

Oracle8*i*

Java Developer's Guide

Release 2 (8.1.6)

December 1999

Part No. A81353-01

Java Developer's Guide, Release 2 (8.1.6)

Part No. A81353-01

Copyright © 1996, 1999, Oracle Corporation. All rights reserved.

Primary Author: Sheryl Maring

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

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

This program contains proprietary information of Oracle Corporation; it is provided under a license agreement containing restrictions on use and disclosure and is also protected by copyright, patent and other intellectual property law. Reverse engineering of the software is prohibited.

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

If this Program is delivered to a U.S. Government Agency of the Department of Defense, then it is delivered with Restricted Rights and the following legend is applicable:

Restricted Rights Legend Programs delivered subject to the DOD FAR Supplement are "commercial computer software" and use, duplication and disclosure of the Programs shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement. Otherwise, Programs delivered subject to the Federal Acquisition Regulations are "restricted computer software" and use, duplication and disclosure of the Programs shall be subject to the restrictions in FAR 52.227-14, Rights in Data -- General, including Alternate III (June 1987). Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065.

Oracle is a registered trademark, and Oracle products mentioned herein are trademarks or registered trademarks of Oracle Corporation. All other company or product names mentioned are used for identification purposes only and may be trademarks of their respective owners.

Contents

Send Us Your Comments	xi
Preface.....	xiii
1 Introduction to Java in Oracle8i	
Contents.....	1-2
Overview of Java.....	1-2
Java and Object-Oriented Programming Terminology.....	1-2
Classes.....	1-2
Attributes.....	1-4
Methods.....	1-5
Class Hierarchy.....	1-5
Interfaces.....	1-6
Polymorphism.....	1-7
The Java Virtual Machine (JVM).....	1-8
Key Features of the Java Language.....	1-11
Why Use Java in Oracle8i?	1-12
Multithreading.....	1-13
Automated Storage Management	1-14
Footprint	1-14
Performance.....	1-15
How Native Compilers Improve Performance.....	1-15
Dynamic Class Loading.....	1-16
Oracle's Java Application Strategy	1-17
Java Stored Procedures	1-18

PL/SQL Integration and Oracle RDBMS Functionality	1-18
JDBC Drivers	1-19
SQLJ – Embedded SQL in Java.....	1-19
Distributed Application Development.....	1-20
Using EJB Components	1-20
Development Tools	1-21
Overview of Oracle8i Java Documentation	1-21

2 Writing Java Applications on Oracle8i

Overview	2-2
Terminology.....	2-2
Database Sessions Imposed on Java Applications.....	2-3
Session Lifetime	2-5
Java Supported APIs	2-5
Execution Control	2-6
Migrating from JDK 1.1 to Java 2.....	2-7
Your Development Environment.....	2-7
JDBC 2.0	2-8
Java 2 Security	2-9
Java 2 ORB APIs.....	2-9
JNDI Lookup.....	2-10
Aurora ORB Interface	2-10
CORBA ORB Interface.....	2-12
Backwards Compatibility for 8.1.5 CORBA and EJB Applications.....	2-13
Java Code, Binaries, and Resources Storage.....	2-13
Preparing Java Class Methods for Execution	2-14
Compiling Java Classes.....	2-14
Compiling Source through javac.....	2-14
Compiling Source through loadjava.....	2-15
Compiling Source at Runtime	2-15
Specifying Compiler Options	2-15
Automatic Recompilation	2-18
Resolving Class Dependencies	2-19
Allowing References to Non-Existent Classes	2-20
ByteCode Verifier	2-21

Loading Classes	2-22
Two Definitions of the Same Class	2-24
Need Database Privileges and JVM Permissions	2-25
Loading JAR or ZIP Files.....	2-25
How to Grant Execute Rights	2-26
Checking Java Uploads.....	2-27
Object Name and Type	2-28
Status	2-29
Publishing	2-29
User Interfaces on the Server	2-30
Shortened Class Names	2-31
Class.forName() on JServer	2-32
Supply the ClassLoader in Class.forName	2-33
Supply Class and Schema Names to classForNameAndSchema	2-34
Supply Class and Schema Names to lookupClass.....	2-35
Supply Class and Schema Names when Serializing	2-35
Class.forName Example	2-35
Managing Your Operating System Resources	2-37
Overview of Operating System Resources	2-37
Operating System Resource Access	2-38
Operating System Resource Lifetime	2-38
Garbage Collection and Operating System Resources.....	2-39
Operating System Resources Affected Across Calls	2-40
Sockets.....	2-42
Threading in JServer	2-43
Thread Lifecycle	2-44

3 Invoking Java in the Database

Overview	3-2
Invoking Java Methods	3-3
Utilizing Java Stored Procedures	3-3
Utilizing Distributed Objects With CORBA and EJB	3-6
IIOP Transport	3-7
Naming	3-7
Creating and Deploying Enterprise JavaBeans.....	3-8

Using an EJB.....	3-9
Session Shell.....	3-9
Utilizing Remote Method Invocation (RMI).....	3-10
Utilizing Java Native Interface (JNI) Support	3-10
Utilizing SQLJ and JDBC for Querying Database.....	3-11
JDBC	3-11
SQLJ.....	3-11
An Example Comparing JDBC and SQLJ	3-12
Complete SQLJ Example	3-13
SQLJ Strong Typing Paradigm	3-15
Translating a SQLJ Program	3-16
Running a SQLJ Program in the Server.....	3-16
Converting a Client Application to Run in the Server.....	3-17
Interacting with PL/SQL.....	3-17
Debugging Server Applications	3-18
Overview.....	3-18
1. Start the Debug Proxy	3-19
Just-in-Time Debugging	3-20
2. Starting, Stopping, and Restarting the Debug Agent.....	3-20
OracleAgent Class	3-21
3. Connecting a Debugger	3-21
How To Tell You Are Executing in the Server	3-23
Redirecting Output on the Server	3-23

4 Java Installation and Configuration

Initializing a Java-Enabled Database.....	4-2
Manual Install	4-2
Requirements	4-3
Package DBMS_JAVA.....	4-3
Configuring JServer.....	4-6
Java Stored Procedure Configuration.....	4-7
Enterprise JavaBeans and CORBA Configuration.....	4-8
Enabling the Java Client.....	4-8
1. Install JDK on the Client	4-8
2. Set up CLASSPATH	4-9

3. Verify the Port/SID	4-9
4. Test Install with Samples.....	4-9

5 Security and Performance

Security	5-2
Network Connection Security	5-2
Database Contents and JVM Security.....	5-3
Java 2 Security.....	5-3
Setting Permissions	5-6
Fine-Grain Definition for Each Permission.....	5-7
Acquiring Administrative Permission to Update Policy Table.....	5-11
Creating Permissions	5-13
Enabling or Disabling Permissions.....	5-17
Permission Types.....	5-18
Initial Permission Grants.....	5-20
.....General Permission Definition Assigned to Roles	5-23
Debugging Permissions.....	5-24
Permission for Loading Classes.....	5-25
Performance	5-25
Natively Compiled Code.....	5-25
Java Memory Usage	5-26
Configuring Memory Initialization Parameters	5-27
Java Pool Memory	5-28
Displaying Used Amounts of Java Pool Memory	5-29
Correcting Out of Memory Errors	5-30
End-of-Call Migration.....	5-31
Oracle-Specific Support for End-of-Call Optimization.....	5-32

A Tools

Schema Object Tools	A-1
What and When to Load.....	A-2
Resolution.....	A-2
Digest Table	A-4
Compilation.....	A-4
loadjava	A-7

Syntax	A-7
Argument Summary	A-8
Argument Details	A-10
dropjava	A-15
Syntax	A-16
Argument Summary	A-16
Argument Details	A-17
Dropping Resources	A-18

Glossary

Send Us Your Comments

Java Developer's Guide, Release 2 (8.1.6)

Part No. A81353-01

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

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

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

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

Please indicate if you would like a reply.

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

Preface

Who Should Read This Book

This book has been written for the following audiences:

- **Management**—You might have purchased Oracle8i for reasons other than Java development within the database. However, if you want to know more about Oracle8i Java features, see "[Overview of Oracle8i Java Documentation](#)" on page 1-21 for a management perspective.
- **Non-Java Developers**—Oracle database programming consists of PL/SQL and other non-Java programming. For experienced PL/SQL developers who are not familiar with Java, a brief overview of Java and object-oriented concepts is discussed in the first part of [Chapter 1, "Introduction to Java in Oracle8i"](#). For more detailed information on Java, see "[Java Information Resources](#)" at the end of this Preface.
- **Java Developers**—Pure Java developers are used to a Java environment that follows Sun Microsystem's specification. However, when Java is combined in the database, both Java and database concepts merge. Thus, the Java environment within Oracle8i is expanded to include database concerns. The bulk of this book discusses the differences you must understand to run Java in the database. The following outlines the two viewpoints that arise from this merge:
 - * **Java environment**—Note that Oracle8i delivers a compliant Java implementation—any 100% pure Java code will work. Oracle8i JServer affects your Java development in the way you manage your classes and the environment in which your classes exist. For example, the classes must be loaded into the database. In addition, there is a clearer separation of client and server in the Oracle8i model.

-
- * Database environment—You must be aware of database concepts for managing your Java objects. This book gives you a comprehensive view of how the two well-defined realms—Oracle8i database and Java environment—fit together. For example, when deciding on your security policies, you must consider both database security and Java security for a comprehensive security policy.

Java API Programming Models

The building blocks Java developers use in Oracle8i are as follows:

- Java stored procedures—You can develop Java applications that are stored in the database. Once loaded, these procedures can be invoked from SQL, PL/SQL, or as triggers. See the *Oracle8i Java Stored Procedures Developer's Guide* for more information.
- JDBC and SQLJ—You can write a Java application that accesses SQL data from the client or directly on the server.
- Distributed Java CORBA or EJB applications—You can develop distributed EJB or CORBA applications that are loaded and invoked in the database.

Each of these models is briefly discussed in [Chapter 1, "Introduction to Java in Oracle8i"](#) and examples are given in [Chapter 3, "Invoking Java in the Database"](#). Both of these chapters should help you decide which model to use for your particular application. Once you decide on the appropriate model, examine the appropriate developer's guide for in depth information on each model. For example, if you decide to use Java stored procedures, you should examine the book *Oracle8i Java Stored Procedures Developer's Guide*.

Java Information Resources

The following table lists the sources of current information discussed in the Java programming documentation suite:

Location	Description
http://www.oracle.com/java	The latest offerings, updates, and news for Java within the Oracle8i database. This site contains Frequently Asked Questions (FAQ), updated JDBC drivers, SQLJ reference implementations, and white papers that detail Java application development. In addition, you can download try-and-buy Java tools from this site.

Location	Description
http://java.sun.com/	Sun Microsystem's web site that is the central source for Java. This site contains Java products and information, such as tutorials, book recommendations, and the Java Developer's Kit (JDK). The JDK is located at http://java.sun.com/products
http://java.sun.com/docs/books/jls http://java.sun.com/docs/books/vmspec	The Oracle8i Java Server (JServer) is based on the Java Language (JLS) and the Java Virtual Machine (JVM) specifications.
comp.lang.java.programmer comp.lang.java.databases	Internet newsgroups can be a valuable source of information on Java from other Java developers. We recommend that you monitor these two newsgroups. Note: Oracle monitors activity on some of these newsgroups and posts responses to Oracle-specific issues.

Your local or on-line bookstore has many useful Java references. You can find another listing of materials that are helpful to beginners and that you can use as general references, in the *Oracle8i Java Stored Procedures Developer's Guide*.



Introduction to Java in Oracle8i

Java applications are supported within the Oracle8i database. Java applications can range from the simple standalone application to large, enterprise solutions using EJB or CORBA. All supported Java APIs cannot be covered within a single document; thus, several books describe the full support for Java within Oracle8i. This book provides a general overview for how you should program your Java applications when loading and running these applications in the database. Secondly, this book helps you choose which type of Java application you might develop, and direct you to the corresponding book for detailed information on that subject.

This chapter contains the following information:

- Introduces the Java language for Oracle database programmers. Oracle PL/SQL developers are accustomed to developing server-side applications that have tight integration with SQL data. You can develop Java server-side applications that take advantage of the scalability and performance of the Oracle database. If you are not familiar with Java, see "[Overview of Java](#)" on page 1-2.
- Examines why you should consider using Java within an Oracle8i database. See "[Why Use Java in Oracle8i?](#)" on page 1-12. In addition, a brief description is given of the Java application development interfaces supported within Oracle8i. These include SQLJ, JDBC, Java stored procedures, EJB, and CORBA. See "[Oracle's Java Application Strategy](#)" on page 1-17.
- Provides a roadmap to the Oracle8i Java documentation. Several Java application types are supported within Oracle8i. Each of these types are described generally in this book, and more intimately in their own books. "[Overview of Oracle8i Java Documentation](#)" on page 1-21 shows you which books cover each Java application type in detail.

Contents

- [Overview of Java](#)
- [Why Use Java in Oracle8i?](#)
- [Oracle's Java Application Strategy](#)
- [Overview of Oracle8i Java Documentation](#)

Overview of Java

Java, which was developed at Sun Microsystems, has emerged over the last several years as the object-oriented programming language of choice. It includes the following concepts:

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

The result is a language easily learned by existing C programmers, but which remains truly object-oriented and efficient for application-level programs.

Java and Object-Oriented Programming Terminology

This section covers some basic terminology for discussing details of Java application development in the Oracle8i environment. The terms should be familiar to experienced Java programmers. A detailed discussion of object-oriented programming or of the Java language is beyond the scope of this book. Many texts, in addition to the complete language specification, are available at your bookstore and on the Internet. See "[Java Information Resources](#)" in the [Preface](#), and "Suggested Reading" in the *Oracle8i Java Stored Procedures Developer's Guide*, for pointers to reference materials and for places to find Java-related information on the Internet.

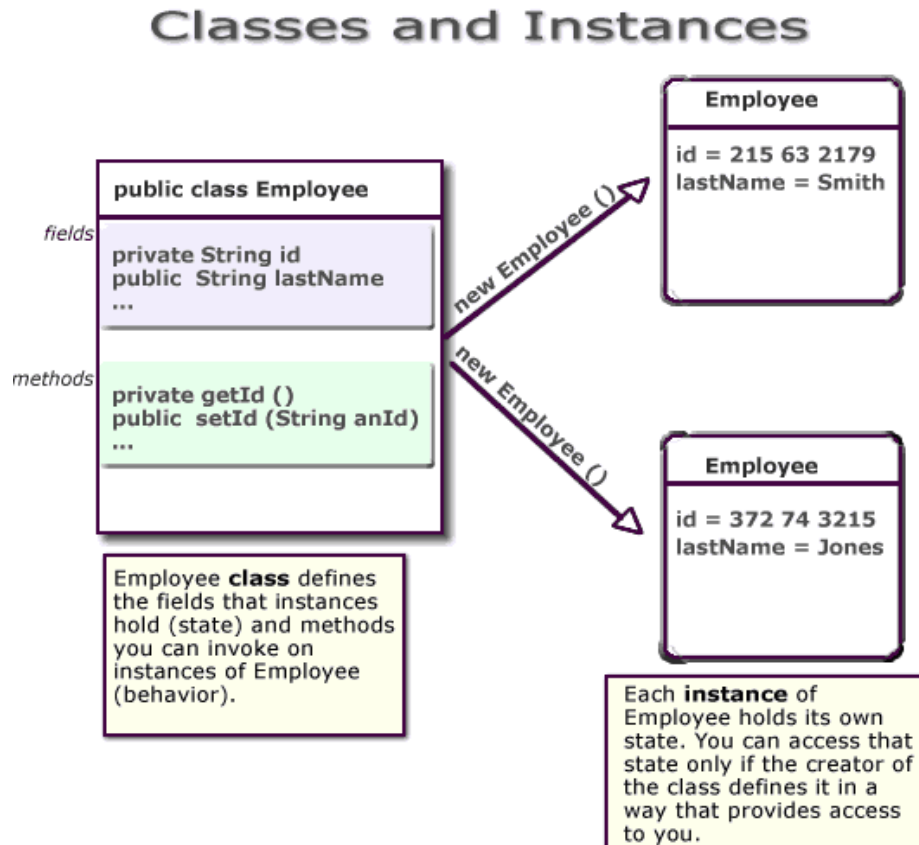
Classes

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

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

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

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 for that single employee. The example in [Figure 1–1](#) shows two instances of employee—Smith and Jones. Each instance contains information relevant to the individual employee.

Attributes

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

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

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

Methods

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

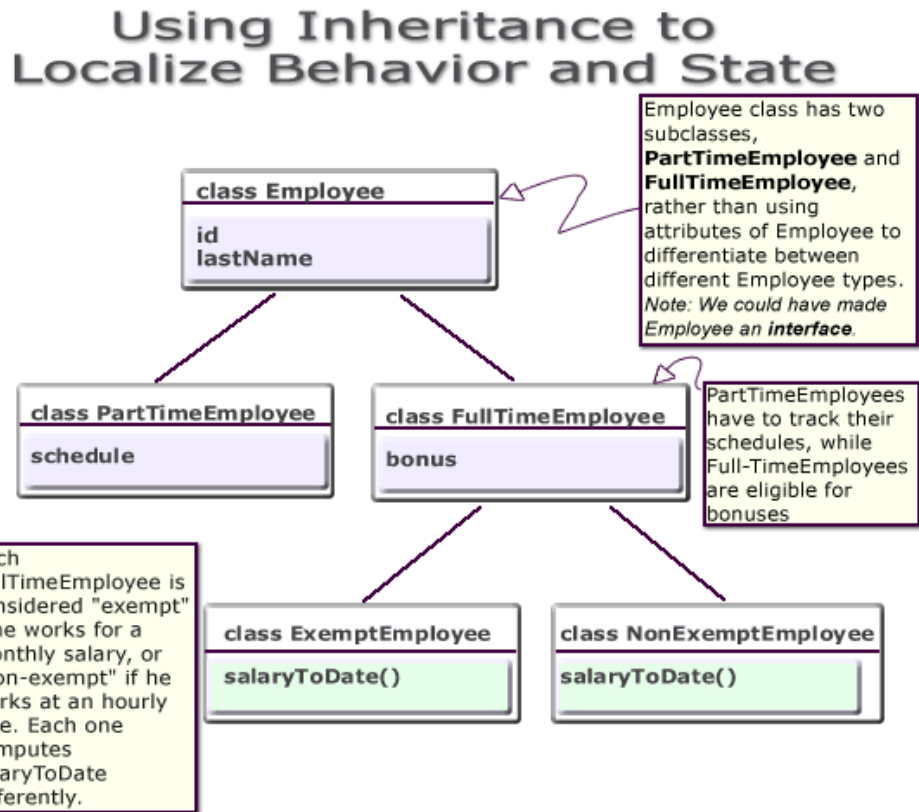
In addition, with encapsulation, you can declare that the `Employee.getId()` method is private, or you can decide not to write an `Employee.getId()` method. Encapsulation helps you write programs that are reusable and not misused. Encapsulation makes public only those features of an object that are declared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

Class Hierarchy

Java defines classes within a large hierarchy of classes. At the top of the hierarchy is the `Object` class. All classes in Java inherit from the `Object` class at some level, as you walk up through the inheritance chain of superclasses. When we say Class B inherits from Class A, each instance of Class B contains all the fields defined in class B, as well as all the fields defined in Class A. For example, in [Figure 1-2](#), the `FullTimeEmployee` class contains the `id` and `lastName` fields defined in the `Employee` class because it inherits from the `Employee` class. In addition, the `FullTimeEmployee` class adds another field, `bonus`, which is contained only within `FullTimeEmployee`.

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

Figure 1–2 Inheritance Hierarchy



Instances of Class B are substitutable for instances of Class A, which makes inheritance another powerful construct of object-oriented languages for improving code reuse. You can create new classes that define behavior and state where it makes sense in the hierarchy, yet make use of pre-existing functionality in class libraries.

Interfaces

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

are implemented in classes declared to implement an interface. Multiple inheritance occurs when a single class simultaneously supports many interfaces.

Polymorphism

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

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

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

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

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

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

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

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

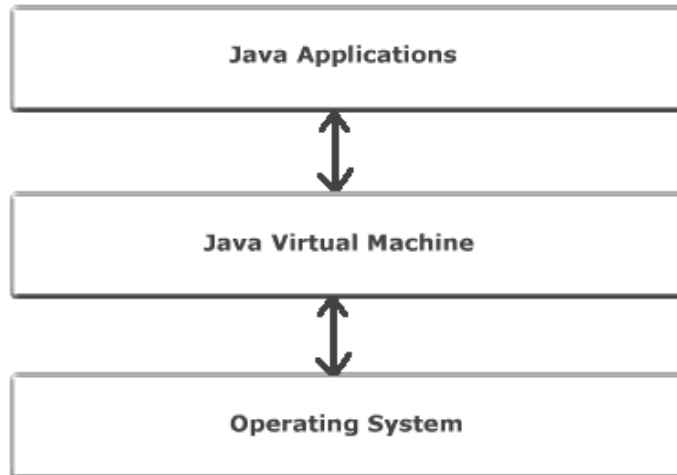
The common usage of the method name `compensationToDate()` allows you to invoke the identical method on different classes and receive different results, without knowing the type of employee you are using. You do not have to write a special method to handle `FullTimeEmployees` and `PartTimeEmployees`. This

ability for the different objects to respond to the identical message in different ways is known as polymorphism.

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

The Java Virtual Machine (JVM)

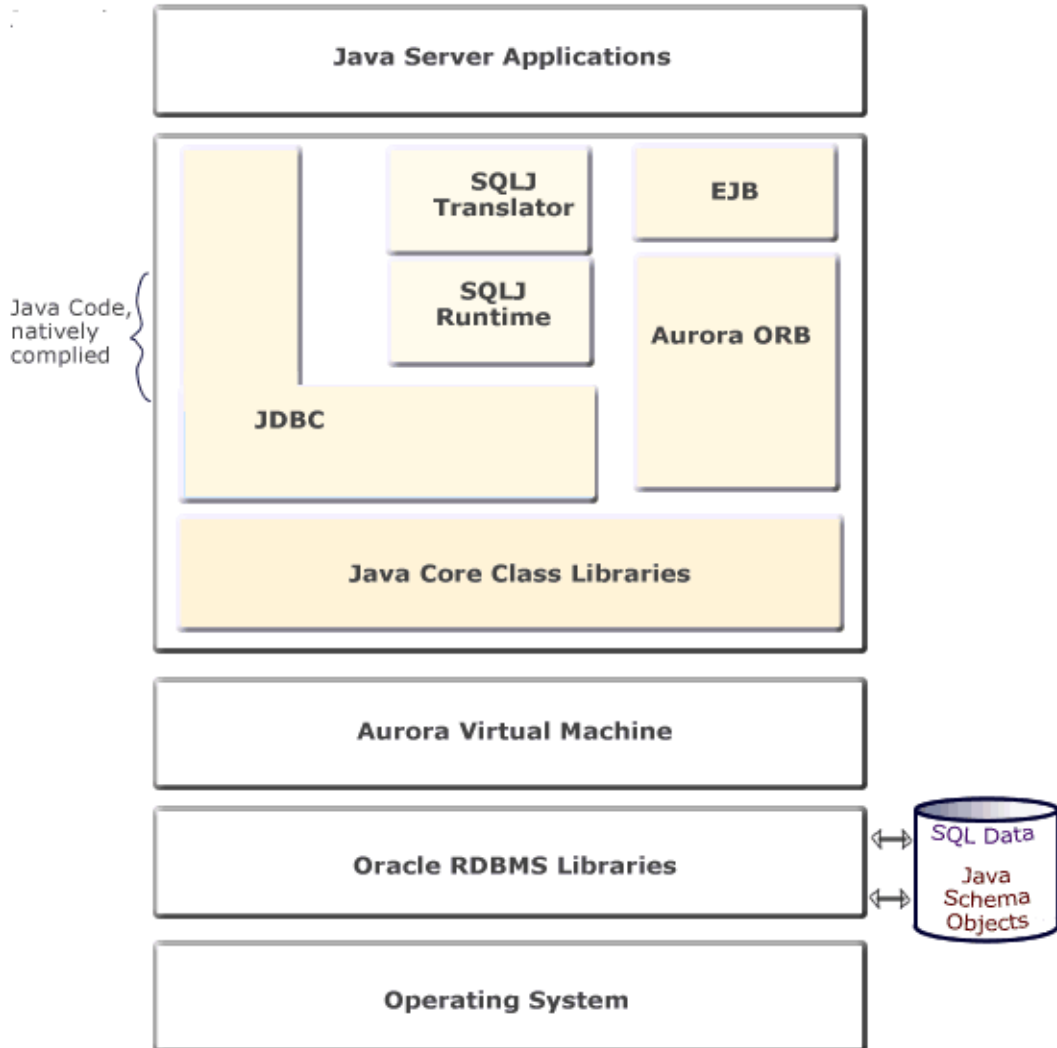
As with other high-level computer languages, your Java source compiles to low-level machine instructions. In Java, these instructions are known as bytecodes (because their size is uniformly one byte of storage). Most other languages, such as C, compile to machine-specific instructions; for example, instructions specific to an Intel or HP processor. Your Java source compiles to a standard, platform-independent set of bytecodes, which interacts with a Java virtual machine (JVM). The JVM is a separate program optimized for the specific platform on which you execute your Java code. [Figure 1-3](#) shows how Java can maintain platform independence. Your Java source is compiled into bytecodes, which are platform independent. Each platform has installed a JVM that is specific to its operating system. The Java bytecodes from your source get interpreted through the JVM into appropriate platform dependent actions.

Figure 1–3 Java Component Structure

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

[Figure 1–4](#) illustrates how Oracle’s Java applications sit on top of the Java core class libraries, which in turn sit on top of the JVM. Because Oracle’s Java support system is located within the database, the JVM interacts with the Oracle database libraries, instead of directly with the operating system.

Figure 1-4 JServer Component Structure



Sun Microsystems furnishes publicly available specifications for both the Java language and the JVM. The Java language specification (JLS) defines things such as syntax and semantics; the JVM specification defines the necessary low-level behavior for the “machine” that executes the bytecodes. In addition, Sun Microsystems provides a compatibility test suite for JVM implementors to determine if they have complied with the specifications. This test suite is known as the Java Compatibility Kit (JCK). Oracle’s 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.

Key Features of the Java Language

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

- **Simplicity**—Java is a simpler language to master than most others you use in server applications because of its consistent enforcement of the object model. The large, standard set of class libraries brings powerful tools to Java developers on all platforms.
- **Portability**—Java is portable across platforms. It is possible to write platform-dependent code in Java, but it is also simple to write programs that move seamlessly across machines. Oracle server applications, which do not support graphical user interfaces directly on the platform that hosts them, also tend to avoid the few platform portability issues that Java has.
- **Automatic Storage Management**—The Java virtual machine automatically performs all memory allocation and deallocation during program execution. Java programmers can neither allocate nor free memory explicitly. Instead, they depend on the JVM to perform these bookkeeping operations, allocating memory as they create new objects and deallocating memory when the objects are no longer referenced. The latter operation is known as garbage collection.
- **Strong Typing**—Before you use a Java variable, you must declare the class of the object it will hold. Java’s strong typing makes it possible to provide a reasonable and safe solution to inter-language calls in the case of Java and PL/SQL and to integrate Java and SQL.
- **No Pointers**—Although Java retains much of the flavor of C in its syntax, it does not support direct pointers or pointer manipulation. You pass all parameters, except primitive types, by reference (that is, object identity is preserved), not by value. Java does not provide C’s low level, direct access to pointers, which eliminates memory corruption and leaks.

- **Exception Handling**—Java exceptions are objects. Java requires developers to declare which exceptions can be thrown by methods in any particular class.
- **Flexible Namespace**—Java defines classes and holds them within a hierarchical structure that mirrors the Internet's domain namespace. You can distribute Java applications and avoid name collisions. Java extensions such as the Java Naming and Directory Interface (JNDI) provide a framework for multiple name services to be federated. Java's namespace approach is flexible enough for Oracle to incorporate the concept of a schema for resolving class names, while fully complying with the language specification.
- **Security**—The design of Java bytecodes and the JVM allow for built-in mechanisms to verify Java binary code has not been tampered with. Oracle8i is installed with an instance of SecurityManager, which, combined with Oracle database security, secures who can invoke any Java methods.
- **Standards for Connectivity to Relational Databases**—JDBC and SQLJ enable Java code to access and manipulate data resident in relational databases. Oracle provides drivers that allow vendor-independent, portable Java code to access the relational database.

Why Use Java in Oracle8i?

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

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

- [Multithreading](#)
- [Automated Storage Management](#)
- [Footprint](#)
- [Performance](#)
- [Dynamic Class Loading](#)

Multithreading

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

As a database server, Oracle8i efficiently schedules work for thousands of users. The Oracle8i Aurora JVM uses the facilities of the RDBMS server to concurrently schedule Java execution for thousands of users. Although Oracle8i supports Java language level threads required by the Java language specification (JLS) and Java Compatibility Kit (JCK), using threads within the scope of the database will not increase your scalability. Using the embedded scalability of the database eliminates the need for writing multithreaded Java servers. You should use the database's facilities for scheduling users by writing single-threaded Java applications. The database will take care of the scheduling between each application; thus, you achieve scalability without having to manage threads. You can still write multithreaded Java applications, but multiple Java threads will not increase your server's performance.

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

- **Non-Oracle8i model**—A single user maps to a single Java language level thread; the same single garbage collector manages all garbage from all users. Different techniques typically deal with allocation and collection of objects of varying lifetimes and sizes. The result in a heavily multithreaded application is, at best, dependent upon operating system support for native threads, which can be unreliable and limited in scalability. High levels of scalability for such implementations have not been convincingly demonstrated.
- **Oracle8i JServer model**—Even when thousands of users connect to the server and execute the same Java code, each user experiences it as if he is executing his own Java code on his own Java virtual machine. The responsibility of the Oracle8i JServer is to make use of operating system processes and threads, using the scalable approach of the Oracle RDBMS. As a result of this approach, the JVM's garbage collector is more reliable and efficient because it never collects garbage from more than one user at any time. Refer to "[Threading in JServer](#)" on page 2-43 for more information on the thread model implementation in JServer.

Automated Storage Management

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

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

- Aurora JVM uses the Oracle8i scheduling facilities, which can manage multiple users efficiently.
- Garbage collection is consistently performant for multiple users because garbage collection is focused on a single user within a single session. The Oracle8i Aurora JVM enjoys a huge advantage because the burden and complexity of the memory manager's job does not increase as the number of users increases. The memory manager performs the allocation and collection of objects within a single session—which typically translates to the activity of a single user.
- Aurora JVM uses different garbage collection techniques depending on the type of memory used. These techniques provide high efficiency and low overhead.

Footprint

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

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

From a scalability perspective, the key to supporting many concurrent clients is a minimum per-user session footprint. Aurora 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. Aurora uses three types of garbage collection algorithms to maintain the user's session memory:

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

Performance

JServer performance is enhanced by implementing a native compiler.

How Native Compilers Improve Performance

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

Compiler	Description
Just In Time (JIT) Compilation	JIT compilers quickly compile Java bytecodes to native (platform-specific) machine code during runtime. This does not produce an executable to be executed on the platform; instead, it provides platform-dependent code from Java bytecodes that is executed directly after it is translated. This should be used for Java code that is run frequently, which will be executed at speeds closer to languages such as C.

Compiler	Description
Static Compilation	Static compilation translates Java bytecodes to platform-independent C code before runtime. Then, a standard C compiler compiles the C code into an executable for the target platform. This approach is more suitable for Java applications that are modified infrequently. This approach takes advantage of the mature and efficient platform-specific compilation technology found in modern C compilers.

Oracle8i uses static compilation to deliver its core Java class libraries, the Aurora/ORB, and JDBC code in natively compiled form. It is applicable across all the platforms Oracle supports, whereas a JIT approach requires low-level, processor-dependent code to be written and maintained for each platform. In a future release, this native compilation technology will be available for use with your own Java code. Refer to "[Natively Compiled Code](#)" on page 5-25 for more information.

Dynamic Class Loading

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

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

Problem	Description	Solution
Reliability	A benefit of dynamic class loading is that it supports program updating. For example, you would update classes on a server, and clients who download the program and load it dynamically see the update whenever they next use the program. Server programs tend to emphasize reliability. As a developer, you must know that every client executes a specific program configuration. You do not want clients to inadvertently load some classes that you did not intend them to load.	Oracle8i separates the upload and resolve operation from the class loading operation at runtime. You upload Java code you developed to the server using the <code>loadjava</code> utility. Instead of using <code>CLASSPATH</code> , you specify a resolver at installation time. The resolver is analogous to <code>CLASSPATH</code> , but allows you to specify the schemas in which the classes reside. This separation of resolution from class loading means you always know what program users execute. Refer to Appendix A, "Tools" , for details on <code>loadjava</code> and resolvers.

Oracle's Java Application Strategy

One appeal of Java is its ubiquity and the growing number of programmers capable of developing applications using it. Oracle furnishes enterprise application developers with an end-to-end Java solution for creating, deploying, and managing Java applications. The total solution consists of client- and server-side programmatic interfaces, tools to support Java development, and a Java virtual machine integrated with the Oracle8i database server. All of these products are 100 percent compatible with Java standards.

In addition to the Aurora JVM, the Oracle8i's Java programming environment consists of:

- Java stored procedures as the Java equivalent and companion for PL/SQL. Java stored procedures are tightly integrated with PL/SQL. You can call a Java stored procedure from a PL/SQL package; you can call PL/SQL procedures from a Java stored procedure.
- SQL data can be accessed through JDBC and SQLJ programming interfaces.
- Distributed enterprise application development through an Object Request Broker (the Aurora/ORB) and Enterprise JavaBeans support.
- Tools and scripts used in assisting in development, class loading, and class management.

To enable your decision making for which Java APIs to use, examine the following table:

Type of functionality you need	Java API to use
To have a Java procedure invoked from SQL, such as a trigger.	Java Stored Procedures
To invoke a static, simple SQL statement from a known table with known column names from a Java object.	SQLJ
To invoke dynamic, complex SQL statements from a Java object.	JDBC
To create a multi-tier Java application.	CORBA or EJB

Java Stored Procedures

If you are a PL/SQL programmer exploring Java, you will be interested in Java stored procedures. A Java stored procedure is a program you write in Java to execute in the server, exactly as a PL/SQL stored procedure. You invoke it directly with products like SQL*Plus or indirectly with a trigger and can access it from any Net8 client—OCI, PRO*, JDBC or SQLJ. The *Oracle8i Java Stored Procedures Developer's Guide* explains how to write stored procedures in Java, how to access them from PL/SQL, and how to access PL/SQL functionality from Java.

In addition, you can use Java to develop powerful programs independently of PL/SQL. Oracle8i provides a fully compliant implementation of the Java programming language and JVM.

PL/SQL Integration and Oracle RDBMS Functionality

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

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

- **JDBC Drivers**—Used to build client/server 2-tier applications.
- **SQLJ - Embedded SQL in Java**—Used to access static SQL. You must know the name of the columns.

JDBC Drivers

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

Driver	Description
JDBC Thin Driver	You can use the JDBC thin driver to write 100% pure Java applications and applets that access Oracle SQL data. The JDBC thin driver is especially well-suited to Web browser-based applications and applets because you can dynamically download it from a Web page just like any other Java applet.
JDBC Oracle Call Interface Driver	The JDBC Oracle Call Interface (OCI) driver accesses Oracle-specific native code (that is, non-Java) libraries on the client or middle tier, providing a richer set of functionality and some performance boost compared to the JDBC thin driver, at the cost of significantly larger size.
JDBC Server-side Internal Driver	Oracle8i uses the JServer server-side internal driver when Java code executes on the server. It allows Java applications executing in the server's Java virtual machine to access locally defined data (that is, on the same machine and in the same process) with JDBC. It provides a further performance boost because of its ability to use underlying Oracle RDBMS libraries directly, without the overhead of an intervening network connection between your Java code and SQL data. By supporting the same Java-SQL interface on the server, Oracle 8i does not require you to rework code when deploying it.

For more information on JDBC, see "[Utilizing SQLJ and JDBC for Querying Database](#)" on page 3-11 or a complete detailed description within the *Oracle8i JDBC Developer's Guide and Reference*.

SQLJ – Embedded SQL in Java

JDBC provides a low-level API for accessing SQL data from Java. It introduces Java classes that mirror their SQL equivalents. 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—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 employs a preprocessor to translate the program to standard Java source

with JDBC calls. At runtime, 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 it in stored procedures, triggers, methods within the JServer environment, and with EJB and CORBA. 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 JServer provides a complete Java environment, you can not only compile SQLJ programs on a client for execution on the JServer, but you can compile them directly on the server. Oracle8i's adherence to Internet standards allows you to choose the development style that fits your needs.

For more information on SQLJ, see "[Utilizing SQLJ and JDBC for Querying Database](#)" on page 3-11 or for a complete detailed description, see the *SQLJ Developer's Guide and Reference*.

Distributed Application Development

In addition to support for traditional RDBMS-stored procedures, JServer comes with a built-in CORBA 2.0 ORB and support for Enterprise JavaBeans (EJB). CORBA and EJB allow you to distribute Java components and application logic between client, middle-tier, and database server.

- | | |
|----------------------------|---|
| OMG CORBA ORB | The ORB allows programs you develop in any language to communicate directly with the Oracle8i database through Internet Inter-ORB Protocol (IIOP), the standard wire-protocol defined by the Object Management Group (OMG). |
| Enterprise JavaBeans (EJB) | For 100% pure Java applications, EJB is the standard framework for deploying component-based, secure, transactional applications on JServer. |

Using EJB Components

"[Java and Object-Oriented Programming Terminology](#)" on page 1-2 discusses encapsulation as a key element of object-oriented programming. Each object maintains its own private state and supports a set of behaviors, which you implement as methods. Java provides a formal way to define components, using JavaBeans. A JavaBean component is a reusable object or group of objects (more precisely, an object graph) that you can manipulate in a builder tool of some type. IDEs, such as JDeveloper, provide tools to build user interfaces that use JavaBeans

and create JavaBean components. Each bean specifies its public interface and properties that can be manipulated. JavaBeans do not have to be visually-oriented components. Virtually any Java programming abstraction can potentially be represented and manipulated as a bean.

A large component library provides the basis for assembling an application from pre-built, pre-tested building blocks. However, beans are limited in their ability to build complex business applications involving transactional logic. To address this limitation, a group of companies, including Oracle, Sun Microsystems, and IBM, developed the Enterprise JavaBean (EJB) specification. EJB introduces a declarative mechanism for specifying how components deal with transactions and security. Refer to the *Oracle8i Enterprise JavaBeans and CORBA Developer's Guide* for detailed information about using EJB components in Oracle8i.

There are alternative component models to JavaBeans and Enterprise JavaBeans—notably, Microsoft's COM and COM+ models. If you have existing Microsoft COM-oriented applications, they can interact with open Internet standards, such as JavaBeans and EJB, with bridge products available from different vendors.

Development Tools

The introduction of Java to the Oracle8i server allows you to use several Java Integrated Development Environments. JServer's adherence to Java compatibility and open Internet standards and protocols ensures that your 100% pure Java programs work when you deploy them on JServer. Oracle delivers JServer with many tools or utilities, all written in Java, that make development and deployment of Java server applications easier. Oracle's JDeveloper has many features designed specifically to make deployment of Java stored procedures and Enterprise JavaBeans easier.

Overview of Oracle8i Java Documentation

This guide is the starting point for Oracle8i Java developers. It outlines some of the unique features of Java programming with Oracle8i, including aspects of the Aurora JVM, explaining how to take advantage of these features in your Java programs.

Once you have mastered the basics of Java development within the Oracle8i database, you might need more information for the specific protocol you will use in implementing your Java application. The following list includes other books within the documentation set that will help you in your application development:

Protocol	Description	Book Title
JDBC	Oracle8i Java developers should become familiar with Oracle's Java Database Connectivity (JDBC) product because it provides the basis for accessing SQL data from Java programs, as well as Oracle-specific extensions to this Java standard. JDBC is an industry standard.	<i>Oracle8i JDBC Developer's Guide and Reference</i>
SQLJ	You may find it easier to develop Java programs that access SQL data using embedded SQL in Java (SQLJ). SQLJ uses a preprocessor, written in Java, to translate embedded SQL statements to standard JDBC-style programs. SQLJ is an industry standard.	<i>Oracle8i SQLJ Developer's Guide and Reference</i>
JPublisher	JPublisher provides a simple and convenient tool to create Java programs that access existing Oracle relational database tables.	<i>Oracle8i JPublisher User's Guide</i>
Java Stored Procedures	<p>If you are a PL/SQL programmer exploring Java, you will be interested in Java stored procedures. A Java stored procedure is a program you write in Java to execute in the server, exactly as a PL/SQL stored procedure. You invoke it directly with products like SQL*Plus or indirectly with a trigger and can access it from any Net8 client—OCI, PRO*, JDBC or SQLJ. The <i>Oracle8i Java Stored Procedures Developer's Guide</i> explains how to write stored procedures in Java, how to access them from PL/SQL, and how to access PL/SQL functionality from Java.</p> <p>In addition, you can use Java to develop powerful programs independently of PL/SQL. Oracle8i provides a fully compliant implementation of the Java programming language and JVM.</p>	<i>Oracle8i Java Stored Procedures Developer's Guide.</i>
EJB and CORBA	For distributed applications, you will utilize either the ORB or EJB technology. Oracle's open distributed object technology is included in its Object Request Broker (the Aurora/ORB) and Enterprise JavaBeans (EJB) functionality. The Aurora/ORB and EJB furnish powerful standards-based frameworks and tools to help you build scalable Java applications that provide seamless transactional access to Oracle data across your intranet or the Internet.	<i>Oracle8i Enterprise JavaBeans and CORBA Developer's Guide</i>

Writing Java Applications on Oracle8i

JServer runs standard Java applications. However, by integrating Java classes within the database server, your environment is different from a typical Java development environment. This chapter describes the basic differences for writing, installing, and deploying Java applications within Oracle8i.

- [Overview](#)
- [Database Sessions Imposed on Java Applications](#)
- [Execution Control](#)
- [Migrating from JDK 1.1 to Java 2](#)
- [Java Code, Binaries, and Resources Storage](#)
- [Preparing Java Class Methods for Execution](#)
- [User Interfaces on the Server](#)
- [Shortened Class Names](#)
- [Class.forName\(\) on JServer](#)
- [Managing Your Operating System Resources](#)
- [Threading in JServer](#)

Note: You should refer to the detailed documentation for the different JServer APIs, to fully explore their usage. The intent of this chapter is to place Java APIs in an overall context, with enough detail for you to see how they fit together and how you use them in the JServer environment.

Overview

As discussed in Chapter 1, the Oracle8i JServer platform is a standard, compatible Java environment, which will execute any 100% pure Java application. It has been implemented by Oracle to be compatible with the Java Language Specification and the Java virtual machine specification. It supports the standard Java binary format and the standard Java APIs. In addition, Oracle8i adheres to standard Java language semantics, including dynamic class loading at runtime. However, unlike other Java environments, the JServer is embedded within the Oracle8i RDBMS and, therefore, introduces a number of new concepts. This section gives an overview of the differences between Sun Microsystems's JDK environment and the environment that occurs when you combine Java within the Oracle8i database.

Terminology

Term	Definition
JServer	Java-enabled Oracle8i database server.
Aurora	Oracle8i JVM.
Session	As a user who executes Java code, you must establish a session in the server. The word <i>session</i> as we employ it here is identical to the standard Oracle (or any other database server) usage. A session is typically, although not necessarily, bounded by the time a single user connects to the server.
Call	<p>When a user causes Java code to execute within a session, we refer to it as a <i>call</i>. You can initiate a call in different ways.</p> <ul style="list-style-type: none">■ A SQL client program executes a Java stored procedure.■ A trigger can execute a Java stored procedure.■ A PL/SQL program calls some Java code.■ A CORBA client invokes a method on a CORBA object.■ An EJB client invokes a method on an EJB object. <p>In all cases, a call begins, some combination of Java, SQL, or PL/SQL code is executed to completion, and the call ends.</p>

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

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

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

- Load and publish your Java applications before execution—See "[Java Code, Binaries, and Resources Storage](#)" and "[Preparing Java Class Methods for Execution](#)" starting on page 2-13.
- Running within a database session—See "[Database Sessions Imposed on Java Applications](#)" on page 2-3.

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

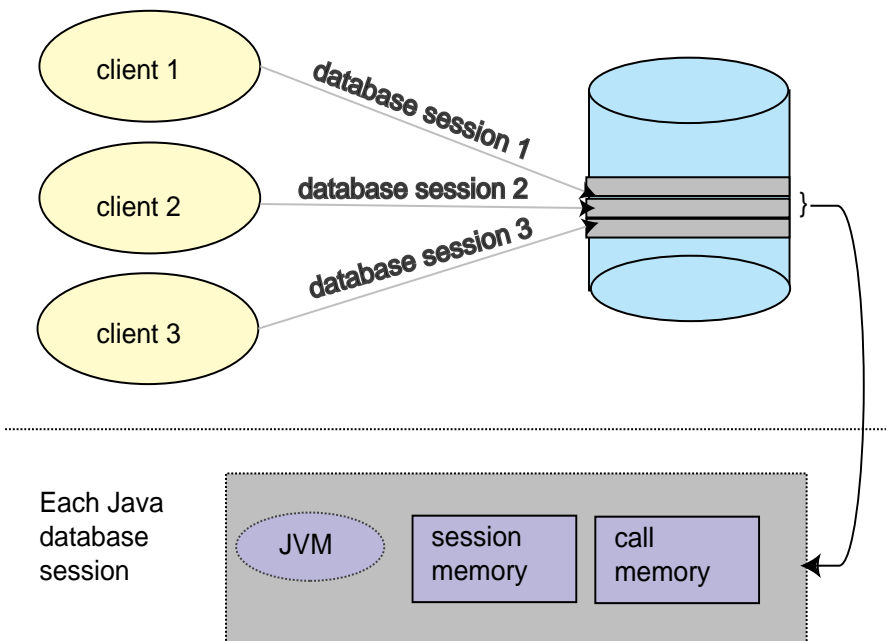
- [Execution Control](#)
- [User Interfaces on the Server](#)
- [Shortened Class Names](#)
- [Class.forName\(\) on JServer](#)
- [Managing Your Operating System Resources](#)
- [Threading in JServer](#)

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

Database Sessions Imposed on Java Applications

In incorporating Java within the Oracle8i database, your Java application exists within the context of a database session. JServer sessions are entirely analogous to traditional Oracle sessions. Each JServer session maintains the client's Java state across calls within the session.

As demonstrated in [Figure 2-1](#), each Java client starts up a database session as the environment for executing Java within the database. Garbage collection, session memory, and call memory exist solely for each client within its session.

Figure 2–1 Java environment within each database session

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

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

Within a single session, the client has its own Java environment, which is separate from every other client's environment. It appears to the client as if a separate, individual JVM was invoked for each session, although the implementation is vastly more efficient than this seems to imply. Within a session, the Aurora JVM manages the scalability for you within the database. Every call executed from a single client is managed within its own session—separately from other clients. The Aurora 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 Java server code. Static variables are all local to the client. No client can access another client's static variables, because the memory is not available across session boundaries. Each client executes its calls within its own session, so each client's activities are separate from any other client. During a call, you can store objects in static fields of different classes, and you can expect this state to be available for your next call. The entire state of your Java program is private to you and exists for your entire session.

The Aurora JVM manages the following within the session:

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

Session Lifetime

When you connect to Oracle8i, you start a database session. A session ends when one of the following events occurs:

1. The user invokes the `oracle.aurora.vm.OracleRuntime.exitSession()` method.
2. The session times out. This is optional for CORBA or EJB sessions.
3. The user takes some action outside of Java code to end the database session.

Java Supported APIs

For the current Oracle8i release, we offer three Java APIs—Java stored procedures, CORBA distributed objects, and Enterprise JavaBeans (EJBs).

API	Lifetime
Java stored procedures	The lifetime of a Java stored procedure session is identical to the SQL session in which it is embedded. This concept is familiar to PL/SQL users. Any state represented in Java transparently persists for the lifetime of the RDBMS session, simplifying the process of writing stored procedures, triggers, and methods for Oracle Abstract Data Types. Individual invocations of Java code within a session are known as calls. For example, a call may be initiated by a SQL call.
CORBA and EJB	CORBA and EJB provide a more object-oriented style of message sending between clients and servers. Clients must implicitly or explicitly establish a session in the server. Every message you send on the client to a server-resident object initiates a call. Refer to the <i>Oracle8i Enterprise JavaBeans and CORBA Developer's Guide and Reference</i> for specifics.

Note: The concepts of call and session apply across all uses of Oracle8i.

In addition, you can access SQL data through SQLJ or JDBC. See [Chapter 3, "Invoking Java in the Database"](#) for examples of each Java API.

Execution Control

In Sun Microsystems's JDK environment, you develop Java applications with a `main()` method, which is called by the interpreter when the class is run. The `main()` method is invoked when you execute `java <classname>` on the command-line. This command starts the java interpreter and passes the desired classname to be executed to the interpreter. The java interpreter loads the class and starts the execution by invoking `main()`. However, Java applications within the database do not start their execution from a `main()` method.

After loading your Java application within the database (see "[Loading Classes](#)" on page 2-22), you can execute your Java code by invoking any static method within the loaded class. The class or methods must be published for you to execute them (see "[Publishing](#)" on page 2-29). Your only entry point is no longer always assumed to be `main()`. Instead, when you execute your Java application, you specify a method name within the loaded class as your entry point.

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

```
java myprogram
```

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

In Oracle8i, 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.

Migrating from JDK 1.1 to Java 2

Java 2 is, for the most part, compatible with JDK 1.1. Sun Microsystems changed certain features, such as the security feature, in Java 2. These changes are documented at the following Sun Microsystems's web site:

<http://java.sun.com/products/jdk/1.2/compatibility.html>

The following sections discuss how the changes made within Java 2 affected Oracle8i:

- [Your Development Environment](#)
- [JDBC 2.0](#)
- [Java 2 Security](#)
- [Java 2 ORB APIs](#)

Your Development Environment

The level of your development environment determines your interoperability with the server. If your development environment is Java 2-based, any code compiled and debugged on your system can be loaded and executed on the database. However, if you are developing applications in a JDK 1.1 development environment, you can only use JDK 1.1 classes. This application can be loaded and executed in the database, which is Java 2-based, with a few exceptions that are discussed in this section. Of course, you always have the option to code your Java 2-based application on your system, load it into the database, and use the Java 2 compiler that exists on the database.

Note: There is another workaround for using Java 2 security even though your code is JDK 1.1-based. The security APIs are provided within a PL/SQL package. You can use these call specifications before your code executes; thus, enabling the correct Java 2 permissions.

JDBC 2.0

Even though 8.1.5 was JDK 1.1-based, JDBC 2.0 support was added to 8.1.5 in an Oracle-specific package—`oracle.jdbc2`. However, the current version of Oracle8i supports Java2, so JDBC 2.0 exists in its intended package—`java.sql`. If you have JDBC programs that used the `oracle.jdbc2` package for JDBC 2.0 APIs, you must modify these programs before executing with JDBC 2.0 drivers.

Note: The assumption is that the Java application in the JDK 1.1 environment uses the `oracle.jdbc2` APIs. If it does not, no migration is necessary to connect to the 8.1.6 database.

With the addition of Java 2 in Oracle8i, the JDBC 2.0 support exists in the `java.sql` package, which is contained in the JDK core libraries. You can continue to use the `oracle.jdbc2` package, and not needing to change your client code, by continuing to use `oracle.jdbc2` within `classes111.zip`.

The following client applications can interoperate with 8.1.6:

- Java application that conforms to JDK 1.1 and uses the JDBC 2.0 APIs contained in `oracle.jdbc2`. This application imports the `oracle.jdbc2` package within `classes111.zip`.
- Java application that conforms to Java 2 and uses the JDBC 2.0 APIs contained in `java.sql`. This application imports the `java.sql` package contained within the `classes12.zip`, not the `oracle.jdbc2` package.

The steps to port your Java application to Java 2 are as follows:

1. Replace all imports and other mentions of `oracle.jdbc2` with `java.sql` in your programs.

The `oracle.jdbc2` package contains the JDBC 2.0 implementation that Oracle implemented in the JDK 1.1 drivers. Because those classes and interfaces are available in Java 2, `oracle.jdbc2` is not included in `classes12.zip`.

2. The return type of the `getTypeMap()` method of `Connection` has been changed from `java.util.Dictionary` to `java.util.Map`. Modify your application accordingly. No change is necessary if you are using `java.util.Hashtable`, because `Hashtable` implements `java.util.Map`.
3. Replace `classes111.zip` with `classes12.zip` in your makefile.
4. Recompile and relink your executable.

For more information, See Chapter 4 in the *Oracle8i JDBC Developer's Guide and Reference*.

Java 2 Security

Java 2 security is implemented in 8.1.6. The JDK 1.1 security sandbox is no longer applicable within Oracle8i. To use the Java 2 security permissions without modifying your code, you can manage these permissions through the PL/SQL package—`DBMS_JAVA`. To execute any of the Java 2 security methods, such as `doPrivileged`, you must have a Java 2 environment when compiling and running the application.

See "[Security](#)" on page 5-2 for more information on Java 2 security.

Java 2 ORB APIs

Oracle8i JServer updated its ORB implementation to Visigenic 3.4. This version is compatible with both JDK 1.1 and Java 2.

Note: All existing CORBA applications must regenerate their stubs and skeletons to work with 8.1.6. You must use the 8.1.6 tools when regenerating your application.

Sun Microsystems's Java 2 contains an ORB implementation; JDK 1.1 did not. Thus, when you imported the Visigenic libraries and invoked the CORBA methods, it always invoked the Visigenic implementation. With Java 2, if you invoke the CORBA methods without any modifications—as discussed below—you will invoke Sun Microsystems's CORBA implementation, which can cause unexpected results.

The following lists the three methods for accessing CORBA server objects in Oracle8i from your client and the recommendations for bypassing Sun Microsystems's CORBA implementation:

- **JNDI Lookup**—The setup for the lookup method is identical for both JDK 1.1 and Java 2. However, you must regenerate the stubs and skeletons.
- **Aurora ORB Interface**—The Aurora ORB provides an interface for initializing the ORB. If you do not use JNDI, your client initializes an ORB on its node to communicate with the ORB in the database. You can use an Aurora ORB on your client through this class.
- **CORBA ORB Interface**—If you want to use OMG's CORBA ORB interface, you must set a few properties to ensure you are accessing the correct implementation. If you do not wish to use the Aurora ORB on your client, you can use the pure CORBA interfaces. However, you need to set up your environment to direct your calls to the correct implementation.

JNDI Lookup

If you are using JNDI on the client to access CORBA objects that reside in the server, no code changes are required. However, you must regenerate your CORBA stubs and skeletons.

Aurora ORB Interface

If your client environment uses JDK 1.1, you do not need to change your existing code. You will need to regenerate your stubs and skeletons.

If your client environment has been upgraded to Java 2, you can initialize the ORB through the `oracle.aurora.jndi.orb_dep.Orb.init` method. This method guarantees that when you initialize the ORB, it will initialize only a single ORB instance. That is, if you use the Java 2 ORB interface, it returns you a new ORB instance each time you invoke the `init` method. Aurora's `init` method initializes a singleton ORB instance. Each successive call to `init` returns an object reference to the existing ORB instance.

In addition, the Aurora ORB interface manages the session-based IIOP connection.

oracle.aurora.jndi.orb_dep.Orb Class There are several `init` methods, each with a different parameter list. The following describes the syntax and parameters for each `init` method.

Note: The returned class for each `init` method are different. You can safely cast the `org.omg.CORBA.ORB` class to `com.visigenic.vbroker.orb.ORB`.

```
public com.visigenic.vbroker.orb.ORB init();
public org.omg.CORBA.ORB init(Properties props);
public org.omg.CORBA.ORB init(String[] args, Properties props);
```

Parameter	Description
Properties props	ORB system properties.
String[] args	Arguments that are passed to the ORB instance.

Example 2-1 Using Aurora ORB init method

The following example shows a client instantiating an ORB using the Aurora Orb class.

```
// Create the client object and publish it to the orb in the client
// Substitute Aurora's Orb.init for OMG ORB.init call
// old way: org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init ();
com.visigenic.vbroker.orb.ORB orb = oracle.aurora.jndi.orb_dep.Orb.init();
```

Joining a Session If your client receives a reference to an object that is created in a session, it can invoke a method on that object within the session. However, since all clients must authenticate, you must provide a username and password to the database. If the server requires client-side authentication in the form of `SSL_CREDENTIALS`, you can provide the client's username, password, and role, which is passed on the connect handshake within the `ORB.init` method.

```
public org.omg.CORBA.ORB init(String un, String pw, String role,
                             boolean ssl, java.util.Properties props);
```

Parameter	Description
String un	The username for client-side authentication.
String pw	The password for client-side authentication.
String role	The role to use after logging on.

Parameter	Description
Boolean ssl	If true, SSL is enabled for the connection. If false, a NON-SSL connection is used.
Properties props	ORB system properties.

CORBA ORB Interface

If you have implemented a pure CORBA client—that is, you do not use JNDI—you need to set the following properties before the ORB initialization call. These properties direct the call to the Aurora implementation, rather than the Java 2 implementation. This ensures the behavior that you expect. The behavior expected from the Visigenic ORB is as follows:

- Even if you invoke `ORB.init` more than once, only a single ORB instance is created. If you do not set these properties, be aware that each invocation of `ORB.init` will create a new ORB instance.
- The session IIOP connection is managed correctly.
- Callbacks from the server are managed correctly.

Note: The Aurora CORBA implementation is based upon Visibroker 3.4.

Property	Assign Value
<code>org.omg.CORBA.ORBClass</code>	<code>com.visigenic.vbroker.orb.ORB</code>
<code>org.omg.CORBA.ORBSingletonClass</code>	<code>com.visigenic.vbroker.orb.ORB</code>

Example 2–2 Assigning Visigenic values to OMG properties

The following example shows how to set up the OMG properties for directing the OMG CORBA `init` method to the Visigenic implementation.

```
System.getProperties().put("org.omg.CORBA.ORBClass",  
                           "com.visigenic.vbroker.orb.ORB");  
System.getProperties().put("org.omg.CORBA.ORBSingletonClass",  
                           "com.visigenic.vbroker.orb.ORB");
```

Or you can set the properties on the command line, as follows:

```
java -Dorg.omg.CORBA.ORBClass=com.visigenic.vbroker.orb.ORB
      -Dorg.omg.CORBA.ORBSingletonClass=com.visigenic.vbroker.orb.ORB
```

Backwards Compatibility for 8.1.5 CORBA and EJB Applications

The tools provided with Oracle8i, such as `publish`, have been modified to work with either a JDK 1.1 or Java 2 environment. However, any CORBA or EJB code that has been generated or loaded with the 8.1.5 version of any tool, will not succeed. Make sure that you always use the 8.1.6 version of all tools. This rule applies to your CORBA stubs and skeletons. You must regenerate all stubs and skeletons with the 8.1.6 IDL compiler.

Java Code, Binaries, and Resources Storage

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

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

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

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

Java code, binaries, and resources	In the JServer environment, source, classes, and resources reside within the Oracle8i database. Because they reside in the database, they are known as Java schema objects, where schema is a known database concept. There are three types of Java objects: source, class, and resource. There are no <code>.java</code> , <code>.class</code> , <code>.sqlj</code> , <code>.properties</code> , or <code>.ser</code> files on the server; instead these files map to source, class, and resource Java schema objects.
Locating Java classes	Instead of a <code>CLASSPATH</code> , you use a resolver to specify one or more schemas to search for source, class, and resource Java schema objects.

To summarize, we use the terms `call` and `session` during our discussions. Although these terms are not Java terms, they are server terms that apply to the Oracle8i

JServer platform. The Aurora memory manager preserves Java program state throughout your session (that is, between calls). The JServer uses the Oracle database to hold Java source, classes, and resources within a schema—Java schema objects. You can use a resolver to specify how Java, when executed in the server, locates source code, classes, and resources.

Preparing Java Class Methods for Execution

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

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

Compiling Java Classes

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

- You can compile the source explicitly on your client machine before loading it into the database through `javac`.
- You can ask `loadjava` to compile the source.
- You can force the compilation to occur dynamically at runtime.

Note: If you decide to compile through `loadjava`, you can specify compiler options. See "[Specifying Compiler Options](#)" on page 2-15 for more information.

Compiling Source through `javac`

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

Compiling Source through loadjava

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

1. The source file is loaded as a source schema object.
2. The source file is compiled.
3. If it does not already exist, a class schema object is created.
4. The compiled code is stored in the class schema object.

JServer logs all compilation errors both to `loadjava`'s logfile and the `USER_ERRORS` view. For more information on the `USER_ERRORS` view, see the *Oracle8i Reference* for a description of this table.

Compiling Source at Runtime

When you load the Java source into the database without the `-resolve` option, JServer will compile the source at runtime. The source file is loaded into a source schema object.

JServer logs all compilation errors both to `loadjava`'s logfile and the `USER_ERRORS` view. For more information on the `USER_ERRORS` view, see the *Oracle8i Reference* for a description of this table.

Specifying Compiler Options

There are two ways to specify options to the compiler.

- Specify compiler options on the `loadjava` command line. You can specify the `encoding` option through `loadjava`. The `online` option is defaulted, unless you override it within the `JAVA$OPTIONS` table. If you run `loadjava` with the `resolve` option, you can specify `encoding` on the command line.
- Specify persistent compiler options in a per-schema database table called `JAVA$OPTIONS`. Every time you compile, the compiler uses these options. However, any specified compiler options on the `loadjava` command override the options defined in this table.

You must create this table yourself if you wish to specify compiler options this way. See "[Compiler Options Specified in a Database Table](#)" on page 2-16 for instructions to create the `JAVA$OPTIONS` table.

The following sections describe your compiler options:

- [Default Compiler Options](#)

- [Compiler Options on the Command Line](#)
- [Compiler Options Specified in a Database Table](#)

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

- `encoding = latin1`
- `online = true`: See the *Oracle8i SQLJ Developer's Guide and Reference* for a description of this option, which applies only to Java sources that contain SQLJ constructs.
- `debug = true`: This option is equivalent to `javac -g`.

Compiler Options on the Command Line The following describes the `loadjava` compiler options:

- `encoding`—Identifies the source file encoding for the compiler. This option overrides any matching value in the `JAVA$OPTIONS` table. The values are identical to the `javac -encoding` option. This option is relevant only when loading a source file.
- `online`—This option is valid for SQLJ only. Setting this option to `TRUE` enables online semantics-checking. Semantics-checking is performed relative to the schema in which the source is loaded. If the `online` option is set to `FALSE`, offline checking is performed.

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

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

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

The parameters for these methods are described below:

Parameter	Description
name	The <code>name</code> parameter is a Java package name, a fully qualified class name, or the empty string. When the compiler searches the <code>JAVA\$OPTIONS</code> table for the options to use for compiling a Java source schema object, it uses the row whose <code>name</code> most closely matches the schema object's fully qualified class name. A name whose value is the empty string matches any schema object name.
option	The <code>option</code> parameter is either 'online', 'encoding' or 'debug'. For the values you can specify for these options, see the <i>Oracle8i SQLJ Developer's Guide and Reference</i> .

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

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

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

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

Table 2–1 Example `JAVA$OPTIONS` Table

JAVA\$OPTIONS Entries			Match Examples
Name	Option	Value	
a.b.c.d	online	true	<ul style="list-style-type: none"> ■ a.b.c.d—matches the pattern exactly. ■ a.b.c.d.e—first part matches the pattern exactly; no other rule matches full name.

Table 2–1 Example JAVA\$OPTIONS Table (Cont.)

JAVA\$OPTIONS Entries			Match Examples
Name	Option	Value	
a.b	online	false	<ul style="list-style-type: none"> ■ a.b—matches the pattern exactly ■ a.b.c.x—first part matches the pattern exactly; no other rule matches beyond specified rule name.
(empty string)	online	true	<ul style="list-style-type: none"> ■ a.c—no pattern match with any defined name; defaults to (empty string) rule ■ x.y—no pattern match with any defined name; defaults to (empty string) rule

Automatic Recompilation

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

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

The system tracks dependencies at a class level of granularity. In the preceding example, you can see that classes A, B, and C depend on one another, because A holds an instance of B, B holds an instance of C, and C holds an instance of A. If you change the definition of class A by adding a new field to it, the dependency mechanism in Oracle8i flags classes B and C as invalid. Before you use any of these classes again, Oracle8i attempts to resolve them again and recompile, if necessary. Note that classes can be recompiled only if source is present on the server.

The dependency system enables you to rely on Oracle8i to manage dependencies between classes, to recompile, and to resolve automatically. You must only force compilation and resolution yourself only if you are developing and you want to find problems early. The `loadjava` utility also provides the facilities for forcing compilation and resolution if you do not want to allow the dependency management facilities to perform this for you.

Resolving Class Dependencies

Many Java classes contain references to other classes, which is the essence of reusing code. A conventional Java virtual machine searches for classes, ZIP, and JAR files within the directories specified in the CLASSPATH. In contrast, the Aurora Java virtual machine searches database schemas for class objects. With Oracle8i, you load all Java classes within the database, so you might need to specify where to find the dependent classes for your Java class within the database.

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

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

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

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

To make searching for dependent classes easier, Oracle8i provides a default resolver and resolver spec that searches first the definer's schema and then PUBLIC. This covers most of the classes loaded within the database. However, if you are accessing classes within a schema other than your own or PUBLIC, you must define your own resolver spec.

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

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

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

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

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

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

Allowing References to Non-Existent Classes

You can specify a special option within a resolver spec that allows an unresolved reference to a non-existent class. Sometimes, internal classes are never used within a product. For example, some ISVs do not remove all references to internal test classes from the JAR file before shipping. In a normal Java environment, this is not a problem, because as long as the methods are not called, Sun Microsystems's JVM ignores them. However, the Oracle8i resolver tries to resolve all classes referenced within the JAR file—even unused classes. If the reference cannot be validated, the classes within the JAR file are marked as invalid.

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

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

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

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

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

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

ByteCode Verifier

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

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

The resolver also issues warnings, as defined below:

- Resolvers containing “-”

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

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

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

- Use of other resolvers

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

For more information on class resolution and loading your classes within the database, see [Appendix A, "Tools"](#).

Loading Classes

This section gives an overview of the main points you should understand when loading your classes into the database. It discusses various options for the `loadjava` tool, but does not go into all the details. See "[loadjava](#)" on page A-7 for complete information.

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

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

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

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

Schema object	<code>loadjava</code> operations on object
<code>.java</code> source files	<ol style="list-style-type: none"> 1. It creates a source schema object within the definer's schema unless another schema is specified 2. It loads the contents of the source file into a schema object 3. It creates a class schema objects for all classes defined in the source file 4. If <code>-resolve</code> is requested, it does the following: <ol style="list-style-type: none"> a. It compiles the source schema object b. It resolves the class and its dependencies c. It stores the compiled class into a class schema object
<code>.sqlj</code> source files	<ol style="list-style-type: none"> 1. It creates a source schema object within the definer's schema unless another schema is specified 2. It loads contents of the source file into the schema object 3. It creates a class schema objects for all classes and resources defined in the source file 4. If <code>-resolve</code> is requested, it does the following: <ol style="list-style-type: none"> a. It translates and compile the source schema object b. It stores the compiled class into a class schema object c. It stores profile into <code>.ser</code> resource schema object and customizes it
<code>.class</code> compiled Java files	<ol style="list-style-type: none"> 1. It creates a class schema object within the definer's schema unless another schema is specified 2. It loads the class file into the schema object 3. It resolves and verify the class and its dependencies if <code>-resolve</code> is specified

<code>.properties</code> Java resource files	<ol style="list-style-type: none"> 1. It creates a resource schema object within the definer's schema unless another schema is specified 2. It loads resource file into a schema object
<code>.ser</code> SQLJ profile	<ol style="list-style-type: none"> 1. It creates a resource schema object within the definer's schema unless another schema is specified 2. It loads <code>.ser</code> resource file into a schema object and customizes it

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

Note: More options for `loadjava` are available. However, this section discusses only the major options. See "[loadjava](#)" on page A-7 for complete information. Also, [dropjava](#) is discussed in full on page A-15.

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

- [Two Definitions of the Same Class](#)
- [Need Database Privileges and JVM Permissions](#)
- [Loading JAR or ZIP Files](#)

After loading, you can access the `USER_OBJECTS` view in your database schema to verify that your classes and resources loaded properly. For more information, see "[Checking Java Uploads](#)" on page 2-27.

Two Definitions of the Same Class

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. JServer tracks whether you loaded a class file or a source file. If you wish to update the class, you must load the same type of file that you originally loaded.

If you wish to update the other type, you must drop the first before loading the second. For example, if you loaded `x.java` as the source for class `y`, to load `x.class`, you must first drop `x.java`.

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

Need Database Privileges and JVM Permissions

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

- `CREATE PROCEDURE` and `CREATE TABLE` privileges to load into your schema.
- `CREATE ANY PROCEDURE` and `CREATE ANY TABLE` privileges to load into another schema.
- `oracle.aurora.security.JServerPermission.loadLibraryInClass.<classname>`. See "[Permission for Loading Classes](#)" on page 5-25 for more information.

Loading JAR or ZIP Files

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

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

How to Grant Execute Rights

When you are loading your classes, you can grant execution rights to another user through an option for `loadjava`. There are two methods for defining who can execute your class:

- granting execution rights to a certain user or schema

The classes that define a Java application are stored within the Oracle8i RDBMS under the SQL schema of their owner. By default, classes that reside in one user's schema are not executable by other users, because of security concerns. You can allow other users (schemas) the right to execute your class through the `loadjava -grant` option.

- specifying what user or schema is evaluated for the execution rights to the class

You can specify that the user evaluated for execution rights is either the invoker or the definer. Invoker's and definer's rights are SQL concepts that are applicable to your Java classes. In SQL, when you execute under definer's rights, you execute with the user identifier for the schema that loaded the PL/SQL. With invoker's rights, your SQL executes with the user identifier for the schema that actually invokes the PL/SQL. This applies to the Java classes in that the schema that is evaluated for JVM security permission to execute is based on whether the class was loaded under invoker's or definer's rights.

Figure 2–2 Invoker’s Versus Definer’s Rights

Method invocation: Class A invokes class B; class B invokes class C.

Execution rights for classes:

- * Class A has execution rights for B.
- * Class A does not have execution rights for C.
- * Class B has execution rights for C.

With the example in [Figure 2–2](#), which class, A or B, is checked when the method in class C is executed? This depends on invoker’s or definer’s rights.

- * **Invoker’s rights**—By default, classes run under the effective identity and rights of the client’s schema that initially invokes the method in C. In this case, the rights for class A is checked. Because class A does not have permission to execute C, the request is rejected. This is the default.
- * **Definer’s rights**—Classes execute under the effective identity of the user that loaded the class. If, when you loaded class B, you specified definer’s rights, class B is checked when the method in class C is invoked. Because class B has the right to execute methods in class C, the method completes successfully. Class B is loaded with definer’s rights by executing `loadjava -definer`.

For information on JVM security permissions, see [Chapter 5, "Security and Performance"](#).

Checking Java Uploads

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

Columns in `USER_OBJECTS` include those contained in [Table 2–2](#) below.

Table 2–2 Key `USER_OBJECT` Columns

Name	Description
<code>OBJECT_NAME</code>	name of the object

Table 2-2 Key USER_OBJECT Columns (Cont.)

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

Object Name and Type

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

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

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

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

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

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

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

```
SVRMGR> select * from javasm;
SHORT                                LONGNAME
-----
/78e6d350_BinaryExceptionHandler  sun/tools/java/BinaryExceptionHandler
/b6c774bb_ClassDeclaration         sun/tools/java/ClassDeclaration
/af5a8ef3_JarVerifierStream1       sun/tools/jar/JarVerifierStream$1
```

Status

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

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

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

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

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

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

For more information about USER_OBJECTS, see the *Oracle8i Java Stored Procedures Developer's Guide*.

Publishing

Oracle8i enables clients and SQL to invoke Java methods loaded within the database, once published. You publish either the object itself or individual methods, depending on the type of Java application it is, as shown below:

Java API	Publishing method	Reference
Java stored procedures	If you write a Java stored procedure that you intend to invoke with a trigger, directly or indirectly in SQL DML or in PL/SQL, you must publish individual methods within the class. You specify how to access it through a call specification. Java programs consist of many methods in many classes; however, only a few static methods are typically exposed with call specifications.	<i>Oracle8i Java Stored Procedures Developer's Guide.</i>
CORBA and EJB development	<p>You do not use call specifications for CORBA or EJB objects. Instead, you publish the object reference for the client to retrieve. Once the object is retrieved, the client can invoke specific methods within the object.</p> <p>Oracle8i's CORBA and EJB implementations support standard CORBA and Java styles of exposing objects by name, with accompanying CORBA and Java-style specifications of the interfaces to those objects.</p> <ul style="list-style-type: none">■ You publish CORBA IOR's through the <code>publish</code> tool.■ You publish EJB Home and Remote interfaces through the <code>deployejb</code> tool.	<i>Oracle8i Enterprise JavaBeans and CORBA Developer's Guide</i>

User Interfaces on the Server

Oracle8i furnishes all core Java class libraries on the server, including those associated with presentation of user interfaces (`java.awt` and `java.applet`). It is, however, inappropriate for code executing in the server to attempt to bring up or materialize a user interface in the server. Imagine thousands of users worldwide exercising an Internet application that executes code that requires someone to click on a dialog presented on the server hardware. You can write Java programs that reference and use `java.awt` classes as long as you do not attempt to materialize a user interface.

When building applets, you test them using the `java.awt` and the Peer implementation, which is a platform-specific set of classes for support of a specific windowing system. When the user downloads an applet, it dynamically loads the proper client Peer libraries, and the user sees a display appropriate for the operating system or windowing system in use on the client side. Oracle8i takes the same approach. We provide an Oracle-specific Peer implementation that throws an exception, `oracle.aurora.awt.UnsupportedOperation`, if you execute Java code on the Oracle8i server that attempts to materialize a user interface.

Oracle8i's lack of support for materializing user interfaces in the server means that we do not pass the Java 2 Compatibility Kit tests for `java.awt`,

`java.awt.manual`, and `java.applet`. In the Oracle RDBMS, all user interfaces are supported only on client applications, although they might be displayed on the same physical hardware that supports the server—for example, in the case of Windows NT. Because it does not make sense for the server to support user interfaces, we exclude these tests from our complete Java Compatibility Kit testing.

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

Shortened Class Names

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

Schema object names, however, have a maximum of only 31 characters, and all characters must be legal and convertible to characters in the database character set. If any full name is longer than 31 characters or contains illegal or non-convertible characters, the Oracle8i server converts the full name to a *short name* to employ as the name of the schema object, keeping track of both names and how to convert between them. If the full name is 31 characters or less and has no illegal or inconvertible characters, then the full name is used as the schema object name.

Because Java classes and methods can have names exceeding the maximum SQL identifier length, Oracle8i uses abbreviated names internally for SQL access. Oracle8i provides a method within the `DBMS_JAVA` package for retrieving the original Java class name for any truncated name.

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

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

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

In addition, you can specify a full name to the database by using the `shortname()` routine of the `DBMS_JAVA` package, which takes a full name as input and returns

the corresponding short name. This is useful when verifying that your classes loaded by querying the `USER_OBJECTS` view.

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

Refer to the *Oracle8i Java Stored Procedures Developer's Guide*, for a detailed example of the use of this function and ways to determine which Java schema objects are present on the server.

Class.forName() on JServer

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

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

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

- The JDK uses one `ClassLoader` that searches the set of directory tree roots specified by the environment variable `CLASSPATH`.
- JServer defines several resolvers, which define how to locate classes. Every class has a resolver associated with it, and each class can, potentially, have a different resolver. When you execute a method that calls `Class.forName()`, the resolver of the currently executing class (`this`) is used to locate the class. See "[Resolving Class Dependencies](#)" on page 2-19 for more information on resolvers.

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

You can solve this resolver problem as follows:

- Avoid any class name lookup by passing the `Class` object itself.
- Supply the `ClassLoader` in the `Class.forName` method.

- Supply the class and the schema it resides in to JServer's `classForNameAndSchema` method.
- Supply the schema and class name to `Class.forName().lookupClass`.
- Serialize your objects with the schema name with the class name.

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

Supply the ClassLoader in Class.forName

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

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

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

Example 2-3 Retrieve Resolver from Current Class

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

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

Example 2-4 Retrieve Resolver from Calling Class

You can retrieve the class of the instance that invoked the executing method through the `oracle.aurora.vm.OracleRuntime.getCallerClass` method. Once you retrieve the class, invoke the `Class.getClassLoader` method on the returned class. The following example retrieves the class of the instance that invoked the `workForCaller` method. Then, its class loader is retrieved and

supplied to the `Class.forName` method. Thus, the resolver used for looking up the class is the resolver of the calling class.

```
void workForCaller() {
    ClassLoader c1 =
        oracle.aurora.vm.OracleRuntime.getCallerClass().getClassLoader();
    ...
    Class c = Class.forName (name, true, c1);
}
```

Supply Class and Schema Names to `classForNameAndSchema`

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

Example 2-5 Providing Schema and Class Names

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

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

void save (Class c1) {
    ClassHandle handle = ClassHandle.lookup(c1);
    Schema schema = handle.schema();
    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 single String, containing both the schema and class names, to the `oracle.aurora.util.Class.forName.lookupClass` method. When invoked, this method locates the class in the specified schema. The string must be in the following format:

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

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

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

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

Supply Class and Schema Names when Serializing

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

JServer 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`. This class can be used on any environment. If used within JServer, the schema names are read out and used when performing the class lookup. If used on a client, the schema names are ignored.

Class.forName Example

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

- To use the resolver of this instance's class, invoke `lookupWithClassLoader`. This method supplies a class loader to the `Class.forName` method in the `from` variable. The class loader specified in the `from` variable defaults to this class.

- To use the resolver from a specific class, call `ForName` with the designated class name followed by `lookupWithClassLoader`. The `ForName` method sets the `from` variable to the specified class. The `lookupWithClassLoader` method uses the class loader from the specified class.
- To use the resolver from the calling class, invoke the `ForName` method without any parameters. It sets the `from` variable to the calling class. Then, invoke the `lookupWithClassLoader` to locate the class using the resolver of the calling class.
- To lookup a class in a specified schema, invoke the `lookupWithSchema` method. This provides the class and schema name to the `classForNameAndSchema` method.

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

public class ForName {

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

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

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

Managing Your Operating System Resources

Operating system resources are a limited commodity on any computer. Because Java is targeted at providing a computing platform as well as a programming language, it contains platform-independent classes and frameworks for accessing platform-specific resources. The Java class methods access operating system resources through the JVM. Java has potential problems with this model, because programmers rely on the garbage collector to manage all resources, when all that the garbage collector manages is Java objects, not the operating system resources that the Java object holds on to.

In addition, because the Aurora JVM is embedded in the database, your operating system resources, which are contained within Java objects, can be invalidated if they are maintained across calls within a session.

The following sections discuss these potential problems:

- [Overview of Operating System Resources](#)
- [Garbage Collection and Operating System Resources](#)
- [Operating System Resources Affected Across Calls](#)

Overview of Operating System Resources

In general, your operating system resources contain the following:

- | | |
|---------|---|
| memory | Aurora manages memory internally, allocating memory as you create new objects and freeing objects as you no longer need them. The language and class libraries do not support a direct means to allocate and free memory. " Automated Storage Management " on page 1-14 discusses garbage collection. |
| files | Java contains classes that represent file resources. Instances of these classes hold on to your operating system's file constructs, such as file handles, which can become invalid between calls in a session. |
| sockets | Java contains classes that represent socket resources. Instances of these classes hold on to socket constructs, some of which can become invalid between calls in a session. See " Sockets " on page 2-42 for information specific to maintaining sockets across calls. |

threads Threads are discouraged within the Aurora JVM because of scalability issues. However, you can have a multi-threaded application within the database. "[Threading in JServer](#)" on page 2-43 discusses in detail the Aurora's JVM threading model.

Operating System Resource Access

By default, a Java user does not have direct access to most operating system resources. A system administrator may give permission to a user to access these resources by modifying the JVM security restrictions. The JVM security enforced upon system resources conforms to Java 2 security. See "[Security](#)" on page 5-2 for more information.

Operating System Resource Lifetime

You access operating system resources using the standard core Java classes and methods. Once you access a resource, the time that it remains active (usable) varies according to the type of resource.

Resource	Lifetime
Files	The system closes all files left open when a database call ends.
Memory	Memory is garbage collected as described in " Automated Storage Management " on page 1-14.
Threads	All threads are terminated when a call ends.
Objects that depend on operating system resources	<p>Regardless of the usable lifetime of the object (for example, the defined lifetime for a thread object), the Java object can be valid for the duration of the session. This can occur, for example, if the Java object is stored in a static class variable or a class variable references it directly or indirectly. If you attempt to use one of these Java objects after its usable lifetime is over, Aurora throws an exception. This is true for the following examples:</p> <ul style="list-style-type: none"> ■ If an attempt is made to read from a <code>java.io.FileInputStream</code> that was closed at the end of a previous call, a <code>java.io.IOException</code> is thrown. ■ <code>java.lang.Thread.isAlive()</code> is false for any Thread object running in a previous call and still accessible in a subsequent call.
Sockets	<ul style="list-style-type: none"> ■ Sockets can exist across calls. ■ ServerSockets on an MTS server terminate when the call ends. ■ ServerSockets on a dedicated server can exist across calls. <p>See "Sockets" on page 2-42 more information.</p>

Garbage Collection and Operating System Resources

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

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

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

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

[Example 2-6](#) shows how you should close the operating system constructs.

Example 2-6 *Closing your operating system resources*

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

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

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

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

Operating System Resources Affected Across Calls

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.

The JVM automatically closes any open operating system constructs—in [Example 2-7](#), the file handle—when the call ends. This can affect any operating system resources within your Java object. For example, if you have a file opened within a static variable, the file handle is closed at the end of the call for you. So, if you hold on to the `File` object across calls, the next usage of the file handle throws an exception.

In [Example 2-7](#), class `Concat` enables multiple files to be written into a single file, `outFile`. On the first call, `outFile` is created. The first input file is opened, read, input into `outFile`, and the call ends. Because `outFile` is statically defined, it is moved into session space between call invocations. However, the file handle—that is, the `FileDescriptor`—is closed at the end of the call. The next time you call `addFile`, you will get an exception.

Example 2-7 *Compromising your operating system resources*

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

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

```

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

```

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

[Example 2-8](#) shows how you can perform concatenation, as in [Example 2-7](#), without compromising your operating system resources. The `addFile` method opens the `outme.txt` file within each call, making sure that anything written into the file is appended to the end. At the end of each call, the file is closed. Two things occur:

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

Example 2-8 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: Net8's TTC or IIOP. See "[Configuring JServer](#)" on page 4-6 for information on how to configure your connection.

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

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

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

`Socket` Because the client-side of the connection is outbound, the `Socket` instance can be serviced across calls within either an MTS or dedicated server.

`ServerSocket` The server-side of the connection is a listener.

- **Dedicated server**—Your `ServerSocket` can listen across calls only within a dedicated server; the dedicated server exists solely for servicing the single client.
- **MTS server**—The `ServerSocket` is closed at the end of a call within an MTS server; the MTS uses shared servers, which move on to another client at the end of every call. You will receive an I/O exception stating that the socket was closed if you try to use the `ServerSocket` outside of the call it was created in.

Threading in JServer

The Aurora JVM implements a non-preemptive threading model. With this model, the JVM runs all Java threads on a single operating system thread. It schedules them in a round-robin fashion and switches between them only when they block.

Blocking occurs when you, for example, invoke the `Thread.yield()` method or wait on a network socket by invoking `mySocket.read()`.

Advantages of JServer's Threading Model	Disadvantages
<ul style="list-style-type: none"> ■ simple to program ■ efficient to implement in the Java virtual machine, because a thread switch does not require any system calls ■ safer, because the JVM can detect a deadlock that would hang a preemptive JVM and can then raise a runtime exception 	<ul style="list-style-type: none"> ■ does not exhibit any concurrency ■ lack of portability ■ performance considerations, because of the system calls required for locking when blocking the thread ■ memory scalability, because efficient multi-threaded memory allocation requires a larger pool of memory

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

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

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

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

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

Thread Lifecycle

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

1. The thread returns to its caller.
2. An exception is thrown and is not caught in Java code.
3. The `System.exit()`, `oracle.aurora.vm.OracleRuntime.exitCall()`, or `oracle.aurora.vm.OracleRuntime.exitSession()` method is invoked.

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

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

When a call ends because of a return and/or uncaught exceptions, Aurora throws a `ThreadDeathException` in all daemon threads. The `ThreadDeathException` essentially forces threads to stop execution.

When a call ends because of a call to `System.exit()`, `oracle.aurora.vm.OracleRuntime.exitCall()`, or

`oracle.aurora.vm.oracleRuntime.exitSession()`, Aurora ends the call abruptly and terminates all threads, but does not throw `ThreadDeathException`.

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

Invoking Java in the Database

We reviewed the basics of writing and deploying Java applications on Oracle8i in [Chapter 2, "Writing Java Applications on Oracle8i"](#). This chapter gives you an overview and examples for how to invoke Java within the database.

- [Overview](#)
- [Invoking Java Methods](#)
- [Utilizing SQLJ and JDBC for Querying Database](#)
- [Debugging Server Applications](#)
- [How To Tell You Are Executing in the Server](#)
- [Redirecting Output on the Server](#)

Overview

In Oracle8i, you utilize Java in one of the following ways:

- [Invoking Java Methods](#)—Invoke Java methods in classes loaded within the database. This includes Java stored procedures, CORBA, and EJB.
- [Utilizing SQLJ and JDBC for Querying Database](#)—You can query the database from a Java client through utilizing [JDBC](#) or [SQLJ](#).

We recommend that you approach Java development in Oracle8i incrementally, building on what you learn at each step. The easiest way to invoke Java within the database is through Java stored procedures. Once you have mastered that, you should move on to CORBA and EJB applications.

1. You should master the process of writing simple Java stored procedures as listed in "[Preparing Java Class Methods for Execution](#)" on page 2-14 and the *Oracle8i Java Developer's Guide*. This includes writing the Java class, deciding on a resolver, loading the class into the database, and publishing the class.
2. You should understand how to access and manipulate SQL data from Java. Most Java server programs, and certainly Java programs executing on Oracle8i, interact with database-resident data. The two standard APIs for accomplishing this are JDBC and SQLJ. Because JDBC forms the foundation for SQLJ, you should understand how the two work together, even though you might be using only SQLJ in your code.
3. If you intend to distribute Java logic between client and server or in an N-tier architecture, you should understand how CORBA and EJB work in Oracle8i. CORBA and EJB provide the simplest solution to this difficult problem, in an Internet-standard manner, enabling you to leverage component-based development for transactional applications. Furthermore, EJB and CORBA utilize Oracle8i's facilities for Java stored procedures and JDBC.

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

As you write more complex Java programs, you can gain performance and scalability by controlling the location where the program logic executes. You can minimize network traffic and maximize locality of reference to SQL data. JDBC and

SQLJ furnish ways to accomplish these goals. However, as you tend to leverage the object model in your Java application, a more significant portion of time is spent in Java execution, as opposed to SQL data access and manipulation. It becomes more important to understand and specify where Java objects reside and execute in an Internet application. Now you have become a candidate for moving into the world of CORBA and Enterprise JavaBeans.

Invoking Java Methods

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

- [Utilizing Java Stored Procedures](#)
- [Utilizing Distributed Objects With CORBA and EJB](#)
- [Utilizing Remote Method Invocation \(RMI\)](#)
- [Utilizing Java Native Interface \(JNI\) Support](#)
- [Utilizing SQLJ and JDBC for Querying Database](#)

Utilizing Java Stored Procedures

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

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

1. Write the Java class.

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

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

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

```
javac Hello.java
```

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

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

- as an individual file in a directory, where the directory is specified in the `CLASSPATH`
 - within a `.jar` or `.zip` file, where the directory is specified in the `CLASSPATH`
3. Decide on the resolver for your class.

In this case, you load `Hello.class` in the server, where it is stored in the database as a Java schema object. When you execute the `world()` method of the `Hello.class` on the server, it finds the necessary supporting classes, such as `String`, using a resolver—in this case, the default resolver. The default resolver looks for classes in the current schema first and then in `PUBLIC`. All core class libraries, including the `java.lang` package, are found in `PUBLIC`. You may need to specify different resolvers, and you can force resolution to occur when you use `loadjava`, to determine if there are any problems earlier, rather than at runtime. Refer to "[Resolving Class Dependencies](#)" on page 2-19 or "[loadjava](#)" on page A-7 for more details on resolvers and `loadjava`.

4. Load the class on the Oracle8i server using `loadjava`. You must specify the username and password.

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

5. Publish the stored procedure through a call specification

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

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

```
connect scott/tiger
create or replace function HELLOWORLD return VARCHAR2 as
  language java name 'Hello.world () return java.lang.String';
```

6. Invoke the stored procedure

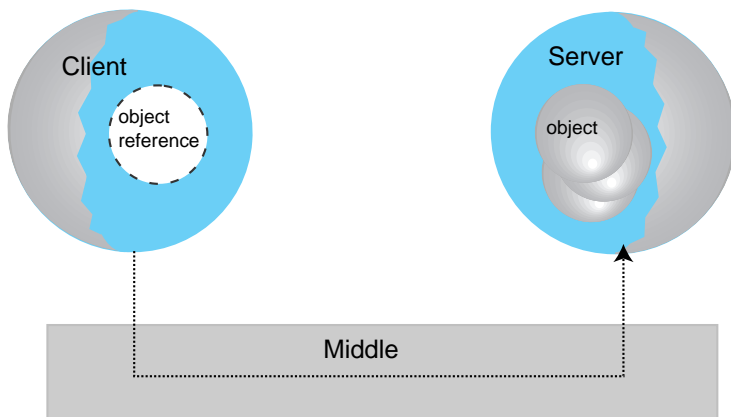
```
variable myString varchar2[20];
call HELLOWORLD() into :myString;
print myString;
```

The call `HELLOWORLD() into :myString` statement performs a top-level call in Oracle8i. The Oracle-specific `select HELLOWORLD from DUAL` also works. Note that SQL and PL/SQL see no difference between a stored procedure written in Java, PL/SQL, or any other language. The call specification provides a means to tie inter-language calls together in a consistent manner. Call specifications are necessary only for entry points invoked with triggers or SQL and PL/SQL calls. Furthermore, JDeveloper can automate the task of writing call specifications.

For more information on Java stored procedures, using Java in triggers, call specifications, rights models, and inter-language calls, refer to the *Oracle8i Java Stored Procedures Developer's Guide*.

Utilizing Distributed Objects With CORBA and EJB

In a program whose logic is distributed, the architecture of choice has three tiers—the client, the middle tier, and the database server.



- | | |
|-------------|--|
| Client tier | Typically limited to display of information provided by the middle tier. |
| Middle tier | Facilitates the communication between client and server. Typically manages the server objects. Marshals and unmarshals the parameters and return values. |
| Server tier | Performs the business or application logic. |

The server object within the three-tier model does the business logic. This may or may not include accessing a database for SQL queries. Oracle8i removes the need for a physical middle tier for distributed applications where the server object requires access to a database. Oracle8i still maintains a three-tier logical architecture, but by combining the middle tier and the database server, the physical architecture is two-tier. The flexibility inherent in this architecture is ideally suited to Internet applications where the client presents information in a Web browser, interacting with servers across the network. Those servers, in turn, can be federated and cooperate in their own client-server interactions to provide information to Web-based clients in an intranet or Internet application.

To use the two-tier distributed object approach for your application, you can use either the CORBA or EJB APIs.

- CORBA uses Interface Definition Language (IDL) to specify, in a language-independent manner, how to access and use a group of objects known

as a component. Oracle8i interacts with each client as if it had its own Java virtual machine running in the server. There is no single ORB in the JServer servicing multiple client requests. Instead, JServer leverages off of Oracle8i's Multithreaded Server (MTS) architecture, providing an ORB per session.

- Enterprise JavaBeans relies on the following:
 - * Java class definitions specify the interface to a component.
 - * RMI-style declarative deployment descriptors define how the component is treated in a transactional, secure application.

An EJB programmer writes business logic and the interfaces to the component; a deployment tool, `deployejb`, loads and publishes the component. No knowledge of IDL is necessary. This portable Java-based server framework provides a fast, scalable, and easy solution to Java-based, three-tier applications.

CORBA and EJB are complementary. The JServer implementation of the Enterprise JavaBeans 1.0 specification builds on the underlying support and services of CORBA.

IIOP Transport

Unlike a session in which the client communicates through Net8, you access CORBA and EJB sessions through IIOP, which is capable of servicing multiple client connections. Although scalable applications generally provide one session per client-server interaction, the ability to service multiple clients extends the flexibility of the session. IIOP enables callouts, callbacks, and loopbacks in your distributed communications.

Naming

You can access components through a name service, which forms a tree, similar to a file system, where you can store objects by name. When you put a CORBA or EJB object into the namespace, you are publishing it.

There are two supported naming protocols within Oracle8i:

- Java Naming and Directory Interface (JNDI)—This package provides a unified interface to name services. Part of JNDI provides a platform-independent abstraction for accessing a file system, which is platform dependent.
- CORBA's CosNaming—If you are a CORBA programmer, you are familiar with bootstrapping your application using CosNaming. Your client Java code obtains handles to objects that reside on the server. Those objects are reachable through the name service. The ORB supplies the name service, which presupposes that

the ORB is running when you attempt to locate the server objects when bootstrapping your client application. JServer provides an activation service based on CORBA's CosNaming. You use a URL-based name within JNDI as an interface to CosNaming when referring and activating CORBA objects in a session. This namespace incorporates the idea of a session directly in the URL, allowing the client to easily manipulate multiple sessions. All bootstrapping is performed by establishing a session with the JServer and using objects always reachable from the Oracle8i database that the standard JNDI and CORBA CosNaming make visible to you. You do not use Inter-ORB References (IORs) as with most CORBA applications.

Creating and Deploying Enterprise JavaBeans

CORBA and EJB application development are complicated topics covered in the *Oracle8i Enterprise JavaBeans and CORBA Developer's Guide*. This section gives an example of how to create an EJB component. This example creates an `EmployeeBean` to look up an employee record in the Oracle RDBMS.

1. Create the home interface. The home interface will reside in the server, enabling you to create instances of your EJB on the server. A home interface is a Java interface that extends `EJBHome`. The home interface is the only object published in the namespace to be visible to clients. The client can access the component through JNDI.
2. Create the remote interface. The remote interface specifies the methods you implement in the EJB, such as instance methods you can invoke from a client. A remote interface is a Java interface that extends `EJBObject`. As an interface, you use it to specify the methods implemented in the bean.
3. Implement the bean class and the methods that the remote interface defines. In the `EmployeeBean` example, the only method is `getEmployee()`. You write a bean by creating a class that implements the `SessionBean` interface.
4. Create the deployment descriptor. The deployment descriptor specifies attributes of the bean, including its transactional properties and security treatment. You specify the attributes, and the `deployejb` tool ensures that the server enforces them.
5. Deploy the EJB. When you deploy the EJB, the `deployejb` tool does the following:
 - Places your home interface and the EJB on the server.
 - Publishes the home interface in the namespace.

- Generates the Java code on the server side to manage transactions and security specified in the deployment descriptor.
- Generates and returns the stub interfaces that provide the client access to the remote functionality of the bean.

Using an EJB

Once you create and deploy an EJB, you will want to use it from a client program. You can use EJBs between servers in n-tier applications also, in which case the client for one server can also be the server for other clients. In your client code, you must perform the following steps:

1. Locate the home interface object that resides on the server. You will locate the object using Java-standard JNDI lookup facilities.
2. Authenticate the client to the server. EJB and CORBA clients use database sessions, just as with any other Oracle client. To initiate a session, you must let the server know you are a valid user. You can use several different approaches to accomplish authentication in a secure manner.
3. Activate an instance of the bean. Because the object you locate with JNDI is the home interface, you will use one of its `create()` methods to return an activated instance of the EJB.
4. Invoke methods on the bean. When you invoke a method on the bean, the method is actually executed in the server, and the appropriate parameter and return objects are transparently transported (by copy) across the underlying IIOP connection. All objects the EJBs return must be serializable—they must implement `java.io.Serializable`.

The *Oracle8i Enterprise JavaBeans and CORBA Developer's Guide* discusses the details of these steps. Java IDEs, as with Oracle's JDeveloper, can automate and simplify the deployment and descriptor process.

Session Shell

Session shell is an example of a tool written completely in Java using Java stored procedures and CORBA. It interacts with server-resident objects that are visible through CORBA within your session by using UNIX shell commands. For more information, see the *Oracle8i Enterprise JavaBeans and CORBA Developer's Guide*. This tool demonstrates how you can use CORBA to build tools that make life simpler for developers and end users.

The session shell provides a shell-like interface to the server. This shell allows users to manipulate the session namespace with familiar UNIX commands, such as `mkdir`, `ls`, and `rm`. In addition, the session shell furnishes a convenient way to run Java programs in the server, using the `java` command. The session shell `java` command takes the name of a class and any arguments the user types in. The session shell calls the static `main(String[])` method on the class, running the Java program in the server. `System.out` and `System.err` are captured and transparently redirected back to the user's console.

Utilizing Remote Method Invocation (RMI)

JServer fully supports Java Remote Method Invocation (RMI). All RMI classes and `java.net` support are in place. In general, RMI is not useful or scalable in JServer applications. CORBA and EJB are the preferred APIs for invoking methods of remote objects. The RMI Server that Sun Microsystems supplies does function on the JServer platform. Because Sun Microsystems's RMI Server uses operating system sockets and is not accessible through a presentation, it is useful only within the context of a single call. It relies heavily on Java language level threads. By contrast, the Oracle8i ORB and EJB rely on the database server to gain scalability. You can efficiently implement an RMI server as a presentation; however, CORBA and EJB currently serves this purpose.

Note: A presentation is an object that accepts either a Net8 or IIOP incoming connection into the database. See "[Configuring JServer](#)" on page 4-6 for more information.

Utilizing Java Native Interface (JNI) Support

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

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

Utilizing SQLJ and JDBC for Querying Database

You can use one of two protocols for querying the database from a Java client. Both protocols establish a session with a given username/password to the database and execute SQL queries against the database.

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 is typically a one-liner that executes against a known table with known column names.

JDBC

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

1. The JDBC Thin driver, a 100% pure Java solution you can use for either client-side applications or applets and requires no Oracle client installation.
2. The JDBC OCI drivers (OCI 8 or OCI 7), which you use for client-side applications and requires an Oracle client installation.
3. The server-side JDBC driver embedded in the Oracle8i server.

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

SQLJ

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

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

SQLJ also allows you to catch errors in your SQL statements before runtime. JDBC code, being pure Java, is compiled directly. The compiler has no knowledge of SQL, so it is unaware of any SQL errors. By contrast, 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.

An Example Comparing JDBC and SQLJ

The following is an example of a simple operation, first in JDBC code and then SQLJ code.

JDBC:

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

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

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

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

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

You can use a prepared statement whenever values within the SQL statement must be dynamically set (you can use the same prepared statement repeatedly with different variable values). The question marks in the prepared statement are placeholders for Java variables and are given values in the

`pstmt.setInt()` and `pstmt.setFloat()` lines of code. The first “?” is set to the `int` variable `id` (with a value of 37115). The second “?” is set to the `float` variable `salary` (with a value of 20000).

3. Execute the query and return the data into a JDBC result set object. (You can use result sets to gather query data.)
4. Retrieve the data of interest (the name) from the result set and print it. A result set usually contains multiple rows of data, although this example has only one row.

By comparison, here is some SQLJ code that performs the same task. Note that all SQLJ statements, both declarations and executable statements, start with the `#sql` token.

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);
```

SQLJ, in addition to allowing SQL statements to be directly embedded in Java code, 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, but 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.

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:@ool1:5521:so12", "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());
        }
    }
}
```

1. Declare your iterators. SQLJ uses a strongly typed version of JDBC result sets, known as iterators. The main difference between the two is that an iterator has a specific number of columns of specific datatypes. You must define your iterator types beforehand, 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`, whose second column maps to a Java `int`, and whose third column maps to a Java `float`. This definition also names the three columns—`ename`, `empno`, and `sal`, respectively—to match the table column names in the database. `MyIter` is a named iterator. See Chapter 3 of the *Oracle8i SQLJ Developer's Guide and Reference* to learn about positional iterators, which do not require column names.

2. Connect to the database.

```
Oracle.connect("jdbc:oracle:thin:@ool1:5521:so12","scott", "tiger");
```

Oracle SQLJ furnishes the `Oracle` class, and its `connect()` method accomplishes three important things:

- a. Registers the Oracle JDBC drivers SQLJ uses to access the database.
- b. Opens a database connection for the specified schema (user `scott`, password `tiger`) at the specified URL (host `oow11`, port `5521`, SID `sol2`, “thin” JDBC driver).
- c. Establishes this connection as the default connection for your 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. Execute a SQL statement.

- 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 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, which allows your 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, allowing 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, you can determine if this affects the application simply by re-running the translator.

Translating a SQLJ Program

Integrated development environments such as Oracle JDeveloper, a Windows-based visual development environment for Java programming, can translate, compile, and customize your SQLJ program for you as you build it. If you are not using an IDE, then you can 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, you must specify an example database schema in your translator option settings. It is not necessary for the schema to have identical data to the one the program will eventually run against; however, the tables should have columns with corresponding names and datatypes. Use the `user` option to enable online checking and specify the username, 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 in the server.

There is almost no difference between coding for a client-side SQLJ program and a server-side SQLJ program. The SQLJ runtime packages are automatically available on the server, and there are just the following few considerations:

- There are no explicit database connections for code running in the server; only a single implicit connection. You do not need the usual connection code. If you are porting an existing client-side application, you do not have to remove your 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 or question only for development because you would not write to `System.out` in a deployed server application.

To run a SQLJ program in 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 (Java classes and resources) to the server. In this case, it is easiest to bundle them into a `.jar` file first.
- Load your SQLJ source code to the server for the embedded translator to translate.

In either case, you can use the Oracle `loadjava` utility to load the file or files to the server. See the *Oracle8i SQLJ Developer's Guide and Reference* for more information.

Converting a Client Application to Run in the Server

The steps in converting an existing SQLJ client-side application to run in the server are as follows. Assume this is an application that has already been translated on the client:

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

```
create or replace procedure SQLJ_MYEXAMPLE as language java
  name 'MyExample.main(java.lang.String[])';
```

You can then execute `SQLJ_MYEXAMPLE` as with 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 allows PL/SQL anonymous blocks to be embedded in SQLJ executable statements, just as with SQL operations.

Debugging Server Applications

JServer furnishes a debugging capability useful for developers who use Oracle's JDeveloper or the JDK's `jdb` debugger. Other independent IDE vendors will be able to integrate their own debuggers with JServer.

Note: You must have the debug permission granted to your user to run a debug agent. See "[Debugging Permissions](#)" on page 5-24 for more information.

The Aurora JVM includes an implementation of the `sun.tools.debug.Agent` Java debugging protocol. Vendors may implement extensions to this protocol. All Oracle-specific extensions in the JServer environment will be publicly specified for use by all vendors.

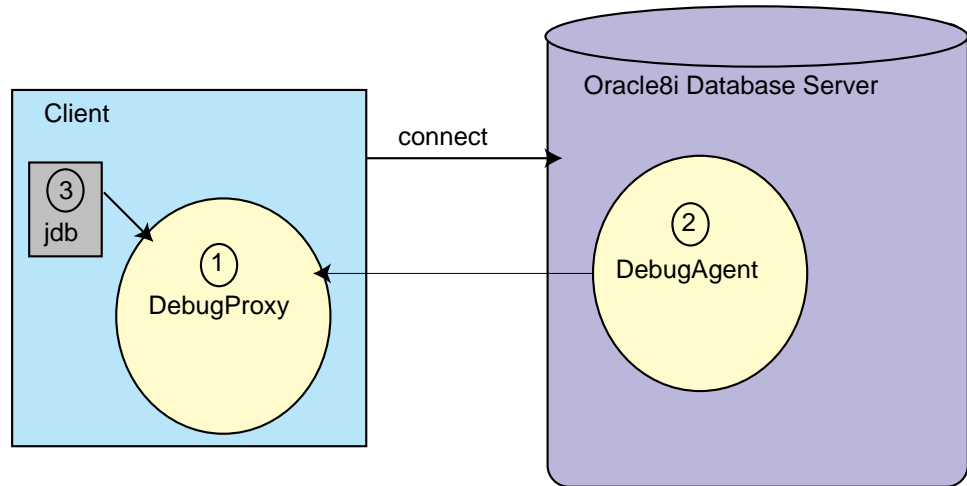
Note: JDeveloper has provided a user interface that utilizes JServer's debugging facilities. You can successfully debug an object loaded into the database by using JDeveloper's debugger.

Overview

In a single-tier environment, you debug a Java application on your own system. Sun Microsystems's `jdb` debugger communicates with the locally executing JVM through a protocol, which enables you to display information about the state of the Java program. In JServer, your Java program executes remotely on a server. The server can reside on the same physical machine, but it typically resides on a separate machine. The `jdb` debugger cannot debug a remote Java application while executing on Sun Microsystems's JDK. Oracle8i provides a method for debugging Java applications that exist on your remote database.

The `DebugProxy` class makes your remote Java application appear to be local. The `DebugProxy` enables a standard debugger that supports the `sun.tools.debug.Agent` protocol to connect to it as if the program is local. The

proxy forwards the standard requests to the server and returns the results to the debugger.



The steps for debugging are, using `jdb` as the example:

1. Start the `DebugProxy`. The `DebugProxy` waits for a `DebugAgent` that executes in the server.
2. Start the `DebugAgent`. Once you connect to the server (starting a session), you can start the `DebugAgent` on the server.

There is no way to cause server-resident code to execute and break, that is, execute and remain indefinitely in a halted mode. Instead, when you start the `DebugAgent`, you must specify a timeout period for the `DebugAgent` to wait before terminating. When the `DebugAgent` starts, the `DebugProxy` displays a password to use when connecting the debugger.

3. Connect a debugger. Start `jdb` using the password provided by the `DebugProxy` when the `DebugAgent` connected to it. Use `jdb` to suspend all threads of the Java program executing in the server. Proceed with debugging using `jdb` facilities.

1. Start the Debug Proxy

The debug proxy is located in the `aurora_client.jar` file, which is located in `$ORACLE_HOME/lib`. The `DebugProxy` class serves as a bridge between Aurora

JVMs running in the Oracle8i server and client-side debuggers running on your system.

A debug proxy waits for connections from the debug agent. Assuming the `aurora_client.jar` file is part of your CLASSPATH, you start the `DebugProxy` as follows:

```
debugproxy
```

You can also specify a particular port to wait on.

```
debugproxy -port 2286
```

The proxy prints out its name, its address, and the port it is waiting on.

```
Proxy Name: yourmachinename  
Proxy Address: aaa.bbb.ccc.ddd  
Proxy Port: 2286
```

Just-in-Time Debugging

Aurora takes any arguments to `DebugProxy`, beyond the optional port, as a command to execute. The agent port password is appended to the command. For instance, on Windows NT, you cause the proxy to execute `jdb` in a new console each time an agent connects to the proxy, by issuing the following command:

```
debugproxy -port 2286 start jdb -password
```

2. Starting, Stopping, and Restarting the Debug Agent

Note: To invoke the debug agent, the caller must have `JAVADEBUGPRIV` permission. For more information, see ["Debugging Permissions"](#) on page 5-24.

Once a proxy is running, you can start a debug agent to connect to the proxy from `SQL*Plus`. You must specify the IP address or URL for a machine running a debug proxy, the port the proxy is waiting on, and a timeout in seconds. You start and stop the debug agent using methods specified within the `DBMS_JAVA` package.

```
SQL> call dbms_java.start_debugging('yourmachinename', 2286, 66);
```

The call waits until the timeout expires or until the main thread is suspended and resumed before it completes. Calculate a timeout that includes enough time for

your debugger to start up, but not so much as to delay your session if you cannot connect a debugger.

Note: If an agent is already running, Aurora stops it and starts a new agent. You can stop the debug agent explicitly through the `stop_debugging` method.

```
SQL> call dbms_java.stop_debugging();
```

Once a debug agent starts, it runs until you stop it, the debugger disconnects, or the session ends.

You can restart a stopped agent with any breakpoints still set with the `restart_debugging` method. The call waits until the timeout expires before it completes. You can also restart a running agent just to buy some seconds to suspend threads and set breakpoints.

```
SQL> call dbms_java.restart_debugging(66);
```

OracleAgent Class

The `DBMS_JAVA` debug agent and proxy calls are published entry points to static methods that reside in `oracle.aurora.debug.OracleAgent` class. You can start, stop, and restart the debug agent in Java code using the class `oracle.aurora.debug.OracleAgent` directly through the following methods:

```
public static void start(String host, int port, long timeout_seconds);
public static void stop();
public static void restart(long timeout_seconds);
```

3. Connecting a Debugger

Each time a debug agent connects to a proxy, the proxy starts a thread to wait for connections from a debugger. The thread prints out the number, name and address of the connecting agent, the port it is waiting on, and the port encoded as a password. Here, a specific port and password are provided for illustration only:

```
Agent Number: 1
Agent Name: servername
Agent Address: eee.fff.jjj.kkk
Agent Port: 2286
Agent Password: 3i65bn
```

You can then pass the password to a `jdb`-compatible debugger (JDK 1.1.6 or later):

```
jdb -password 3i65bn
```

The first thing you should do in the debugger is suspend all threads. Otherwise, your `start_debugging` call might time out and complete before you get your breakpoints set.

If your code writes to `System.out` or `System.err`, then you may also want to use the `dbgtrace` flag to `jdb`, which redirects these streams to the debugging console:

```
jdb -dbgtrace -password 3i65bn
```

How To Tell You Are Executing in the Server

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

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

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

The `getProperty` method returns the following:

- If executing in the server, returns a `String` that represents the Oracle8i database version ("8.1.5" or "8.1.6").
- If executing on the client, returns `null`.

Redirecting Output on the Server

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

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

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

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

Output prints at the end of the call.

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

Java Installation and Configuration

This chapter details what you need to know to install and configure JServer within your Oracle8i database. To configure Java memory, see the "[Java Memory Usage](#)" section in [Chapter 5, "Security and Performance"](#).

- [Initializing a Java-Enabled Database](#)
- [Configuring JServer](#)
- [Enabling the Java Client](#)

Initializing a Java-Enabled Database

If you installed Oracle8i with JServer, the database is Java-enabled. That is, it is ready to run Java stored procedures, JDBC, SQLJ, and CORBA/EJB objects. If you are using your own scripts to create your Oracle instance, you must initialize the JServer explicitly.

You install JServer in one of three ways:

- Oracle8i Typical or Minimal Install—Choose the Typical or Minimal Oracle8i installation, which results in JServer being automatically installed.
- Oracle8i Custom Install—Choose the JServer option within a "Custom" Oracle8i installation.
- [Manual Install](#)—Install JServer by invoking the `initjvm.sql` script.

Manual Install

If you did not install JServer through any of the Oracle8i install options, you can add JServer to an existing database with the `initjvm.sql` script in `ORACLE_HOME/javavm/install`.

The `initjvm.sql` script loads the initial set of Java classes necessary to support Java, initializes the tables for supporting Java and for the CORBA namespace, and publishes top-level entry points through call-specifications. The `initjvm.sql` script loads the support Java classes into the database, which include the following:

- the standard Java runtime
- bytecode verifier and optimizer
- Java and SQLJ compilers
- JDBC runtime
- CORBA ORB and EJB runtime
- some additional support classes, such as `DBMS_JAVA`, which are described in "[Package DBMS_JAVA](#)" on page 4-3

The `initjvm.sql` script performs the following actions:

1. Loads the classes to the SYS schema.
2. Creates public synonyms for the loaded classes to be accessible to all users.
3. Alters some of these classes to run with definer's rights to support CORBA callouts.

4. Defines database start up and shut down triggers.

Note: The `initjvm.sql` script can take up to an hour to execute.

Requirements

Initializing a Java-enabled database requires a `SHARED_POOL_SIZE` of 50 MB, a `JAVA_POOL_SIZE` of about 20 MB, an additional 30 MB of system tablespace, and enough rollback segments. If the script fails for some reason, such as a lack of resources, you can adjust resources as necessary and re-execute `initjvm.sql`. Refer to "[Java Memory Usage](#)" on page 5-26 and the `/javavm/README.txt` file for the most up-to-date information on database initialization file configuration parameters and requirements.

In addition, there are specific requirements for enabling EJB and CORBA communications. The initial settings that the Oracle8i JServer installations furnish should be sufficient to get you started. Consult the specifics of the documentation in the *Oracle8i Enterprise JavaBeans and CORBA Developer's Guide* and the *Net8 Administrator's Guide* for more details.

Package DBMS_JAVA

When initializing the JServer, the `initjvm.sql` script creates the PL/SQL package `DBMS_JAVA`. Some entrypoints of `DBMS_JAVA` are for your use; others are only for internal use. The corresponding Java class `DbmsJava` provides methods for accessing RDBMS functionality from Java.

The `DBMS_JAVA` package supplies the following entrypoints:

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

Return the full name from a Java schema object. Because Java classes and methods can have names exceeding the maximum SQL identifier length, Aurora uses abbreviated names internally for SQL access. This function simply returns the original Java name for any (potentially) truncated name. An example of this function is to print the fully qualified name of classes that are invalid:

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

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

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

Refer to "[Shortened Class Names](#)" on page 2-31 and *Oracle8i Java Stored Procedures Developer's Guide* for examples of these functions.

```
FUNCTION get_compiler_option(what VARCHAR2, optionName VARCHAR2)
PROCEDURE set_compiler_option(what VARCHAR2, optionName VARCHAR2,
                             value VARCHAR2)
PROCEDURE reset_compiler_option(what VARCHAR2, optionName VARCHAR2)
```

These three entry points control the options of the JServer Java and SQLJ compiler Oracle8i delivers. See "[Compiling Java Classes](#)" on page 2-14 for an example of these options. Additionally, both the *Oracle8i Java Stored Procedures Developer's Guide* and the *Oracle8i SQLJ Developer's Guide and Reference* document the options and these entry points.

```
PROCEDURE set_output (bufferSize NUMBER)
```

This procedure redirects the output of Java stored procedures and triggers to the `DBMS_OUTPUT` package. See "[Redirecting Output on the Server](#)" on page 3-23 for an example.

```
PROCEDURE loadjava(options varchar2)
PROCEDURE loadjava(options varchar2, resolver varchar2)
PROCEDURE dropjava(options varchar2)
```

These procedures allow you to load and drop classes within the database using a call rather than through the `loadjava` or `dropjava` command-line tools. To execute within your Java application, do the following:

```
call dbms_java.loadjava('... options...');
call dbms_java.dropjava('... options...');
```

The options are identical to those specified for the `loadjava` and `dropjava` command-line tools. Each option should be separated by a blank. You should not separate the options with a comma. The only exception for this is the `loadjava -resolver` option, which contains blanks. For `-resolver`, you should specify all other options first, separate these options by a comma, and then specify the `-resolver` option with its definition. You should not specify the following options, because they relate to the database connection for the `loadjava` command-line tool: `-thin`, `-oci8`, `-user`, `-password`. The output is directed to `stderr`.

For more information on the available options, see ["loadjava"](#) on page A-7.

```
PROCEDURE grant_permission( grantee varchar2,
                           permission_type varchar2,
                           permission_name varchar2,
                           permission_action varchar2 )
```

```
PROCEDURE restrict_permission( grantee varchar2,
                               permission_type varchar2,
                               permission_name varchar2,
                               permission_action varchar2)
```

```
PROCEDURE grant_policy_permission( grantee varchar2,
                                   permission_schema varchar2,
                                   permission_type varchar2,
                                   permission_name varchar2)
```

```
PROCEDURE revoke_permission(permission_schema varchar2,
                            permission_type varchar2,
                            permission_name varchar2,
                            permission_action varchar2)
```

```
PROCEDURE disable_permission(key number)
```

```
PROCEDURE enable_permission(key number)
```

```
PROCEDURE delete_permission(key number)
```

These entry points control the JVM permissions. See ["Setting Permissions"](#) on page 5-6 for a description and example of these options.

```
PROCEDURE start_debugging(host varchar2, port number,
                          timeout number)
```

```
PROCEDURE stop_debugging
```

```
PROCEDURE restart_debugging(timeout number)
```

These entry points start and stop the debug agent when debugging. See ["Debugging Server Applications"](#) on page 3-18 for a description and example of these options.

Configuring JServer

When you install JServer as part of your normal Oracle8i installation, you will encounter configuration requirements for JServer within the Oracle8i Database Configuration Assistant and the Net8 Assistant. However, if you install using `initjvm.sql`, you must configure either by bringing up certain configuration tools or manually editing the initialization files.

The main configuration for Java classes within Oracle8i includes configuring Java memory requirements, the type of database processes, and the underlying connection protocol to the server.

- **Java memory requirements**—You must have at least 20 MB of `JAVA_POOL_SIZE` and 50 MB of `SHARED_POOL_SIZE`. If you installed JServer through the Oracle installer, these parameters are configured correctly. If you installed using the SQL script `initjvm.sql`, see "[Java Memory Usage](#)" on page 5-26 for information on configuring these parameters within the database initialization files.
- **Database processes**—You must decide whether to use dedicated server processes or MTS processes for your database server.
- **Connection protocol**—The presentation layer within the database defines the type of connection your client is using to access the database. In most networking protocols, the presentation layer is responsible for making sure data is represented in a format the application and session layers can accommodate. Within the database, a presentation is a service protocol that accepts incoming network requests and activates routines in the database kernel layer or in the Aurora JVM to process the requests. Currently, the two basic presentation types are Net8 and GIOP. The GIOP presentation is used for IIOP connections for EJB and CORBA applications. Most database connections default to the Net8 connection type.

Presentation protocol	Description
GIOP	Accepts IIOP or IOP over SSL requests for CORBA or EJB applications. See the <i>Oracle8i Enterprise JavaBeans and CORBA Developer's Guide</i> for configuration information.
Net8	Processes incoming Net8 requests for database SQL services from Oracle tools (such as SQL*Plus) and customer-written applications (using Forms, Pro*C, or the OCI). See "Configuring Multi-Threaded Server" in Chapter 9 of the <i>Net8 Administrator's Guide</i> for configuration information.

You will require a different configuration for your database type and connection configuration, depending on the type of Java application, as listed below:

Java API	Database type	Connection configuration
Java stored procedures	Java stored procedures can run either in dedicated server mode or multi-threaded server (MTS) mode. If you are primarily developing Java stored procedures, you can run them in the dedicated server configuration.	Java clients or PL/SQL clients that trigger a Java stored procedure connect over a Net8 connection. See the <i>Net8 Administrator's Guide</i> for information on configuring a Net8 connection.
Enterprise Java Beans (EJB) or CORBA	EJB and CORBA applications run only in the MTS configuration. See <i>Net8 Administrator's Guide</i> for information on configuring MTS.	EJB and CORBA clients use the CORBA Internet Inter-Orb Protocol (IIOP). See the <i>Oracle8i Enterprise JavaBeans and CORBA Developer's Guide</i> for information on IIOP configuration.
Both Java stored procedures and EJB or CORBA applications	<p>If you are combining both EJB and CORBA applications with Java stored procedures in a single application, you can configure both application types as follows:</p> <ul style="list-style-type: none"> ■ Configure your database to support EJB and CORBA applications in an MTS configuration ■ Configure your database to support stored procedures in a dedicated server configuration. 	You must configure both a Net8 and an IIOP connection.

Java Stored Procedure Configuration

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

- **Dedicated server mode**—You must configure the database and clients, as described in the *Oracle8i Java Stored Procedures Developer's Guide*.
- **MTS mode**—You must configure the server for MTS mode with the MTS_DISPATCHERS parameter, as described in Chapter 9 of the *Net8 Administrator's Guide*.

Java, SQL, or PL/SQL clients, which execute Java stored procedures on the server, connect to the database over a Net8 connection. For a full description of how to configure the Net8 connection, see the *Net8 Administrator's Guide*.

Enterprise JavaBeans and CORBA Configuration

Clients access EJB and CORBA applications in the database over an Inter-Orb Protocol (IIOP) connection. IIOP is an implementation of GIOP over TCP/IP. To support an IIOP connection, you must configure the database in MTS mode with the General Inter-Orb Protocol (GIOP) presentation.

Oracle8i also supports the use of authentication data such as certificates and private keys required for use by SSL in combination with both types of GIOP protocols—regular GIOP and session-based GIOP.

For a complete description of how to configure MTS, see the *Net8 Administrator's Guide*. For a full description of how to configure the GIOP presentation, see the *Oracle8i Enterprise JavaBeans and CORBA Developer's Guide*.

Enabling the Java Client

In order to run Java between the client and server, your client system must perform the following:

1. [Install JDK on the Client.](#)
2. [Set up CLASSPATH.](#)
3. [Verify the Port/SID.](#)
4. [Test Install with Samples.](#)

1. Install JDK on the Client

The client system is defined as the system where you execute the JServer tools, such as `loadjava`, `deployejb`. You can use the same system for both your client and server.

The client system requires JDK 1.1.6 or later. Solaris 2.6 bundles JDK 1.1.3, which does not work with our samples. Verify that your `PATH` includes JDK 1.1.6 or later and does not include JDK 1.1.3. To confirm what version of the JDK you are using, perform the following:

```
$ which java
/usr/local/packages/jdk1.1.6/bin/java
$ which javac
/usr/local/packages/jdk1.1.6/bin/javac
$ java -version
java version "1.1.6"
```

If JDK 1.1.6 does not appear within these commands, either put your JDK 1.1.6 installation at the start of PATH or remove the 1.1.3 installation. In addition, check your CLASSPATH for references to the incorrect JDK version.

Note: All Oracle8i Java-based client tools work in the Java 2 environment.

2. Set up CLASSPATH

If your client is a Java client involved with a distributed application—CORBA, EJB, or RMI—you must perform one of the following before compiling your client code:

- Set up CLASSPATH to include support JAR or ZIP files.
- Include support JAR or ZIP files within an option on compile line.

For the Java client to work across nodes in a distributed application, it must be compiled with appropriate server stubs. See the documentation for the required JAR or ZIP files within the *Oracle8i Enterprise JavaBeans and CORBA Developer's Guide*. You can also find out what JAR and ZIP files are required by examining the makefile for the associated sample.

3. Verify the Port/SID

If you do not configure the default listener port numbers or database SID in your installation—such as ports 1521, 2481, and SID `orcl`—the samples will not work correctly. All the samples expect the default port numbers and database SID provided by an Oracle8i Typical install. If you have different values, specify the new port numbers as follows:

Host type	Directions
UNIX	\$ make SERVICE=sess_iiop://localhost:myportnum:mysid
Windows NT	In the control panel, set the system environment variable ORACLE_SERVICE to sess_iiop://localhost:myportnum:mysid

4. Test Install with Samples

We provide a set of samples in `$ORACLE_HOME/javavm/demo/demo.tar` (or `demo.zip` for Windows NT). These samples compile and run for a database installed with the Oracle8i Typical install option. Execute these samples as a test of your installation.

```
$ORACLE_HOME/javavm/demo/examples/jsp/helloworld  
$ORACLE_HOME/javavm/demo/examples/corba/basic/helloworld  
$ORACLE_HOME/javavm/demo/examples/ejb/basic/helloworld
```

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

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

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

Security and Performance

This chapter details how to provide security for your Java applications—both for the connection and for the loaded classes—and how to make your application more effective.

The following features are discussed in this chapter:

- [Security](#)
- [Performance](#)

Security

Security is a large arena that includes network security for the connection, access and execution control of operating system resources or of JVM and user-defined classes, and bytecode verification of imported JAR files from an external source. You should be aware of what type of security you desire for your Java applications. The following sections describe the various security support available for Java applications within Oracle8i.

- [Network Connection Security](#)
- [Database Contents and JVM Security](#)

Network Connection Security

There are two major aspects to network security: authentication and data confidentiality. The type of authentication and data confidentiality is dependent on how you connect to the database—through Net8, JDBC, or distributed object (EJB or CORBA) connection.

Connection Security	Description
Net8	<p>The database can require both authentication and authorization before allowing a user to connect to the database. Net8 database connection security can require one or more of the following:</p> <ul style="list-style-type: none">▪ Use a username and password for client verification. Each incoming connection into the database has to provide the correct username/password configured within Net8. For more information, see the <i>Net8 Administrator's Guide</i>.▪ Use Advanced Networking Option for encryption, kerberos, or secureId. See the <i>Oracle Advanced Security Administrator's Guide</i>.▪ Use SSL for certificate authentication. See the <i>Oracle Advanced Security Administrator's Guide</i>.
JDBC	<p>JDBC connection security required is similar to the constraints required on a Net8 database connection. In addition to the books listed in the Net8 database connection section, see the <i>Oracle8i JDBC Developer's Guide and Reference</i>.</p>
Distributed Object	<p>Encryption and authentication might be required for distributed applications, such as EJB and CORBA. For more information, see the <i>Oracle8i Enterprise JavaBeans and CORBA Developer's Guide</i>.</p>

Database Contents and JVM Security

Once you are connected to the database, you still must have the correct Java 2 permissions and database privileges to access the resources stored within the database. These resources include the following:

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

These resources can be protected by the following two methods:

Resource Security	Description
Database Resource Security	Authorization for database resources requires that database privileges (not the same as the Java 2 security permissions) are granted to resources. For example, database resources include tables, classes, or PL/SQL packages. For more information, see the <i>Oracle8i Application Developer's Guide - Fundamentals</i> .
JVM Security	<p>JServer uses Java 2 security, which uses permissions to protect operating system resources and all loaded classes—both JVM and user-defined classes. Java 2 security is automatically installed upon startup and protects all operating system resources and JVM classes from all users, except JAVA_ADMIN. JAVA_ADMIN can grant permission to other users to access these classes.</p> <p>All user-defined classes are secured against users from other schemas. You can grant execution permission to other users/schemas through an option on the <code>loadjava</code> command. For more information on setting execution rights when loading the class, see the <code>-grant</code> option discussed in "Loading Classes" on page 2-22 or "loadjava" on page A-7.</p> <p>See "Java 2 Security" on page 5-3 for how to manage and modify Java 2 permissions and policies.</p>

Java 2 Security

Each user or schema must be assigned the proper permissions to access JVM classes and operating system resources. For example, this includes sockets, files, and system properties.

Java 2 security was created to provide a flexible, configurable security for Java applications. With Java 2 security, you can define exactly what permissions on each

loaded object a schema or role will have. In 8.1.5, the security provided you the choice of two secure roles:

- `JAVAUSERPRIV`—few permissions, including examining properties
- `JAVASYSPRIV`—major permissions, including updating JVM protected packages

Note: Both roles still exist within 8.1.6 for backward compatibility; however, Oracle recommends that you specify each permission explicitly, rather than utilize these roles.

Because JServer security is based on Java 2 security, you assign permissions to users on a class by class basis.

Java security was created for the non-database world. When you apply the Java 2 security model within the database, certain differences manifest themselves. For example, Java 2 security defines that all applets are implicitly untrusted and all classes within the `CLASSPATH` are trusted. Within Oracle8i, all classes are loaded within a secure database; thus, no classes are trusted.

The following table briefly describes the differences between Sun Microsystem's Java 2 security and Oracle8i's implementation. This table assumes that you already understand Sun Microsystem's Java 2 security model. For more information, we recommend the following books:

- *Inside Java 2 Platform Security* by Li Gong
- *Java Security* by Scott Oaks

Java 2 Security standard	Oracle8i implementation
Java classes located within the <code>CLASSPATH</code> are trusted.	All Java classes are loaded within the database. Classes are trusted on a class by class basis according to the permission granted.
You can specify the policy through the <code>-usepolicy</code> flag on the <code>java</code> command line.	You must specify the policy within the <code>PolicyTable</code> .

Java 2 Security standard	Oracle8i implementation
You can write your own SecurityManager or use the Launcher.	You can write your own SecurityManager; Oracle recommends that you use only Aurora's SecurityManager or that you extend Aurora's SecurityManager. If you want to modify the behavior, you should not define a SecurityManager; instead, you should extend <code>oracle.aurora.rdbms.SecurityManagerImpl</code> and override specific methods.
SecurityManager is not initialized for you. You must initialize the SecurityManager.	Aurora always initializes SecurityManager at startup.
Permissions are determined by the location where the application or applet is loaded from (the URL) or keycode (signed code).	Permissions are determined by the schema in which the class is loaded. JServer does not support signed code.
The security policy is defined in a file.	The PolicyTable definition is contained within a secure database table.
You can update the security policy file through a text editor (if you have the correct permissions) or through a tool.	You update the PolicyTable through DBMS_JAVA procedures. After initialization, only JAVA_ADMIN has permission to modify the PolicyTable. JAVA_ADMIN must grant you the right to modify the PolicyTable for you to grant permissions to others.
Permissions are assigned to a protection domain, which classes can belong to.	All classes within the same schema are within the same protection domain.
Use <code>CodeSource</code> class for identifying code.	Use <code>CodeSource</code> class for identifying schema.
<ul style="list-style-type: none"> ■ The <code>equals</code> method returns true if the URL and certificates are equal. ■ The <code>implies</code> method returns true if the first <code>CodeSource</code> is a generic representation that includes the specific <code>CodeSource</code> object. 	<ul style="list-style-type: none"> ■ The <code>equals</code> method returns true if the schemas are the same. ■ The <code>implies</code> method returns true if the schemas are the same.
Supports positive permissions only (grant).	Supports both positive (grant) and negative (restrict) permissions.

Setting Permissions

As designed in Java 2 security, Oracle8i supports the security classes. Normally, you set the permissions for the code base either through a tool or by editing the security policy file. In Oracle8i, you set the permissions dynamically through DBMS_JAVA procedures. These procedures modify a policy table, which is a new table within the database that exclusively manages Java 2 security permissions.

Two views have been created for you to view the policy table: USER_JAVA_POLICY and DBA_JAVA_POLICY. Both views contain information about granted and restricted permissions. The DBA_JAVA_POLICY view can see all rows within the policy table; the USER_JAVA_POLICY table can only see permissions relevant to the current user. The following is a description of the rows within each view:

Table entry	Description
Kind	GRANT or RESTRICT. Shows whether this permission is a positive (GRANT) or negative (RESTRICT) permission.
Grantee	The name of the user, schema, or role that the permission is assigned to.
Permission schema	The schema that the permission is loaded in.
Permission type	The Permission class that you assign. The syntax for the permission name is a string containing the entire class name, such as, <code>java.io.FilePermission</code> .
Permission name	The permission name of the assigned Permission class. You use this name when defining the permission. When defining the name for a permission of type <code>PolicyTablePermission</code> , the name can become quite complicated. See " Acquiring Administrative Permission to Update Policy Table " on page 5-11 for more information.
Permission action	The type of action you grant or restrict for this permission. If no action is appropriate for the permission, supply a null value.
Status	ACTIVE or INACTIVE. After granting the permission, you can disable or re-enable the permission. This field shows the status of whether the permission is enabled (ACTIVE) or disabled (INACTIVE).
Key	Sequence number you use to identify this row. This number should be supplied when disabling, enabling, or deleting the permission.

There are two ways to set your permissions:

- **Fine-Grain Definition for Each Permission**—You grant each permission individually for specific users or roles. If you do not grant a permission for access, the schema will be denied access.
- **General Permission Definition Assigned to Roles**—If you do not want to grant specific permissions for each user, you can grant roles, which grants a collection of permissions to the user. Oracle8i supplies the roles: JAVAUSERPRIV or JAVASYSPRIV.

Note: For absolute certainty about your security, you should implement the fine-grain definition. The general definition is easier; but you might not get the exact security you require.

Fine-Grain Definition for Each Permission

To set individual permissions within the policy table, you must provide the following information:

Parameter	Description
Grantee	The name of the user, schema, or role that you want to assign the permission to.
Permission type	The Permission class that you are granting permission on. For example, if you are defining access to a file, the permission type would be <code>FilePermission</code> . This parameter requires a fully-qualified name of a class that extends <code>java.lang.security.Permission</code> . If the class is not within <code>SYS</code> , the name should be prefixed by <code><schema>.</code> For example, <code>mySchema:myPackage.MyPermission</code> is a valid name for a user generated permission.
Permission name	The permission name is defined within the Permission class. Examine the appropriate Permission class for the relevant name. You use this name when the Permission object is created.
Permission action	The type of actions that you can specify vary according to the type of permission. For example, <code>FilePermission</code> can have the actions of read or write.
Key	Number returned from grant or restrict to use on enable, disable, or delete methods.

You can grant either Java 2 Permissions or create your own. The Java 2 Permissions are listed in [Table 5-1](#). If you would like to create your own permissions, see ["Creating Permissions"](#) on page 5-13.

Table 5-1 Permission Types

Permission type

- `java.util.PropertyPermission`
 - `java.io.SerializablePermission`
 - `java.io.FilePermission`
 - `java.net.NetPermission`
 - `java.net.SocketPermission`
 - `java.lang.RuntimePermission`
 - `java.lang.reflect.ReflectPermission`
 - `java.security.SecurityPermission`
 - `oracle.aurora.rdbms.security.PolicyTablePermission`
 - `oracle.aurora.security.JServerPermission`
-

Granting permissions using the DBMS_JAVA package:

```
procedure grant_permission( grantee varchar2, permission_type varchar2,  
                           permission_name varchar2,  
                           permission_action varchar2 )
```

```
procedure grant_permission( grantee varchar2, permission_type varchar2,  
                           permission_name varchar2,  
                           permission_action varchar2, key OUT number)
```

Granting permissions using Java:

```
long oracle.aurora.rdbms.security.PolicyTableManager.grant(  
    java.lang.String grantee,  
    java.lang.String permission_type,  
    java.lang.String permission_name,  
    java.lang.String permission_action);
```

```
void oracle.aurora.rdbms.security.PolicyTableManager.grant(  
    java.lang.String grantee,  
    java.lang.String permission_type,
```

```
java.lang.String permission_name,
java.lang.String permission_action,
long[] key);
```

Note: In the Java version of DBMS_JAVA, each method returns the row key identifier, either as a returned parameter or as an OUT variable in the parameter list. In the PL/SQL DBMS_JAVA package, the row key is returned only in the procedure that defines the key OUT parameter. This key is used to enable and disable specific permissions. See "[Enabling or Disabling Permissions](#)" on page 5-17 for more information.

Restricting permissions using the DBMS_JAVA package:

```
procedure restrict_permission( grantee varchar2, permission_type varchar2,
                             permission_name varchar2,
                             permission_action varchar2)
```

```
procedure restrict_permission( grantee varchar2, permission_type varchar2,
                             permission_name varchar2,
                             permission_action varchar2, key OUT number)
```

Restricting permissions using Java:

```
long oracle.aurora.rdbms.security.PolicyTableManager.restrict(
    java.lang.String grantee,
    java.lang.String permission_type,
    java.lang.String permission_name,
    java.lang.String permission_action);
```

```
void oracle.aurora.rdbms.security.PolicyTableManager.restrict(
    java.lang.String grantee,
    java.lang.String permission_type,
    java.lang.String permission_name,
    java.lang.String permission_action,
    long[] key);
```

Example 5-1 Granting permissions

Assuming that you have appropriate permissions to modify the policy table, you use the `grant_permission` method within the DBMS_JAVA package to modify the PolicyTable to allow the user access to the indicated file. In this example, the user,

Larry, has PolicyTable modification permission. Within a SQL package, Larry grants permission to read and write a file to the user Dave.

```
connect larry/larry

REM Grant DAVE permission to read and write the Test1 file.
call dbms_java.grant_permission('DAVE',
                                'java.io.FilePermission', '/test/Test1',
                                'read,write');

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

Example 5-2 Restricting permissions

You use the restrict method for specifying exceptions for general rules. You create general rules to grant or restrict access to a larger arena. That is, if you have defined a general rule that no one can read or write for an entire directory, you can use restrict for eliminating one aspect of this rule. In addition, you can override this restriction with a more specific grant. For example, if you want to allow access to all files within the /tmp directory—except for your password file that exists in that directory—you would grant permission for read and write to all files within /tmp and restrict read and write access to the password file. Furthermore, if you want the file owner to still be able to modify the password file, you can grant a more specific permission to allow access to one user, which will override the restriction. JServer security combines all rules to understand who really has access to the password file. The following is the code that implements this example:

1. Grants everyone read and write permission to all files in /tmp.
2. Restricts everyone from reading or writing one file in /tmp—password.
3. Grants Larry explicit permission to read and write the password file.

```
connect larry/larry

REM Grant permission to all users (PUBLIC) to be able to read and write
REM all files in /tmp.
call dbms_java.grant_permission('PUBLIC',
                                'java.io.FilePermission',
                                '/tmp/*',
                                'read,write');

REM Restrict 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 restriction,
REM Larry can read and write /tmp/password.

```
call dbms_java.grant_permission('LARRY',
    'java.io.FilePermission',
    '/tmp/password',
    'read,write');
```

```
commit;
```

The explicit rule for this scenario is as follows:

If the restrict permission implies the request, then for a grant to be effective, the restrict permission must also imply the grant.

Acquiring Administrative Permission to Update Policy Table

After the first initialization for JServer, only a single role—`JAVA_ADMIN`—is allowed to modify the policy table. The `JAVA_ADMIN` role is immediately assigned to `DBA`; thus, if you are assigned to the `DBA` group, you will automatically take on all `JAVA_ADMIN` permissions.

In order for you to be able to add permissions to this table, `JAVA_ADMIN` must grant you administrative permission to change the policy table. `JAVA_ADMIN` grants your schema update rights for the permission, `PolicyTablePermission`. This permission defines that your schema can update certain specific permission types. For example, in order for you to add a permission that controls access to a file, you must have a `PolicyTablePermission` that allows you to add grant or restrict permission on a `FilePermission`. To grant administrative permission, use the following method within the `DBMS_JAVA` package.

Granting policy table administrative permissions using `DBMS_JAVA`:

```
procedure grant_policy_permission( grantee varchar2, permission_schema varchar2,
    permission_type varchar2,
    permission_name varchar2)
```

```
procedure grant_policy_permission( grantee varchar2, permission_schema varchar2,
    permission_type varchar2,
    permission_name varchar2,
    key OUT number)
```

Granting policy table administrative permission using Java:

```
long oracle.aurora.rdbms.security.PolicyTableManager.grantPolicyPermission(  
    java.lang.String grantee,  
    java.lang.String permission_type,  
    java.lang.String permission_name);
```

```
void oracle.aurora.rdbms.security.PolicyTableManager.grantPolicyPermission(  
    java.lang.String grantee,  
    java.lang.String permission_type,  
    java.lang.String permission_name,  
    long[] key);
```

Parameter	Description
Grantee	The name of the user, schema, or role that you want to assign the permission to.
Permission schema	The <i><schema></i> where the permission class is loaded.
Permission type	The Permission class that you are granting permission on. For example, if you are defining access to a file, the permission type would be <code>FilePermission</code> . This parameter requires a fully-qualified name of a class that extends <code>java.lang.security.Permission</code> .
Permission name	The permission name is defined within the Permission class. Examine the appropriate Permission class for the relevant name. You use this name when the Permission object is created.
Row number	Number returned from grant or restrict to use on enable, disable, or delete methods.

Note: When looking at the policy table, the name within the PolicyTablePermission rows contains both the permission type and the permission name, which are separated by a '#'. For example, to grant a user administrative rights for reading a file, the name in the row contains `java.io.FilePermission#read`. The '#' separates the permission class from the permission name.

Example 5–3 Granting PolicyTable permission

The following example shows JAVA_ADMIN (as SYS) giving Larry permission to update the PolicyTable for FilePermission. Once this permission is granted, Larry can grant permissions to other users for reading, writing, and deleting files.

```
REM Connect as SYS, which is assigned JAVA_ADMIN role, to give Larry permission
REM to modify the PolicyTable
connect SYS/SYS
```

```
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:

1. [Create and load the user permission.](#)
2. [Grant administrative and action permissions to specified users.](#)
3. [Implement security checks for the permission.](#)

1. Create and load the user permission You can create your own permission by extending the Java 2 Permission class. Any user created permission must extend Permission. The following example creates MyPermission. MyPermission extends BasicPermission, which in turn extends Permission.

```
package test.larry;
import java.security.Permission;
import java.security.BasicPermission;

public class MyPermission extends BasicPermission {

    public MyPermission(String name) {
```

```
        super(name);
    }

    public boolean implies(Permission p) {
        boolean result = super.implies(p);
        return result;
    }
}
```

2. Grant administrative and action permissions to specified users When you create a permission, you are designated as owner of that permission. The owner always has administrative rights for the permission. This means that the owner can grant permission, including administrative rights, to other users. Administrative rights permits the user to update the PolicyTable for the user-defined permission. For example, if LARRY creates a permission, MyPermission, only LARRY can invoke **grant_policy_permission** for himself or another user. This method updates the PolicyTable on who can grant rights to MyPermission. The following code demonstrates this:

```
REM Since Larry is the user that creates MyPermission, Larry connects to
REW the database to assign permissions for MyPermission.
connect larry/larry
```

```
REM As the owner of MyPermission, Larry grants himself the right to
REM administer permissions for test.larry.MyPermission within the JVM
REM security PolicyTable. Only the owner of the user-defined permission
REM can grant administrative rights.
call dbms_java.grant_policy_permission('LARRY', 'LARRY',
                                     'test.larry.MyPermission', '*');
```

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

Once you have granted administrative rights, you can grant action permissions for the user created permission. For example, the following SQL grants permission for LARRY to execute anything within MyPermission and DAVE to be able to only execute actions that start with "act."

```
REM Since Larry is the user that creates MyPermission, Larry connects to
REW the database to assign permissions for MyPermission.
connect larry/larry
```

```
REM Once able to modify the PolicyTable for MyPermission, Larry grants himself
REM full permission for MyPermission. Notice that the Permission is prepended
REM with its owner schema.
```

```

call dbms_java.grant_permission( 'LARRY',
                                'LARRY:test.larry.MyPermission', '**', null);

REM Larry grants Dave permission to do any actions that start with 'act.*'.
call dbms_java.grant_permission
    ('DAVE', 'LARRY:test.larry.MyPermission', 'act.*', null);

REM commit the changes to the PolicyTable
commit;

```

3. Implement security checks for the permission Once you have created, loaded, and assigned permissions for `MyPermission`, you must implement the call to `SecurityManager` for having the permission checked. There are four methods in the following example: `sensitive`, `act`, `print`, and `hello`. Because of the permissions granted in the SQL example in step 2, the following users can execute methods within the example class:

- LARRY can execute any of the methods.
- DAVE is given permission to execute only the `act` method.
- Anyone can execute the `print` and `hello` methods. The `print` method does not check any permissions, so anyone can execute the `print` method. The `hello` method executes `AccessController.doPrivileged`, which means that the method executes with LARRY's permissions. This is referred to as `definer's rights`.

```

package test.larry;
import java.security.AccessController;
import java.security.Permission;
import java.security.PrivilegedAction;

import java.sql.Connection;
import java.sql.SQLException;

/**
 * MyActions is a class with a variety of public methods that
 * have some security risks associated with them. We will rely
 * on the Java security mechanisms to ensure that they are
 * performed only by code that is authorized to do so.
 */

public class Larry {

    private static String secret = "Larry's secret";

```

```
MyPermission sensitivePermission = new MyPermission("sensitive");

/**
 * This is a security sensitive operation. That is it can
 * compromise our security if it is executed by a "bad guy".
 * Only larry has permission to execute sensitive.
 */
public void sensitive() {
    checkPermission(sensitivePermission);
    print();
}

/**
 * Will print a message from Larry. We need to be
 * careful about who is allowed to do this
 * because messages from Larry may have extra impact.
 * Both larry and dave have permission to execute act.
 */
public void act(String message) {
    MyPermission p = new MyPermission("act." + message);
    checkPermission(p);
    System.out.println("Larry says: " + message);
}

/**
 * Print our secret key
 * No permission check is made; anyone can execute print.
 */
private void print() {
    System.out.println(secret);
}

/**
 * Print "Hello"
 * This method invokes doPrivileged, which makes the method run
 * under definer's rights. So, this method runs under Larry's
 * rights, so anyone can execute hello.
 * Only Larry can execute hello
 */
public void hello() {
    AccessController.doPrivileged(new PrivilegedAction() {
        public Object run() { act("hello"); return null; }
    });
}
```

```

/**
 * If a security manager is installed ask it to check permission
 * otherwise use the AccessController directly
 */
void checkPermission(Permission permission) {
    SecurityManager sm = System.getSecurityManager();
    sm.checkPermission(permission);
}
}

```

Enabling or Disabling Permissions

Once you have defined a permission, you can disable it so that it is no longer valid. However, if you decide you want the permission validated again, you can enable the permission. You can delete the permission from the table if you believe that it will never be used again. To delete, you must first disable the permission. If you do not disable the permission, the deletion will not occur.

To disable permissions, you can use either the `disable_permission` or the `revoke` method.

- The `revoke_permission` method takes in parameters similar to the `grant` and `restrict`. It searches the entire policy table for all rows that apply to the supplied parameters.
- The `disable_permission` method disables only a single row within the policy table. To do this, it takes in the policy table key. This key is also necessary to enable or delete a permission. To retrieve the permission key number, perform one of the following:
 - * Save the key when it is returned on the `grant` or `restrict` calls. If you do not foresee a need to ever enable or disable the permission, you can use the `grant` and `restrict` calls that do not return the permission number.
 - * View `DBA_JAVA_POLICY` for the appropriate permission key number.

Disabling permissions using `DBMS_JAVA`:

```

procedure revoke_permission(permission_schema varchar2,
                             permission_type varchar2,
                             permission_name varchar2,
                             permission_action varchar2)

procedure disable_permission(key number)

```

Disabling permissions using Java:

```
void revoke(String schema, String type, String name, String action);  
  
void oracle.aurora.rdbms.security.PolicyTableManager.disable(long number);
```

Enabling permissions using DBMS_JAVA:

```
procedure enable_permission(key number)
```

Enabling permissions using Java:

```
void oracle.aurora.rdbms.security.PolicyTableManager.enable(long number);
```

Deleting permissions using DBMS_JAVA:

```
procedure delete_permission(key number)
```

Deleting permissions using Java:

```
void oracle.aurora.rdbms.security.PolicyTableManager.delete(long number);
```

Permission Types

[Table 5–2](#) lists the installed permission types. Whenever you want to grant or restrict permission, you must provide the permission type within the DBMS_JAVA method. The permission types with which you control access are the following:

- Oracle-provided permission types listed in [Table 5–2](#)
- user created permission types that extend `java.security.Permission`

Table 5–2 *Permission types*

Permission type

- `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`

Table 5–2 Permission types (Cont.)

-
- `oracle.aurora.rdbms.security.PolicyTablePermission`
 - `oracle.aurora.security.JServerPermission`
-

All the Java permission types are documented in Sun Microsystem's Java 2 documentation.

Note: SYS is granted permission to load libraries that come with Oracle. However, Aurora does not support other users loading libraries, because loading C within the database is insecure. So, you are not allowed to grant permission for `loadLibrary.*` of `RuntimePermission`.

The Oracle-specific permissions, `PolicyTablePermission` and `JServerPermission` are described below:

oracle.aurora.rdbms.security.PolicyTablePermission You use this permission to control who can update the policy table. Once granted the right to update the policy table for a certain `Permission` type, the user can control another user's access to some resource.

After JServer initialization, only the `JAVA_ADMIN` role can grant administrative rights for the policy table through `PolicyTablePermission`. Once it grants this right to other users, these users can in turn update the policy table with their own grant and restrict permissions.

To grant policy table updates, you use the `DBMS_JAVA` method: `grant_policy_permission`, as discussed in "[Acquiring Administrative Permission to Update Policy Table](#)" on page 5-11. Once you have updated the table, you can view either the `DBA_JAVA_POLICY` or `USER_JAVA_POLICY` views to see who has been granted permissions.

oracle.aurora.security.JServerPermission You use this permission to grant and restrict access to Aurora JVM resources. The `JServerPermission` extends from `BasicPermission`. The following table lists the names that `JServerPermission` grants access for:

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 <code>oracle.aurora.vm.OracleRuntime</code> on call settings
Memory.Stack	grants rights to call certain methods in <code>oracle.aurora.vm.OracleRuntime</code> on stack settings
Memory.SGASIntern	grants rights to call certain methods in <code>oracle.aurora.vm.OracleRuntime</code> on SGA settings
Memory.GC	grants rights to call certain methods in <code>oracle.aurora.vm.OracleRuntime</code> on garbage collector settings

Initial Permission Grants

When you first initialize JServer, several roles are populated with certain permission grants. The following tables show these roles and their initial permissions:

1. The `JAVA_ADMIN` role is given access to modify the policy table for all permissions. All DBAs, including `SYS`, are granted `JAVA_ADMIN`. Full administrative rights to update the policy table are granted for the following permissions:

Permission type

```
java.util.PropertyPermission
java.io.SerializablePermission
java.io.FilePermission
java.net.NetPermission
java.net.SocketPermission
java.lang.RuntimePermission
```

```

java.lang.reflect.ReflectPermission
java.security.SecurityPermission
oracle.aurora.rdbms.security.PolicyTablePermission
oracle.aurora.security.JServerPermission

```

2. In addition to the JAVA_ADMIN permissions, SYS is also granted the following permissions:

Note: Within the RuntimePermission grants, there seems to be unnecessary granting of more specific permission for loadlibrary.<package>. The reason for this is to override the restriction given to PUBLIC for loadLibrary.*.

Table 5–3 SYS Initial Permissions

Permission type	Permission name	Action granted
oracle.aurora.rdbms.security.PolicyTablePermission	*	Administrative rights to modify the policy table
oracle.aurora.security.JServerPermission	*	null
java.net.NetPermission	*	null
java.security.SecurityPermission	*	null
java.util.PropertyPermission	*	write
java.lang.reflect.ReflectPermission	*	null
java.lang.RuntimePermission	*	null
	loadLibrary.xaNative	null
	loadLibrary.corejava	null
	loadLibrary.corejava_d	null

3. All users are initially granted the following permissions. For the JServerPermission, all users can load classes, except for the list specified in the table. These exceptions are RESTRICT permissions. For more information on RESTRICT permissions, see [Example 5–2](#).

Table 5–4 PUBLIC Default Permissions

Permission type	Permission name	Granted action
oracle.aurora.rdbms.security. PolicyTablePermission	java.lang.RuntimePermission loadLibrary.*	null
java.util.PropertyPermission	*	read
	user.language	write
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 5–5 JAVAUSERPRIV Permissions

Permission type	Permission name	Action
java.net.SocketPermission	*	connect, resolve
java.io.FilePermission	<<ALL FILES>>	read
java.lang.RuntimePermission	modifyThreadGroup, stopThread, getProtectionDomain, readFileDescriptor, accessClassInPackage.*, and defineClassInPackage.*	null

Table 5–6 JAVASYSPRIV Permissions

Permission type	Permission name	Action
java.io.SerializablePermission	*	no applicable action
java.io.FilePermission	<<ALL FILES>>	read ,write, execute, delete
java.net.SocketPermission	*	accept, connect, listen, resolve

Table 5–6 JAVASYSPRIV Permissions (Cont.)

Permission type	Permission name	Action
java.lang.RuntimePermission	createClassLoader	null
	getClassLoader	null
	setContextClassLoader	null
	setFactory	null
	setIO	null
	setFileDescriptor	null
	readFileDescriptor	null
	writeFileDescriptor	null

Table 5–7 JVADEBUGPRIV Permissions

Permission type	Permission name	Action
oracle.aurora.security.JServerPermission	Debug	null
java.net.SocketPermission	*	connect, resolve

General Permission Definition Assigned to Roles

In 8.1.5, JVM security was controlled by granting the roles of JAVASYSPRIV, JAVAUSERPRIV, or JVADEBUGPRIV to schemas. In the current version, these roles still exist as permission groups. See the previous section, "[Initial Permission Grants](#)" on page 5-20 for the explicit permissions set for each role. You can set up and define your own collection of permissions. Once defined, you can grant any collection of permissions to any user. That user will then have the same permissions that exist within the role. In addition, if you need additional permissions, you can add individual permissions to either your specified user or role. Permissions defined within the policy table have a cumulative effect. See "[Fine-Grain Definition for Each Permission](#)" on page 5-7 for information on how to grant permissions to a user or a role.

Note: The ability to write to properties, granted through the write action on `PropertyPermission`, is no longer granted to all users. Instead, you must either have `JAVA_ADMIN` grant you this permission or you can receive this permission by being granted the role of `JAVASYSPRIV`.

The following example gives Larry and Dave the following permissions:

- Larry receives `JAVASYSPRIV` permissions.
- Dave receives `JAVADEBUGPRIV` permissions and the ability to read and write all files on the system.

```
REM Granting Larry the same permissions as exist within JAVASYSPRIV
grant javasyspriv to larry;
```

```
REM Granting Dave the ability to debug
grant javadebugpriv to dave;
```

```
commit;
```

```
REM I also want Dave to be able to read and write all files on the system
call dbms_java.grant_permission('DAVE', 'SYS:java.io.FilePermission',
                               '<<ALL FILES>>', 'read,write', null);
```

Debugging Permissions

A debug role, `JAVADEBUGPRIV`, was created to grant permissions for running the debugger. The permissions assigned to this role are listed in [Table 5-7](#). In order to have permission to invoke the debug agent, the caller must have been granted `JAVADEBUGPRIV` or the debug `JServerPermission` as follows:

```
REM Granting Dave the ability to debug
grant javadebugpriv to dave;
```

```
REM Larry grants himself permission to start the debug agent.
call dbms_java.grant_permission
    ('LARRY', 'oracle.aurora.security.JServerPermission', 'Debug', null);
```

A debugger provides extensive access to both code and data on the server, but at this time, we envision its use to be restricted to development environments. Refer to

the discussion in the section "[Debugging Server Applications](#)" on page 3-18 for information on using the debugging facilities in this release.

Permission for Loading Classes

In order to load classes, you must have the following permission:

```
JServerPermission("LoadClassInPackage." + <class_name>)
```

The class name is the fully qualified name of the class that you are loading.

This excludes loading into system packages or replacing any system classes. Even if you are granted permission to load a system class, JServer prevents you from performing the load. System classes are those classes that are installed with the database.

The following shows the ability of each user after database installation, including permissions and JServer restrictions:

- SYS can load any class, except for system classes.
- Any user can load classes in its own schema that do not start with the following patterns: `java.*`, `oracle.aurora.*`, `oracle.jdbc.*`. If the user wants to load into another schema, it must be granted the `JServerPermission(LoadClassInPackage.<class>)` permission.

The following example shows how to grant SCOTT permission to load classes into the `oracle.aurora.*` package:

```
dbms_java.grant_permission('SCOTT', 'SYS:oracle.aurora.tools.*', null);
```

Performance

You can increase your Java application performance through one of the following methods:

- [Natively Compiled Code](#)
- [Java Memory Usage](#)
- [End-of-Call Migration](#)

Natively Compiled Code

All core Java class libraries and Oracle-provided Java code within JServer is natively compiled for greater execution speed. Java classes exist as shared libraries in

`$ORACLE_HOME/javavm/admin`, where each shared library corresponds to a Java package. For example, `libjox8java_lang.so` on Solaris and `libjox8java_lang.dll` on Windows/NT hold `java.lang` classes. Specifics of packaging and naming can vary by platform. The Aurora JVM uses natively compiled Java files internally and opens them, as necessary, at runtime.

Native compilation provides a speed increase ranging from two to ten times 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

In general, natively compiled code consumes more memory than interpreted code, by a factor of two to three. Caching in an adaptive optimization technique produces a similar trade-off. This is particularly true when a Java server is executing independent code or stored procedures from thousands of users. Using the JServer's static compilation approach for delivering natively compiled Java code provides a large, consistent performance gain, regardless of the number of users or the code paths they traverse on the server.

Java native compilation technology will be available for your Java code in subsequent releases. In the current JServer release, Java code you load to the server is interpreted; the underlying core classes upon which your code relies (`java.lang.*`) are natively compiled. Until the native compiler is available for user programs, the net speed benefit of native compilation to your executing program is dependent upon how much native code is traversed, as opposed to interpreted code. The more Java code from core classes and Oracle-provided class libraries you use, the more benefit you will see from native compilation.

Java Memory Usage

The typical and custom database installation process furnishes a database that has been configured for reasonable Java usage during development. However, runtime use of Java should be determined by the usage of system resources for a given deployed application. Resources you use during development can vary widely, depending on your activity. The following sections describe how you can configure

memory depending on your performance needs, how to tell how much SGA memory you are using, and what errors denote a Java memory issue:

- [Configuring Memory Initialization Parameters](#)
- [Java Pool Memory](#)
- [Displaying Used Amounts of Java Pool Memory](#)
- [Correcting Out of Memory Errors](#)

Configuring Memory Initialization Parameters

You can modify the following database initialization parameters to tune your memory usage to reflect more accurately your application needs:

- **SHARED_POOL_SIZE**—Shared pool memory is used by the class loader within the JVM. The class loader uses an average of about 8 KB for each loaded class. Shared pool memory is used when loading and resolving classes into the database. It is also used when compiling source in the database or when using Java resource objects in the database.

The memory specified in `SHARED_POOL_SIZE` is consumed transiently when you use `loadjava`. The database initialization process (executing `initjvm.sql` against a clean database, as opposed to the installed seed database) requires `SHARED_POOL_SIZE` to be set to 50 MB as it loads the Java binaries for approximately 8,000 classes and resolves them. The `SHARED_POOL_SIZE` resource is also consumed when you create call specifications and as the system tracks dynamically loaded Java classes at runtime.

- **JAVA_POOL_SIZE**—Aurora's memory manager allocates all other Java state during runtime execution from the amount of memory allocated using `java_pool_size`. This memory includes the shared in-memory representation of Java method and class definitions, as well as the Java objects migrated to session space at end-of-call. In the first case, you will be sharing the memory cost with all Java users. In the second case, under MTS, you must adjust `JAVA_POOL_SIZE` allocation based on the actual amount of state held in static variables for each session. See "[Java Pool Memory](#)" on page 5-28 for more information on `JAVA_POOL_SIZE`.
- **JAVA_SOFT_SESSIONSPACE_LIMIT**—This parameter allows you to specify a soft limit on Java memory usage in a session, which will warn you if you must increase your Java memory limits. The memory that you request in this parameter is allocated when the session is started.

When a user's session-duration Java state exceeds this size, Aurora generates a warning that is written into the trace files. The default is 1 MB. You should understand the memory requirements of your deployed classes, especially as they relate to usage of session space.

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

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

JServer's unique memory management facilities and sharing of read-only artifacts (such as bytecodes) enables HelloWorld to execute with a per-session incremental memory requirement of only 35 KB. More stateful server applications, such as the Aurora/ORB that CORBA and EJB applications use, have a per-session incremental memory requirement of approximately 200 KB. Such applications must retain a significant amount of state in static variables across multiple calls. Refer to the discussion in the section, "[End-of-Call Migration](#)" on page 5-31, for more information on understanding and controlling migration of static variables at end-of-call.

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 Oracle8i server is running in.

Java pool memory used within a dedicated server

- The shared part of each Java class used per session. This includes readonly memory, such as code vectors, and methods. In total, this can average about 4-8 KB for each class.
- None of the per-session Java state of each session. For a dedicated server, this is stored in UGA within the PGA—not within the SGA.

Under dedicated servers, which is probably the case for applications using only Java Stored Procedures, the total required Java pool memory is not much more than 10 MB.

Java pool memory used within a Multi-Threaded Server (MTS)

- The shared part of each Java class that is used per session. This includes readonly memory, such as vectors, and methods. In total, this can average about 4-8 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. Since Java pool memory size is fixed, you must estimate the total requirement for your applications and multiply by the number of concurrent sessions they want to create a 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 MTS servers, which is the case for applications using CORBA or EJB, this figure could be very large. Java-intensive, multi-user benchmarks could require more than 1 GB. Current size limitations are unknown; however, it is platform dependent.

Note: If you are compiling code on the server, rather than compiling on the client and loading to the server, you might need a bigger `JAVA_POOL_SIZE` than the default 20 MB. EJB deployment uses the Java compiler on the server; therefore, it also requires a larger `JAVA_POOL_SIZE`.

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 what you configured in the database initialization file.

```
SVRMGR> select * from v$sgastat;
POOL          NAME                                BYTES
-----
fixed_sga     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

- [Running out of memory while compiling](#)
- [Running out of memory while loading](#)

Running out of memory while compiling If you run out of memory while compiling (within loadjava or deployejb), you should see an error:

```
A SQL exception occurred while compiling: : ORA-04031: unable to allocate bytes
of shared memory ("shared pool","unknown object","joxlod: init h", "JOX: ioc_
allocate_pal")
```

The cure is to shut down your database and to reset `JAVA_POOL_SIZE` to a larger value. The mention of "shared pool" in the error message is a misleading reference

to running out of memory in the "Shared Global Area". It does not mean you should increase your SHARED_POOL_SIZE. Instead, you must increase your JAVA_POOL_SIZE, restart your server, and try again.

Running out of memory while loading If you run out of memory while loading classes, it can fail silently, leaving invalid classes in the database. Later, if you try to invoke or resolve any invalid classes, you will see `ClassNotFoundException` or `NoClassDefFoundException` exceptions being thrown at runtime. You would get the same exceptions if you were to load corrupted class files. You should perform the following:

- Verify that the class was actually included in the set you are loading to the server. Many people have accidentally forgotten to load just one class out of hundreds and spend considerable time chasing this down.
- Use the `loadjava -force` option to force the new class being loaded to replace the class already resident in the server.
- Use the `loadjava -resolve` option to attempt resolution of a class during the load process. This allows you to catch missing classes at load time, not run time.
- Double check the status of a newly loaded class by connecting to the database in the schema containing the class, and execute the following:

```
select * from user_objects where object_name = dbms_java.shortname('');
```

The STATUS field should be "VALID". If `loadjava` complains about memory problems or failures such as "connection lost", increase SHARED_POOL_SIZE and JAVA_POOL_SIZE, and try again.

End-of-Call Migration

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

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

to only what is necessary, you help the memory manager and improve your server's performance.

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

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

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

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

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

Oracle-Specific Support for End-of-Call Optimization

You can register the static variables that you want cleared at the end of the call when the buffer, field, or data structure is created. Within the Oracle-specified

`oracle.aurora.memoryManager.EndOfCallRegistry` class, the `registerCallback` method takes in an object that implements a `Callback` object. The `registerCallback` object stores this object until the end of the call. When end-of-call occurs, Aurora invokes the `act` method within all registered `Callback` objects. The `act` method within the `Callback` object is implemented to clear the user-defined buffer, field, or data structure. Once cleared, the `Callback` is removed from the registry.

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

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

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

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

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

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

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

```
        thunk = null;
    }

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

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

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

- Instance fields—Use `EndOfCallRegistry` to clear state in data structures held in instance fields. For example, when a state is associated with each instance of a class, each instance has a field that holds the cached state for the instance and fills in the cached field as necessary. You can access the cached field with a method that ensures the state is cached.
 1. Implements the instance as a `Callback` object.
 2. Implements the `Callback.act` method to free up the instance's fields.
 3. When the user requests a cache, the `Callback` object registers itself for end-of-call migration.
 4. Provides a method, `createCachedField`, for lazily recreating the cache.

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

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

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

class Example2 implements Callback {
    private Object cachedField = null;

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

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

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

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

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

EndOfCallRegistry.registerCallback method

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

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

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

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

EndOfCallRegistry.runCallbacks method

```
static void runCallbacks()
```

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

Callback Interface

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

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

Callback.act method

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

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

Virtually all JServer developers use the `loadjava` and `dropjava` tools. See "[Preparing Java Class Methods for Execution](#)" on page 2-14 for more information. Each of the individual books for Oracle8i Java Support (*Oracle8i Java Stored Procedures and Developer's Guide*, *SQLJ Developer's Guide and Reference*, *Enterprise JavaBeans and CORBA Developer's Guide*, *Oracle8i JDBC Developer's Guide and Reference*, and *Oracle8i JPublisher User's Guide*) include loading information specific to their particular areas.

Schema Object Tools

Unlike a conventional Java virtual machine, which compiles and loads Java files, the Aurora Java virtual machine compiles and loads schema objects. The three kinds of Java schema objects are:

- *Java class schema objects*, which correspond to Java class files.
- *Java source schema objects*, which correspond to Java source files.
- *Java resource schema objects*, which correspond to Java resource files.

To make a class file runnable by the Aurora Java virtual machine, you use the `loadjava` tool to create a Java class schema object from the class file or the source file and load it into a schema. To make a resource file accessible to the Aurora Java virtual machine, you use `loadjava` to create and load a Java resource schema object from the resource file.

The `dropjava` tool does the reverse of the `loadjava` tool; it deletes schema objects that correspond to Java files. You should always use `dropjava` to delete a Java schema object that was created with `loadjava`; dropping by means of SQL DDL commands will not update auxiliary data maintained by `loadjava` and `dropjava`.

What and When to Load

You must load resource files with `loadjava`. If you create `.class` files outside the database with a conventional compiler, then you must load them with `loadjava`. The alternative to loading class files is to load source files and let the Oracle8i system compile and manage the resulting class schema objects. In the current Oracle8i release, most developers will find that compiling and debugging most of their code outside the database and then loading `.class` files to debug those files which must be tested inside the database, is the most productive approach. For a particular Java class, you can load either its `.class` file or its `.java` file, but not both.

`loadjava` accepts JAR files that contain either source and resource files or class and resource files (recall that you can load a class's source or its class file but not both). When you pass `loadjava` a JAR file or a ZIP file, `loadjava` opens the archive and loads its members individually; there is no JAR or ZIP schema object. A file whose content has not changed since the last time it was loaded is not re-loaded, therefore there is little performance penalty for loading JARs. Loading JAR files is the simplest and most foolproof way to use `loadjava`.

It is illegal for two schema objects in the same schema to define the same class. For example, suppose `a.java` defines class `x` and you want to move the definition of `x` to `b.java`. If `a.java` has already been loaded, then `loadjava` will reject an attempt to load `b.java` (which also defines `x`). Instead, do either of the following:

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

Resolution

All Java classes contain references to other classes. A conventional Java virtual machine searches for classes in the directories, ZIP files, and JARs named in the CLASSPATH. The Aurora Java virtual machine, by contrast, searches schemas for class schema objects. Each Oracle8i class has a *resolver spec*, which is the Oracle8i counterpart to the CLASSPATH. For a hypothetical class `alpha`, its resolver spec is a list of schemas to search for classes `alpha` uses. Notice that resolver specs are per-class, whereas in a classic Java virtual machine, CLASSPATH is global to all classes.

In addition to a resolver spec, each class schema object has a list of interclass reference bindings. Each reference list item contains a reference to another class, and one of the following:

- the name of the class schema object to invoke when class uses the reference
- a code indicating that the reference is unsatisfied; in other words, the referent schema object is not known

An Oracle8i facility called the *resolver* maintains reference lists. For each interclass reference in a class, the resolver searches the schemas specified by the class's resolver spec for a valid class schema object that satisfies the reference. If all references are resolved, the resolver marks the class *valid*. A class that has never been resolved, or has been resolved unsuccessfully, is marked *invalid*. A class that depends on a schema object that becomes invalid is also marked invalid at the same time; in other words, invalidation cascades upward from a class to the classes that use it and the classes that use them, and so on. When resolving a class that depends on an invalid class, the resolver first tries to resolve the dependency because it may be marked invalid only because it has never been resolved. The resolver does not re-resolve classes that are marked valid.

A class developer can direct `loadjava` to resolve classes, or can defer resolution until run time. (The resolver runs automatically when a class tries to load a class that is marked invalid.) It is best to resolve before run time to learn of missing classes early; unsuccessful resolution at run time produces a "class not found" exception. Furthermore, run-time resolution can fail for lack of database resources if the tree of classes is very large.

The `loadjava` has two resolution modes:

1. Load-and-resolve (`-resolve` option): Loads all classes you specify on the command line, marks them invalid, and then resolves them. Use this mode when initially loading classes that refer to each other, and in general when reloading isolated classes as well. By loading all classes and then resolving them, this mode avoids the error message that occurs if a class refers to a class that will be loaded later in the execution of the command.
2. Load-then-resolve (no `-resolve` option): Resolves each class when compiled at runtime.

Note: Like a Java compiler, `loadjava` resolves references to classes but not to resources; be sure to correctly load the resource files your classes need.

If you can, it is best to defer resolution until all classes have been loaded; this technique avoids the situation in which the resolver marks a class invalid because a class it uses has not yet been loaded.

Digest Table

The schema object digest table is an optimization that is usually invisible to developers. The digest table enables `loadjava` to skip files that have not changed since they were last loaded. This feature improves the performance of makefiles and scripts that invoke `loadjava` for collections of files, only some of which need to be re-loaded. A re-loaded archive file might also contain some files that have changed since they were last loaded and some that have not.

The `loadjava` tool detects unchanged files by maintaining a digest table in each schema. The digest table relates a file name to a *digest*, which is a shorthand representation of the file's content (a hash). Comparing digests computed for the same file at different times is a fast way to detect a change in the file's content—much faster than comparing every byte in the file. For each file it processes, `loadjava` computes a digest of the file's content and then looks up the file name in the digest table. If the digest table contains an entry for the file name that has the identical digest, then `loadjava` does not load the file because a corresponding schema object exists and is up to date. If you invoke `loadjava` with the `-verbose` option, then it will show you the results of its digest table lookups.

Normally, the digest table is invisible to developers because `loadjava` and `dropjava` keep it 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 with DDL. If the digest table becomes corrupted (`loadjava` does not update a schema object whose file has changed), use `loadjava`'s `-force` option to bypass the digest table lookup.

Compilation

Loading a source file creates or updates a Java source schema object and invalidates the class schema object(s) previously derived from the source. (If the class schema objects don't exist, `loadjava` creates them.) `loadjava` invalidates the old class schema objects because they were not compiled from the newly loaded source. Compilation of a newly loaded source, called for instance A, is automatically triggered by any of the following conditions:

- The resolver, working on class B, finds that it refers to class A but class A is invalid.

- The compiler, compiling source B, finds that it refers to class A but A is invalid.
- The class loader, trying to load class A for execution, finds that it is invalid.

To force compilation when you load a source file, use `loadjava -resolve`.

The compiler writes error messages to the predefined `USER_ERRORS` view; `loadjava` retrieves and displays the messages produced by its compiler invocations. See the *Oracle8i Reference* for a description of this table.

The compiler recognizes compiler options. There are two ways to specify options to the compiler. If you run `loadjava` with the `-resolve` option (which may trigger compilation), you can specify compiler options on the command line.

You can additionally specify persistent compiler options in a per-schema database table called `JAVA$OPTIONS` which you create as described shortly. You can use the `JAVA$OPTIONS` table for default compiler options, which you can override selectively with a `loadjava` command-line option.

Note: A command-line option both overrides and clears the matching entry in the `JAVA$OPTIONS` table.

A `JAVA$OPTIONS` row contains the names of source schema objects to which an option setting applies; you can use multiple rows to set the options differently for different source schema objects. The compiler looks up options in the `JAVA$OPTIONS` table when it has been invoked without a command line (that is, by the class loader), or when the command line does not specify an option. When compiling a source schema object for which there is neither a `JAVA$OPTIONS` entry nor a command line value for an option, the compiler assumes a default value as follows:

- `encoding = latin1`
- `online = true`: see the *Oracle8i SQLJ Developer's Guide and Reference* for a description of this option, which only applies to Java sources that contain SQLJ constructs.

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

- PROCEDURE `set_compiler_option(name VARCHAR2, option VARCHAR2, value VARCHAR2);`

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

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

The `option` parameter is either `'online'` or `'encoding'`. For the values you can specify for these options, see the *Oracle8i SQLJ Developer's Guide and Reference*.

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

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

[Table A-1](#) represents a hypothetical `JAVA$OPTIONS` database table. Because the table has no entry for the `encoding` option, the compiler will use the default or the value specified on the command line. The `online` options shown in the table match schema object names as follows:

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

Table A-1 Example JAVA\$OPTIONS Table and Matching Examples

JAVA\$OPTIONS Entries			Match Examples
Name	Option	Value	
<code>a.b.c.d</code>	<code>online</code>	<code>true</code>	<code>a.b.c.d</code> , <code>a.b.c.d.e</code>
<code>a.b</code>	<code>online</code>	<code>false</code>	<code>a.b</code> , <code>a.b.c.x</code>
(empty string)	<code>online</code>	<code>true</code>	<code>a.c</code> , <code>x.y</code>

loadjava

The `loadjava` tool creates schema objects from files and loads them into a schema. Schema objects can be created from Java source, class, and data files. `loadjava` can also create schema objects from SQLJ files; the *Oracle8i SQLJ Developer's Guide and Reference* describes how to use `loadjava` with SQLJ.

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

- `CREATE PROCEDURE` and `CREATE TABLE` privileges to load into your schema.
- `CREATE ANY PROCEDURE` and `CREATE ANY TABLE` privileges to load into another schema.
- `oracle.aurora.security.JServerPermission.loadLibraryInClass.<classname>`. See ["Database Contents and JVM Security"](#) on page 5-3 for more information.

You can execute the `loadjava` tool either through the command line (as described below) or through the `loadjava` method contained within the `DBMS_JAVA` class. To execute within your Java application, do the following:

```
call dbms_java.loadjava('... options...');
```

where the options are the same as specified below. Each option should be separated by a blank. You should not separate the options with a comma. The only exception for this is the `-resolver` option, which contains blanks. For `-resolver`, you should specify all other options first, a comma, then the `-resolver` option with its definition. You should not specify the following options as they relate to the database connection for the `loadjava` command-line tool: `-thin`, `-oci8`, `-user`, `-password`. The output is directed to `stderr`. Set `serveroutput` on and call `dbms_java.set_output` as appropriate.

Note: The `loadjava` tool is located in the `bin` subdirectory under `$ORACLE_HOME`.

Syntax

```
loadjava {-user | -u} <user>/<password>[@<database>] [options]
<file>.java | <file>.class | <file>.jar | <file>.zip |
<file>.sqlj | <resourcefile>} ...
[-debug]
[{-d | -definer}]
[{-e | -encoding} <encoding_scheme>]
[{-f | -force}]
[{-g | -grant} <user> [, <user>]...]
```

```

[{-o | -oci8}]
[ -order ]
[-noverify]
[{-r | -resolve}]
[{-R | -resolver} "resolver_spec"]
[{-S | -schema} <schema>]
[ -stdout ]
[{-s | -synonym}]
[{-t | -thin}]
[{-v | -verbose}]

```

Argument Summary

Table A-2 summarizes the `loadjava` arguments. If you execute `loadjava` multiple times specifying the same files and different options, the options specified in the most recent invocation hold. There are two exceptions:

1. If `loadjava` does not load a file because it matches a digest table entry, most options on the command line have no effect on the schema object. The exceptions are `-grant`, and `-resolve`, which are always obeyed. Use the `-force` option to direct `loadjava` to skip the digest table lookup.
2. The `-grant` option is cumulative; every user specified in every `loadjava` invocation for a given class in a given schema has the EXECUTE privilege. You cannot grant to a role; you can only grant to specified schemas or users.

Table A-2 *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 in any order.
-debug	Turns on SQL logging and is equivalent to <code>javac -g</code> .
-definer	By default, class schema objects run with the privileges of their invoker. This option confers definer (the developer who invokes <code>loadjava</code>) privileges upon classes instead. (This option is conceptually similar to the UNIX <code>setuid</code> facility.)
-encoding	Identifies the source file encoding for the compiler, overriding the matching value, if any, in the <code>JAVA\$OPTIONS</code> table. Values are the same as for the <code>javac -encoding</code> option. If you do not specify an encoding on the command line or in a <code>JAVA\$OPTIONS</code> table, the encoding is assumed to be <code>latin1</code> . The <code>-encoding</code> option is relevant only when loading a source file.

Table A-2 loadjava Argument Summary (Cont.)

Argument	Description
-force	Forces files to be loaded even if they match digest table entries.
-grant	<p>Grants the EXECUTE privilege on loaded classes to the listed users. (To call the methods of a class, users must have the EXECUTE privilege.) Any number and combination of user names can be specified, separated by commas but not spaces (-grant Bob,Betty not -grant Bob, Betty). Note: -grant is a “cumulative” option; users are added to the list of those with the EXECUTE privilege. To remove privileges, either drop and reload the schema object with the desired privileges or change the privileges with the SQL REVOKE command. Also, you cannot grant to a role. All grants must be explicit in granting to specific users.</p> <p>To grant the EXECUTE privilege on an object in someone else’s schema requires that the original CREATE PROCEDURE privilege was granted with WITH GRANT options.</p>
-noverify	<p>Causes the classes to be loaded without bytecode verification. You must be granted oracle.aurora.security.JServerPermission(Verifier) to execute this option. In addition, this option must be used in conjunction with -r.</p>
-oci8	Directs loadjava to communicate with the database using the OCI JDBC driver. -oci8 and -thin are mutually exclusive; if neither is specified -oci8 is used by default. Choosing -oci8 implies the syntax of the -user value. You do not need to provide the URL.
-order	Directs loadjava load the classes in an order that facilitates resolution of those classes. Classes are loaded in a manner where any dependent class is loaded before the class that includes it as a dependency.
-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 spec, which is bound to the newly loaded classes. If -resolver is not specified, the default resolver spec, which includes current user’s schema and PUBLIC, is used. See "resolver" in this section for details.

Table A-2 loadjava Argument Summary (Cont.)

Argument	Description
-schema	Designates the schema where schema objects are created. If not specified, the logon 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 <code>stdout</code> , rather than to <code>stderr</code> .
-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 <code>-synonym</code> is specified for source files, classes compiled from the source files are treated as if they had been loaded with <code>-synonym</code> .
-thin	Directs <code>loadjava</code> to communicate with the database using the thin JDBC driver. <code>-oci8</code> and <code>-thin</code> are mutually exclusive; if neither is specified, then <code>-oci8</code> is used by default. Choosing <code>-thin</code> implies the syntax of the <code>-user</code> value. You do need to specify the appropriate URL through the <code>-user</code> option.
-user	Specifies a user, password, and database connect string; the files will be loaded into this database instance. The argument has the form <code><username>/<password>[@<database>]</code> .
-verbose	Directs <code>loadjava</code> to emit detailed status messages while running. Use <code>-verbose</code> to learn when <code>loadjava</code> does not load a file because it matches a digest table entry.

Argument Details

This section describes the details of `loadjava` arguments whose behavior is more complex than the summary descriptions contained in [Table A-2](#).

File Names

You can specify as many `.class`, `.java`, `.sqlj`, `.jar`, `.zip`, and resource files as you like, in any order. If you specify a JAR or ZIP file, then `loadjava` processes the files in the JAR or ZIP; there is no JAR or ZIP schema object. If a JAR or ZIP contains a JAR or ZIP, `loadjava` does not process them.

The best way to load files is to put them in a JAR or ZIP and then load the archive. Loading archives avoids the resource schema object naming complications

described later in this section. If you have a JAR or ZIP that works with the JDK, then you can be sure that loading it with `loadjava` will also work, without having to learn anything about resource schema object naming.

Schema object names are slightly different from file names, and `loadjava` names different types of schema objects differently. Because class files are self-identifying (they contain their names), `loadjava`'s mapping of class file names to schema object names is invisible to developers. Source file name mapping is also invisible to developers; `loadjava` gives the schema object the fully qualified name of the first class defined in the file. JAR and ZIP files also contain the names of their files; however, resource files are not self-identifying. `loadjava` generates Java resource schema object names from the *literal* names you supply as arguments (or the literal names in a JAR or ZIP file). Because running classes use resource schema objects, it is important that you specify resource file names correctly on the command line, and the correct specification is not always intuitive. The surefire way to load individual resource files correctly is:

Run `loadjava` from the top of the package tree and specify resource file names relative to that directory. (The “top of the package tree” is the directory you would name in a Java CLASSPATH list.)

If you do not want to follow this rule, observe the details of resource file naming that follow. When you load a resource file, `loadjava` generates the resource schema object name from the resource file name *as literally specified on the command line*. Suppose, for example you type:

```
% cd /home/scott/javastuff
% loadjava options alpha/beta/x.properties
% loadjava options /home/scott/javastuff/alpha/beta/x.properties
```

Although you have specified the same file with a relative and an absolute path name, `loadjava` creates *two* schema objects, one called `alpha/beta/x.properties`, the other `ROOT/home/scott/javastuff/alpha/beta/x.properties`. (`loadjava` prepends `ROOT` because schema object names cannot begin with the “/” character; however, that is an implementation detail that is unimportant to developers.) The important point is that a resource schema object's name is generated from the file name *as entered*.

Classes can refer to resource files relatively (for example, `b.properties`) or absolutely (for example, `/a/b.properties`). To ensure that `loadjava` and the class loader use the same name for a schema object, follow this rule when loading resource files:

Enter the name on the command line that the class passes to `getResource()` or `getResourceAsString()`.

Instead of remembering whether classes use relative or absolute resource names and changing directories so that you can enter the correct name on the command line, you can load resource files in a JAR as follows:

```
% cd /home/scott/javastuff
% jar -cf alpharesources.jar alpha/*.properties
% loadjava options alpharesources.jar
```

Or, to simplify further, put both the class and resource files in a JAR, which makes the following invocations equivalent:

```
% loadjava options alpha.jar
% loadjava options /home/scott/javastuff/alpha.jar
```

The two `loadjava` commands in this example make the point that you can use any pathname to load the contents of a JAR file. Note as well that even if you did execute the redundant commands shown above, `loadjava` would realize from the digest table that it did not need to load the files twice. That means that re-loading JAR files is not as time-consuming as it might seem even when few files have changed between `loadjava` invocations.

definer

```
{-definer | -d}
```

The `-definer` option is identical to `definer`'s rights in stored procedures and is conceptually similar to the UNIX `setuid` facility; however, whereas `setuid` applies to a complete program, you can apply `-definer` class by class. Moreover, different `definers` may have different privileges. Because an application may consist of many classes, you must apply `-definer` with care to achieve the results desired, namely classes that run with the privileges they need but no more. For more information on `definer`'s rights, see the *Oracle8i Java Stored Procedures Developer's Guide*.

noverify

```
{-noverify}
```

Causes the classes to be loaded without bytecode verification. You must be granted `oracle.aurora.security.JServerPermission(Verifier)` to execute this option. In addition, this option must be used in conjunction with `-r`.

The verifier ensures that incorrectly formed Java binaries cannot be loaded for execution in the server. If you know that the JAR or classes you are loading are valid, use of this option will speed up the `loadjava` process. Some JServer-specific

optimizations for interpreted performance are put in place during the verification process. Thus, interpreted performance of your application may be adversely affected by using this option.

resolve

```
{-resolve | -r}
```

Use `-resolve` to force `loadjava` to compile (if necessary) and resolve a class that has previously been loaded. It is not necessary to specify `-force` because resolution is performed after, and independently of, loading.

resolver

```
{-resolver | -R} "resolver spec"
```

This option associates an explicit resolver spec with the class schema objects that `loadjava` creates or replaces.

A resolver spec consists of one or more items, each of which consists of a *name spec* and a *schema spec* expressed in the following syntax:

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

- A name spec is similar to a name in a Java `import` statement. It can be a fully qualified Java class name, or a package name whose final element is the wildcard character “*”, or (unlike an imported package name) simply the wildcard character “*”; however, the elements of a name spec must be separated by “/” characters, not periods. For example, the name spec `a/b/*` matches all classes whose names begin with `a.b..` The special name `*` matches all class names.
- A schema spec can be a schema name or the wildcard character “-”. The wildcard does not identify a schema but directs the resolve operation to not mark a class invalid because a reference to a matching name cannot be resolved. (Without a “-” wildcard in a resolver spec, an unresolved reference in the class makes the class invalid and produces an error message.) Use a “-” wildcard when you must test a class that refers to a class you cannot or do not want to load; for example, GUI classes that a class refers to but does not call because when run in the server there is no GUI.

The resolution operation interprets a resolver spec item as follows:

When looking for a schema object whose name matches the name spec, look in the schema named by the partner schema spec.

The resolution operation searches schemas in the order in which the resolver spec lists them. For example,

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

means the following:

Search for any reference first in SCOTT and then in PUBLIC. If a reference is not resolved, then mark the referring class invalid and display an error message; in other words, call attention to missing classes.

The following example:

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

means the following:

Search for any reference first in SCOTT and then in PUBLIC. If the reference is not found, and is to a class in the package my.gui then mark the referring class valid, and do not display an error; in other words, ignore missing classes in this package. If the reference is not found and is not to a class in my.gui, then mark the referring class invalid and produce an error message.

user

```
{-user | -u} <user>/<password>[@<database>]
```

By default, `loadjava` loads into the login schema specified by the `-user` option. Use the `-schema` option to specify a different schema to load into. This does not involve a login into that schema, but does require that you have sufficient permissions to alter it.

The permissible forms of `@<database>` depend on whether you specify `-oci8` or `-thin`; `-oci8` is the default.

- `-oci8`: `@<database>` is optional; if you do not specify, `loadjava` uses the user's default database. If specified, `<database>` can be a TNS name or a Net8 name-value list.
- `-thin`: `@<database>` is required. The format is `<host>:<lport>:<SID>`.
 - `<host>` is the name of the machine running the database.
 - `<lport>` is the listener port that has been configured to listen for Net8 connections; in a default installation, it is 5521.
 - `<SID>` is the database instance identifier; in a default installation it is ORCL.

Here are examples of `loadjava` commands:

- Connect to the default database with the default `oci8` driver, load the files in a JAR into the TEST schema, then resolve them.

```
loadjava -u joe/shmoe -resolve -schema TEST ServerObjects.jar
```

- Connect with the thin driver, load a class and a resource file, and resolve each class:

```
loadjava -thin -u scott/tiger@dbhost:5521:orcl \  
-resolve alpha.class beta.props
```

- Add Betty and Bob to the users who can execute `alpha.class`:

```
loadjava -thin -schema test -u scott/tiger@localhost:5521:orcl \  
-grant Betty,Bob alpha.class
```

dropjava

The `dropjava` tool is the converse of `loadjava`. It transforms command-line file names and JAR or ZIP file contents to schema object names, then drops the schema objects and deletes their corresponding digest table rows. You can enter `.java`, `.class`, `.sqlj`, `.ser`, `.zip`, `.jar`, and resource file names on the command line in any order.

Alternatively, you can specify a schema object name (full name, not short name) directly to `dropjava`. A command-line argument that does not end in `.jar`, `.zip`, `.class`, `.java`, or `.sqlj` is presumed to be a schema object name. If you specify a schema object name that applies to multiple schema objects (such as a source schema object `FOO` and a class schema object `FOO`), all will be removed.

Dropping a class invalidates classes that depend on it, recursively cascading upwards. Dropping a source drops classes derived from it.

Note: You must remove Java schema objects in the same way that you first loaded them. If you load a `.sqlj` source file and translate it in the server, you must run `dropjava` on the same source file. If you translate on a client and load classes and resources directly, run `dropjava` on the same classes and resources.

You can execute the `dropjava` tool either through the command line (as described below) or through the `dropjava` method contained within the `DBMS_JAVA` class. To execute within your Java application, do the following:

```
call dbms_java.dropjava('... options...');
```

where the options are the same as specified below. Each option should be separated by a blank. You should not separate the options with a comma. The only exception for this is the `-user` option. The connection is always made to the current session, so you cannot specify another username through the `-user` option.

For `-resolver`, you should specify all other options first, a comma, then the `-resolver` option with its definition. You should not specify the following options as they relate to the database connection for the `loadjava` command-line tool: `-thin`, `-oci8`, `-user`, `-password`. The output is directed to `stderr`. Set `serveroutput` on and call `dbms_java.set_output` as appropriate.

Syntax

```
dropjava {-u | -user} <user>/<password>[@<database>] [options]
{<file>.java | <file>.class | file.sqlj |
<file>.jar | <file.zip> | <resourcefile>} ...
  [{-o | -oci8}]
  [{-S | -schema} <schema>]
  [ -stdout ]
  [{-s | -synonym}]
  [{-t | -thin}]
  [{-v | -verbose}]
```

Argument Summary

[Table A-3](#) summarizes the `dropjava` arguments.

Table A-3 *dropjava* Argument Summary

Argument	Description
<code>-user</code>	Specifies a user, password, and optional database connect string; the files will be dropped from this database instance.
<code><filenames></code>	You can specify any number and combination of <code>.java</code> , <code>.class</code> , <code>.sqlj</code> , <code>.ser</code> , <code>.jar</code> , <code>.zip</code> , and resource file names in any order.
<code>-oci8</code>	Directs <code>dropjava</code> to connect with the database using the <code>oci8</code> JDBC driver. <code>-oci8</code> and <code>-thin</code> are mutually exclusive; if neither is specified, then <code>-oci8</code> is used by default. Choosing <code>-oci8</code> implies the form of the <code>-user</code> value.
<code>-schema</code>	Designates the schema from which schema objects are dropped. If not specified, the logon schema is used. To drop a schema object from a schema that is not your own, you need the <code>DROP ANY PROCEDURE</code> and <code>UPDATE ANY TABLE</code> privileges.

Table A-3 *dropjava Argument Summary (Cont.)*

Argument	Description
-stdout	Causes the output to be directed to <code>stdout</code> , rather than to <code>stderr</code> .
-synonym	Drops a PUBLIC synonym that was created with <code>loadjava</code> .
-thin	Directs <code>dropjava</code> to communicate with the database using the thin JDBC driver. <code>-oci8</code> and <code>-thin</code> are mutually exclusive; if neither is specified, then <code>-oci8</code> is used by default. Choosing <code>-thin</code> implies the form of the <code>-user</code> value.
-verbose	Directs <code>dropjava</code> to emit detailed status messages while running.

Argument Details

File Names

`dropjava` interprets most file names as `loadjava` does:

- `.class` files: `dropjava` finds the class name in the file and drops the corresponding schema object.
- `.java` and `.sqlj` files: `dropjava` finds the first class name in the file and drops the corresponding schema object.
- `.jar` and `.zip` files: `dropjava` processes the archived file names as if they had been entered on the command line.

If a file name has another extension or no extension, then `dropjava` interprets the file name as a schema object name and drops all source, class, and resource objects that match the name. For example, the hypothetical file name `alpha` drops whichever of the following exists: the source schema object named `alpha`, the class schema object named `alpha`, and the resource schema object named `alpha`. If the file name begins with the “/” character, then `dropjava` prepends `ROOT` to the schema object name.

If `dropjava` encounters a file name that does not match a schema object, it displays a message and processes the remaining file names.

user

```
{-user | -u} <user>/<password>[@<database>]
```

The permissible forms of `@<database>` depend on whether you specify `-oci8` or `-thin`; `-oci8` is the default.

- `-oci8: @<database>` is optional; if you do not specify, then `dropjava` uses the user's default database. If specified, then `<database>` can be a TNS name or a Net8 name-value list.
- `-thin: @<database>` is required. The format is `<host>: <lport>: <SID>`.
 - `<host>` is the name of the machine running the database.
 - `<lport>` is the listener port that has been configured to listen for Net8 connections; in a default installation, it is 5521.
 - `<SID>` is the database instance identifier; in a default installation, it is ORCL.

Here are some `dropjava` examples.

- Drop all schema objects in schema `TEST` in the default database that were loaded from `ServerObjects.jar`:

```
dropjava -u scott/tiger -schema TEST ServerObjects.jar
```

- Connect with the thin driver, then drop a class and a resource file from the user's schema:

```
dropjava -thin -u scott/tiger@dbhost:5521:orcl alpha.class beta.props
```

Dropping Resources

Care must be taken if you are removing a resource that was loaded directly into the server. This includes profiles if you translated on the client without using the `-ser2class` option. When dropping source or class schema objects, or resource schema objects that were generated by the server-side SQLJ translator, the schema objects will be found according to the package specification in the applicable `.sqlj` source file. However, the fully qualified schema object name of a resource that was generated on the client and loaded directly into the server depends on path information in the `.jar` file or on the command line at the time you loaded it. If you use a `.jar` file to load resources and use the same `.jar` file to remove resources, there will be no problem. If, however, you use the command line to load resources, then you must be careful to specify the same path information when you run `dropjava` to remove the resources.

Glossary

API

Application Programming Interface. As applied to Java, a well-defined set of classes and methods that furnish a specific set of functionality to the Java programmer. JDBC and SQLJ are APIs for accessing SQL data.

Bytecodes

The set of single-byte, machine-independent instructions to which Java source code is compiled using the Java compiler.

Call Memory

The memory that the memory manager uses to allocate new objects.

CLASSPATH

The environment variable (or command line argument) the JDK or JRE uses to specify the set of directory tree roots in which Java source, classes, and resources are located.

Context switch

In a uniprocessor system, the current thread is interrupted by a higher priority thread or by some external event, and the system switches to a different thread. The choice of which thread to dispatch is usually made on a priority basis or based on how long a thread has been waiting.

Cooperative Multitasking

The programmer places calls to the `Thread.yield()` method in locations in the code where it is appropriate to suspend execution so that other threads can run.

This is quite error-prone because it is often difficult to assess the concurrent behavior of a program as it is being written.

CORBA

Common Object Request Broker Architecture. Specified by the Object Management Group (OMG), CORBA provides a language-independent architecture for distributing object-oriented programming logic between logical and physical tiers in a network, connected through ORBs.

Core Class Libraries

Generally, the Java packages delivered with Sun Microsystem's JDK, java.*. We also use this term to denote some sun.* packages.

Deadlock

The conflict state where two or more synchronized Java objects depend on locking each other but cannot because they themselves are locked by the dependent object. For example, object A tries to lock object B while object B is trying to lock object A. This situation is difficult to debug because a preemptive Java virtual machine can neither detect nor prevent deadlock. Without deadlock detection, a deadlocked program simply hangs.

Dispatch

The system saves the state of the currently executing thread, restores the state of the thread to be executed, and branches to the stored program counter for the new thread, effectively continuing the new thread as if it had not been interrupted.

Driver

As used with JDBC, a layer of code that determines the low-level libraries employed to access SQL data and/or communicate across a network. The three JDBC drivers in JServer are: thin, OCI, and server.

EJB

Enterprise JavaBeans. JServer provides an implementation of the Enterprise JavaBeans 1.0 Specification. JServer supports only Session Beans; Entity Beans are an optional part of the EJB 1.0 Specification, and JServer does not support them.

End-of-Call

Within your session, you may invoke Java many times. Each time you perform this, end-of-call occurs at the point at which Java code execution completes. The memory manager migrates static variables to session space at end-of-call.

Garbage Collection

The popular name for the automatic storage reclamation facility provided by the Java virtual machine.

IDE

Integrated Development Environment. A Java IDE runs on a client workstation, providing a graphical user interface for access to the Java class library and development tools.

Interface Definition Language (IDL)

The platform-independent language CORBA specifies for defining the interface to a CORBA component. You use a tool like `idl2java` to convert IDL to Java code.

Java Schema Object

The term JServer uses to denote either Java source, binary, or resources when stored in the Oracle8i database. These three Java schema objects correspond to files under the JDK—`.java`, `.class`, or other files (such as `.properties` files) used in the JDK CLASSPATH.

JCK

Java Compatibility Kit. The set of Java classes that test a Java virtual machine and Java compiler's compliance with the Java standard. JCK releases correspond to Sun Microsystem's JDK releases, although in the case of JServer, only the Java classes and not the virtual machine, are identical to Sun Microsystem's JDK.

JDBC

Java Database Connectivity. The standard Java classes that provide vendor-independent access to databases.

JDBC Driver

The vendor-specific layer of JDBC that provides access to a particular database. Oracle provides three JDBC drivers—`thin`, `OCI`, and `server`.

JDK

Java Development Kit. The Java virtual machine, together with the set of Java classes and tools Sun Microsystems furnishes to support Java application and applet development. The JDK includes a Java compiler; the JRE does not.

JRE

Java Runtime Environment. The set of Java classes supporting a Java application or applet at runtime. The JRE classes are a subset of the JDK classes.

JServer

Oracle's scalable Java server platform, composed of the Aurora Java virtual machine running within the Oracle8i database server, the Java runtime environment and Oracle extensions, including the Aurora/ORB and Enterprise JavaBeans implementation.

Lazy Initialization

A technique for initializing data, typically used in accessor methods. The technique checks to see if a field has been initialized (is non-null) before returning the initialized object to it. The overhead associated with the check is often small, especially in comparison to initializing a data structure that may never be accessed. You can employ this technique in conjunction with end-of-call processing to minimize session space overhead.

Object Graph

An object is said to reference the objects held in its fields. This collection of objects forms an object graph. The memory manager actually migrates the object graphs held in static variables; that is, it migrates not only the objects held in static fields, but the objects that those objects reference, and so on.

ORB

Object Request Broker. An ORB is a program that executes on the server, receiving encoded messages from clients for execution by server-side objects and returning objects to the client. ORBs typically support different services that clients can use, such as a name service. The JServer ORB is known as the Aurora/ORB.

Preemptive Multitasking

The operating system preempts, or takes control away from a thread, under certain conditions, such as when another thread of higher priority is ready to run, or when an external interrupt occurs, or when the current thread waits on an I/O operation, such as a socket accept or a file read. Some Java virtual machines implement a type of round-robin preemption by preempting the current thread on certain virtual machine instructions, such as backward branches, method calls, or other changes in control flow. For a Java virtual machine that maps Java threads to actual operating system threads, the preemption takes place in the operating system kernel, outside

the control of the virtual machine. Although this yields decent parallelism, it complicates garbage collection and other virtual machine activities.

Process

An address space and one or more threads.

Session Memory

The memory that the memory manager uses to hold objects that survive past the end-of-call—those objects reachable from Java static variables within your session.

SQLJ

Embedded SQL in Java. The standard that defines how SQL statements can be embedded in Java programs to access SQL data. A translator transforms the SQLJ programs to standard JDBC programs.

Strong Typing

In Java, the requirement that the class of each field and variable, and the return type of each method be explicitly declared.

Symmetric Multiprocessing (SMP)

The hardware has multiple processors, and the operating system maps threads to different processors depending on their load and availability. This assumes that the Java virtual machine maps OS threads to Java threads. This mechanism provides true concurrency among the threads but can lead to subtle programming errors and deadlock conflicts on synchronized objects.

System

Often used in discussion as the combination of the hardware, the operating system and the Java virtual machine.

Thread

An execution context consisting of a set of registers, a program counter, and a stack.

Virtual Machine

A program that emulates the functionality of a traditional processor. A Java virtual machine must conform to the requirements of the Java Virtual Machine Specification. The JServer virtual machine is known as the Aurora virtual machine.

Symbols

#sql, 3-13, 3-14

A

act method, 5-33

Agent protocol, 3-18

application

 compiling, 2-14

 development, 2-3

 distributed, 1-20

 executing in a session, 2-3

 execution control, 2-6

 execution rights, 2-26

 invoking, 3-3, 3-23

 threading, 2-43

attributes

 definition, 1-3

 types of, 1-4

Aurora

 definition, 2-2

aurora_client.jar file, 3-19

authentication, 5-2

B

BasicPermission, 5-13

bean, 1-20

bytecode

 defined, 1-8

 verification, 2-21

C

call

 definition, 2-2

 managing resources across calls, 2-40

 static fields, 2-5

call specification, 3-4, 3-5

Callback class

 act method, 5-33

class

 attributes, 1-3, 1-5

 definition, 1-2

 dynamic loading, 1-16

 execution, 2-2

 hierarchy, 1-5

 inheritance, 1-5, 1-6

 loading, 2-2, 2-6, 2-22

 marking valid, 2-19

 methods, 1-3, 1-5

 name, 2-31

 publish, 2-2

 resolving references, 2-19

Class class

 getClassLoader method, 2-34

.class files, 2-13, 2-22, 2-23

Class interface

 forName method, 2-32

class schema object, 2-13, 2-19, 2-22, 2-23, A-1, A-3

classes

 loading, 3-2

 protected, 5-25

 publishing, 2-29, 3-2

 resolving, 3-2

classes111.zip, 2-8

- classes12.zip, 2-8
- Class.forName class
 - lookupClass method, 2-35
- class.forNameAndSchema method, 2-34
- ClassNotFoundException, 2-32
- CLASSPATH, 2-13, 2-32, 4-9
- client
 - setup, 4-8
- code
 - native compilation, 5-25
- CodeSource class, 5-5
 - equals method, 5-5
 - implies method, 5-5
- COM, 1-21
- compiling, 2-14
 - error messages, 2-15, A-5
 - memory problems, 5-30
 - options, 2-15, A-5
 - runtime, 2-14
- component, 1-20
- configuration, 4-1
 - JServer, 4-6 to 4-8
 - performance, 5-27
- connection
 - configuration, 4-6
 - security, 5-2
- CORBA
 - configuring, 4-7, 4-8
 - CosNaming, 3-7
 - defined, xiv, 1-17, 1-20, 2-5, 3-2, 3-6
 - documentation, 1-22
 - example, 3-9
 - invoking, 3-2
 - Java 2 support, 2-9
 - pure CORBA using Java 2, 2-12
 - security, 5-2
- CosNaming, 3-7

D

- data confidentiality, 5-2
- database
 - configuration, 4-6
 - privileges, 5-3
- DBA_JAVA_POLICY view, 5-6, 5-17, 5-19

- DBMS_JAVA package, 3-20, 4-3
 - defined, 5-5
 - delete_permission method, 4-5, 5-18
 - disable_permission method, 4-5, 5-17
 - dropjava method, 4-4
 - enable_permission method, 4-5, 5-18
 - get_compiler_option method, 4-4
 - grant_permission method, 4-5, 5-8, 5-9
 - grant_policy_permission method, 4-5, 5-11, 5-19
 - loadjava method, 4-4
 - longname method, 2-28, 2-31, 4-3
 - manipulating security, 2-9
 - modifying permissions, 5-18
 - modifying PolicyTable permissions, 5-9, 5-11
 - reset_compiler_option method, 4-4
 - restart_debugging method, 3-21, 4-5
 - restrict_permission method, 4-5, 5-9, 5-10
 - revoke_permission method, 4-5, 5-17
 - set_compiler_option method, 4-4
 - set_output method, 3-23, 4-4
 - setting permissions, 5-6
 - shortname method, 2-28, 2-31, 4-3
 - start_debugging method, 3-20, 4-5
 - stop_debugging method, 4-5
- DBMS_OUTPUT package, 4-4
- DbmsJava class, see DBMS_JAVA package
- DbmsObjectInputStream class, 2-35
- DbmsObjectOutputStream class, 2-35
- deadlock, 2-44
- DeadlockError, 2-44
- DeadlockError exception, 2-44
- debug
 - compiler option, 2-16
 - loadjava option, A-8
- DebugAgent class, 3-19
- debugging, 4-5, 5-24
 - agent, 3-19, 3-20
 - connecting a debugger, 3-21
 - Java applications, 3-18
 - necessary permissions, 5-24
 - starting Debug Agent, 3-20
 - starting proxy, 3-19
 - using JDeveloper, 3-18
 - using OracleAgent class, 3-21
- DebugProxy class, 3-18, 3-19

- debugproxy command, 3-20
- definer
 - loadjava option, A-8, A-10, A-12
- definer rights, 2-26
- delete method, 5-18
- delete_permission method, 4-5, 5-18
- deployejb tool, 2-30
- digest table, A-4
- disable method, 5-17
- disable_permission method, 4-5, 5-17
- distributed objects, 1-20, 3-6
- documentation, 1-1, 1-21
- dropjava method, 4-4
- dropjava tool, 2-24, A-15

E

EJB

- component, 1-20
- configuring, 4-7, 4-8
- defined, xiv, 1-17, 1-20, 2-5, 3-2, 3-6
- documentation, 1-22
- example, 3-8
- invoking, 3-2
- security, 5-2
- state, 1-20
- enable method, 5-18
- enable_permission method, 4-5, 5-18
- encoding
 - compiler option, 2-16, A-5
 - loadjava option, A-8
- end-of-call migration, 5-31
- EndOfCallRegistry class, 5-32
- registerCallback method, 5-33
- Enterprise Java Beans, see EJB
- equals method, 5-5
- errors
 - compilation, 2-15
- exception, 2-44
 - ClassNotFoundException, 2-32
 - IOException, 2-38
 - LimboError, 2-44
 - ThreadDeathException, 2-45
- execution rights, 2-26
- exitCall method, 2-44

- exitSession method, 2-5, 2-44

F

file names

- dropjava, A-17
- loadjava, A-10
- FilePermission, 5-8, 5-18, 5-20, 5-22
- files, 2-37
 - across calls, 2-40
 - lifetime, 2-38
- finalizers, 2-40
- footprint, 1-14, 2-4
- force
 - loadjava option, A-9
- forName method, 2-32

G

- garbage collection, 1-13, 1-14, 2-5
 - managing resources, 2-37
 - misuse, 2-39
 - purpose, 2-39
- General Inter-Orb Protocol, see GIOP
- get_compiler_option method, 2-16, 4-4, A-6
- getCallerClass method, 2-34
- getClassLoader method, 2-34
- GIOP
 - configuring, 4-6, 4-7
 - presentation, 4-8
- grant
 - loadjava option, A-8, A-9
- grant method, 5-8
- grant_permission method, 4-5, 5-8, 5-9
- grant_policy_permission method, 4-5, 5-11, 5-19
- granting permission, 5-5
- grantPolicyPermission method, 5-12
- GUI, 2-30

I

IIOP

- configuring, 4-7
- defined, 3-7
- SSL, 4-8

- implies method, 5-5
- inheritance, 1-5, 1-6
- init method, 2-10
- initjvm.sql, 4-2, 4-3, 4-6
- installation, 4-1, 4-2
- integrity, 5-2
- interfaces
 - defined, 1-6
 - user, 2-30
- internet newsgroups, xv
- invoker rights, 2-26
- IOException, 2-38

J

Java

- applications, 2-1, 2-14
 - loading, 2-22
- attributes, 1-3
- class, 1-2
- client
 - CLASSPATH, 4-9
 - setup, 4-8
- compiling, 2-14
- development environment, 2-13
- differences from Sun JDK, 2-3
- distributed applications, xiv, 3-2
- documentation, xiv, 1-1, 1-21
- execution control, 2-6
- execution rights, 2-26
- features, 1-11
- in the database, 1-1, 1-12, 2-1, 2-2
- interpreter, 2-2
- introduction, xiii
- invoking, 2-2, 3-3
- loading classes, 2-6, 3-2
 - checking results, 2-27
- methods, 1-3
- overview, 1-1, 1-2
- permissions, 4-5
- polymorphism, 1-6
- programming models, xiv
- publishing, 2-7
- resolving classes, 2-19
- resources, 1-2

- stored procedures, see Java stored procedures
- Java 2
 - migrating from JDK 1.1, 2-7
 - migrating security, 2-9
 - security, 5-3
- Java Compatibility Kit, see JCK
- .java files, 2-13, 2-22, 2-23
- java interpreter, 2-2, 2-6
- Java language specification, see JLS
- Java Naming and Directory Interface, see JNDI
- Java Native Interface, see JNI
- Java Remote Method Invocation, see RMI
- Java stored procedures, xiv, 2-5
 - configuring, 4-7
 - defined, 1-17, 1-18, 3-3
 - documentation, 1-22
 - invoking, 3-2
 - publishing, 2-29
- Java virtual machine, see JVM
- JAVASOPTIONS table, 2-15, A-5
- JAVA_ADMIN role
 - assigned permissions, 5-20
 - example, 5-13
 - granting permission, 5-3, 5-5, 5-11, 5-19
- JAVA_MAX_SESSIONSPACE_SIZE
 - parameter, 5-28
- JAVA_POOL_SIZE parameter
 - default, 4-6
 - defined, 5-27, 5-28
 - errors, 5-30
 - minimum value, 4-3
- JAVA_SOFT_SESSIONSPACE_LIMIT
 - parameter, 5-27
- JAVADEBUGPRIV role, 5-23, 5-24
- java.sql package, 2-8
- JAVASYSPRIV role, 5-4, 5-22, 5-23
- JAVAUSERPRIV role, 5-4, 5-22, 5-23
- JCK, 1-11
- jdb debugging tool, 3-18, 3-22
- JDBC
 - 2.0 support, 2-8
 - accessing SQL, 1-18
 - defined, 1-17, 3-2, 3-11
 - documentation, 1-22
 - driver types, 1-19, 3-11

- example, 3-12
- interacting with SQL, 3-17
- security, 5-2
- web information, xiv

JDeveloper

- debugging, 3-18
- development environment, 1-21, 3-16, 4-10

JDK

- requirements, 4-8
- web location, xv

JLS

- specification, 1-11
- web information, xv

JNDI

- defined, 3-7
- name lookup, 3-8

JNI support, 3-10

JPublisher

- documentation, 1-22

JServer

- configure, 4-1
- definition, xv, 2-2
- install, 4-1, 4-2

JServerPermission, 5-8, 5-19, 5-21, 5-22, 5-23

- defined, 5-19

JVM

- bytecodes, 1-8
- defined, 1-2, 1-8
- garbage collection, 1-13, 1-14
- multithreading, 1-13
- responsibilities, 2-4
- security, 4-5
- specification, 1-11
- web information, xv

L

LimboError exception, 2-44

loading, 2-22 to 2-29

- checking results, 2-24, 2-27
- class, 1-16, 2-6, 2-14
- compilation option, 2-14
- granting execution, 2-26
- JAR or ZIP files, 2-25
- necessary privileges and permissions, 2-25

- reloading classes, 2-26
- restrictions, 2-24

loadjava method, 4-4

loadjava tool, 2-23 to 2-25, A-1 to A-2

- compiling source, 2-14, 5-31
- example, 3-4
- execution rights, 2-26, 5-3
- loading class, 2-22
- loading ZIP or JAR files, 2-25
- restrictions, 2-24
- using memory, 5-27

logging, 2-15

longname method, 2-28, 2-31, 4-3

lookupClass method, 2-35

M

main method, 2-6

memory

- across calls, 2-39
- call, 2-5
- java pool, 5-29
- leaks, 2-39
- lifetime, 2-37, 2-38
- manager, 2-14
- performance configuration, 5-27
- running out of, 5-30
- session, 2-5, 5-33

methods, 1-3, 1-5

multithreading, 1-13

N

name service, 3-7

- CosNaming, 3-7
- JNDI, 3-7

native compilation, 1-15, 5-25

Net8

- configuring, 4-6

NetPermission, 5-8, 5-18, 5-20, 5-21

networking

- configuration, 4-6

noverify

- loadjava option, A-9, A-12

O

object

- full to short name conversion, 2-28
- lifetime, 2-38
- schema, 2-13
- serialization, 2-35
- short name, 2-28

ObjectInputStream class, 2-35

ObjectOutputStream class, 2-35

oci8

- dropjava option, A-16
- loadjava option, A-9

online

- compiler option, 2-16, A-5

operating system

- resources, 2-37
 - across calls, 2-40
 - lifetime, 2-38
 - performance, 5-26
 - permission, 2-38

OracleAgent class

- restart method, 3-21
- start method, 3-21
- stop method, 3-21

oracle.jdbc2 package, 2-8

OracleRuntime class

- exitCall method, 2-44
- exitSession method, 2-5, 2-44
- getCallerClass method, 2-34

Orb Class

- init method, 2-10

ORBClass property, 2-12

ORBSingletonClass property, 2-12

order

- loadjava flag, A-9

output

- redirecting, 3-23

P

packages

- DBMS_JAVA, 4-3
- oracle.jdbc2, 2-8
- protected, 5-25

performance, 1-15, 5-25 to 5-37

Permission class, 5-7, 5-13, 5-18

permissions, 4-5, 5-3 to 5-25

- administrating, 5-11
- assigning, 5-4, 5-6
- creating, 5-13
- deleting, 5-18
- disabling, 5-17
- enabling, 5-17
- granting, 5-5, 5-8, 5-9
- granting policy, 5-11
- grouped into roles, 5-23
- JAVA_ADMIN role, 5-20
- JAVADEBUGPRIV role, 5-23
- JAVASYSPRIV role, 5-22
- JAVAUSERPRIV role, 5-22
- PUBLIC, 5-22
- restricting, 5-5, 5-9, 5-10
- specifying policy, 5-4
- SYS permissions, 5-21
- types, 5-8, 5-18

policy table

- managing, 5-11
- modifying, 5-6
- setting permissions, 5-6
- viewing, 5-6

PolicyTable class

- specifying policy, 5-4
- updating, 5-5, 5-14

PolicyTableManager class

- delete method, 5-18
- disable method, 5-17
- enable method, 5-18
- revoke method, 5-17

PolicyTablePermission, 5-8, 5-11, 5-19, 5-21, 5-22

polymorphism, 1-6

presentation

- compatibility for RMI, 3-10

presentation layer

- GIOP, 4-8

privileges

- database, 5-3

.properties files, 2-13, 2-22, 2-24

property

- ORBClass, 2-12

- ORBSingletonClass, 2-12
- PropertyPermission, 5-8, 5-18, 5-20, 5-21, 5-22, 5-24
- PUBLIC permissions, 5-22
- publish tool, 2-30
- publishing, 2-7, 2-14, 2-29, 3-2
 - example, 3-4

R

- ReflectPermission, 5-8, 5-18, 5-21
- registerCallback method, 5-33
- requirements
 - JDK version, 4-8
- reset_compiler_option method, 2-16, 4-4, A-6
- resolve
 - loadjava option, A-3, A-9, A-13
- resolver, 2-19 to 2-22, A-3
 - default, 2-20
 - defined, 2-13, 2-14, 2-20, 2-32, 3-2
 - example, 3-4
 - ignoring non-existent references, 2-20, 2-22
 - loadjava option, A-9, A-13
 - spec, A-2
- resource schema object, 2-13, 2-22, 2-24, A-1
- restart method, 3-21
- restart_debugging method, 3-21, 4-5
- restrict method, 5-9
- restrict_permission method, 4-5, 5-9, 5-10
- revoke method, 5-17
- revoke_permission method, 4-5, 5-17
- RMI
 - support, 3-10
- RuntimePermission, 5-8, 5-18, 5-20, 5-21, 5-22, 5-23

S

- schema
 - dropjava option, A-16
 - loadjava option, A-10
- schema object
 - defined, 2-22, A-1
 - name, 2-31
 - using, 2-13
- security, 5-2 to 5-25
 - book recommendations, 5-4

- CORBA, 5-2
- EJB, 5-2
- Java 2, 5-3
- JDBC, 5-2
- JVM, 4-5
 - network, 5-2
 - operating system resources, 2-38
- SecurityManager class, 5-5
- SecurityPermission, 5-8, 5-18, 5-21
- .ser files, 2-13, 2-22, 2-24
- SerializablePermission, 5-8, 5-18, 5-20, 5-22
- serialization, 2-35
- ServerSocket class, 2-42
- session
 - coordination with JVM, 2-4
 - definition, 2-2
 - footprint, 1-14
 - lifetime, 2-5
 - role in Java execution, 2-3
 - timeout, 2-5
- session shell tool, 3-9
- set_compiler_option method, 2-16, 4-4, A-5
- set_output method, 3-23, 4-4
- SHARED_POOL_SIZE parameter
 - default, 4-6
 - defined, 5-27
 - errors, 5-30
 - minimum value, 4-3
- shortname method, 2-28, 2-31, 4-3
- Socket class, 2-42
- SocketPermission, 5-8, 5-18, 5-20, 5-22, 5-23
- sockets
 - across calls, 2-37, 2-42
 - defined, 2-42
 - lifetime, 2-38, 2-42
- source schema object, 2-13, 2-22, 2-23, A-1
 - compiling, A-4
- SQL
 - query, 3-2, 3-11
- SQLJ
 - accessing SQL, 1-18
 - converting, 3-17
 - defined, xiv, 1-17, 1-19, 3-2, 3-11
 - documentation, xiv, 1-22
 - example, 3-13

- interoperates with PL/SQL, 3-17
- running, 3-16
- translating, 3-16
- typing paradigm, 3-15
- using JDBC, 1-19
- .sqlj files, 2-13, 2-22, 2-23
- sqlj utility, 3-16
- SSL, 4-8
 - configuring, 4-6
- start method, 3-21
- start_debugging method, 3-20, 4-5
- static compilation, 5-26
- static variable, 2-5
 - end of call migration, 5-31
- stdout
 - loadjava flag, A-10, A-17
- stop method, 3-21
- stop_debugging method, 3-20, 4-5
- synonym
 - dropjava option, A-17
 - loadjava option, A-10
- SYS
 - assigned permissions, 5-21
 - security permissions, 5-19
- System class
 - getProperty method, 3-23

T

- thin
 - dropjava option, A-17
 - loadjava option, A-10
- ThreadDeathException, 2-45
- threading
 - applications, 2-43
 - lifecycle, 2-44
 - model, 1-13, 2-43
 - using in JServer, 2-37
- timeout, 2-5
- trigger
 - using Java stored procedures, 3-3

U

- user

- dropjava option, A-16, A-17
- loadjava option, A-10, A-14
- user interface, 2-30
- USER_ERRORS view, 2-15
- USER_JAVA_POLICY view, 5-6, 5-19
- USER_OBJECTS view, 2-24, 2-27, 4-4

V

- V\$SGASTAT table, 5-29
- variables
 - static, 2-5
- verbose
 - dropjava option, A-17
- version
 - retrieving, 3-23
 - Visigenic, 2-9
- Visigenic
 - version supported, 2-9

W

- web sites, xiv