` command. This tool prints the status of each class either to the screen or to a designated file. In addition, it always saves the output in the `JACCELERATOR$STATUS` table. The values can be the following:

Native Compilation Class Status	Description
`ALREADY_NCOMPED`	The class is currently natively compiled.
`NEED_NCOMPING`	A class within the shared library was reloaded after native compilation. As a result, you should recompile this shared library.
`INVALID`	A class loaded in the database is invalid. The 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 run of the statusnc command. When run, statusnc cleans this table before writing the new records into it.

Syntax

```
statusnc [ options ] class_designation_file
  -user user/password[@database_url]
  [-output | -o filename]
  [-projectDir | -d directory]
  [-thin]
  [-oci | -oci8]
```

Argument Summary

Table 9–3 summarizes the statusnc arguments. The class_designation_file can be a .jar, .zip, or .classes file.

Table 9–3 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.
-user | -u *username/password* [@*database_url*]	Specifies a user name, password, and database connect string where the files are loaded. If you specify the database URL on this option, then you must specify it with OCI syntax. To provide a JDBC Thin database URL, use the -thin option.
-output *filename*	Designates that statusnc should send the 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, the accelerator uses the directory from which ncomp is called 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. However, if neither -oci nor -thin are specified, then the default assumes 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, run-time 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 System Global Area (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 your application needs more accurately:

- `SHARED_POOL_SIZE`

  Shared pool memory is used by the class loader within the JVM. The class loader, on an average, uses about 8 KB of memory for each loaded class. Shared pool memory is used when loading and resolving classes into the database. It is also used when compiling the 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 requires `SHARED_POOL_SIZE` to be set to 50 MB because 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 run time.

- `JAVA_POOL_SIZE`

  The Oracle JVM memory manager allocates all other Java state during run time 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.

- `JAVA_SOFT_SESSIONSPACE_LIMIT`

  This parameter lets you 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 Java state exceeds this size, Oracle JVM generates a warning that is written into the trace files. Although this warning is 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`

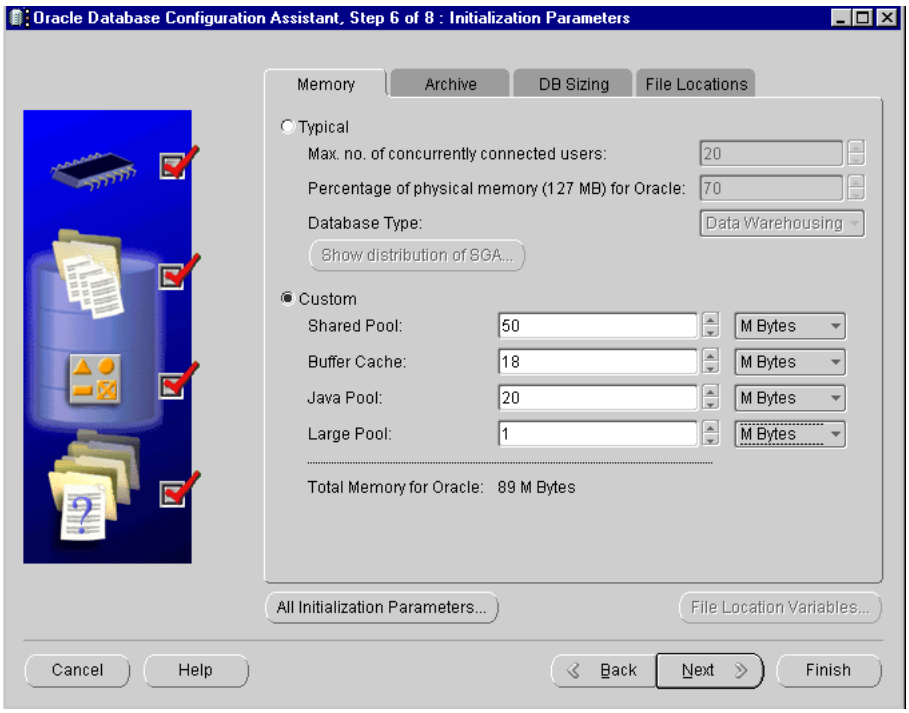
  If a Java program, which can be called by a user, running in the server can be used in a way that is not self-limiting in its memory usage, then this setting may be useful to place a hard limit on the amount of session space made available to it. The default is 4 GB. This limit is purposely set extremely high to be normally invisible.

When a user's session Java state attempts to exceeds this size, the application can receive an out-of-memory failure.

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 enables you to modify these values in the Memory section, as shown in Figure 9–2.

Figure 9–2 Configuring Oracle JVM 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 to 8 KB for each class.

- None of the per-session Java state of each session.

  For a dedicated server, this is stored in the User Global Area (UGA) within the Program Global Area (PGA), and 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 to 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 the 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, multiuser 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, then 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, then it can fail silently, leaving invalid classes in the database. Later, if you try to call or resolve any invalid classes, then a `ClassNotFoundException` or `NoClassDefFoundException` instance will be thrown at run time. 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.
- 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 enables you to catch missing classes at load time, rather than at run time.
- Double check the status of the newly loaded class by connecting to the database in the schema containing the class, and run 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 lost connection, then increase `SHARED_POOL_SIZE` and `JAVA_POOL_SIZE`, and try again.

10

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 Java virtual machine (JVM)-defined and user-defined classes. Security also includes bytecode verification of Java Archive (JAR) files imported from an external source. The following sections describe the various security support available for Java applications within Oracle Database:

- Network Connection Security
- Database Contents and Oracle JVM Security
- Database Authentication Mechanisms

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, either through Oracle Net or Java Database Connectivity (JDBC) connection. The following table provides the security description for Oracle Net and JDBC connections:

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: ■ A user name and password for client verification. For each connection request, a user name and password configured within Oracle Net has to be provided. ■ Advanced Networking Option for encryption, kerberos, or secureId. ■ SSL for certificate authentication.
JDBC	The JDBC connection security that is required is similar to the constraints required on an Oracle Net database connection.

See Also:

- *Oracle Database Net Services Administrator's Guide*
- *Oracle Database Advanced Security Administrator's Guide*
- *Oracle Database JDBC Developer's Guide and Reference*

Database Contents and Oracle JVM Security

Once you are connected to the database, you must have the appropriate Java2 security permissions and database privileges to access the resources stored within the database. These resources include:

- Database resources, such as tables and PL/SQL packages
- Operating system resources, such as files and sockets
- Oracle JVM classes
- User-loaded classes

These resources can be protected by the following methods:

Resource Security	Description
Database Resource Security	Authorization for database resources requires that database privileges, which are not the same as the Java2 security permissions, are granted to resources. For example, database resources include tables, classes, and PL/SQL packages. All user-defined classes are secured against users from other schemas. You can grant execution permission to other users or schemas through an option on the `loadjava` command.
JVM Security	Oracle JVM uses Java2 security, which uses `Permission` objects to protect operating system resources. Java2 security is automatically installed upon startup and protects all operating system resources and Oracle JVM classes from all users, except `JAVA_ADMIN`. The `JAVA_ADMIN` user can grant permission to other users to access these classes.

This section covers the following topics:

- Java2 Security
- Setting Permissions
- Debugging Permissions
- Permission for Loading Classes

See Also:

- *Oracle Database Application Developer's Guide - Fundamentals*
- Chapter 11, "Schema Objects and Oracle JVM Utilities"

Java2 Security

Each user or schema must be assigned the proper permissions to access operating system resources, such as sockets, files, and system properties.

Java2 security provides a flexible and configurable security for Java applications. With Java2 security, you can define exactly what permissions on each loaded object that a schema or role will have. In Oracle8*i* Database release 8.1.5, the following secure roles are available:

- `JAVAUSERPRIV`

  Few permissions, including examining properties

- `JAVASYSPRIV`

Major permissions, including updating the Oracle JVM-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 Oracle JVM security is based on Java2 security, you assign permissions on a class-by-class basis. These permissions are assigned through database management tools. Each permission is encapsulated in a `Permission` object and is stored within a `Permission` table. `Permission` contains the `target` and `action` attributes, which take `String` values.

Java2 security was created for the non-database world. When you apply the Java2 security model within the database, certain differences manifest themselves. For example, Java2 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. As a result, no classes are trusted.

The following table describes the differences between the Sun Microsystems Java2 security and the Oracle Database security implementation:

Java2 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 using the `-usepolicy` flag on the `java` command.	You must specify the policy within `PolicyTable`.
You can write your own `SecurityManager` or use the Launcher.	You can write your own `SecurityManager`. However, Oracle recommends that you use only the Oracle Database `SecurityManager` or that you extend it. If you want to modify the behavior, then you should not define a `SecurityManager`. Instead, you should extend `oracle.aurora.rdbms.SecurityManagerImpl` and override specific methods.
`SecurityManager` is not initialized by default. You must initialize `SecurityManager`.	The Oracle JVM always initializes `SecurityManager` at startup.
Permissions are determined by the location or the URL, where the application or applet is loaded, or keycode, that is, 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 in a secure database table.
You can update the security policy file using a text editor or a tool, if you have the appropriate permissions.	You can update `PolicyTable` through `DBMS_JAVA` procedures. After initialization, only `JAVA_ADMIN` has permission to modify `PolicyTable`. `JAVA_ADMIN` must grant you the right to modify `PolicyTable` so that you can grant permissions to others.
Permissions are assigned to a protection domain, which classes can belong to.	All classes within the same schema are in the same protection domain.

Java2 Security Standard	Oracle Database Security Implementation
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, that is, `grant`.	Supports both positive and limitation permissions, that is, `grant` and `restrict`.

Setting Permissions

As with Java2 security, Oracle Database supports the security classes. Normally, you set the permissions for the code base either using a tool or by editing the security policy file. In Oracle Database, you set the permissions dynamically using `DBMS_JAVA` procedures, which modify a policy table in the database.

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` view 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 or a limitation 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 of the `Permission` object. You use this when defining the permission. When defining the target for a `Permission` object of type `PolicyTablePermission`, the name can become quite complicated. **See Also:** "Acquiring Administrative Permission to Update Policy Table" on page 10-8
Permission_action	The `action` attribute of the `Permission` object. 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` object, you can disable or re-enable it. This column shows whether the permission is enabled or disabled.
Key	Sequence number you use to identify this row. This number should be supplied when disabling, enabling, or deleting a permission.

There are two ways to set permissions:

- Fine-Grain Definition for Each Permission
- General Permission Definition Assigned to Roles

Note: For absolute certainty about the security settings, implement the fine-grain definition. The general definition is easy to implement, but you may not get the exact security settings you require.

Fine-Grain Definition for Each Permission

Using fine-grain definition, you can grant each permission individually to specific users or roles. If you do not grant a permission for access, then the schema will be denied access. 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`, then 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 as defined by the `Permission` class. Examine the appropriate `Permission` class for the relevant name.
Permission action	The type of action that you can specify. This can vary according to the permission type. For example, `FilePermission` can have the action, read or write.
Key	Number returned from grant or limit to use on enable, disable, or delete methods.

You can grant permissions using either SQL or Java. Each version 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.

After running the grant, if a row already exists for the exact permission, then no update occurs, but the key for that row is returned. If the row was disabled, then running the grant enables the existing row.

Note: If you are granting `FilePermission`, then 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. 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/-
```

You can grant permissions using the DBMS_JAVA package, as follows:

```
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)
```

You can grant permissions using Java, as follows:

```
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);
```

You can limit permissions using the DBMS_JAVA package, as follows:

```
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)
```

You can limit permissions using Java, as follows:

```
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 10–1 shows how to use the `grant_permission()` method to grant permissions. Example 10–2 shows how to limit permissions using the `restrict()` method.

Example 10–1 Granting Permissions

Assuming that you have appropriate permissions to modify the policy table, you can use the `grant_permission()` method, which is in the DBMS_JAVA package, to modify PolicyTable to allow user access to the indicated file. In this example, the user, Larry, has modification permission on PolicyTable. Within a SQL package, Larry can grant permission to Dave to read and write a file, as follows:

```
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 PolicyTable
commit;
```

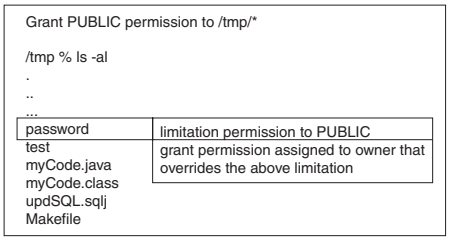
Example 10–2 Limiting Permissions

You can use the `restrict()` method to specify a limitation or exception to general rules. A general rule is a rule where, in most cases, the permission is true or granted.

However, there may be exceptions to this rule. For these exceptions, you specify a limitation permission.

If you have defined a general rule that no one can read or write an entire directory, then 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, then 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, then you need to create an explicit grant permission to override the limitation permission. In the previously mentioned scenario, if you want the file owner to still be able to modify the password file, then you can grant a more explicit permission to allow access to one user, which will override the limitation. Oracle JVM 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 permission to be effective, the limitation permission must also imply the grant.

The following code implements this example:

```
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;
```

The preceding code performs the following actions:

1. Grants everyone read and write permission to all files in `/tmp`.
2. Limits everyone from reading or writing only the `password` file in `/tmp`.
3. Grants only `Larry` explicit permission to read and write the `password` file.

Acquiring Administrative Permission to Update Policy Table

All permissions are rows in `PolicyTable`. Because it is a table in the database, you need appropriate permissions to modify it. Specifically, the `PolicyTablePermission` object is required to modify the table. After initializing Oracle JVM, only a single role, `JAVA_ADMIN`, is granted `PolicyTablePermission` to modify `PolicyTable`. The `JAVA_ADMIN` role is immediately assigned to the database administrator (DBA). Therefore, if you are assigned to the DBA group, then you will automatically take on all `JAVA_ADMIN` permissions.

If you need to add permissions as rows to this table, `JAVA_ADMIN` must grant your schema update rights using `PolicyTablePermission`. This permission defines that your schema can add rows to the table. Each `PolicyTablePermission` is for a specific type of permission. For example, to add a permission that controls access to a file, you must have `PolicyTablePermission` that lets you grant or limit a permission on `FilePermission`. Once this occurs, you have administrative permission for `FilePermission`.

An administrator can grant and limit `PolicyTablePermission` in the same manner as other permissions, but the syntax is complicated. For ease of use, you can use the `grant_policy_permission()` or `grantPolicyPermission()` method to grant administrative permissions.

You can grant policy table administrative permission using `DBMS_JAVA`, as follows:

```
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 )
```

You can grant policy table administrative permission using Java, as follows:

```
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 as 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 in 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 10–3 shows how you can modify `PolicyTable`.

Example 10–3 Granting PolicyTable Permission

This example shows `SYS`, which has the `JAVA_ADMIN` role assigned, giving `Larry` permission to update `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

You can create your own permission type by performing the following steps:

1. Create and load the user permission

   Create your own permission by extending the `java.security.Permission` class. Any user-defined 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 the 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 run `grant_policy_permission()`. Administrative permission enable the user to update the policy table for the user-defined permission.

For example, if `LARRY` creates a permission, `MyPermission`, then only he can call `grant_policy_permission()` for himself or another user. This method updates `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 PolicyTable
commit;
```

Once you have granted administrative rights, you can grant action permissions for the created permission. For example, the following SQL statements grant `LARRY` the permission to run anything within `MyPermission` and `DAVE` the permission to run 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 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 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 using SQL in the preceding steps, the following users can run methods within the example class:

   - `LARRY` can run any of the methods.
   - `DAVE` is given permission to run only the `act()` method.
   - Anyone can run the `print()` and `hello()` methods. The `print()` method does not check any permissions. As a result, anyone can run it. The `hello()` method runs `AccessController.doPrivileged()`, which means that the

method runs with the permissions assigned to LARRY. This is referred to as the 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 no longer applies. However, if you decide that you want the row action again, then 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, then the deletion will not occur.

To disable rows, you can use either of the following methods:

- `revoke_permission()`

  This method accepts parameters similar to the `grant()` and `restrict()` methods. It searches the entire policy table for all rows that match the parameters provided.

- `disable_permission()`

  This method disables only a single row within the policy table. To do this, it accepts the policy table key as parameter. 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, then you can use the grant and limit calls that do not return the permission number.

  – Look up `DBA_JAVA_POLICY` or `USER_JAVA_POLICY` for the appropriate permission key number.

You can disable permissions using `DBMS_JAVA`, as follows:

```
procedure revoke_permission (permission_schema varchar2, permission_type varchar2,
permission_name varchar2, permission_action varchar2)

procedure disable_permission (key number)
```

You can disable permissions using Java, as follows:

```
void revoke (String schema, String type, String name, String action);

void oracle.aurora.rdbms.security.PolicyTableManager.disable (long number);
```

You can enable permissions using DBMS_JAVA, as follows:

```
procedure enable_permission (key number)
```

You can enable permissions using Java, as follows:

```
void oracle.aurora.rdbms.security.PolicyTableManager.enable (long number);
```

You can delete permissions using DBMS_JAVA, as follows:

```
procedure delete_permission (key number)
```

You can delete permissions using Java, as follows:

```
void oracle.aurora.rdbms.security.PolicyTableManager.delete (long number);
```

Permission Types

Table 10–1 lists the installed permission types. Whenever you want to grant or limit a permission, you must provide the permission type. The permission types with which you control access are the following:

- Java2 permission types
- Oracle-specific permission types
- User-defined permission types that extend `java.security.Permission`

Table 10–1 Predefined Permissions

Type	Permissions
Java2	▪ `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 specific	▪ `oracle.aurora.rdbms.security.PolicyTablePermission` ▪ `oracle.aurora.security.JServerPermission`

Note: SYS is granted permission to load libraries that come with Oracle Database. However, Oracle JVM does not support other users loading libraries, because loading C libraries within the database is insecure. As a result, you are not allowed to grant `RuntimePermission` for `loadLibrary.*`.

The Oracle-specific permissions are:

- `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, you can control the access to few resources.

After the initialization of Oracle JVM, 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, you can use the grant_policy_permission() method, which is in the DBMS_JAVA package. Once you have updated the table, you can view either the DBA_JAVA_POLICY or USER_JAVA_POLICY view to see who has been granted permissions.

- oracle.aurora.security.JServerPermission

  This permission is used to grant and limit access to Oracle JVM resources. The JServerPermission extends BasicPermission. The following table lists the permission 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
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 Oracle JVM, several roles are populated with certain permission grants. The following tables show these roles and their initial Permissions:

- Table 10–1, " Predefined Permissions"
- Table 10–2, " SYS Initial Permissions"
- Table 10–3, " PUBLIC Default Permissions"
- Table 10–4, " JAVAUSERPRIV Permissions"
- Table 10–5, " JAVASYSPRIV Permissions"
- Table 10–6, " JAVADEBUGPRIV Permissions"

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 permissions listed in Table 10–1. In addition to the JAVA_ADMIN permissions, SYS is granted the permissions listed in Table 10–2. All users are initially granted the permissions defined in Table 10–3.

Table 10–2 SYS Initial Permissions

Permission Type	Permission Name	Action
`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

Table 10–3 PUBLIC Default Permissions

Permission Type	Permission Name	Action
`oracle.aurora.rdbms.security.PolicyTablePermission`	`java.lang.RuntimePermission.loadLibrary.*`	null
`java.util.PropertyPermission`	*	read
	`user.language`	write
`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 10–4 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 10–5 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
java.lang.RuntimePermission	createClassLoader	null
	getClassLoader	null
	setContextClassLoader	null
	setFactory	null
	setIO	null
	setFileDescriptor	null
	readFileDescriptor	null
	writeFileDescriptor	null

Table 10–6 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 Oracle8*i* Database release 8.1.5, the Oracle JVM security was controlled by granting the JAVASYSPRIV, JAVAUSERPRIV, or JAVADEBUGPRIV role to schemas. In Oracle Database 10*g*, these roles still exist as permission groups. You can set up and define your own collection of permissions. Once defined, you can grant any collection of permissions to any user or role. That user will then have the same permissions that exist within the role. In addition, if you need additional permissions, then you can add individual permissions to either your specified user or role. Permissions defined within the policy table have a cumulative effect.

> **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 JAVASYSPRIV role.

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 those existing 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);
```

See Also: "Fine-Grain Definition for Each Permission" on page 10-5.

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 10–6. To receive permission to call 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.

See Also: "Debugging Server Applications" on page 3-9

Permission for Loading Classes

To load classes, you must have the following permission:

```
JServerPermission("LoadClassInPackage." + class_name)
```

where, *`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 using the `CREATE JAVA SYSTEM` statement. The following error is thrown if you try to replace a system class:

```
ORA-01031 "Insufficient privileges"
```

The following describes what each user can do after database installation:

- `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.*`, and `oracle.jdbc.*`. If the user wants to load such classes into another schema, then 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

The following database authentication mechanisms are available:

- Password authentication
- Strong authentication
- Proxy authentication
- Single sign-on

11

Schema Objects and Oracle JVM Utilities

This chapter describes the schema objects that you use in the Oracle Database Java environment and the Oracle JVM utilities. You run these utilities from a UNIX shell or from the Microsoft Windows DOS prompt.

> **Note:** All names supplied to these tools are case-sensitive. As a result, the schema, user name, and password should not be changed to uppercase.

This chapter contains the following sections:

Schema Objects Overview

Unlike a conventional Java virtual machine (JVM), which compiles and loads Java files, the Oracle JVM compiles and loads schema objects. The following kinds of Java schema objects are loaded:

- 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 ensure that a class file can be run by the Oracle JVM, you must 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 Oracle JVM, you must use `loadjava` to create and load a Java resource schema object from the resource file.

The `dropjava` tool 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 schema objects using SQL data definition language (DDL) commands will not update auxiliary data maintained by `loadjava` and `dropjava`.

What and When to Load

You must load resource files using `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 Oracle Database 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 the corresponding `.java` file, but not both.

The `loadjava` tool accepts Java Archive (JAR) files that contain either source and resource files or class and resource files. When you pass a JAR or ZIP file to `loadjava`, it 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. As a result, there is little performance penalty for loading JAR files. Loading JAR files is a simple, fool-proof way to use `loadjava`.

It is illegal for two schema objects in the same schema to define the same class. For example, assume that `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`. Instead, do either of the following:

- Drop `a.java`, load `b.java`, and then load the new `a.java`, which does not define `x`.
- Load the new `a.java`, which does not define `x`, and then load `b.java`.

Resolution

All Java classes contain references to other classes. A conventional JVM searches for classes in the directories, ZIP files, and JAR files named in the `CLASSPATH`. In contrast, the Oracle JVM searches schemas for class schema objects. Each class in the database has a resolver specification, which is the Oracle Database counterpart to `CLASSPATH`. For example, the resolver specification of a class, `alpha`, lists the schemas to search for classes that `alpha` uses. Notice that resolver specifications are per-class, whereas in a classic JVM, `CLASSPATH` is global to all classes.

In addition to a resolver specification, 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 call when the class uses the reference
- A code indicating whether the reference is unsatisfied, that is, whether the referent schema object is known

An Oracle Database facility known as **resolver** maintains reference lists. For each interclass reference in a class, the resolver searches the schemas specified by the resolver specification of the class for a valid class schema object that satisfies the reference. If all references are resolved, then the resolver marks the class valid. A class that has never been resolved, or has been resolved unsuccessfully, is marked invalid. A class that 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 these classes, 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 resolve classes that are marked valid.

A 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 `ClassNotFound` 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:

- Load-and-resolve

  The `-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 while the command is being carried out.

- Load-then-resolve

  This mode resolves each class at run time. The `-resolve` option is not specified.

> **Note:** As with a Java compiler, `loadjava` resolves references to classes but not to resources. Ensure that you correctly load the resource files that your classes need.

If you can, defer resolution until all classes have been loaded. This avoids a 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 call `loadjava` for collections of files, 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 or a hash, of the content of the file. Comparing digests computed for the same file at different times is a fast way to detect a change in the content of the file. This is much faster than comparing every byte in the file. For each file it processes, `loadjava` computes a digest of the content of the file and then looks up the file name in the digest table. If the digest table contains an entry for the file name that has an identical digest, then `loadjava` does not load the file, because a corresponding schema object exists and is up to date. If you call `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, then use the `loadjava -force` option to bypass the digest table lookup or delete all rows from the table `JAVA$CLASS$MD5$TABLE`.

Compilation

Loading a source file creates or updates a Java source schema object and invalidates the class schema objects previously derived from the source. If the class schema objects do not exist, then `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, for example, class `A`, is automatically triggered by any of the following conditions:

- The resolver, while working on class `B`, finds that class `B` refers to class `A`, but class `A` is invalid.
- The compiler, while compiling the source of class `B`, finds that class `B` refers to class `A`, but class `A` is invalid.
- The class loader, while trying to load class `A` for running it, finds that class `A` 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. The `loadjava` tool 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, then you can specify compiler options on the command line. You can additionally specify persistent compiler options in a per-schema database table, `JAVA$OPTIONS`. You can use the `JAVA$OPTIONS` table for default compiler options, which you can override selectively using a `loadjava` command-line option.

Note: A command-line option overrides and clears the matching entry in the `JAVA$OPTIONS` table.

A `JAVA$OPTIONS` row contains the names of source schema objects to which an option setting applies. You can use multiple rows to set the options differently for different source schema objects. The compiler looks up options in `JAVA$OPTIONS` when it has been called by the class loader or when called from the command line without specifying any 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 applies only to Java sources that contain SQLJ constructs.
- `debug = true`

  This option is equivalent to `javac -g`.

  See Also: "Compiler Options Specified in a Database Table" on page 2-7

The loadjava Tool

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. `loadjava` can also create schema objects from SQLJ 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`*.

You can run the `loadjava` tool either from the command line or by using the `loadjava` method contained in the `DBMS_JAVA` class. To run the tool from within your Java application, do the following:

```
call dbms_java.loadjava('... options...');
```

The options are the same as those that can be specified on the command line with the `loadjava` tool. Separate each option with a space. Do not separate the options with a comma. The only exception for this is the `-resolver` option, which contains spaces. For `-resolver`, specify all other options in the first input parameter and the `-resolver` options in the second parameter, as follows:

```
call dbms_java.loadjava('..options...', 'resolver_options');
```

Do not specify the `-thin`, `-oci`, `-user`, and `-password` options, because they relate to the database connection for the `loadjava` command-line tool. 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` directory under `$ORACLE_HOME`.

Just before the `loadjava` tool exits, it checks whether the processing 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. These errors are printed with the following syntax:

```
exiting: error_reason
```

This section covers the following:

Syntax

The syntax of the `loadjava` command is as follows:

```
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 11–1 summarizes the `loadjava` arguments. If you run `loadjava` multiple times specifying the same files and different options, then the options specified in the most recent invocation hold. However, there are two exceptions to this, as follows:

- If `loadjava` does not load a file because it matches a digest table entry, then most options on the command line have no effect on the schema object. The exceptions are `-grant` and `-resolve`, which always take effect. You must use the `-force` option to direct `loadjava` to skip the digest table lookup.
- 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 11–1 loadjava Argument Summary

Argument	Description
filenames	You can specify any number and combination of `.java`, `.class`, `.sqlj`, `.ser`, `.jar`, `.zip`, and resource file name arguments.
`-action`	Perform all actions. This is the default behavior. It is 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. 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, then you should use this option instead of `-resolve`.
`-casesensitivepub`	Publishing will create case-sensitive names. Unless the names are already all uppercase, 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, then `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 only `SCOTT`, then the proper options are: `-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 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. Therefore, this option will only have an effect for resources.
`-encoding`	Identifies the source file encoding for the compiler, overriding the matching value, if any, in `JAVA$OPTIONS`. Values are the same as for the `javac -encoding` option. If you do not specify an encoding on the command line or in `JAVA$OPTIONS`, then the encoding is assumed to be the value returned by: `System.getProperty("file.encoding");` This 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.

Table 11–1 (Cont.) loadjava Argument Summary

Argument	Description
`-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 the referred 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` option, even when referring classes are created in some other schema. The created classes will be flagged so that tools can recognize them. In particular, this is needed, so that the verifier can recognize the generated classes.
`-genmissingjar` *`jar_file`*	This option performs the same actions as `-genmissing`. In addition, it creates a JAR file, *`jar_file`*, that contains the definitions of any generated classes.
`-grant`	Grants the `EXECUTE` privilege on loaded classes to the listed users. Any number and combination of user names can be specified, separated by commas, but not spaces. Granting the `EXECUTE` privilege on an object in another schema requires that the original `CREATE PROCEDURE` privilege was granted with the `WITH GRANT` options. **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. ■ The schema name should be used in uppercase.
`-help`	Prints usage message on how to use `loadjava` 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. The `-action` option in the option file will override the `-noaction` option specified on the command line.
`-nocasesensitivepub`	All lowercase characters are converted to uppercase. Transitions from lowercase to uppercase 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 the `-cleargrants` option.
`-nodefiner`	Define invoker rights on the loaded classes. This is the default behavior. This option is used to override the `-definer` option.
`-nogrant`	Do not grant `EXECUTE` privileges to the loaded classes. This is the default behavior. This option is used to override the `-grant` option.

Table 11–1 (Cont.) loadjava Argument Summary

Argument	Description
`-norecursivejars`	Treat JAR files contained in other JAR files as resources. This is the default behavior. This option is used to override the `-recursivejars` option.
`-noschema`	Place the loaded classes, sources, and resources into the schema associated with the user specified in the `-user` option. This is the default behavior. It is used to override the `-schema` option.
`-nosynonym`	Do not create a public synonym for the classes. This is the default behavior. This overrides the `-synonym` option.
`-noserverside`	Changes the behavior of `dbms_java.loadjava` to use a Java Database Connectivity (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 `loadjava` to use a JDBC driver, then 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. `oracle.aurora.security.JServerPermission(Verifier)` `must be granted` to use 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 JDBC Oracle Call Interface (OCI) driver. `-oci` and `-thin` are mutually exclusive. If neither is specified, then `-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.
`-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, `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 are the same as the corresponding command-line options and take an 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 and for any 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`, then it is unspecified which will prevail. If two rows have the same pattern and option columns, then 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.
`-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 arguments up to and including *`number`*. The default value is `3`. This option applies to `main`, as well as any method that has exactly one argument of type `java.lang.String`.

Table 11–1 (Cont.) loadjava Argument Summary

Argument	Description
`-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 JAR files as if they were top-level JAR files. 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. If you do not specify `-resolve`, `loadjava` loads files, but does not compile or resolve them.
`-resolver`	Specifies an explicit resolver specification, which is bound to the newly loaded classes. If `-resolver` is not specified, then the default resolver specification, which includes current user's schema and `PUBLIC`, is used.
`-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, then 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 class could not be resolved. Therefore, these errors are a secondary effect of whatever caused a referred class to be unresolved.
`-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, then the classes compiled from the source files are treated as if they had been loaded with `-synonym`.
`-tableschema` *`schema`*	Creates the `loadjava` internal tables within the specified schema, rather than in the Java file destination schema.
`-thin`	Directs `loadjava` to communicate with the database using the JDBC Thin driver. 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.
`-unresolvedok`	When combined with `-resolve`, will ignore unresolved errors.
`-user`	Specifies a user name, password, and database connect string. The files will be loaded into this database instance.
`-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 some of the `loadjava` arguments whose behavior is more complex than the summary descriptions contained in Table 11–1.

File Names

You can specify as many `.class`, `.java`, `.sqlj`, `.jar`, `.zip`, and resource files as you want and 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 another 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. If you have a JAR or ZIP that works with the Java Development Kit (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 different from file names, and `loadjava` names different types of schema objects differently. Because class files are self-identifying, the mapping of class file names to schema object names done by `loadjava` 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. Because classes use resource schema objects and the correct specification of resources is not always intuitive, it is important that you specify resource file names correctly on the command line.

The perfect way to load individual resource files correctly is to run `loadjava` from the top of the package tree and specify resource file names relative to that directory.

Note: The top of the package tree is the directory you would name in a `CLASSPATH`.

If you do not want to follow this rule, then 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. For example, if 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, `alpha/beta/x.properties` and `ROOT/home/scott/javastuff/alpha/beta/x.properties`. The name of the resource schema object is generated from the file name as entered.

Classes can refer to resource files relatively or absolutely. To ensure that `loadjava` and the class loader use the same name for a schema object, enter the name on the command line, which 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
```

To simplify the process further, place 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 preceding `loadjava` commands imply that you can use any path name to load the contents of a JAR file. Even if you run the redundant commands, `loadjava` would realize from the digest table that it need not load the files twice. This implies 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}
```

This option is identical to the definer rights in stored procedures and is conceptually similar to the UNIX `setuid` facility. However, you can apply the `-definer` option to individual classes, in contrast to `setuid`, which applies to a complete program. Moreover, different definers may have different privileges. Because an application can consist of many classes, you must apply `-definer` with care to achieve the desired results. That is, classes run with the privileges they need, but no more.

See Also: "Controlling the Current User" on page 2-14

noverify

```
[-noverify]
```

This option causes the classes to be loaded without bytecode verification. `oracle.aurora.security.JServerPermission(Verifier)` must be granted to run this option. Also, this option must be used in conjunction with `-resolve`.

The verifier ensures that incorrectly formed Java binaries cannot be loaded for running on the server. If you know that the JAR or classes you are loading are valid, then the 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. Therefore, the interpreted performance of your application may be adversely affected by using this option.

optionfile

```
[-optionfile <file>]
```

This option enables you to specify a file with `loadjava` options. This file is read and processed by `loadjava` before any other `loadjava` options are processed. The file can 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 is checked against the patterns. Patterns can end in a wildcard (`*`) to indicate an arbitrary sequence of characters, or 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`.

You can use Java comments in this file. A line comment begins with a `#`. Empty lines are ignored. The quote character is a double quote (`"`). That is, options containing spaces should be surrounded by double quotes. Certain options, such as `-user` and `-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.

To help package applications, `loadjava` looks for the `META-INF/loadjava-options` entry in each JAR it processes. If it finds such an entry, then 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.

If `loadjava` has partially processed entities before it processes `META-INF/loadjava-options`, then `loadjava` will attempt to patch up the schema object to conform to the applicable options. For example, `loadjava` alters classes that were created with invoker rights when they should have been created with definer 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, then 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:

- It must be a member of a `public` class.
- It must be declared `public` and `static`.
- The method signature should satisfy the following rules so that it can be mapped:
  - Java arithmetic types for arguments and return values 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 `-pubmain` option description.
  - If the return type is `void`, then a procedure is created.
  - If the return type is an arithmetic, `char`, or `java.lang.String` type, then a function is created.

Methods that take arguments or return types that are not covered by the preceding rules are not eligible. No provision is made for `OUT` and `IN OUT` SQL arguments, `OBJECT` types, and many other SQL features.

resolve

```
{-resolve | -r}
```

Use `-resolve` to force `loadjava` to compile and resolve a class that has previously been loaded. It is not necessary to specify `-force`, because resolution is performed after, and independent of, loading.

resolver

```
{-resolver | -R} resolver_specification
```

This option associates an explicit resolver specification with the class schema objects that `loadjava` creates or replaces.

A resolver specification consists of one or more items, each of which consists of a name specification and a schema specification expressed in the following syntax:

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

A name specification is similar to a name in an `import` statement. It can be a fully qualified Java class name or a package name whose final element is the wildcard character asterisk (`*`) or simply an asterisk (`*`). However, the elements of a name specification must be separated by slashes (`/`), not periods (`.`). For example, the name specification `a/b/*` matches all classes whose names begin with `a.b`. The special name `*` matches all class names.

A schema specification can be a schema name or the wildcard character dash (`-`). The wildcard does not identify a schema, but directs the resolve operation not to mark a class invalid, because a reference to a matching name cannot be resolved. Use dash (`-`) 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.

When looking for a schema object whose name matches the name specification, the resolution operation looks in the schema named by the partner schema specification.

The resolution operation searches schemas in the order in which the resolver specification lists them. For example,

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

This implies that 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.

Consider the following example:

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

This implies that search for any reference first in `SCOTT` and then in `PUBLIC`. If the reference is to a class in the package `my.gui` and is not found, then mark the referring class valid and do not display an error. If the reference is not to a class in `my.gui` and is not found, then mark the referring class invalid and produce an error message.

user

```
{-user | -u} user/password[@database_url]
```

By default, `loadjava` loads into the logged in schema specified by the `-user` option. You use the `-schema` option to specify a different schema to load into. This does not

require you to log in to that schema, but does require that you have sufficient permissions to alter the schema.

The permissible forms of `@database_url` depend on whether you specify `-oci` or `-thin`, as described:

- `-oci:@database_url` is optional. If you do not specify, then `loadjava` uses the user's default database. If specified, `database_url` can be a TNS name or an Oracle Net Services name-value list.
- `-thin:@database_url` is required. The format is `host:lport:SID`.

  where:

  - `host` is the name of the computer 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`.

The following 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, and then resolve them:

```
loadjava -u joe/shmoe -resolve -schema TEST ServerObjects.jar
```

- Connect with the JDBC 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 run `alpha.class`:

```
loadjava -thin -schema test -u SCOTT/TIGER@localhost:5521:orcl \
  -grant BETTY,BOB alpha.class
```

The dropjava Tool

The `dropjava` tool is the converse of `loadjava`. It transforms command-line file names and JAR or ZIP file contents to schema object names, drops the schema objects, and deletes their corresponding digest table rows. You can enter `.java`, `.class`, `.sqlj`, `.ser`, `.zip`, `.jar`, and resource file names on the command line and in any order.

Alternatively, you can specify a schema object name directly to `dropjava`. A command-line argument that does not end in `.jar`, `.zip`, `.class`, `.java`, or `.sqlj` is presumed to be a schema object name. If you specify a schema object name that applies to multiple schema objects, then 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 load a `.sqlj` source file and translate it in the server, then you must run `dropjava` on the same source file. If you translate on a client and load classes and resources directly, then run `dropjava` on the same classes and resources.

You can run `dropjava` either from the command line or by using the `dropjava` method in the `DBMS_JAVA` class. To run `dropjava` from within your Java application, use the following command:

```
call dbms_java.dropjava('... options...');
```

The options are the same as specified on the command line. Separate each option with a space. Do not separate the options using commas. The only exception to this is the `-resolver` option. The connection is always made to the current session. Therefore, 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 `-thin`, `-oci`, `-user`, and `-password` options, because they relate to the database connection for `loadjava`. The output is directed to `stderr`. Set `serveroutput` on and call `dbms_java.set_output`, as appropriate.

This section covers the following topics:

- Syntax
- Argument Summary
- Argument Details
- Dropping Resources

Syntax

The syntax of the `dropjava` command is:

```
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 11–2 summarizes the `dropjava` arguments.

Table 11–2 dropjava Argument Summary

Argument	Description
`-user`	Specifies a user name, password, and optional database connect string. The files will be dropped from this database instance.
`filenames`	Specifies any number and combination of `.java`, `.class`, `.sqlj`, `.ser`, `.jar`, `.zip`, and resource file names.

Table 11–2 (Cont.) dropjava Argument Summary

Argument	Description
`-genmissingjar` *`JARfile`*	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 schemas. 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 `loadjava` 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`*	Has the same usage as for `loadjava`.
`-optiontable` *`table_name`*	Has the same usage as for `loadjava`.
`-schema` *`schema`*	Designates the schema from which schema objects are dropped. If not specified, then 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 JDBC Thin driver. 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

This section describes few of the `dropjava` argument, which are complex

File Names

`dropjava` interprets most file names as `loadjava` does:

- `.class` files

  Finds the class name in the file and drops the corresponding schema object.

- `.java` and `.sqlj` files

  Finds the first class name in the file and drops the corresponding schema object.

- `.jar` and `.zip` files

  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.

If dropjava encounters a file name that does not match a schema object, then 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:@*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 an Oracle Net Services name-value list.
- -thin:@*database* is required. The format is *host*:*lport*:*SID*.

  where:

  - *host* is the name of the computer 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.

The following are examples of dropjava.

- Drop all schema objects in the TEST schema in the default database that were loaded from ServerObjects.jar:

  ```
  dropjava -u SCOTT/TIGER -schema TEST ServerObjects.jar
  ```

- Connect with the JDBC 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 them 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 that specified 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, then there will be no problem. However, if 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.

The ojvmjava Tool

The ojvmjava tool is an interactive interface to the session namespace of a database instance. 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.

This section covers the following topics:

Syntax

The syntax of the `ojvmjava` command is:

```
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 11–3 summarizes the `ojvmjava` arguments.

Table 11–3 ojvmjava Argument Summary

Argument	Description
`-user | -u`	Specifies user name for connecting to the database. This name is not case-sensitive. The name will always be converted to uppercase. If you provide the database information, then the default syntax used is OCI. You can also specify the default database.
`-password | -p`	Specifies the password for connecting to the database. This is not case-sensitive and will always be converted to uppercase.
`@`*`filename`*	Specifies a script file that contains the `ojvmjava` commands to be run.
`-batch`	Disables all messages printed to the screen. No help messages or prompts will be printed. Only responses to commands entered are printed.
`-command`	Runs the desired command. If you do not want to run `ojvmjava` in interpretive mode, but only want to run a single command, then run it with this option followed by a string that contains the command and the arguments. Once the command runs, `ojvmjava` exits.
`-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.
`-o schema` *`schema`*	Use this schema for class lookup.

Table 11–3 (Cont.) ojvmjava Argument Summary

Argument	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

Open a shell on the session namespace of the database `orcl` on listener port `2481` on the host `dbserver`, as follows.

```
ojvmjava -thin -user SCOTT/TIGER@dbserver:2481:orcl
```

Functionality

The `ojvmjava` commands span several different types of functionality, which are grouped as follows:

- ojvmjava Options
- Shell Commands

ojvmjava Options

This section describes the options for the `ojvmjava` command-line tool

The ojvmjava Tool Output Redirection

You can direct any output generated by the `ojvmjava` tool to a file by appending `&>`*`filename`* at the end of the command options. The following command directs all output to the `listDir` file:

```
ls -lR &>/tmp/listDir
```

Scripting ojvmjava Commands in the @filename Option

This `@`*`filename`* 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 the file and runs all commands on the designated server. In addition, because the script file is run on the server, any interaction with the operating system in the script file, such as redirecting output to a file or running another script, will occur on the server. If you direct `ojvmjava` to run another script file, then this file must exist in `$ORACLE_HOME` on the server.

Enter the `ojvmjava` command followed by any options and any expected input arguments.

The script file contains the `ojvmjava` command followed by options and input parameters. The input parameters can be passed to `ojvmjava` on the command line. `ojvmjava` processes all known options and passes on any other options and arguments to the script file.

To access arguments within the commands in the script file, use `&1...&`*`n`* to denote the arguments. If all input parameters are passed to a single command, then you can type `&*` 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, `ojvmjava` passes `contexts` as argument to the `cd` command.

To run this file, do the following:

```
ojvmjava -user SCOTT -password TIGER -thin -database dbserver:2481:orcl \
  @execShell alpha beta
```

`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. The actual command is run as follows:

```
java testhello alpha beta
```

You can add any comments in your script file using hash (`#`). Comments are ignored by `ojvmjava`. For example:

```
#this whole line is ignored by ojvmjava
```

Running sess_sh Within Applications

You can run `sess_sh` commands from within a Java or PL/SQL application using the following commands:

Application Type	Command and Description

PL/SQL applications	`dbms_namespace.shell(in command VARCHAR2)`
	
	Your application can run individual commands on a unique session instance. The state of the shell is preserved between different calls of `sess_sh` within the same session. The following examples run the `cd` command of the `sess_sh` tool within a PL/SQL application:
	`dbms_namespace.shell('cd /webdomains');`
	
	`dbms_namespace.shell('ls &> /tmp/test');`
	
	To reset the state of the shell instance, run the `exit` command, as follows:
	`dbms_namespace.shell('exit');`

Application Type	Command and Description

Java applications	Instantiate `oracle.aurora.namespace.shell.Shell` within a Java server object. After creation, you must initialize the `Shell` object using its initialize method. Once initialized, you can run `sess_sh` commands, as follows:
	`String commands="cd /webdomains\nls -l";`
	`StringReader commandReader = new StringReader(commands);`
	`Shell sh = new oracle.aurora.namespace.shell.Shell();`
	
	`try`
	`{`
	`  sh.initialize();`
	`  sh.invoke(new BufferedReader(commandReader), false);`
	`}`
	`catch (ToolsException te)`
	`{`
	`  //Error executing the commands`
	`}`

Shell Commands

This section describes the commands used for manipulating and viewing contexts and objects in the namespace.

The following shell commands function similar to their UNIX counterparts:

- echo
- exit
- help
- java
- version
- whoami

Each of these shell commands have some options in common, which are summarized in Table 11–4:

Table 11–4 ojvmjava Command Common Options

Option	Description
`-describe | -d`	Summarizes the operation of the tool.
`-help | -h`	Summarizes the syntax of the tool.
`-version`	Shows the version.

echo

This commnad prints to `stdout` exactly what is indicated. This is used mostly in script files.

The syntax is as follows:

```
echo [echo_string] [args]
```

`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 command prints out a notification:

```
echo "Adding an owner to the schema" &1
```

If the input argument is SCOTT, then the output would be:

```
Adding an owner to the schema SCOTT
```

exit

This command terminates ojvmjava. The syntax is as follows:

```
exit
```

For example, to leave a shell, use the following command:

```
$ exit
%
```

help

This command summarizes the syntax of the shell commands. You can also use the help command to summarize the options for a particular command. The syntax is as follows:

```
help [command]
```

java

This command is analogous to the JDK java command. It calls the static main() method of the class. The class must be loaded with loadjava. The command provides a convenient way to test Java code that runs in the database. In particular, the command catches exceptions and redirects the standard output and standard error of the class to the shell, which displays them as with any other command output. The destination of standard out and standard error for Java classes that run in the database is one or more database server process trace files, which are inconvenient and may require DBA privileges to read.

The syntax of this command is:

```
java [-schema schema] class [arg1 ... argn]
```

Table 11–5 summarizes the arguments of this command.

Table 11–5 java Argument Summary

Argument	Description
class	Names the Java class schema object that is to be run.
-schema	Names the schema containing the class to be run. The default is the invoker's schema. The schema name is case-sensitive.
arg1 ... argn	Arguments to the static main() method of the class.

Consider the following Java file, World.java:

```
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]);
  }
}
```

You can compile, load, publish, and run the class, as follows:

```
% 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

This command shows the version of the `ojvmjava` tool. You can also show the version of a specified command. The syntax of this command is:

```
version [options] [command]
```

For example, you can display the version of the shell, as follows:

```
$ version
1.0
```

whoami

This command prints the user name of the user who logged in to the current session. The syntax of the command is:

```
whoami
```

12

Database Web Services

This chapter provides an overview of database Web services and discusses how to call existing Web services. This chapter contains the following sections:

Overview of Database Web Services

Web services enable application-to-application interaction over the Web, regardless of platform, language, or data formats. The key ingredients, including Extensible Markup Language (XML), Simple Object Access Protocol (SOAP), Web Services Description Language (WSDL), and Universal Description, Discovery, and Integration (UDDI), have been adopted across the entire software industry. Web services 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.

The Database Web services technology is a database approach to Web services. It works in the following two directions:

- Accessing database resources as a Web service
- Consuming external Web services from the database

Oracle Database can access Web services through PL/SQL packages and Java classes deployed within the database. Turning Oracle Database into a Web service provider leverages investment in Java stored procedures, PL/SQL packages, predefined SQL queries, and data manipulation language (DML). Conversely, consuming external Web services from the database, together with integration with the SQL engine, enables Enterprise Information Integration.

Using Oracle Database as Service Provider for Web Services

Web Services use industry-standard mechanisms to provide easy access to remote content and applications, regardless of the platform and location of the provider and implementation and data format. Client applications can query and retrieve data from Oracle Database and call 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 disconnected environments.

You can call into the database from a Web service, using the database as a service provider. This enables you to leverage existing or new SQL, PL/SQL, Java stored procedures, or Java classes within Oracle Database. You can access and manipulate database tables from a Web service client.

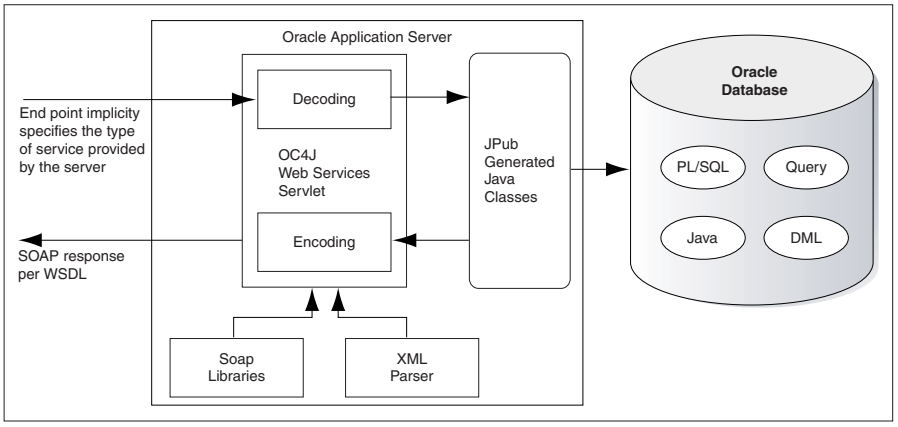
This section covers the following topics:

- How to Use JPublisher for Web Services Call-Ins
- Features of Oracle Database as a Web Service Provider
- JPublisher Support for Web Services Call-Ins to Oracle Database

How to Use JPublisher for Web Services Call-Ins

You can use JPublisher to generate Java wrappers that correspond to database operations and deploy the wrappers as Web services in Oracle Application Server. Figure 12–1 illustrates how you use JPublisher to publish PL/SQL packages, SQL objects, collections, and packages as Java classes. Once published, these classes can be accessed by any Web service through an Oracle Application Server Containers for J2EE (OC4J) Web services servlet.

Figure 12–1 Web Services Call-In to the Database



See Also: *Oracle Database JPublisher User's Guide*

Features of Oracle Database as a Web Service Provider

Using Oracle Database as a Web service provider offers the following features:

- Enhances PL/SQL Web services

  Improves PL/SQL Web services by extending the Web services support for additional PL/SQL types, including `CLOB`, `BLOB`, `XMLTYPE`, `REF CURSOR`, and PL/SQL records and tables. This enables you to use most of your existing PL/SQL packages as Web services.

- Exposes Java in the database as Web services

  Exposes existing Java classes deployed in Oracle 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.

- Provides SQL query Web services

Leverages warehousing or business intelligence queries, data monitoring queries, and any predefined SQL statements as Web services.

- Enables DML Web services

  Offers secure, persistent, transactional, and scalable logging, auditing, and tracking operations implemented through SQL DML, as Web services. DML Web services are implemented as atomic or group, or batch, `INSERT`, `UPDATE`, and `DELETE` operations.

JPublisher Support for Web Services Call-Ins to Oracle Database

The following JPublisher features support Web services call-ins to the code running in Oracle Database:

- 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 specifications
- Support for publishing SQL queries or DML statements
- Support for unique method names
- Support for Oracle Streams AQ

See Also: *Oracle Database JPublisher User's Guide*

Using Oracle Database as Service Consumer for Web Services

You can extend the storage, indexing, and searching capabilities of a relational database 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 Oracle Database as a service consumer would be to call external Web services from a predefined database job to retrieve inventory information from multiple suppliers, and then update your local inventory database. Another example is that of a Web crawler, where a database job can be scheduled to collate product and price information from a number of sources.

This section covers the following topics:

How to Use Oracle Database for Web Services Call-Outs

The Web services client code is written in SQL, PL/SQL, or Java to run inside Oracle Database, which then calls the external Web service. Figure 12–2 illustrates how you

can call a Web service from a Java client within the database, using one of the following methods:

- SQL and PL/SQL call specifications

  Start a Web service through a user-defined function call, which is generated through JPublisher, either directly within a SQL statement or view or through a variable.

- Pure Java `static` proxy class

  Use JPublisher to generate a client proxy class, which uses Java API for XML-based remote procedure call (JAX-RPC). This method simplifies the Web service invocation because 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 the work required to construct the SOAP request, including marshalling and unmarshalling parameters.

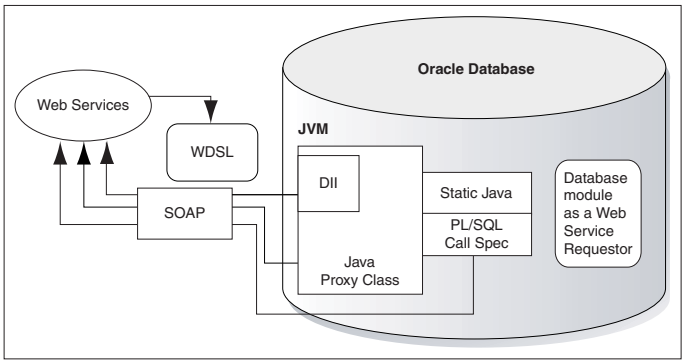
- Pure Java using dynamic invocation interface (DII) over JAX-RPC

  Dynamic invocation provides the ability to construct the SOAP request and access the service without the client proxy.

For Web services call-outs using PL/SQL, use the `UTL_DBWS` PL/SQL package. This package essentially uses the same application programming interfaces (APIs) as the DII classes.

You can use a Web services data source to process the results from any Web service request.

Figure 12–2 Calling Web Services From Within the Database



Web Service Data Sources (Virtual Table Support)

To access data that is returned from single or multiple Web service invocations, create a virtual table using a Web service data source. This table lets you query a set of returned rows as though it were a table.

The client calls a Web service and the results are stored in a virtual table in the database. You can pass result sets from function to function. This enables 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` statement creates rows whose columns are used as arguments for calling the `CALL_WS` Web service call-out.

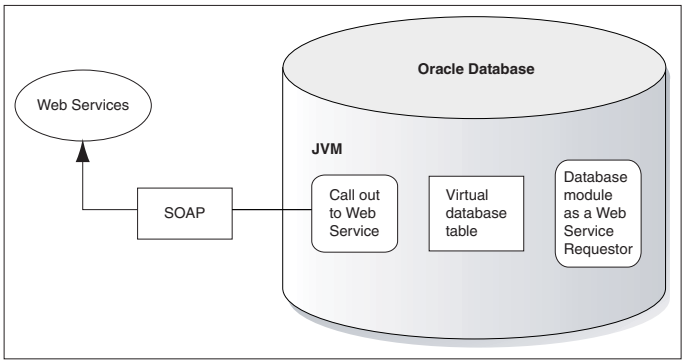
```
SELECT column1, cloumn2, ...
```

```
FROM TABLE(WS_TABFUN(CURSOR(SELECT s FROM table_name)))
WHERE ...
```

The table expression in the preceding example can be used in other SQL queries, for constructing views, and so on.

Figure 12–3 illustrates the support for virtual table.

Figure 12–3 Storing Results from Request in a Virtual Table



Features of Oracle Database as a Web Service Consumer

Using Oracle Database as a Web service consumer provides the following features:

- Consuming Web services form Java

  Provides an easy-to-use interface for Web services call-outs, thereby insulating developers from low-level SOAP programming. Java classes running in the database can directly call external Web services by using the previously loaded Java proxy class or through dynamic invocation.

- Consuming Web services from SQL and PL/SQL

  Enables 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 procedure hides all Java and SOAP programming details from the SQL client.

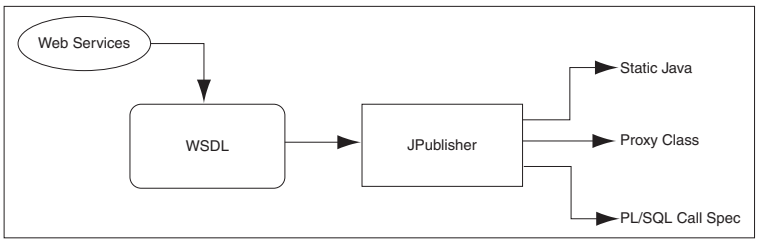
- Using Web services data source

  Enables application and data integration by turning external Web service into a SQL data source, making the external Web service appear as regular SQL table. This table function represents the output of calling external Web services and can be used in a SQL query.

JPublisher Generation Overview

JPublisher can receive the WSDL file from a Web service and create the static Java, proxy class, or PL/SQL call specification, as shown in Figure 12–4.

Figure 12–4 Creating Web Services Call-Out 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 generates additional wrapper classes to expose instance methods as `static` methods and generates PL/SQL wrappers. It performs 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 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 specifications or through the JAX-RPC client proxy classes. The PL/SQL wrappers use the `static` methods. A client would *not* normally access any Web service using 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 accepts 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 may want to use the special `-proxyopts` settings.

- `-endpoint=`*`Web_services_endpoint`*

Use this option in conjunction with the `-proxywsdl` option to specify the Web services endpoint.

- `-sysuser=`*`superuser_name/superuser_password`*

  Use this option to specify the name and password for the superuser account used to grant permissions for the client proxy classes to access Web services using HTTP.

Adjusting the Mapping of SQL Types

Although Oracle Application Server does not currently support `LOB` types, `XMLTYPE`, `REF CURSORS`, and `OUT` and `IN OUT` arguments, you can use an alternative approach to expose PL/SQL methods and SQL types as Web services.

You can change the default action of JPublisher to generate code that uses a user-provided subclass. For example, if you have a PL/SQL method that returns a `REF CURSOR`, then JPublisher automatically maps the return type to `java.sql.ResultSet`. However, this `ResultSet` type cannot be published as a Web service. To solve this, create a new method that can return the result set in a Web service-supported format, as follows:

```
public String [] readRefCursorArray(String arg1, Integer arg2)
{
  java.sql.ResultSet rs = getRefCursor(arg1,arg2);
  ...
  //create a String[] from rs and return it
  ...
}
```

After creating a method, create an interface that contains the exact methods to publish. You can use JPublisher to easily accomplish this mapping by using the following command:

```
jpub -sql=MYAPP:MyAppBase:MyApp#MyAppInterf...
```

In the preceding command:

- `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 Java Archive (JAR) file and use the Web Services Assembler to create a deployable Web service Enterprise Archive (EAR) file.

See Also: *Oracle Database JPublisher User's Guide*

A

DBMS_JAVA Package

This chapter provides a description of the DBMS_JAVA package. The functions and procedures in this package provide an entry point for accessing RDBMS functionality from Java.

longname

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

The function returns the fully qualified name of the specified Java schema object. Because Java classes and methods can have names exceeding the maximum SQL identifier length, Oracle JVM uses abbreviated names internally for SQL access. This function returns the original Java name for any 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';
```

shortname

```
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.

get_compiler_option

```
FUNCTION get_compiler_option(name VARCHAR2, optionName VARCHAR2) RETURN VARCHAR2
```

This function returns the value of the option specified through the optionName parameter. It is one of the functions used to control the options of the Java and SQLJ compiler supplied with Oracle Database.

> **See Also:** "Compiler Options Specified in a Database Table" on page 2-7

set_compiler_option

```
PROCEDURE set_compiler_option(name VARCHAR2, optionName VARCHAR2, value VARCHAR2)
```

This procedure is used to set the options of the Java and SQLJ compiler supplied with Oracle Database.

> **See Also:** "Compiler Options Specified in a Database Table" on page 2-7

reset_compiler_option

```
PROCEDURE reset_compiler_option(name VARCHAR2, optionName VARCHAR2)
```

This procedure is used to reset the specified compiler option to the default value.

> **See Also:** "Compiler Options Specified in a Database Table" on page 2-7

resolver

```
FUNCTION resolver (name VARCHAR2, owner VARCHAR2, type VARCHAR2) RETURN VARCHAR2
```

This function returns the resolver specification for the object specified in `name` and in the schema specified in `owner`, where the object is of the type specified in `type`. The caller must have `EXECUTE` privilege and have access to the given object to use this function.

The `name` parameter is the short name of the object.

The value of `type` can be either `SOURCE` or `CLASS`.

If there is an error, then `NULL` is returned. If the underlying object has changed, then `ObjectTypeChangedException` is thrown.

You can call this function as follows:

```
SELECT dbms_java.resolver('tst', 'SCOTT', 'CLASS') FROM DUAL;
```

This would return:

```
DBMS_JAVA.RESOLVER('TST','SCOTT','CLASS')
-----------------------------------------
((* SCOTT)(* PUBLIC))
```

derivedFrom

```
FUNCTION derivedFrom (name VARCHAR2, owner VARCHAR2, type VARCHAR2) RETURN
VARCHAR2
```

This function returns the source name of the object specified in `name` of the type specified in `type` and in the schema specified in `owner`. The caller must have `EXECUTE` privilege and have access to the given object to use this function.

The `name` parameter, as well as the returned source name, is the short name of the object.

The value of `type` can be either `SOURCE` or `CLASS`.

If there is an error, then `NULL` is returned. If the underlying object has changed, then `ObjectTypeChangedException` is thrown.

The returned value will be `NULL` if the object was not compiled in Oracle JVM.

You can call this function as follows:

```
SELECT dbms_java.derivedFrom('tst', 'SCOTT', 'CLASS') FROM DUAL;
```

This would return:

```
DBMS_JAVA.DERIVEDFROM('TST','SCOTT','CLASS')
----------------------------------------
tst
```

fixed_in_instance

```
FUNCTION fixed_in_instance (name VARCHAR2, owner VARCHAR2, type VARCHAR2) RETURN
NUMBER
```

This function returns the permanently kept status for object specified in `name` of the type specified in `type` and in the schema specified in `owner`. The caller must have `EXECUTE` privilege and have access to the given object to use this function.

The `name` parameter is the short name for the object.

The value of `type` can be either of `RESOURCE`, `SOURCE`, `CLASS`, or `SHARED_DATA`.

The number returned is either `0`, indicating the status is not kept, or `1`, indicating the status is kept.

You can call this function as follows:

```
SELECT dbms_java.fixed_in_instance('tst', 'SCOTT', 'CLASS') FROM DUAL;
```

This would return:

```
DBMS_JAVA.FIXED_IN_INSTANCE('TST','SCOTT','CLASS')
----------------------------------------
0
```

Consider the following statement:

```
SELECT dbms_java.fixed_in_instance('java/lang/String', 'SYS', 'CLASS') FROM DUAL;
```

This would return:

```
DBMS_JAVA.FIXED_IN_INSTANCE('JAVA/LANG/STRING','SYS','CLASS')
------------------------------------------------------------
1
```

set_output

```
PROCEDURE set_output (buffersize NUMBER)
```

This procedure redirects the output of Java stored procedures and triggers to the `DBMS_OUTPUT` package.

start_debugging

```
PROCEDURE start_debugging(host VARCHAR2, port NUMBER, timeout NUMBER)
```

This procedure is used to start the debug agent on the specified host at the specified port

stop_debugging

```
PROCEDURE stop_debugging
```

This procedure is used to stop the debug agent

restart_debugging

```
PROCEDURE restart_debugging(timeout NUMBER)
```

This procedure is used to restart the debug agent.

export_source

```
PROCEDURE export_source(name VARCHAR2, schema VARCHAR2, blob BLOB)

PROCEDURE export_source(name VARCHAR2, blob BLOB)

PROCEDURE export_source(name VARCHAR2, clob CLOB)
```

These procedures are used to export the Java source as a Java source schema object to Oracle Database. The source is specified through the `name` parameter. The source can be exported into a `BLOB` or `CLOB` object. The internal representation of the source uses the UTF8 format, so that format is used to store the source in the `BLOB` as well. The source schema object is created in the specified schema. If the schema is not specified then the current schema is used.

export_class

```
PROCEDURE export_class(name VARCHAR2, schema VARCHAR2, blob BLOB)

PROCEDURE export_class(name VARCHAR2, blob BLOB)
```

These procedures are used to export Java classes specified through the `name` parameter as Java class schema objects to Oracle Database. You cannot export a class into a `CLOB` object, only into a `BLOB` object. If the schema is specified, then the class schema object is created in this schema, else in the current schema.

export_resource

```
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)
```

The resource specified through the name parameter is exported to Oracle Database as a resource schema object in the schema specified through the `schema` parameter. If the schema is not specified then the current schema is used. The resource can be exported into either a `CLOB` object or `BLOB` object.

loadjava

```
PROCEDURE loadjava(options VARCHAR2)
```

```
PROCEDURE loadjava(options VARCHAR2, resolver VARCHAR2)
```

These procedures enable you to load classes in to the database using a call, rather than through the `loadjava` command-line tool. You can call this procedure within your Java application as follows:

```
CALL dbms_java.loadjava('... options...');
```

The options are identical to those specified on the command line. Each option should be separated by a space. Do not separate the options with a comma. The only exception to this is the `loadjava -resolver` option, which contains spaces. 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 `-thin`, `-oci`, `-user`, and `-password` options, because they relate to the database connection. The output is directed to `System.err`. The output typically goes to a trace file, but can be redirected.

dropjava

```
PROCEDURE dropjava(options VARCHAR2)
```

This procedure enables you to drop classes within the database using a call, rather than through the `dropjava` command-line tool. You can call this procedure within your Java application as follows:

```
CALL dbms_java.dropjava('... options...');
```

grant_permission

```
PROCEDURE grant_permission(grantee VARCHAR2, permission_type VARCHAR2,
permission_name VARCHAR2, permission_action VARCHAR2)
```

This method is used to grant permission to specific users or roles.

See Also: "Fine-Grain Definition for Each Permission" on page 10-5

restrict_permission

```
PROCEDURE restrict_permission(grantee VARCHAR2, permission_type VARCHAR2,
permission_name VARCHAR2, permission_action VARCHAR2)
```

This method is used to specify limitations or exceptions to general rules.

See Also: "Fine-Grain Definition for Each Permission" on page 10-5

grant_policy_permission

```
PROCEDURE grant_policy_permission(grantee VARCHAR2, permission_schema VARCHAR2,
permission_type VARCHAR2, permission_name VARCHAR2)
```

This method is used to grant and limit `PolicyTablePermission`.

See Also: "Acquiring Administrative Permission to Update Policy Table" on page 10-8

revoke_permission

```
PROCEDURE revoke_permission(permission_schema VARCHAR2, permission_type VARCHAR2,
permission_name VARCHAR2, permission_action VARCHAR2)
```

This method is used to disable a granted permission.

See Also: "Enabling or Disabling Permissions" on page 10-12

disable_permission

```
PROCEDURE disable_permission(key NUMBER)
```

This method is used to disable a granted permission.

See Also: "Enabling or Disabling Permissions" on page 10-12

enable_permission

```
PROCEDURE enable_permission(key NUMBER)
```

This method is used to enable a permission.

See Also: "Enabling or Disabling Permissions" on page 10-12

delete_permission

```
PROCEDURE delete_permission(key NUMBER)
```

This method is used to delete a granted permission.

See Also: "Enabling or Disabling Permissions" on page 10-12

set_preference

```
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 `user` parameter specifies the name of the schema to which the preference should be attached. If the logged in schema is not `SYS`, then `user` must specify the current logged in schema or the `INSERT` will fail. The `type` parameter can take either the value `U`, indicating user preference, or `S`, indicating system preference. The `abspath` parameter specifies the absolute path for the preference. `key` is the preference key used for the lookup, and `value` is the value of the preference key.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W