

Oracle® Database

Java Developer's Guide,

11g Release 2 (11.2)

E10588-03

December 2009

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

Oracle Database Java Developer's Guide, 11g Release 2 (11.2)

E10588-03

Copyright © 1999, 2009, Oracle and/or its affiliates. All rights reserved.

Primary Author: Tulika Das, Sheryl Maring, Rick Sapir, Michael Wiesenber

Contributing Author: Venkatasubramaniam Iyer, Brian Wright, Timothy Smith

Contributor: Malik Kalfane, Kuassi Mensah, Mark Jungerman, Suresh Srinivasan, Ernest Tucker, Robert H Lee, Dmitry Nizhegorodov, David Unietis, Paul Lo, Steve Harris, Ellen Barnes, Peter Benson, Greg Colvin, Bill Courington, Matthieu Devin, Jim Haungs, Hal Hildebrand, Susan Kraft, Thomas Kurian, Scott Meyer, Tom Portfolio, Dave Rosenberg, Jerry Schwarz, Harlan Sexton, Xuhua Li

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this software or related documentation is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle USA, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

This software is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications which may create a risk of personal injury. If you use this software in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure the safe use of this software. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software in dangerous applications.

Oracle is a registered trademark of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

This software and documentation may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

Contents

Preface	xv
Audience	xv
Documentation Accessibility	xv
Related Documents	xvi
Conventions	xvi
1 Introduction to Java in Oracle Database	
Overview of Java	1-1
Java and Object-Oriented Programming Terminology	1-2
Classes	1-2
Objects	1-2
Interfaces	1-3
Encapsulation	1-3
Inheritance	1-3
Polymorphism	1-4
Key Features of the Java Language	1-5
JVM.....	1-6
Java Class Hierarchy.....	1-7
Using Java in Oracle Database	1-8
Java and RDBMS: A Robust Combination	1-8
Multithreading.....	1-9
Memory Spaces Management	1-10
Footprint.....	1-11
Performance of Oracle JVM.....	1-12
Dynamic Class Loading	1-13
Overview of Oracle JVM	1-14
Process Area.....	1-15
The main() Method	1-15
The GUI	1-15
The IDE	1-15
New Features in This Release	1-15
Internet Protocol Version 6 Support.....	1-16
Feature List of Oracle JVM	1-16
Main Components of Oracle JVM	1-17
Library Manager.....	1-18

Compiler	1-18
Interpreter.....	1-18
Class Loader.....	1-18
Verifier	1-18
Server-Side JDBC Internal Driver	1-19
Server-Side SQLJ Translator	1-19
System Classes	1-19
Java Application Strategy of Oracle	1-19
Java in Database Application Development	1-20
Java Programming Environment	1-20
Java Stored Procedures.....	1-20
PL/SQL Integration and Oracle RDBMS Functionality	1-21
JDBC Drivers	1-21
SQLJ	1-21
JPublisher	1-22
Development Tools.....	1-22
Memory Model for Dedicated Mode Sessions.....	1-22

2 Java Applications on Oracle Database

Database Sessions Imposed on Java Applications	2-1
Execution Control of Java Applications.....	2-3
Java Code, Binaries, and Resources Storage	2-3
Preparing Java Class Methods for Execution.....	2-5
Compiling Java Classes	2-5
Compiling Source Through javac	2-6
Compiling Source Through the loadjava Tool	2-6
Compiling Source at Run Time.....	2-6
Specifying Compiler Options.....	2-6
Recompiling Automatically.....	2-8
Resolving Class Dependencies.....	2-9
Loading Classes.....	2-12
Granting Execute Rights	2-15
Controlling the Current User	2-16
Checking Java Uploads	2-18
Publishing.....	2-19
Auditing.....	2-19
User Interfaces on the Server	2-20
Shortened Class Names.....	2-21
Class.forName() in Oracle Database.....	2-22
Supply ClassLoader in Class.forName().....	2-23
Supply Class and Schema Names to classForNameAndSchema().....	2-24
Supply Class and Schema Names to lookupClass().....	2-24
Supply Class and Schema Names when Serializing.....	2-24
Class.forName Example	2-25
Managing Your Operating System Resources	2-26
Overview of Operating System Resources	2-26
Garbage Collection and Operating System Resources	2-27

Managing Your Applications Using JMX	2-28
Overview of JMX.....	2-28
Enabling and Starting JMX in a Session.....	2-29
OJVM JMX Defaults and Configurability.....	2-30
Examples of SQL calls to dbms_java.start_jmx_agent.....	2-31
Using JConsole to Monitor and Control Oracle JVM.....	2-32
The jconsole Command.....	2-32
The JConsole interface.....	2-32
The OracleRuntime MBean	2-42
Memory Thresholds	2-44
Important Security Notes.....	2-46
Threading in Oracle Database	2-46
Shared Servers Considerations	2-49
End-of-Call Migration	2-49
Oracle-Specific Support for End-of-Call Optimization	2-50
The EndOfCallRegistry.registerCallback() Method	2-53
The EndOfCallRegistry.runCallbacks() Method	2-53
The Callback Interface	2-54
The Callback.act() method	2-54
Operating System Resources Affected Across Calls.....	2-54

3 Calling Java Methods in Oracle Database

Invoking Java Methods	3-1
Using PL/SQL Wrappers.....	3-1
JNI Support	3-3
Utilizing SQLJ and JDBC with Java in the Database.....	3-3
JDBC.....	3-4
SQLJ	3-4
Example Comparing JDBC and SQLJ	3-5
Complete SQLJ Example.....	3-6
SQLJ Strong Typing Paradigm.....	3-7
Translating a SQLJ Program.....	3-7
Running a SQLJ Program in the Server	3-8
Converting a Client Application to Run on the Server	3-8
Interacting with PL/SQL	3-8
Using Command-Line Interface.....	3-9
Using the Client-Side Stub.....	3-10
Debugging Server Applications	3-13
How To Tell You Are Running on the Server	3-13
Redirecting Output on the Server	3-14
Calling Java in the Database Directly	3-20

4 Java Installation and Configuration

Initializing a Java-Enabled Database	4-1
Configuring with Oracle Database Template	4-1
Modifying an Existing Oracle Database to Include Oracle JVM.....	4-1

Configuring Oracle JVM.....	4-2
Using The DBMS_JAVA Package.....	4-2
Enabling the Java Client	4-2
Install J2SE on the Client	4-2
Set Up Environment Variables.....	4-2
Test Install with Samples	4-4
Two-Tier Duration for Java Session State	4-4
Setting System Properties.....	4-5

5 Developing Java Stored Procedures

Stored Procedures and Run-Time Contexts	5-1
Functions and Procedures.....	5-2
Database Triggers.....	5-2
Object-Relational Methods.....	5-2
Advantages of Stored Procedures	5-3
Performance	5-3
Productivity and Ease of Use	5-3
Scalability	5-3
Maintainability	5-4
Interoperability.....	5-4
Replication.....	5-4
Security	5-4
Java Stored Procedures Steps	5-5
Step 1: Create or Reuse the Java Classes	5-5
Step 2: Load and Resolve the Java Classes	5-6
Step 3: Publish the Java Classes	5-6
Step 4: Call the Stored Procedures.....	5-6
Step 5: Debug the Stored Procedures, if Necessary	5-7

6 Publishing Java Classes With Call Specifications

Understanding Call Specifications	6-1
Defining Call Specifications	6-2
Setting Parameter Modes	6-2
Mapping Data Types	6-3
Using the Server-Side Internal JDBC Driver	6-4
Writing Top-Level Call Specifications	6-6
Writing Packaged Call Specifications	6-10
Writing Object Type Call Specifications	6-12
Declaring Attributes	6-13
Declaring Methods.....	6-13
Map and Order Methods	6-14
Constructor Methods.....	6-14
Examples	6-14

7 Calling Stored Procedures

Calling Java from the Top Level.....	7-1
--------------------------------------	-----

Redirecting Output	7-2
Examples of Calling Java Stored Procedures From the Top Level	7-2
Calling Java from Database Triggers	7-4
Calling Java from SQL DML	7-7
Calling Java from PL/SQL	7-8
Calling PL/SQL from Java	7-10
How Oracle JVM Handles Exceptions	7-10
8 Java Stored Procedures Application Example	
Drawing the Entity-Relationship Diagram	8-1
Planning the Database Schema	8-4
Creating the Database Tables	8-4
Writing the Java Classes	8-5
Loading the Java Classes	8-9
Publishing the Java Classes	8-10
Calling the Java Stored Procedures	8-11
9 Oracle Database Java Application Performance	
Oracle JVM Just-in-Time Compiler (JIT)	9-1
Overview of Oracle JVM JIT	9-1
Advantages of JIT Compilation	9-2
Methods Introduced in Oracle Database 11g	9-2
Java Memory Usage	9-4
Configuring Memory Initialization Parameters	9-4
Initializing Pool Sizes within Database Templates	9-5
Java Pool Memory	9-6
Displaying Used Amounts of Java Pool Memory	9-7
Correcting Out of Memory Errors	9-8
Displaying Java Call and Session Heap Statistics	9-8
10 Security for Oracle Database Java Applications	
Network Connection Security	10-1
Database Contents and Oracle JVM Security	10-2
Java2 Security	10-2
Setting Permissions	10-4
Fine-Grain Definition for Each Permission	10-5
General Permission Definition Assigned to Roles	10-17
Debugging Permissions	10-17
Permission for Loading Classes	10-18
Database Authentication Mechanisms	10-18
11 Schema Objects and Oracle JVM Utilities	
Overview of Schema Objects	11-1
What and When to Load	11-2
Resolution of Schema Objects	11-2

Schema Object Digest Table.....	11-3
Compilation of Schema Objects.....	11-4
The ojvmtc Tool.....	11-5
The loadjava Tool.....	11-6
Syntax.....	11-7
Argument Summary.....	11-8
Argument Details.....	11-12
The dropjava Tool.....	11-17
Syntax.....	11-18
Argument Summary.....	11-18
Argument Details.....	11-19
Dropping Resources.....	11-21
The ojvmjava Tool.....	11-21
Syntax.....	11-21
Argument Summary.....	11-21
Example.....	11-22
Functionality.....	11-23
ojvmjava Tool Command-Line Options.....	11-23
Shell Commands.....	11-25

12 Database Web Services

Overview of Database Web Services.....	12-1
Using Oracle Database as Web Services Provider.....	12-1
How to Use JPublisher for Web Services Call-Ins.....	12-2
Features of Oracle Database as a Web Service Provider.....	12-2
JPublisher Support for Web Services Call-Ins to Oracle Database.....	12-3
Using Oracle Database as Web Services Consumer.....	12-3
How to Use Oracle Database for Web Services Call-Outs.....	12-3
Web Service Data Sources (Virtual Table Support).....	12-4
Features of Oracle Database as a Web Service Consumer.....	12-5
Overview of JPublisher Generation.....	12-5
Adjusting the Mapping of SQL Types.....	12-7

A DBMS_JAVA Package

longname.....	A-1
shortname.....	A-1
get_compiler_option.....	A-1
set_compiler_option.....	A-2
reset_compiler_option.....	A-2
resolver.....	A-2
derivedFrom.....	A-2
fixed_in_instance.....	A-3
set_output.....	A-3
export_source.....	A-4
export_class.....	A-4
export_resource.....	A-4
loadjava.....	A-4

dropjava.....	A-5
grant_permission.....	A-5
restrict_permission.....	A-5
grant_policy_permission.....	A-5
revoke_permission.....	A-5
disable_permission.....	A-6
enable_permission.....	A-6
delete_permission.....	A-6
set_preference.....	A-6
runjava.....	A-6
runjava_in_current_session.....	A-6
set_property.....	A-7
get_property.....	A-7
remove_property.....	A-7
show_property.....	A-7
set_output_to_sql.....	A-8
remove_output_to_sql.....	A-8
enable_output_to_sql.....	A-8
disable_output_to_sql.....	A-8
query_output_to_sql.....	A-8
set_output_to_java.....	A-9
remove_output_to_java.....	A-9
enable_output_to_java.....	A-9
disable_output_to_java.....	A-9
query_output_to_java.....	A-10
set_output_to_file.....	A-10
remove_output_to_file.....	A-10
enable_output_to_file.....	A-10
disable_output_to_file.....	A-10
query_output_to_file.....	A-10
enable_output_to_trc.....	A-11
disable_output_to_trc.....	A-11
query_output_to_trc.....	A-11
endsession.....	A-11
endsession_and_related_state.....	A-11
set_native_compiler_option.....	A-11
unset_native_compiler_option.....	A-12
compile_class.....	A-12
compile_class.....	A-12
uncompile_class.....	A-12
uncompile_class.....	A-12
compile_method.....	A-12
compile_method.....	A-13
uncompile_method.....	A-13
uncompile_method.....	A-13
start_jmx_agent.....	A-13

B Classpath Extensions and User Classloaded Metadata

Classpath Extensions	B-1
jserverQuotedClassPathTermPrefix	B-1
jserverURLPrefix	B-1
jserverSpecialTokenPrefix.....	B-1
JSERVER_CP.....	B-2
JSERVER_SCHEMAc.....	B-2
jserver:/CP general syntax	B-2
User Classloaded Metadata	B-3

Index

List of Examples

2-1	Retrieve Resolver from Current Class	2-23
2-2	Retrieve Resolver from Calling Class	2-23
2-3	Providing Schema and Class Names	2-24
2-4	Closing Your Operating System Resources	2-27
2-5	Compromising Your Operating System Resources	2-55
2-6	Correctly Managing Your Operating System Resources	2-55
3-1	Basic Configuration of listener.ora with the Default Service Defined	3-12
6-1	Publishing a Simple JDBC Stored Procedure.....	6-7
6-2	Publishing the main() Method	6-8
6-3	Publishing a Method That Returns an Integer Value	6-8
6-4	Publishing a Method That Switches the Values of Its Arguments	6-9
6-5	Packaged Call Specification.....	6-10
7-1	A Simple JDBC Stored Procedure.....	7-2
7-2	Fibonacci Sequence	7-3
7-3	Calling Java Stored Procedure from Database Trigger - I.....	7-4
7-4	Calling Java Stored Procedure from Database Trigger - II	7-6
10-1	Granting Permissions	10-7
10-2	Limiting Permissions.....	10-7
10-3	Granting PolicyTable Permission	10-9

List of Tables

1-1	Feature List of Oracle JVM	1-16
2-1	Description of Java Code and Classes.....	2-4
2-2	Definitions for the Name and Option Parameters	2-7
2-3	Example JAVA\$OPTIONS Table.....	2-8
2-4	ORA Errors	2-11
2-5	Description of Java Files.....	2-12
2-6	loadjava Operations on Schema Objects.....	2-13
2-7	Key USER_OBJECT Columns	2-18
2-8	Statement Auditing Options Related to Java Schema Objects	2-20
2-9	Object Auditing Options Related to Java Schema Options	2-20
2-10	Description of the Summary Tab Fields in JConsole Interface.....	2-34
3-1	Command Line Argument Summary	3-10
3-2	set_output_to_sql Argument Summary	3-15
3-3	set_output_to_java Argument Summary	3-17
6-1	Legal Data Type Mappings	6-3
10-1	Predefined Permissions.....	10-14
10-2	SYS Initial Permissions.....	10-15
10-3	PUBLIC Default Permissions	10-16
10-4	JAVAUSERPRIV Permissions	10-16
10-5	JAVASYSPRIV Permissions	10-16
10-6	JAVADEBUGPRIV Permissions	10-17
11-1	ojvmtc Argument Summary.....	11-5
11-2	loadjava Argument Summary.....	11-8
11-3	dropjava Argument Summary	11-18
11-4	ojvmjava Argument Summary.....	11-22
11-5	ojvmjava Command Common Options.....	11-26
11-6	java Argument Summary	11-27
11-7	connect Argument Summary	11-28
11-8	runjava Argument Summary	11-29
11-9	jdwp Argument Summary.....	11-29

List of Figures

1-1	Classes and Instances	1-3
1-2	Java Component Structure	1-6
1-3	Oracle Database Java Component Structure.....	1-7
1-4	Class Hierarchy	1-8
1-5	Two-Tier Client/Server Configuration.....	1-9
1-6	Garbage Collection	1-11
1-7	Interpreter versus Accelerator	1-13
1-8	Main Components of Oracle JVM	1-17
2-1	Java Environment Within Each Database Session	2-2
2-2	Loading Java into Oracle Database	2-4
2-3	Rights to Run Classes	2-16
2-4	Invoker's rights Solution.....	2-17
2-5	Indirect Access.....	2-17
2-6	The Summary Tab of the JConsole Interface.....	2-34
2-7	The Memory Tab of the JConsole Interface.....	2-35
2-8	The Threads Tab of the JConsole Interface	2-37
2-9	The Classes tab of the JConsole interface	2-38
2-10	Displaying the Attributes of an MBean	2-39
2-11	Displaying a Chart of the Values of an Attribute.....	2-39
2-12	Displaying Details of a Complex Attribute in the MBeans Tab	2-40
2-13	Operations Tab of the MBeans Tab of the JConsole Interface	2-41
2-14	The VM Tab of the JConsole Interface	2-41
2-15	Attributes Tab of the OracleRuntime MBean	2-42
2-16	OracleRuntime MBean	2-43
2-17	Operation sessionsRunningJava	2-43
2-18	Operation sessionDetailsBySID	2-44
2-19	Setting the Usage Threshold and Collection Usage Threshold in the MBeans Tab.....	2-45
2-20	Memory Tab of the JConsole Interface When Used Memory Exceeds the Usage Threshold..	2-46
3-1	Client-Side Stub Interface	3-11
5-1	Calling a Stored Procedure.....	5-1
6-1	Calling a Java Method	6-2
8-1	Rule for Drawing an E-R Diagram	8-2
8-2	E-R Diagram for Purchase Order Application.....	8-3
8-3	Schema Plan for Purchase Order Application	8-4
9-1	Configuring Oracle JVM Memory Parameters.....	9-6
12-1	Web Services Call-In to the Database.....	12-2
12-2	Calling Web Services From Within the Database	12-4
12-3	Storing Results from Request in a Virtual Table	12-5
12-4	Creating Web Services Call-Out Stubs.....	12-6

Preface

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

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

This preface contains the following topics:

- [Audience](#)
- [Documentation Accessibility](#)
- [Related Documents](#)
- [Conventions](#)

Audience

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

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

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

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible to all users, including users that are disabled. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Accessibility standards will continue to evolve over time, and Oracle is actively engaged with other market-leading

technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For more information, visit the Oracle Accessibility Program Web site at <http://www.oracle.com/accessibility/>.

Accessibility of Code Examples in Documentation

Screen readers may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, some screen readers may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation

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

Deaf/Hard of Hearing Access to Oracle Support Services

To reach Oracle Support Services, use a telecommunications relay service (TRS) to call Oracle Support at 1.800.223.1711. An Oracle Support Services engineer will handle technical issues and provide customer support according to the Oracle service request process. Information about TRS is available at <http://www.fcc.gov/cgb/consumerfacts/trs.html>, and a list of phone numbers is available at <http://www.fcc.gov/cgb/dro/trsphonebk.html>.

Related Documents

For more information, refer to the following Oracle resources:

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

Conventions

The following conventions are also used in this manual:

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

Introduction to Java in Oracle Database

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

This chapter contains the following sections:

- [Overview of Java](#)
- [Using Java in Oracle Database](#)
- [Overview of Oracle JVM](#)
- [New Features in This Release](#)
- [Feature List of Oracle JVM](#)
- [Main Components of Oracle JVM](#)
- [Java Application Strategy of Oracle](#)
- [Memory Model for Dedicated Mode Sessions](#)

Overview of Java

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

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

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

This section covers the following topics:

- [Java and Object-Oriented Programming Terminology](#)
- [Key Features of the Java Language](#)
- [JVM](#)
- [Java Class Hierarchy](#)

Java and Object-Oriented Programming Terminology

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

- [Classes](#)
- [Objects](#)
- [Interfaces](#)
- [Encapsulation](#)
- [Inheritance](#)
- [Polymorphism](#)

Classes

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

- **Attributes**

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

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

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

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

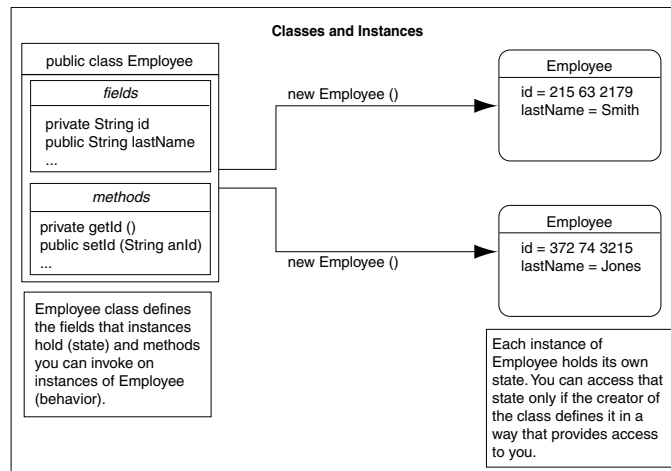
- **Methods**

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

Objects

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

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

Figure 1–1 Classes and Instances

When you create an instance, the attributes store individual and private information relevant only to the employee. That is, the information contained within an employee instance is known only to that particular employee. The example in [Figure 1–1](#) shows two instances of the `Employee` class, one for the employee Smith and one for Jones. Each instance contains information relevant to the individual employee.

Interfaces

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

Encapsulation

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

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

Inheritance

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

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

Polymorphism

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

In the example in [Figure 1-1](#), assume that the different types of employees must be able to respond with their compensation to date. Compensation is computed differently for different types of employees:

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

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

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

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

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

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

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

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

In addition, you can create a `Contractor` class that does not inherit properties from `Employee` and implements a `compensationToDate()` method in it. A program that calculates total payroll to date would iterate over all people on payroll, regardless of whether they were full-time or part-time employees or contractors, and add up the

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

Key Features of the Java Language

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

- **Simplicity**

Java is simpler than most other languages that are used to create server applications, because of its consistent enforcement of the object model. The large, standard set of class libraries brings powerful tools to Java developers on all platforms.

- **Portability**

Java is portable across platforms. It is possible to write platform-dependent code in Java, and it is also simple to write programs that move seamlessly across systems.

See Also: ["JVM"](#) on page 1-6

- **Automatic storage management**

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

- **Strong typing**

Before you use a variable, you must declare the type of the variable. Strong typing in Java makes it possible to provide a reasonable and safe solution to interlanguage calls between Java and PL/SQL applications, and to integrate Java and SQL calls within the same application.

- **No pointers**

Although Java is quite similar to C in its syntax, it does not support direct pointers or pointer manipulation. You pass all parameters, except primitive types, by reference and not by value. As a result, the object identity is preserved. Java does not provide low level, direct access to pointers, thereby eliminating any possibility of memory corruption and leaks.

- **Exception handling**

Java exceptions are objects. Java requires developers to declare which exceptions can be thrown by methods in any particular class.

- **Flexible namespace**

Java defines classes and places them within a hierarchical structure that mirrors the domain namespace of the Internet. You can distribute Java applications and avoid name collisions. Java extensions, such as the Java Naming and Directory Interface (JNDI), provide a framework for multiple name services to be federated. The namespace approach of Java is flexible enough for Oracle to incorporate the concept of a schema for resolving class names in full compliance with the JLS.

- **Security**

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

- Standards for connectivity to relational databases

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

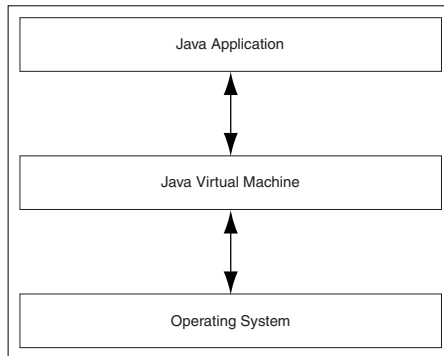
JVM

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

When compiled, the Java code gets converted to a standard, platform-independent set of bytecodes, which interacts with a JVM. JVM is a separate program that is optimized for the specific platform on which you run your Java code.

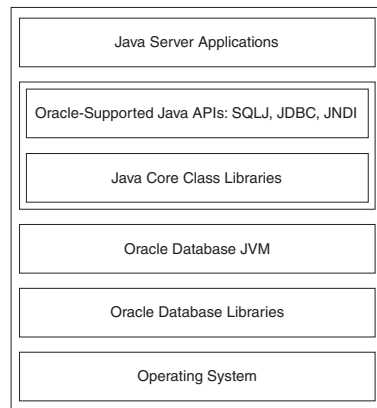
Figure 1–2 illustrates how Java can maintain platform independence. Each platform has JVM installed that is specific to the operating system. The Java bytecodes get interpreted through JVM into the appropriate platform dependent actions.

Figure 1–2 Java Component Structure



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

Figure 1–3 illustrates how Oracle Java applications reside on top of the Java core class libraries, which reside on top of JVM. Because the Oracle Java support system is located within the database, JVM interacts with Oracle Database libraries, instead of directly interacting with the operating system.

Figure 1–3 Oracle Database Java Component Structure

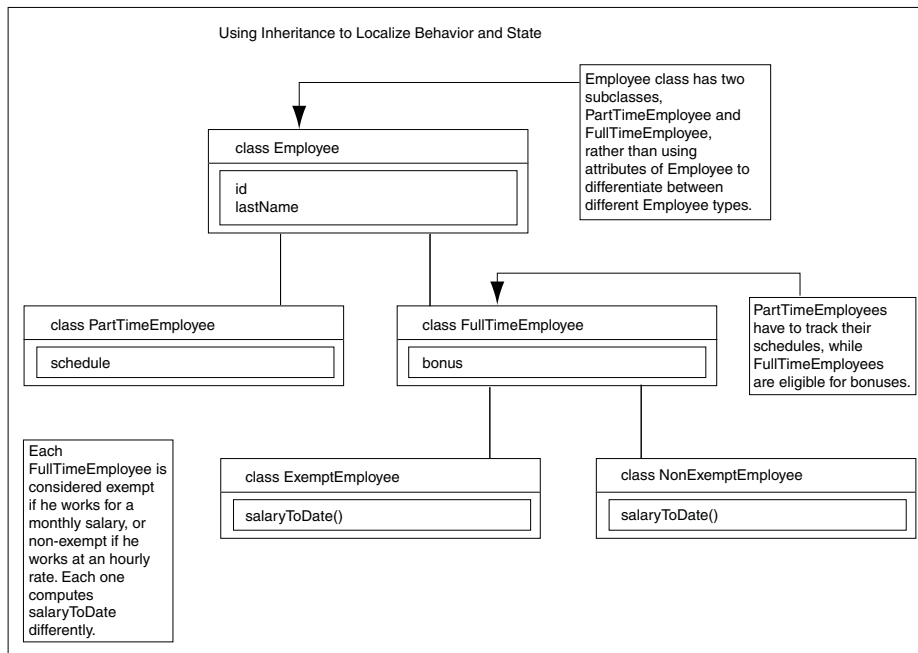
Sun Microsystems provides publicly available specifications for both the Java language and the JVM. The JLS defines the syntax and semantics, and JVM specification defines the necessary low-level actions for the system that runs the application. In addition, Sun Microsystems provides a compatibility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

Java Class Hierarchy

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

[Figure 1–4](#) illustrates a generic Java class hierarchy. The `FullTimeEmployee` class contains the `id` and `lastName` fields defined in the `Employee` class, because it inherits from the `Employee` class. In addition, the `FullTimeEmployee` class adds another field, `bonus`, which is contained only within `FullTimeEmployee`.

You can call any method on an instance of Class B that was defined in either Class A or Class B. In the example, the `FullTimeEmployee` instance can call methods defined only in the `FullTimeEmployee` class and methods defined in the `Employee` class.

Figure 1–4 Class Hierarchy

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

Using Java in Oracle Database

You can write and load Java applications within the database because it is a safe language with a lot of security features. Java has been developed to prevent anyone from tampering with the operating system where the Java code resides in. Some languages, such as C, can introduce security problems within the database. However, Java, because of its design, is a robust language that can be used within the database.

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

- [Java and RDBMS: A Robust Combination](#)
- [Multithreading](#)
- [Memory Spaces Management](#)
- [Footprint](#)
- [Performance of Oracle JVM](#)
- [Dynamic Class Loading](#)

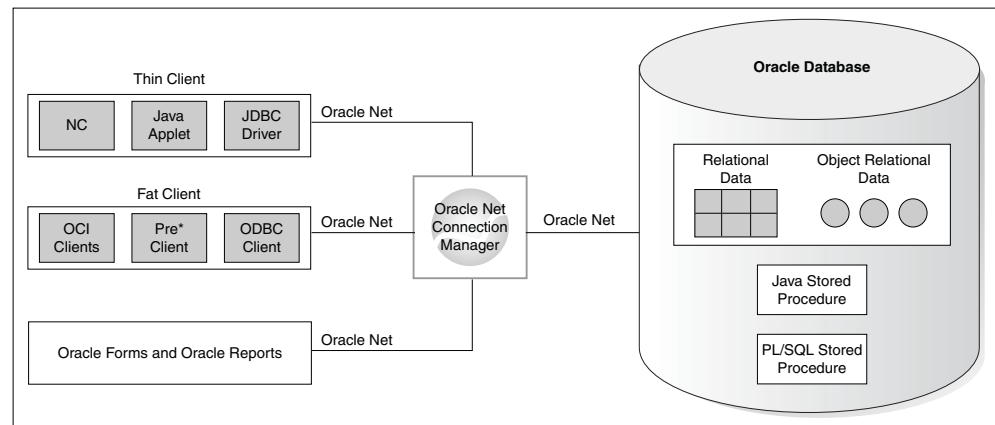
Java and RDBMS: A Robust Combination

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

The combination of Java and Oracle Database helps you to create component-based, network-centric applications that can be easily updated as business needs change. In addition, you can move applications and data stores off the desktop and onto intelligent networks and network-centric servers. More important, you can access those applications and data stores from any client device.

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

Figure 1–5 Two-Tier Client/Server Configuration



Multithreading

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

Oracle Database server efficiently schedules work for thousands of users. Oracle JVM takes advantage of the session architecture of Oracle database to concurrently run Java applications for hundreds to thousands of users. Although Oracle Database supports Java language-level threads required by the JLS and JCK, scalability will not increase by using threads within the scope of the database. By using the embedded scalability of the database, the need for writing multithreaded Java servers is eliminated.

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

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

- Non-Oracle Database model

A single user maps to a single Java thread and a single garbage collector manages all garbage from all users. Different techniques typically deal with allocation and

collection of objects of varying lifetimes and sizes. The result in a heavily multithreaded application is, at best, dependent upon operating system support for native threads, which can be unreliable and limited in scalability. High levels of scalability for such implementations have not been convincingly demonstrated.

- Oracle JVM model

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

See Also: ["Threading in Oracle Database"](#) on page 2-46

Memory Spaces Management

Garbage collection is a major function of the automated storage management feature of Java, eliminating the need for Java developers to allocate and free memory explicitly. Consequently, this eliminates a large source of memory leaks that are commonly found in C and C++ programs. However, garbage collection contributes to the overhead of program execution speed and footprint.

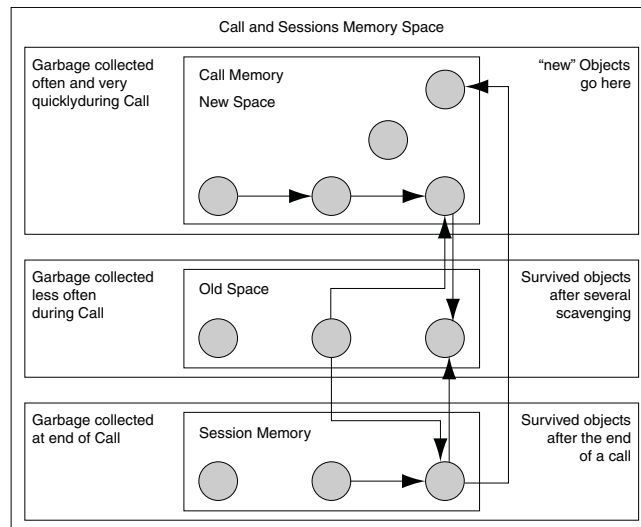
Garbage collection imposes a challenge to the JVM developer seeking to supply a highly scalable and fast Java platform. Oracle JVM meets these challenges in the following ways:

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

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

Memory space	Description
Call space	It is a fast and inexpensive type of memory. It primarily exists for the length of a call. Call memory space is divided into new and old segments. All new objects are created within new memory. Objects that have survived several scavenges are moved into old memory.
Session space	It is an expensive, performance-wise memory. It primarily exists for the length of a session. All <code>static</code> variables and any objects that exist beyond the lifetime of a call exist here.

[Figure 1–6](#) illustrates the different actions performed by the garbage collector.

Figure 1–6 Garbage Collection

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

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

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

Footprint

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

- Size of the program
The size of the program depends on the number of classes and methods and how much code they contain.
- Complexity of the program
The complexity of the program depends on the number of core class libraries that Oracle JVM uses as the program runs, as opposed to the program itself.
- Amount of space JVM uses
The amount of space JVM uses depends on the number of objects JVM allocates, how large these objects are, and how many objects must be retained across calls.

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

This can not be determined often. The speed with which objects are allocated and the way they are held on to by other objects influences the importance of this factor.

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

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

Performance of Oracle JVM

The performance of Oracle JVM is enhanced by Oracle JVM embedding an innovative Just-In-Time compiler similar to HotSpot on Sun's JVM. The platform-independent Java bytecodes run on top of a JVM, and JVM interacts with the specific hardware platform. Any time you add levels within software, the performance is degraded. Because Java requires going through an intermediary to interpret the bytecodes, a degree of inefficiency exists for Java applications as compared to applications developed using a platform-dependent language, such as C. To address this issue, several JVM suppliers create native compilers. Native compilers translate Java bytecodes into platform-dependent native code, which eliminates the interpreter step and improves performance.

The following table describes two methods for native compilation:

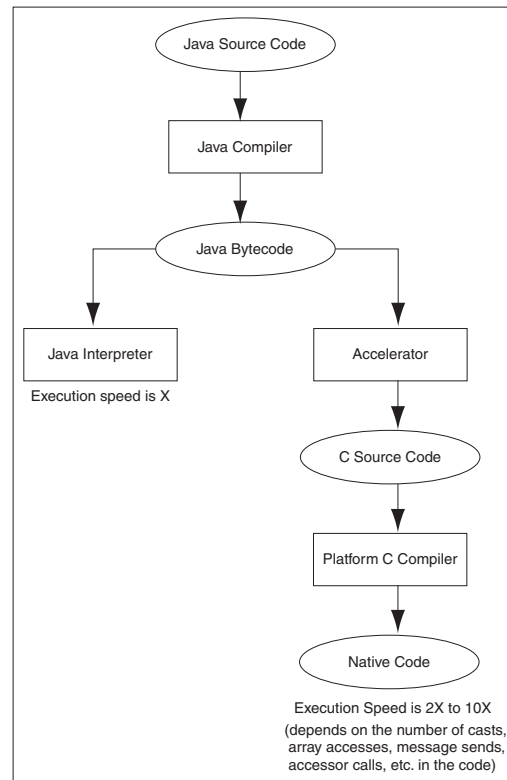
Compiler	Description
Just-In-Time (JIT) Compilation	JIT compilers quickly compile Java bytecodes to platform-specific, or native, machine code during run time. These compilers do not produce an executable file to be run on the platform. Instead, they provide platform-dependent code from Java bytecodes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

Oracle Database uses Just-In-Time (JIT) compilation to deliver its core Java class libraries, such as JDBC code, in natively compiled form. The JIT compiler is enabled without the support of any plug-ins and it is applicable across all the platforms that Oracle supports.

See Also: [Oracle JVM Just-in-Time Compiler \(JIT\)](#)

Figure 1–7 illustrates how natively compiled code runs up to 10 times faster than interpreted code. As a result, the more native code your program uses, the faster it runs.

Figure 1–7 Interpreter versus Accelerator



Dynamic Class Loading

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

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

Problem	Description	Solution
Reliability	A benefit of dynamic class loading is that it supports program updating. For example, you would update classes on a server, and clients, who download the program and load it dynamically, see the update whenever they next use the program. Server programs tend to emphasize reliability. As a developer, you must know that every client runs a specific program configuration. You do not want clients to inadvertently load some classes that you did not intend them to load.	Oracle Database separates the upload and resolve operation from the class loading operation at run time. You upload Java code you developed to the server using the <code>loadjava</code> tool. Instead of using <code>CLASSPATH</code> , you specify a resolver at installation time. The resolver is analogous to <code>CLASSPATH</code> , but enables you to specify the schemas in which the classes reside. This separation of resolution from class loading ensures that you always know what programs users run. See Also: Chapter 11, "Schema Objects and Oracle JVM Utilities"

Overview of Oracle JVM

Oracle JVM is a standard, Java-compatible environment that runs any pure Java application. It is compatible with the JLS and the JVM specification laid down by Sun Microsystems. It supports the standard Java binary format and the standard Java APIs. In addition, Oracle Database adheres to standard Java language semantics, including dynamic class loading at run time.

Java in Oracle Database introduces the following terms:

- **Session**
A session in Oracle Database Java environment is identical to the standard Oracle Database usage. A session is typically, although not necessarily, bounded by the time a single user connects to the server. As a user who calls a Java code, you must establish a session in the server.
- **Call**
When a user causes a Java code to run within a session, it is termed as a call. A call can be started in the following different ways:
 - A SQL client program runs a Java stored procedure.
 - A trigger runs a Java stored procedure.
 - A PL/SQL program calls a Java code.

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

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

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

- [Process Area](#)
- [The `main\(\)` Method](#)
- [The GUI](#)
- [The IDE](#)

Process Area

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

```
java classname
```

This command causes the application to run within a process on your operating system. However, if you are not using the command-line interface, you must load the application into the database, publish the interface, and then run the application within a database session.

See Also: [Chapter 2, "Java Applications on Oracle Database"](#) for information about loading, publishing, and running Java applications

The main() Method

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

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

The GUI

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

See Also: ["User Interfaces on the Server"](#) on page 2-20

The IDE

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

See Also: ["Development Tools"](#) on page 1-22

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

New Features in This Release

This section describes the new features of Oracle JVM in Oracle Database 11g Release 2 (11.2).

Internet Protocol Version 6 Support

This release of Oracle JVM supports Internet Protocol Version 6 (IPv6) addresses in the URL and machine names of the Java code in the database, which resolve to IPv6 addresses. IPv6 is a new Network layer protocol designed by the Internet Engineering Task Force (IETF) to replace the current version of Internet Protocol, Internet Protocol Version 4 (IPv4). The primary benefit of IPv6 is a large address space, derived from the use of 128-bit addresses. IPv6 also improves upon IPv4 in areas such as routing, network autoconfiguration, security, quality of service, and so on.

The following system properties enable you to configure IPv6 preferences:

java.net.preferIPv4Stack

If IPv6 is available on the operating system, then the underlying native socket will be an IPv6 socket. This enables Java applications to connect to, and accept connections from both IPv4 and IPv6 hosts. If you have an application that has a preference to use only IPv4 sockets, then you can set this property to `true`. If you set the property to `true`, then it implies that the application will not be able to communicate with IPv6 hosts.

java.net.preferIPv6Addresses

Even if IPv6 is available on the operating system, then for backward compatibility reasons, the addresses are mapped to IPv4. For example, applications that depend on access to only an IPv4 service, benefit from this type of mapping. If you want to change the preferences to use IPv6 addresses over IPv4 addresses, then you can set the `java.net.preferIPv6Addresses` property to `true`. This allows applications to be tested and deployed in environments, where the application is expected to connect to IPv6 services.

Note: All the new System classes that are required for IPv6 support are loaded when Java is enabled during database initialization. So, if your application does not have any addresses that are included in the software code, then you do not need to change your code to use IPv6 functionality.

Feature List of Oracle JVM

Table 1–1 lists the features of Oracle JVM and the versions in which they were first supported.

Table 1–1 Feature List of Oracle JVM

Feature	Supported Since Oracle JVM Release
JDK 1.5 support	11.1
<code>loadjava</code> URL support	11.1
List-Based operations with <code>dropjava</code> support	11.1
<code>ojvmtc</code> Tool	11.1
<code>Runjava</code> command-line interface (JDK-like interface)	11.1
Database-Resident JARs	11.1
Sharing of user classloaded classes metadata support	11.1
Two-tier duration for the Java session state support	11.1

Table 1–1 (Cont.) Feature List of Oracle JVM

Feature	Supported Since Oracle JVM Release
Default service feature	11.1
Just-in-Time compiler (JIT)	11.1
Internet Protocol Version 6 (IPv6) Support	11.2

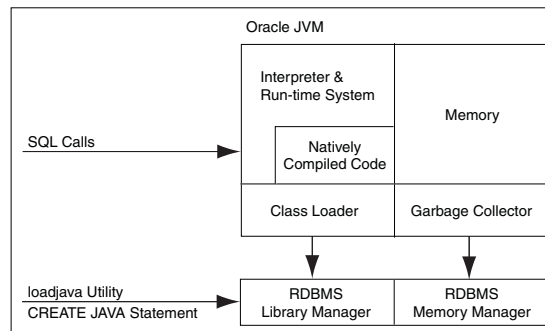
Main Components of Oracle JVM

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

Oracle JVM is a complete, Java2-compliant environment for running Java applications. It runs in the same process space and address space as the database kernel by sharing its memory heaps and directly accessing its relational data. This design optimizes memory use and increases throughput.

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

Figure 1–8 shows the main components of Oracle JVM.

Figure 1–8 Main Components of Oracle JVM

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

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

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

- [Library Manager](#)
- [Compiler](#)
- [Interpreter](#)
- [Class Loader](#)
- [Verifier](#)

- [Server-Side JDBC Internal Driver](#)
- [Server-Side SQLJ Translator](#)
- [System Classes](#)

See Also: ["Memory Spaces Management"](#) on page 1-10 and ["Performance of Oracle JVM"](#) on page 1-12

Library Manager

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

Compiler

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

Interpreter

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

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

Class Loader

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

Verifier

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

Server-Side JDBC Internal Driver

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

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

Server-Side SQLJ Translator

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

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

System Classes

A set of classes that constitute a significant portion of the implementation of Java in Oracle Database environment is known as the **System classes**. These classes are defined in the SYS schema and exported for all users by public synonym. A class with the same name as one of the System classes can be defined in a schema other than the SYS schema¹. But, this is a bad practice because the alternate version of the class may behave in a manner that violates assumptions about the semantics of that class that are present in other System classes or in the underlying implementation of Java Virtual Machine. Oracle strongly discourages this practice.

Java Application Strategy of Oracle

Oracle provides enterprise application developers an end-to-end Java solution for creating, deploying, and managing Java applications. The total solution consists of client-side and server-side programmatic interfaces, tools to support Java development, and a JVM integrated with Oracle Database. All these products are fully compatible with Java standards. This section covers the following topics:

- [Java in Database Application Development](#)
- [Java Programming Environment](#)

¹ You cannot always define a class with the same name as one of the System classes. For the classes present in some packages, for example, `java.lang`, such definitions are explicitly prohibited by the code.

- [Java Stored Procedures](#)
- [PL/SQL Integration and Oracle RDBMS Functionality](#)
- [Development Tools](#)

Java in Database Application Development

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

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

Java Programming Environment

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

- Java stored procedures as the Java equivalent and companion for PL/SQL. Java stored procedures are tightly integrated with PL/SQL. You can call Java stored procedures from PL/SQL packages and PL/SQL procedures from Java stored procedures.
- The JDBC and SQLJ programming interfaces for accessing SQL data.
- Tools and scripts that assist in developing, loading, and managing classes.

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

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

Java Stored Procedures

Java stored procedures are Java programs written and deployed on a server and run from the server, exactly like a PL/SQL stored procedure. You invoke it directly with products like SQL*Plus, or indirectly with a trigger. You can access it from any Oracle Net client, such as OCI and PRO*, or JDBC or SQLJ.

See Also: [Chapter 5, "Developing Java Stored Procedures"](#)

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

PL/SQL Integration and Oracle RDBMS Functionality

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

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

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

- [JDBC Drivers](#)
- [SQLJ](#)
- [JPublisher](#)

JDBC Drivers

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

Driver	Description
JDBC Thin driver	You can use the JDBC Thin driver to write pure Java applications and applets that access Oracle SQL data. The JDBC Thin driver is especially well-suited for Web-based applications and applets, because you can dynamically download it from a Web page, similar to any other Java applet.
JDBC OCI driver	The JDBC OCI driver accesses Oracle-specific native code, that is, non-Java code, and libraries on the client or middle tier, providing performance boost compared to the JDBC Thin driver, at the cost of significantly larger size and client-side installation.
JDBC server-side internal driver	Oracle Database uses the server-side internal driver when the Java code runs on the server. It allows Java applications running in Oracle JVM on the server to access locally defined data, that is, data on the same system and in the same process, with JDBC. It provides a performance boost, because of its ability to use the underlying Oracle RDBMS libraries directly, without the overhead of an intervening network connection between the Java code and SQL data. By supporting the same Java-SQL interface on the server, Oracle Database does not require you to rework code when deploying it.

See Also:

- ["Utilizing SQLJ and JDBC with Java in the Database"](#) on page 3-3
- *Oracle Database JDBC Developer's Guide*

SQLJ

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

program to standard Java source with JDBC calls. At run time, the program can communicate with multi-vendor databases using standard JDBC drivers.

SQLJ provides a simple, but powerful, way to develop both client-side and middle-tier applications that access databases from Java. You can use SQLJ in stored procedures, triggers, and methods within the Oracle Database 11g environment. In addition, you can combine SQLJ programs with JDBC.

The SQLJ translator is a Java program that translates embedded SQL in Java source code to pure JDBC-based Java code. Oracle Database 11g provides a complete Java environment, so you cannot compile SQLJ programs on a client that will run on the server. Instead, you can compile them directly on the server. The adherence of Oracle Database to the Internet standards enables you to choose the development style as per your requirements.

See Also:

- "Utilizing SQLJ and JDBC with Java in the Database" on page 3-3
- *Oracle Database SQLJ Developer's Guide*

JPublisher

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

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

Development Tools

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

<http://www.oracle.com/technology/software/products/jdev/index.html>

Memory Model for Dedicated Mode Sessions

Since Oracle Database 10g, Oracle JVM has a new memory model for sessions that connect to the database through a dedicated server. The basic memory structures associated with Oracle include:

- System Global Area (SGA)

The SGA is a group of shared memory structures, known as SGA components, that contain data and control information for one Oracle Database instance. The SGA is shared by all server and background processes. Examples of data stored in the SGA include cached data blocks and shared SQL areas.

- Program Global Areas (PGA)

A PGA is a memory region that contains data and control information for a server process. It is nonshared memory created by Oracle when a server process is started. Access to the PGA is exclusive to the server process. There is one PGA for each server process. Background processes also allocate their own PGAs. The total

PGA memory allocated for all background and server processes attached to an Oracle instance is referred to as the **aggregate PGA**.

The simplest way to manage memory is to allow the database to automatically manage and tune it for you. To do so, you set only a target memory size initialization parameter (`MEMORY_TARGET`) and a maximum memory size initialization parameter (`MEMORY_MAX_TARGET`), on most platforms. The database then tunes to the target memory size, redistributing memory as needed between the SGA and aggregate PGA. Because the target memory initialization parameter is dynamic, you can change the target memory size at any time without restarting the database. The maximum memory size serves as an upper limit so that you cannot accidentally set the target memory size too high. Because certain SGA components either cannot easily shrink or must remain at a minimum size, the database also prevents you from setting the target memory size too low.

See Also: *Oracle Database Administrator's Guide*

Java Applications on Oracle Database

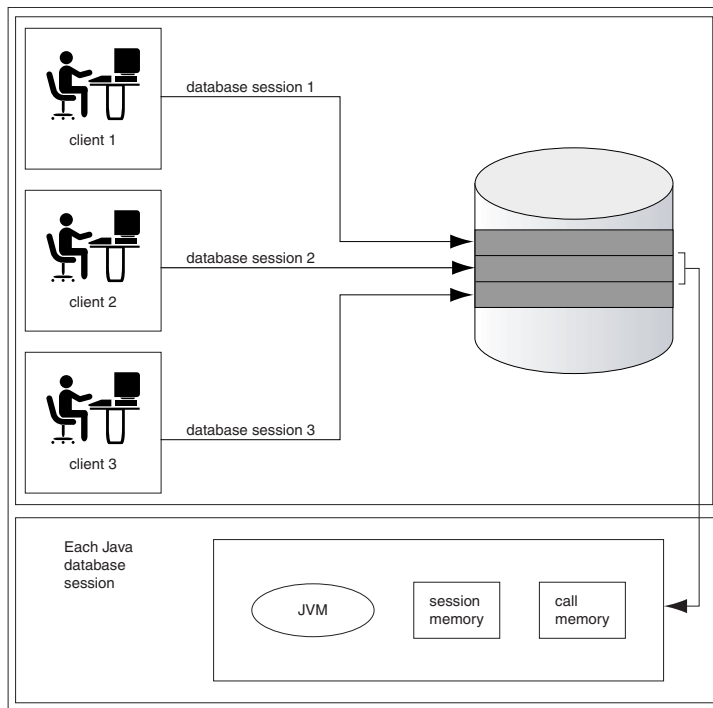
Oracle Database runs standard Java applications. However, the Java-integrated Oracle Database environment is different from a typical Java development environment. This chapter describes the basic differences for writing, installing, and deploying Java applications within Oracle Database. This chapter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

- [Database Sessions Imposed on Java Applications](#)
- [Execution Control of Java Applications](#)
- [Java Code, Binaries, and Resources Storage](#)
- [Preparing Java Class Methods for Execution](#)
- [User Interfaces on the Server](#)
- [Shortened Class Names](#)
- [Class.forName\(\) in Oracle Database](#)
- [Managing Your Operating System Resources](#)
- [Managing Your Applications Using JMX](#)
- [Threading in Oracle Database](#)
- [Shared Servers Considerations](#)

Database Sessions Imposed on Java Applications

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

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

Figure 2–1 Java Environment Within Each Database Session

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

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

Within a session, the client has its own Java environment. It appears to the client as if a separate, individual JVM was started for each session, although the implementation is more efficient than this seems to imply. Within a session, Oracle JVM manages the scalability of applications. Every call from a single client is managed within its own session, and calls from each client is handled separately. Oracle JVM maximizes sharing read-only data between clients and emphasizes a minimum amount of per-session incremental footprint, to maximize performance for multiple clients.

The underlying server environment hides the details associated with session, network, state, and other shared resource management issues from the Java code. Variables defined as `static` are local to the client. No client can access the `static` variables of other clients, because the memory is not available across session boundaries. Because each client runs the Java application calls within its own session, activities of each client are separate from any other client. During a call, you can store objects in `static` fields of different classes, which will be available in the next call. The entire state of your Java program is private and exists for your entire session.

Oracle JVM manages the following within the session:

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

- Call memory for variables that exist within a call

Execution Control of Java Applications

In the Java2 Platform, Standard Edition (J2SE) environment, you develop Java applications with a `main()` method, which is called by the interpreter when the class is run. The `main()` method is called when you enter the following command on the command-line:

```
java classname
```

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

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

For example, in a standard Java environment, you would start the Java object on the server by running the following command:

```
java myprogram
```

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

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

Java Code, Binaries, and Resources Storage

In the standard Java development environment, Java source code, binaries, and resources are stored as files in a file system, as follows:

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

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

Both these concepts are different in Oracle Database environment.

[Table 2–1](#) describes how Oracle Database handles Java classes and locates dependent classes.

Table 2–1 Description of Java Code and Classes

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

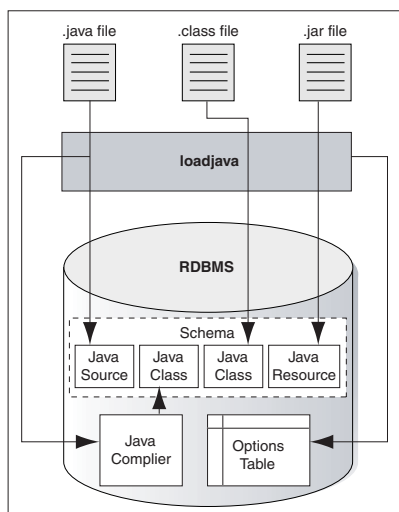
Java Classes Loaded in the Database

If you are not using the command-line interface, you must load Java files into the database as schema objects, to make them available to Oracle JVM. As [Figure 2–2](#) illustrates, the loadjava tool can call the Java compiler of Oracle JVM, which compiles source files into standard class files.

See Also: ["Using Command-Line Interface"](#) on page 3-9

[Figure 2–2](#) shows that the loadjava tool can set the values of options stored in a system database table. Among other things, these options affect the processing of Java source files.

Figure 2–2 Loading Java into Oracle Database



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

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

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

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

Oracle Database accepts Java names up to 4000 characters long. However, the names of Java schema objects cannot be longer than 30 characters. Therefore, if a schema object name is longer than 30 characters, then the system generates a short name, or alias, for the schema object. Otherwise, the fully qualified name, also called full name, is used. You can specify the full name in any context that requires it. When needed, name mapping is handled by Oracle Database.

See Also: ["Shortened Class Names"](#) on page 2-21 and ["System Classes"](#) on page 1-19

Preparing Java Class Methods for Execution

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

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

This sections covers the following topics:

- [Compiling Java Classes](#)
- [Resolving Class Dependencies](#)
- [Loading Classes](#)
- [Granting Execute Rights](#)
- [Controlling the Current User](#)
- [Checking Java Uploads](#)
- [Publishing](#)
- [Auditing](#)

Compiling Java Classes

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

- You can compile the source explicitly on a client system before loading it into the database, through a Java compiler, such as `javac`.
- You can ask the database to compile the source during the loading process, which is managed by the `loadjava` tool.
- You can force the compilation to occur dynamically at run time.

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

This section includes the following topics:

- [Compiling Source Through `javac`](#)

- [Compiling Source Through the loadjava Tool](#)
- [Compiling Source at Run Time](#)
- [Specifying Compiler Options](#)
- [Recompiling Automatically](#)

Compiling Source Through javac

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

Compiling Source Through the loadjava Tool

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

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

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

Compiling Source at Run Time

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

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

Specifying Compiler Options

You can specify the compiler options in the following ways:

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

See Also: ["Compiler Options Specified in a Database Table"](#)

Default Compiler Options

When compiling a source schema object for which neither a `JAVA$OPTIONS` entry exists nor a command-line value for any option is specified, the compiler assumes a default value as follows:

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

This option applies only to Java sources that contain SQLJ constructs.

- `debug=true`

This option is equivalent to:

```
javac -g
```

Compiler Options on the Command Line

The `encoding` compiler option specified with the `loadjava` tool identifies the encoding of the `.java` file. This option overrides any matching value in the `JAVA$OPTIONS` table. The values are identical to:

```
javac -encoding
```

This option is relevant only when loading a source file.

Compiler Options Specified in a Database Table

Each `JAVA$OPTIONS` entry contains the names of source schema objects to which an option setting applies. You can use multiple rows to set the options differently for different source schema objects.

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

```
PROCEDURE set_compiler_option(name VARCHAR2, option VARCHAR2, value VARCHAR2);
```

```
FUNCTION get_compiler_option(name VARCHAR2, option VARCHAR2) RETURNS VARCHAR2;
```

```
PROCEDURE reset_compiler_option(name VARCHAR2, option VARCHAR2);
```

Table 2–2 describes the parameters for these methods.

Table 2–2 *Definitions for the Name and Option Parameters*

Parameter	Description
name	This is a Java package name, a fully qualified class name, or an empty string. When the compiler searches the <code>JAVA\$OPTIONS</code> table for the options to use for compiling a Java source schema object, it uses the row that has a value for name that most closely matches the fully qualified class name of a schema object. A name whose value is the empty string matches any schema object name.
option	The option parameter is either <code>online</code> , <code>encoding</code> , or <code>debug</code> .

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

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

Table 2–3 represents a hypothetical `JAVA$OPTIONS` database table. The pattern match rule is to match as much of the schema name against the table entry as possible. The schema name with a higher resolution for the pattern match is the entry that applies. Because the table has no entry for the `encoding` option, the compiler uses the default or the value specified on the command line. The `online` option shown in the table matches schema object names as follows:

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

Table 2–3 Example JAVA\$OPTIONS Table

Name	Option	Value	Match Examples
<code>a.b.c.d</code>	<code>online</code>	<code>true</code>	<ul style="list-style-type: none"> ■ <code>a.b.c.d</code> Matches the pattern exactly. ■ <code>a.b.c.d.e</code> First part matches the pattern exactly. No other rule matches the full qualified name.
<code>a.b</code>	<code>online</code>	<code>false</code>	<ul style="list-style-type: none"> ■ <code>a.b</code> Matches the pattern exactly ■ <code>a.b.c.x</code> First part matches the pattern exactly. No other rule matches beyond this rule.
Empty string	<code>online</code>	<code>true</code>	<ul style="list-style-type: none"> ■ <code>a.c</code> No pattern match with any defined name. Defaults to the empty string rule. ■ <code>x.y</code> No pattern match with any defined name. Defaults to the empty string rule.

Recompiling Automatically

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

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



```

{
  A a;
  public void assignA()
  {
    a = new A()
  }
}

```

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

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

Resolving Class Dependencies

Many Java classes contain references to other classes, which is the essence of reusing code. A conventional JVM searches for `.class`, `.zip`, and `.jar` files within the directories specified in `CLASSPATH`. In contrast, Oracle JVM searches database schemas for class objects. In Oracle Database, because you load all Java classes into the database, you may need to specify where to find the dependent classes for your Java class within the database.

All classes loaded within the database are referred to as class schema objects and are loaded within certain schemas. All predefined Java application programming interfaces (APIs), such as `java.lang.*`, are loaded within the `PUBLIC` schema. If your classes depend on other classes you have defined, then you will probably load them all within your own schema. For example, if your schema is `SCOTT`, the database resolver searches the `SCOTT` schema before searching the `PUBLIC` schema. The listing of schemas to search is known as a **resolver specification**. Resolver specifications are defined for each class. This is in contrast to a classic JVM, where `CLASSPATH` is global to all classes.

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

Note: As with the Java compiler, the `loadjava` tool resolves references to classes, but not to resources. Ensure that you correctly load the resource files that your classes require.

For each interclass reference in a class, the resolver searches the schemas specified by the resolver specification for a valid class schema object that satisfies the reference. If all references are resolved, then the resolver marks the class valid. A class that has

never been resolved, or has been resolved unsuccessfully, is marked invalid. A class that depends on a schema object that becomes invalid is also marked invalid.

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

Classes can be resolved in the following ways:

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

```
loadjava -resolve
```

- Loading using your own resolver specification definition:

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

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

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

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

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

Allowing References to Nonexistent Classes

You can specify a special option within a resolver specification that allows an unresolved reference to a nonexistent class. Sometimes, internal classes are never used within a product. In a standard Java environment, this is not a problem, because as long as the methods are not called, JVM ignores them. However, when resolving a class, Oracle JVM tries to resolve all names referenced by that class, including names that may never be used. If Oracle JVM cannot find a matching class for each such names referenced by that class, then the class being resolved is marked as invalid and cannot be run.

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

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

Without the wildcard, if a dependent class is not found within one of the schemas, your class is listed as invalid and cannot be run.

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

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

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

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

Even when a minus sign (-) wildcard is used, the super class of a class can never be nonexistent. If the super class is not found, then the class will be invalid regardless of the use of a minus sign (-) wildcard in the resolver.

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

Bytecode Verifier

According to JVM specification, `.class` files are subject to verification before the class they define is available in a JVM. In Oracle JVM, the verification process occurs at class resolution.

Table 2–4 describes the problems the resolver may find and the appropriate Oracle error code issued.

Table 2–4 ORA Errors

Error Code	Description
ORA-29545	<p>If the resolver determines that the class is malformed, then the resolver does not mark it valid. When the resolver rejects a class, it issues an ORA-29545 error. The <code>loadjava</code> tool reports the error. For example, this error is thrown if the contents of a <code>.class</code> file are not the result of a Java compilation or if the file has been corrupted.</p> <p>The ORA-29545 error may also show up if you used the minus sign (-) wild card expression with the resolver and the validity of some classes was not verified.</p>
ORA-29552	<p>In some situations, the resolver allows a class to be marked valid, but will replace bytcodes in the class to throw an exception at run time. In these cases, the resolver issues an ORA-29552 warning that the <code>loadjava</code> tool reports. The <code>loadjava</code> tool issues this warning when the Java Language Specification (JLS) requires an <code>IncompatibleClassChangeError</code> to be thrown. Oracle JVM relies on the resolver to detect these situations, supporting the proper run-time behavior 'that the JLS requires.</p>

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

```
B myMethod(A a)
```

```
{
    return a;
}
```

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

A resolver without the minus sign (-) wildcard ensures that the class definitions of A and B are found and resolved properly if they are present in the schemas they specifically identify. The only time you may consider using the alternative resolver is if you must load an existing JAR file containing classes that reference other nonsystem classes, which are not included in the JAR file.

See Also: [Chapter 11, "Schema Objects and Oracle JVM Utilities"](#) for more information about class resolution and loading your classes within the database.

Loading Classes

This section gives an overview of loading your classes into the database using the loadjava tool. You can use the loadjava tool to create schema objects from files and load the schema objects to different schemas. For example,

```
loadjava -u scott -schema TEST MyClass.java
Password: password
```

Note: You do *not* have to load the classes to the database as schema objects if you use the command-line interface. For example,

```
C:\oraclehome\bin>loadjava -u scott MyClass.java
Password: password
```

For more information about the command-line interface, refer to ["Using Command-Line Interface"](#) on page 9.

See Also: [Chapter 11, "Schema Objects and Oracle JVM Utilities"](#)

You can also run the loadjava tool from within SQL commands. Unlike a conventional JVM, which compiles and loads from files, Oracle JVM compiles and loads from database schema objects.

[Table 2-5](#) describes database schema objects that correspond to the files used by a conventional JVM.

Table 2-5 Description of Java Files

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

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

Table 2–6 describes the activities the `loadjava` tool performs for each type of file.

Table 2–6 *loadjava Operations on Schema Objects*

Schema Object	loadjava Operations on Objects
<code>.java</code> source files	<ol style="list-style-type: none"> 1. Creates a Java source schema object in the definer's schema unless another schema is specified. 2. Loads the contents of the source file into a schema object. 3. Creates a class schema object for all classes defined in the source file. 4. If <code>-resolve</code> is requested, compiles the source schema object and resolves the class and its dependencies. It then stores the compiled class into a class schema object.
<code>.sqlj</code> source files	<ol style="list-style-type: none"> 1. Creates a source schema object in the definer's schema unless another schema is specified. 2. Loads contents of the source file into the schema object. 3. Creates a class schema object for all classes and resources defined in the source file. 4. If <code>-resolve</code> is requested, translates and compiles the source schema object and stores the compiled class into a class schema object. It then stores the profile into a <code>.ser</code> resource schema object and customizes it.
<code>.class</code> compiled Java files	<ol style="list-style-type: none"> 1. Creates a class schema object in the definer's schema unless another schema is specified. 2. Loads the class file into the schema object. 3. Resolves and verifies the class and its dependencies if <code>-resolve</code> is specified.
<code>.properties</code> Java resource files	<ol style="list-style-type: none"> 1. Creates a resource schema object in the definer's schema unless another schema is specified. 2. Loads a resource file into a schema object.
<code>.ser</code> SQLJ profile	<ol style="list-style-type: none"> 1. Creates a resource schema object in the definer's schema unless another schema is specified. 2. Loads the <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. Always use the `dropjava` tool to delete a Java schema object created with the `loadjava` tool. For example,

```
dropjava -u SCOTT -schema TEST MyClass.java
Password: password
```

Dropping with SQL data definition language (DDL) commands will not update the auxiliary data maintained by the `loadjava` tool and the `dropjava` tool. You can also run the `dropjava` tool from within SQL commands.

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

Sharing of Metadata for User Classloaded Classes

Classes loaded by the built-in mechanism for loading database resident classes are known as **system classloaded**, whereas those loaded by other means are called **user classloaded**. When you load a class into the database, a representation of the class is created in memory, part of which is referred to here as the class metadata. The class metadata is the same for any session using the class and is potentially sharable. Earlier, such sharing was available only for system classloaded classes. From Oracle Database 11g, you can also share class metadata of user classloaded classes, at the discretion of the system administrator.

See Also: [Appendix B, "Classpath Extensions and User Classloaded Metadata"](#)

Defining the Same Class Twice

You cannot have two class objects with the same name in the same schema. This rule affects you in two ways:

Note: An exception to this rule is when you use the `-prependjarnames` option for database resident JARs. If you use this option, then you can have two classes with the same class name in the same schema. For more information about database resident JARs, see ["Database Resident JARs"](#) on page 2-15.

- You can load either a particular Java `.class` file or its `.java` file, but not both. Oracle Database tracks whether you loaded a class file or a source file. If you want to update the class, then you must load the same type of file that you originally loaded. If you want to update the other type, then you must drop the first before loading the second. For example, if you loaded `x.java` as the source for class `y`, then to load `x.class`, you must first drop `x.java`.
- You cannot define the same class within two different schema objects in the same schema. For example, suppose `x.java` defines class `y` and you want to move the definition of `y` to `z.java`. If `x.java` has already been loaded, then the `loadjava` tool rejects any attempt to load `z.java`, which also defines `y`. Instead, do either of the following:
 - Drop `x.java`, load `z.java`, which defines `y`, and then load the new `x.java`, which does not define `y`.
 - Load the new `x.java`, which does not define `y`, and then load `z.java`, which defines `y`.

See Also: ["System Classes"](#) on page 1-19

Designating Database Privileges and JVM Permissions

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

- `CREATE PROCEDURE` and `CREATE TABLE` privileges to load into your schema.
- `CREATE ANY PROCEDURE` and `CREATE ANY TABLE` privileges to load into another schema.
- `oracle.aurora.security.JServerPermission.loadLibraryInClass.classname`.

See Also: ["Permission for Loading Classes"](#) on page 10-18

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 a JAR or ZIP file to the `loadjava` tool, it opens the archive and loads the members of the archive individually. There is no JAR or ZIP schema object. If the JAR or ZIP content has not changed since the last time it was loaded, then it is not reloaded. Therefore, there is little performance penalty for loading JAR or ZIP files. In fact, loading JAR or ZIP files is the simplest way to use the `loadjava` tool.

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

Database Resident JARs

Starting with 11g release 1 (11.1), when you load the contents of a JAR into the database, you have the option of creating a database object representing the JAR itself. In this way, you can retain an association between this JAR object and the class, resource, and source objects loaded from the JAR. This enables you to:

- Use signed JARs and JAR namespace segregation in the same way as you use them in standard JVM.
- Manage the classes that you have derived from a JAR while loading it into the database as a single unit. This helps you to prevent individual redefinition of the classes loaded from the JAR. It also enables you to drop the whole set of classes loaded from the JAR, irrespective of the contents or the continued existence of the JAR on the external file system, at the time of dropping it.

In order to load a JAR into the database, you have the following options of the `loadjava` tool:

- `-jarsasdbobjects`
- `-prependjarnames`

For more information regarding the `loadjava` tool options, refer to "[The loadjava Tool](#)" section on page 11-6.

Granting Execute Rights

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

The classes that define a Java application are stored within Oracle Database under the SQL schema of their owner. By default, classes that reside in one user's schema cannot be run by other users, because of security concerns. You can provide other users the right to run your class in the following ways:

- Using the `loadjava -grant` option

See Also: "[The loadjava Tool](#)" on page 11-6

- Using the following command:

```
SQL> grant execute on myclass to scott;
```

where, `myclass` is the name of the underlying Java class.

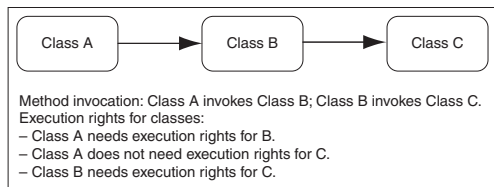
If you want to shorten a classname, then you can use the following command:

```
SQL> select dbms_java.shortname('long classname') from dual;
```

Note: Prior to Oracle Database 11g release 1 (11.1), granting execute right to a stored procedure would mean granting execute right to both the stored procedure and the Java class referred by the stored procedure. Since Oracle Database 11g release, if you want to grant execute right on the underlying Java class as well, then you must grant execute right on the class explicitly. This is implemented for better security.

Figure 2–3 illustrates the rights required to run classes.

Figure 2–3 Rights to Run Classes



See Also: [Chapter 9, "Oracle Database Java Application Performance"](#) for information about JVM security permissions

Controlling the Current User

During the execution of PL/SQL code, there is always a current user. The same concept is used for the execution of Java code. Initially, the current user is the user, who creates the session that invokes the Java code. A Java method is called from SQL or PL/SQL through a corresponding wrapper. Java wrappers are special PL/SQL entities, which expose Java methods to SQL and PL/SQL as PL/SQL stored procedures or functions. Such a wrapper might change the current effective user. The wrappers that change the current effective user to the owner of the wrapper are called definer's rights wrappers. If a wrapper does not change the current effective user, then the effective user remains unchanged.

By default, Java wrappers are definer's rights wrappers. If you want to override this, then create the wrapper using the `AUTHID CURRENT_USER` option.

See Also: ["Writing Top-Level Call Specifications"](#) on page 6-6 for more information about the `AUTHID CURRENT_USER` option

At any time during the execution of Java code, a Java call stack is maintained. The stack contains frames corresponding to Java methods entered, with the innermost frame corresponding to the currently executing method. By default, Java methods execute on the stack without changing the current user, that is, with the privileges of their current effective invoker, not their definer.

You can load a Java class to the database with the `loadjava -definer` option. Any method of a class having the definer attribute marked, becomes a definer's rights method. When such a method is entered, a special kind of frame called a definer's

frame is created onto the Java stack. This frame switches the current effective user to the owner (definer) of such a class. A new user ID remains effective for all inner frames until either the definer's frame is popped off the stack or a nested definer's frame is entered.

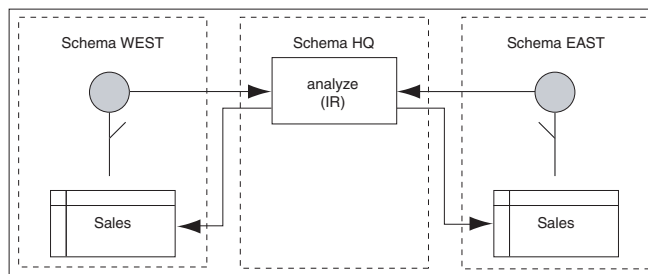
Thus, at any given time during the execution of a Java method that is called from SQL or PL/SQL through its wrapper, the effective user is one of the following:

- The innermost definer's frame on the Java stack
- Either the owner of the PL/SQL wrapper of the topmost Java method, if it is definer's rights, or the user who called the wrapper.

Consider a company that uses a definer's rights procedure to analyze sales. To provide local sales statistics, the procedure `analyze` must access `sales` tables that reside at each regional site. To do this, the procedure must also reside at each regional site. This causes a maintenance problem. To solve the problem, the company installs an invoker's rights version of the procedure `analyze` at headquarters.

Figure 2-4 shows how all regional sites can use the same procedure to query their own `sales` tables.

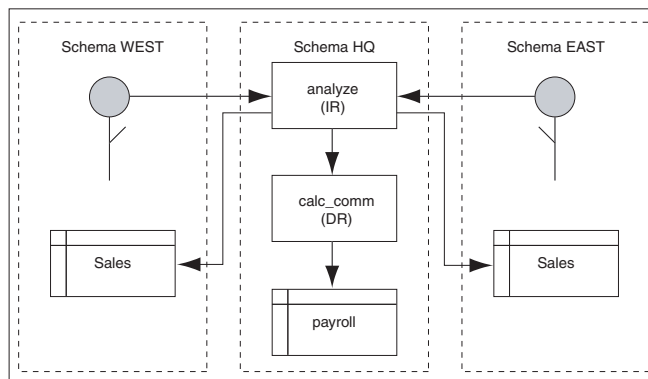
Figure 2-4 Invoker's rights Solution



Occasionally, you may want to override the default invoker's rights behavior. Suppose headquarters wants the `analyze` procedure to calculate sales commissions and update a central `payroll` table. This presents a problem, because invokers of `analyze` should not have direct access to the `payroll` table, which stores employee salaries and other sensitive data.

Figure 2-5 illustrates the solution, where the `analyze` procedure calls the definer's rights procedure, `calcComm`, which in turn updates the `payroll` table.

Figure 2-5 Indirect Access



Checking Java Uploads

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

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

Table 2-7 Key USER_OBJECT Columns

Name	Description
<code>OBJECT_NAME</code>	Name of the object
<code>OBJECT_TYPE</code>	Type of the object, such as <code>JAVA_SOURCE</code> , <code>JAVA_CLASS</code> , or <code>JAVA_RESOURCE</code> .
<code>STATUS</code>	Status of the object. The values can be either <code>VALID</code> or <code>INVALID</code> . It is always <code>VALID</code> for <code>JAVA_RESOURCE</code> .

Object Name and Type

An `OBJECT_NAME` in `USER_OBJECTS` is the alias. The fully qualified name is stored as an alias if it exceeds 30 characters.

See Also: ["Shortened Class Names"](#) on page 2-21 for information about fully qualified names and aliases.

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

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

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

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

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

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

This statement displays the `OBJECT_TYPE` of the schema object with the specified fully qualified name. This presumes that the fully qualified name is representable in the database character set.

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

This statement displays all the data stored in the `javasnm` view.

Status

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

Example: Accessing USER_OBJECTS

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

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

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

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

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

Publishing

Oracle Database enables clients and SQL to call Java methods that are loaded in the database after they are published. You publish either the object itself or individual methods. If you write a Java stored procedure that you intend to call with a trigger, directly or indirectly in SQL data manipulation language (DML) or in PL/SQL, then you must publish individual methods in the class. Using a call specification, specify how to access the method. Java programs consist of many methods in many classes. However, only a few static methods are typically exposed with call specifications.

See Also: [Chapter 6, "Publishing Java Classes With Call Specifications"](#)

Auditing

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

Since Oracle Database 10g release 2 (10.2), you can audit DDL statements for creating, altering, or dropping Java source, class, and resource schema objects, as with any other DDL statement. Oracle Database 10g release 2 (10.2) and Oracle Database 11g provides auditing options for auditing Java activities easily and directly. You can also audit any modification of Java sources, classes, and resources.

You can audit database activities related to Java schema objects at two different levels, statement level and object level. At the statement level you can audit all activities related to a special pattern of statements.

[Table 2–8](#) lists the statement auditing options and the corresponding SQL statements related to Java schema objects.

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

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

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

```
AUDIT ALTER JAVA SOURCE
```

Object level auditing provides finer granularity. It enables you to identify specific problems by zooming into specific objects.

Table 2–9 lists the object auditing options for each Java schema object. The entry X in a cell indicates that the corresponding SQL command can be audited for that Java schema object. The entry NA indicates that the corresponding SQL command is not applicable for that Java schema object.

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

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

See Also:

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

User Interfaces on the Server

Oracle Database furnishes all core Java class libraries on the server, including those associated with presentation of the user interfaces. However, it is inappropriate for code running on the server to attempt to materialize or display a user interface on the server. Users running applications in Oracle JVM environment should not be expected nor allowed to interact with or depend on the display and input hardware of the server where Oracle Database is running.

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

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

See Also:

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

Oracle JVM takes a similar approach for sound support. Applications in Oracle JVM are not allowed to access the underlying sound system for purposes of sound playback or recording. Instead, the system sound resources appear to be unavailable in a manner consistent with the sound API specification of the methods that are trying to access the resources. For example, methods in `javax.sound.midi.MidiSystem` that attempt to access the underlying system sound resources throw the `MidiUnavailableException` checked exception to signal that the system is unavailable. However, similar to the Headless AWT support, Oracle Database supports the APIs that allow sound file manipulation, free of the native sound devices. Oracle JVM also fully complies with the JCK, when it implements the sound API.

Shortened Class Names

Each Java source, class, and resource is stored in its own schema object in the server. The name of the schema object is derived from the fully qualified name, which includes relevant path or package information. Dots are replaced by slashes.

Schema object names, however, have a maximum of only 30 characters, and all characters must be legal and convertible to characters in the database character set. If any fully qualified name is longer than 30 characters or contains illegal or nonconvertible characters, then Oracle Database converts it to a short name, or alias, to use as the name of the schema object. Oracle Database keeps track of both the names and how to convert between them. If the fully qualified name is 30 characters or less and has no illegal or inconvertible characters, then it is used as the schema object name.

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

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

This function returns the fully qualified name of the Java schema object, which is specified using its alias. The following is an example of a statement used to display the fully qualified name of classes that are invalid:

```
SELECT dbms_java.longname (object_name) FROM user_objects WHERE object_type =  
'JAVA CLASS' and status = 'INVALID';
```

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

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

See Also: ["System Classes"](#) on page 1-19

Class.forName() in Oracle Database

The JLS provides the following description of `Class.forName()`:

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

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

- The JDK uses one instance of `ClassLoader` that searches the set of directory tree roots specified by the `CLASSPATH` environment variable.
- Oracle JVM defines several resolvers that specify how to locate classes. Every class has a resolver associated with it, and each class can, potentially, have a different resolver. When you run a method that calls `Class.forName()`, the resolver of the currently running class, which is `this`, is used to locate the class.

See Also: ["Resolving Class Dependencies"](#) on page 2-9

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

You can solve this resolver problem as follows:

- Avoid any class name lookup by passing the `Class` object itself.
- Supply the `ClassLoader` instance in the `Class.forName()` method.
- Supply the class and the schema it resides in to the `classForNameAndSchema()` method.
- Supply the schema and class name to `Class.forName().lookupClass()`.
- Serialize your objects with the schema name and the class name.

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

This section covers the following topics:

- [Supply ClassLoader in Class.forName\(\)](#)
- [Supply Class and Schema Names to class.forNameAndSchema\(\)](#)
- [Supply Class and Schema Names to lookupClass\(\)](#)
- [Supply Class and Schema Names when Serializing](#)
- [Class.forName Example](#)

Supply ClassLoader in Class.forName()

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

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

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

Example 2-1 Retrieve Resolver from Current Class

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

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

Example 2-2 Retrieve Resolver from Calling Class

You can retrieve the class of the instance that called the running method by using the `oracle.aurora.vm.OracleRuntime.getCallerClass()` method. After you retrieve the class, call the `Class.getClassLoader()` method on the returned class. The following example retrieves the class of the instance that called the `workForCaller()` method. Then, its class loader is retrieved and supplied to the `Class.forName()` method. As a result, the resolver used for looking up the class is the resolver of the calling class.

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

Supply Class and Schema Names to classForNameAndSchema()

You can resolve the problem of where to find the class by supplying the resolver, which can identify the schemas to be searched. Alternatively, you can supply the schema in which the class is loaded. If you know in which schema the class is loaded, then you can use the `classForNameAndSchema()` method, which is in the `DbmsJava` class provided by Oracle Database. This method takes both the name of the class and the schema in which the class resides and locates the class within the designated schema.

Example 2-3 Providing Schema and Class Names

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

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

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

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

Supply Class and Schema Names to lookupClass()

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

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

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

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

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

Supply Class and Schema Names when Serializing

When you deserialize a class, part of the operation is to lookup a class based on a name. To ensure that the lookup is successful, the serialized object must contain both the class and schema names.

Oracle Database provides the following classes for serializing and deserializing objects:

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

Class.forName Example

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

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

public class ForName
{
    private Class from;

    /* Supply an explicit class to the constructor */
    public ForName(Class from)
    {
        this.from = from;
    }

    /* Use the class of the code containing the "new ForName()" */
    public ForName()
    {
        from = OracleRuntime.getCallerClass();
    }

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

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

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

- To use the resolver of the class of an instance, call `lookupWithClassLoader()`. This method supplies a class loader to the `Class.forName()` method in the `from` variable. The class loader specified in the `from` variable defaults to this class.

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

Managing Your Operating System Resources

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

In addition, when you use shared servers, your operating system resources, which are contained within Java objects, can be invalidated if they are maintained across calls within a session.

See Also: ["Operating System Resources Affected Across Calls"](#) on page 2-54

The following sections discuss these potential problems:

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

Overview of Operating System Resources

In general, your operating system resources contain the following:

Operating System Resources	Description
memory	Oracle Database manages memory internally, allocating memory as you create objects and freeing objects as you no longer need them. The language and class libraries do not support a direct means to allocate and free memory. See Also: "Memory Spaces Management" on page 1-10.
files and sockets	Java contains classes that represent file or socket resources. Instances of these classes hold on to the file or socket constructs, such as file handles, of the operating system.
threads	Oracle JVM threads provide no additional scalability over what is provided by the database support of multiple concurrently executing sessions. However, Oracle JVM supports the full Java threading API. See Also: "Threading in Oracle Database" on page 2-46.

Operating System Resource Access

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

See Also: ["Java2 Security"](#) on page 10-2

Operating System Resource Lifetime

You can access operating system resources using the standard core Java classes and methods. Once you access a resource, the time that it remains active varies according to the type of resource. Memory is garbage collected. Files, threads, and sockets persist across calls when you use a dedicated mode server. In shared server mode, files, threads, and sockets terminate when the call ends.

See Also: ["Operating System Resources Affected Across Calls"](#) on page 2-54.

Garbage Collection and Operating System Resources

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

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

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

For performance reasons, the garbage collector cannot examine each object to see if it contains a handle. As a result, the garbage collector collects Java objects and variables, but does not issue the appropriate operating system methods for freeing any handles.

[Example 2-4](#) shows how to close the operating system constructs.

Example 2-4 Closing Your Operating System Resources

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

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

    /*closing the file, which frees up the operating system file handle*/
    in.close();
}
```

```
}
```

If you do not close `inFile`, then eventually the `File` object will be garbage collected. Even after the `File` object is garbage collected, the operating system treats the file as if it were in use, because it was not closed.

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

Managing Your Applications Using JMX

This section contain the following topics:

- [Overview of JMX](#)
- [Enabling and Starting JMX in a Session](#)
- [OJVM JMX Defaults and Configurability](#)
- [Examples of SQL calls to `dbms_java.start_jmx_agent`](#)
- [Using JConsole to Monitor and Control Oracle JVM](#)
- [Important Security Notes](#)

Overview of JMX

JMX (Java Management Extensions) is a Java technology that supplies tools for managing and monitoring applications, system objects, devices, service-oriented networks, and JVM (Java Virtual Machine). This API allows its classes to be dynamically constructed and changed. So, you can use this technology to monitor and manage resources as they are created, installed, and implemented. The JMX API also includes remote access, so a remote management program can interact with a running application for these purposes.

In JMX, a given resource is instrumented by one or more Java objects known as MBeans (Managed Beans). These MBeans are registered in a core managed object server, known as an MBean server, that acts as a management agent and can run on most devices enabled for the Java programming language. A JMX agent consists of an MBean server, in which MBeans are registered, and a set of services for handling MBeans.

See Also:

- <http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement/>
- <http://java.sun.com/j2se/1.5.0/docs/guide/security/jsse/JSSERefGuide.html>
- <http://java.sun.com/j2se/1.4.2/docs/guide/security/jaas/JAASLMDevGuide.html>

Enabling and Starting JMX in a Session

Oracle Database 11g release 1 (11.1) introduced a new role `JMXSERVER` and a new procedure `dbms_java.start_jmx_agent` to support JMX in the database. The `JMXSERVER` role is granted specific Java permissions that enable you to start and run `MBeanServer` and JMX agent in a session. The procedure `dbms_java.start_jmx_agent` starts the agent in a specific session that generally remains active for the duration of the session. Perform the following to enable and start JMX:

1. Obtain `JMXSERVER` from `SYS` or `SYSTEM`:

```
SQL> grant jmxserver to scott;
```

where, `scott` is the user name.

2. Invoke the procedure `dbms_java.start_jmx_agent` to startup JMX in the session. The `dbms_java.start_jmx_agent` procedure can be invoked with the following arguments:

`port`: the port for the JMX listener. The value of this parameter sets the Java property `com.sun.management.jmxremote.port`.

`ssl`: sets the value for the Java property `com.sun.management.jmxremote.ssl`. Case for `true` and `false` values is ignored.

`auth`: the value for the property `com.sun.management.jmxremote.authenticate`, otherwise a semicolon-separated list of Java Authentication and Authorization Service (JAAS) credentials. The value is not case-sensitive.

Each of these arguments can be `null` or omitted, with `null` as the default value. when an argument is `null`, it does not alter the previously present value of the corresponding property in the session.

Note: The Java properties corresponding to the parameters of `dbms_java.start_jmx_agent` are from the set of Java properties specified in Sun Java 5.0 JMX documentation. For the full list of Java JMX properties please refer to <http://java.sun.com/j2se/1.5.0/docs/guide/management/agent.html>

The `dbms_java.start_jmx_agent` procedure starts an agent activating OJVM JMX server and a listener. The JMX server runs as one or more daemon threads in the current session and in general is available for the duration of the session. Once JMX Agent is started in a session, Java code running in the session can be monitored.

The `dbms_java.start_jmx_agent` procedure is a PL/SQL wrapper for the Java method `oracle.aurora.rdbms.JMXAgent.startOJVMAgent`, which by itself can also be called programmatically from Java stored procedures. This method usually exits immediately after the JMX agent is started. So, the following statement ends the Java call immediately after the `startOJVMAgent` method returns.

```
call dbms_java.start_jmx_agent(...);
```

But, the MBeanServer and the JMX Agent remain initialized and fully functional for the remaining lifetime of the session. Note that any Java code runs in a given RDBMS session only when there is a Java call on the stack, and JMX connectivity is provided by means of daemon threads; so, no JMX interaction with clients can occur during the time intervals between two subsequent Java calls. Typically, JMX connectivity freezes at the end of each call, unfreezes automatically at the start of the next call in dedicated (shadow) processes, and drops entirely at the end of the call on the shared server processes. In the default dedicated process configuration, daemon threads go into a quiescent state for the time duration between calls and can often wake up to resume the same socket connection transparently, unless the time interval between Java calls exceeds the timeout intervals of the client and the server. This enables clients such as JConsole to use the same physical connection across calls. In shared server processes, socket connections cannot survive call boundaries, and JMX clients such as JConsole need to reconnect during each following call. This connectivity issue relates only to physical socket connections. In both modes, the `startOJVMAgent` method needs to be invoked only once per session.

A different mode of JMX monitoring is possible with the `EXIT_CALL_WHEN_ALL_THREADS_TERMINATE` policy. By setting the call exit policy to `OracleRuntime.OracleRuntime.EXIT_CALL_WHEN_ALL_THREADS_TERMINATE`, you can configure the session to run JMX server continuously in a call that invokes the `startOJVMAgent` method till the Java call is exited programmatically. This mode is convenient when various Java tasks are fired up from a JMX client as operations of specific MBeans. This way, continuous JMX management and monitoring is driven by these operations. Please refer to the JVM JMX demo for such a bean, for example, `jmxserv.Load`.

OJVM JMX Defaults and Configurability

When `dbms_java.start_jmx_agent` is activated, the property `com.sun.management.jmxremote` is set to `true`. Before invoking `start_jmx_agent`, a JMXSERVER-privileged user can preset various management properties in the following ways:

- Using the PL/SQL function `dbms_java.set_property`
- Invoking method `java.lang.System.setProperty`

The JMXSERVER role user can also preset the properties in database resident Java resource specified by Java property `com.sun.management.config.file`. The default name for this resource, tried when `com.sun.management.config.file` is not set, is `lib.management.management.properties`. This resource mechanism is Oracle JVM extension of Sun's file-based JMX configuration management. This mechanism is superior for Oracle JVM as it provides more security and per-schema management. When the resource does not exist in schema, a file-read is attempted as a fall-back. The default file path, tried when `com.sun.management.config.file` is not set, is `$(java.home)/lib/management/management.properties`. In Oracle Database 11g this file contains the following presets:

```
com.sun.management.jmxremote.ssl.need.client.auth = true
com.sun.management.jmxremote.authenticate = false
```

The property `com.sun.management.jmxremote.ssl.need.client.auth` in conjunction with `com.sun.management.jmxremote.ssl`, sets JMX for two-way encrypted SSL authentication with client and server certificates. The default value of

`com.sun.management.jmxremote.ssl` is `true`. This configuration is the default and is preferred over JAAS password authentication.

Note: For more information visit the following:

- <http://java.sun.com/j2se/1.5.0/docs/guide/management/agent.html>
 - <http://java.sun.com/j2se/1.5.0/docs/guide/security/jsse/JSSERefGuide.html#CustomizingStores>
-

Note: The default Sun JMX Login Module providing file-based store for passwords is not supported in Oracle JVM for security reasons¹. So, if JAAS password authentication must be used instead of SSL client authentication, then pass transient JAAS credentials securely as the `auth` parameter to `dbms_java.start_jmx_agent` as illustrated in this section, or configure JMX to use a secure custom LDAP login module.

¹ An RMI-related JAAS vulnerability identified in early versions of JDK 5.0 is fixed in JDK 5.0_10, on which the 11g OJVM is based. But, still the default JMX login module uses clear-text files to store passwords and is not supported. Files `jmxremote.access` do not store passwords and are supported.

Examples of SQL calls to `dbms_java.start_jmx_agent`

Following are some examples of starting the JMX server:

- Starts the JMX server and the listener using the default settings as described in the preceding sections or the values set earlier in the same session:

```
call dbms_java.start_jmx_agent;
```

- Starts the JMX server and the listener using the default settings as described in the preceding sections or the values set earlier in the same session:

```
call dbms_java.start_jmx_agent(null, null, null);
```

- Starts the JMX server and the listener on port 9999 with the other JMX settings having the default values or the values set earlier in the same session:

```
call dbms_java.start_jmx_agent('9999');
```

- Starts the JMX server and the listener on port 9999 with the other JMX settings having the default values or the values set earlier in the same session:

```
call dbms_java.start_jmx_agent('9999', null, null);
```

- Starts the JMX server and the listener with the JMX settings having the default values or the values set earlier in the same session and with JAAS credentials `monitorRole/1z2x` and `controlRole/2p3o`:

```
call dbms_java.start_jmx_agent(null, null,
'monitorRole/1z2x;controlRole/2p3o');
```

These credentials are transient. The property `com.sun.management.jmxremote.authenticate` is set to `true`.

- Starts JMX listener on port 9999 with no SSL and no JAAS authentication. Used only for development or demonstration.

See Also: ["Important Security Notes"](#) on page 2-46

```
call dbms_java.start_jmx_agent('9999', 'false', 'false');
```

Using JConsole to Monitor and Control Oracle JVM

This section describes how to use JConsole, a Sun JMX client tool, for monitoring and controlling Oracle JVM. JConsole is a part of Sun Java JDK.

This section discusses the following topics:

- [The jconsole Command](#)
- [The JConsole interface](#)
- [The OracleRuntime MBean](#)
- [Memory Thresholds](#)

Note: To start JConsole, you should have a server-side Java session running JMX Agent. For more information refer [Enabling and Starting JMX in a Session](#).

The jconsole Command

Use the `jconsole` command syntax to start JConsole. The simplest format to start the JConsole tool is the following:

```
jconsole [hostName:portNum]
```

where:

- `hostname` is the name of the system running the application
- `portNum` is the port number of the JMX listener

In the following examples, we connect to a host with name `example.com` through default port 9999. This mode assumes no authentication and encryption. This mode is adequate only for demo or testing, and can be used to connect to Oracle JVM JMX sessions that are started with the following command:

```
call dbms_java.start_jmx_agent(portNum, false);
```

Remember that you can connect to and interact with Oracle JVM from JConsole, only when the daemon threads of the server are running and are not dormant. This means that there should be an active Java call in the session, which is running the JMX server on the specified port. During the time interval between subsequent Java calls, JMX server preserves its state and statistics, but is unable to communicate with JConsole.

See Also: [Important Security Notes](#)

The JConsole interface

The JConsole interface consists of the following tabs:

- Summary tab
It displays summary information on Oracle JVM and the values monitored by JMX. For more information see [Viewing Oracle JVM Summary Information](#).
- Memory tab

It displays information on memory usage. For more information see [Monitoring Memory Consumption](#).

- Threads tab

It displays information on thread usage. For more information see [Monitoring Thread Use](#).

- Classes tab

It displays information on class loading. For more information see [Monitoring Class Loading](#).

- MBeans tab

It displays information on MBeans. For more information see [Monitoring and Managing MBeans](#).

- VM tab

It displays information on Oracle JVM. For more information see [Viewing VM Information](#).

Important: In Oracle Database 11g Release 2, the data collected and passed to JConsole is limited to the Oracle JVM session that runs the JMX agent. This data does not include the attributes of other sessions that may be running in Oracle JVM. One exception is the `OracleRuntime` MBean that provides information about many `WholeJVM_` Attributes and operations of Oracle JVM. For more information on `OracleRuntime` MBean, refer to "[The OracleRuntime MBean](#)" on page 2-42.

Viewing Oracle JVM Summary Information

You can use the Summary tab of the JConsole interface to view Oracle JVM Summary Information. This tab displays key monitoring information on thread usage, memory consumption, class loading, and other VM and operating system specifics.

If JConsole successfully connects to an Oracle JVM session running a JMX Agent, then the Summary Tab looks the following figure:

Figure 2–6 The Summary Tab of the JConsole Interface

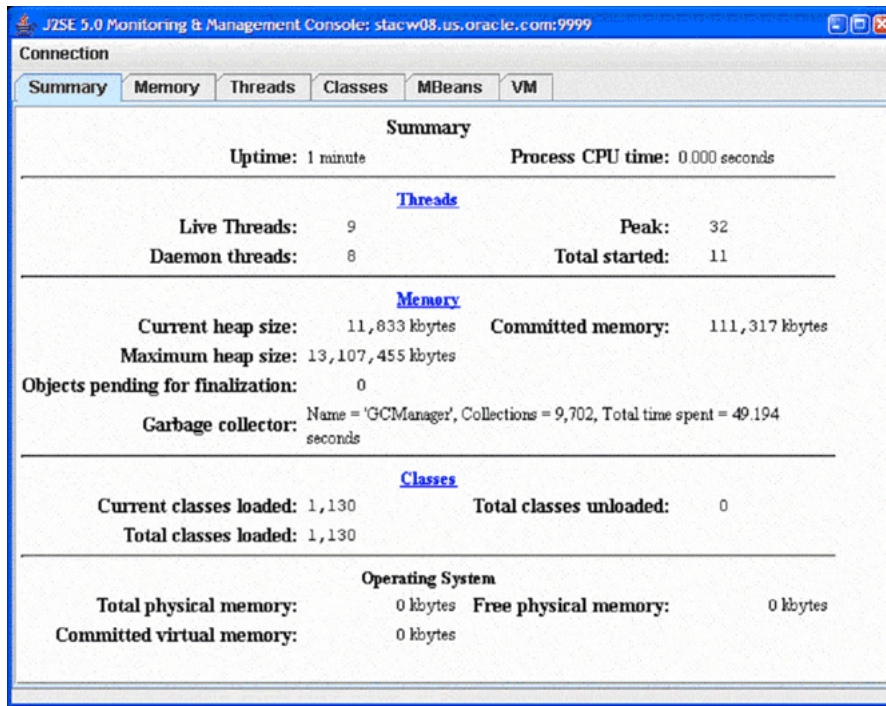


Table 2–10 provides description of the fields present in the Summary tab.

Table 2–10 Description of the Summary Tab Fields in JConsole Interface

Field	Description
Uptime	The duration for which the Oracle JVM session has been running.
Process CPU time	This information is not gathered for Oracle JVM sessions in Oracle Database 11g Release 2.
Live threads	The current number of live daemon and non-daemon threads.
Peak	Highest number of live threads since Oracle JVM started.
Daemon threads	Current number of live daemon threads.
Total started	Total number of threads started since Oracle JVM started. It includes daemon, non-daemon, and terminated threads.
Current heap size	Number of kilobytes currently occupied by the heap.
Committed memory	Total amount of memory allocated for use by the heap.
Maximum heap size	Maximum number of kilobytes occupied by the heap.
Objects pending for finalization	Number of objects pending for finalization.
Garbage collector information	Information about the garbage collector, which includes name, number of collections performed, and total time spent performing garbage collection.
Current classes loaded	Number of classes currently loaded into memory for execution.
Total classes loaded	Total number of classes loaded into session memory since the session started.

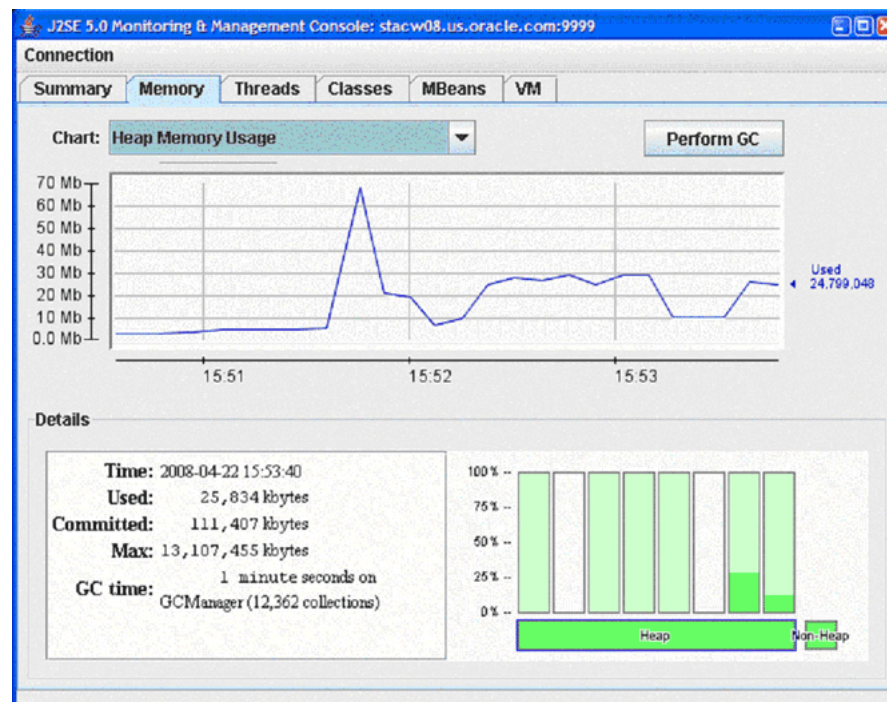
Table 2–10 (Cont.) Description of the Summary Tab Fields in JConsole Interface

Field	Description
Total classes unloaded	Number of classes unloaded from memory. Typically this is zero for Oracle Database 11g Release 2.
Total physical memory	This information is not gathered for Oracle JVM sessions in Oracle Database 11g Release 2. So, the value displayed is zero.
Free physical memory	This information is not gathered for Oracle JVM sessions in Oracle Database 11g Release 2. So, the value displayed is zero.
Committed virtual memory	This information is not gathered for Oracle JVM sessions in Oracle Database 11g Release 2. So, the value displayed is zero.

Monitoring Memory Consumption

You can use the Memory tab of the JConsole interface to monitor memory consumption. This tab provides information on memory consumption and memory pools.

Figure 2–7 shows the Memory tab.

Figure 2–7 The Memory Tab of the JConsole Interface

The chart on the Memory tab shows Oracle JVM memory usages versus time, for the whole memory space and also for specific memory pools. The memory pools available for Oracle JVM reflect the internal organization of Oracle JVM and correspond to object memories of Oracle JVM Memory Manager. The available memory pools in this release of Oracle Database are:

- New Generation Space
 - This is the memory pool from which memory is initially allocated for most objects. This pool is also referred to as the Eden Space.
- Old Generation Space

This memory pool contains objects that have survived the garbage collection process in Eden Space. This pool is also referred to as the Survival Space.

- Malloc/Free Space

This memory pool contains objects for which memory is allocated and freed in malloc/free fashion.

- End of Migration Space

This memory pool contains objects surviving end-of-session migration.

- Dedicated Session Space

This memory pool is used to allocate memory to session objects in Oracle Dedicated Sessions mode.

- Paged Session Space

This memory pool is used to allocate memory to session objects that are big and paged.

- Run space

This memory pool is used to allocate memory to temporary and auxiliary objects.

- Stack space

This memory pool is used to allocate memory to temporary objects for which memory is allocated and freed in stack-like fashion.

The Details area in the Memory tab displays current memory matrixes that include the following:

- Used

This matrix indicates the amount of memory currently used by the process running the session.

- Committed

This matrix indicates the amount of memory guaranteed to be available for use by Oracle JVM, as if the memory has already been allocated. The amount of Committed memory may change over time. But Committed memory will always be greater than or equal to Used memory.

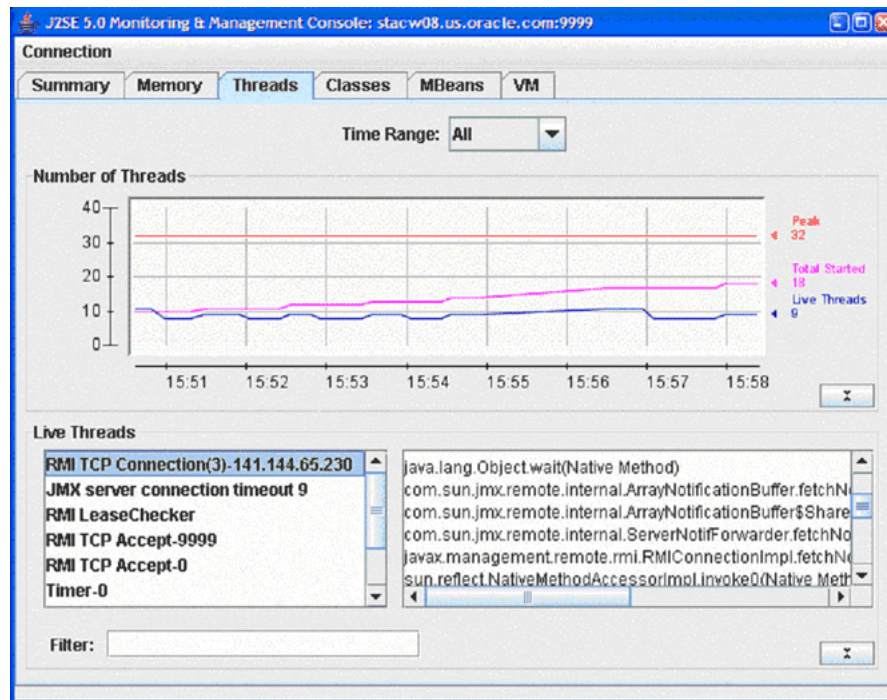
- Max

This matrix indicates the maximum amount of memory that can be used for memory management. It usually corresponds to the initial configuration of Oracle JVM.

The bar chart at the lower right corner of the Memory tab shows memory consumed by the individual memory pools. The bar turns red when the memory used exceeds the memory usage threshold. You can set the memory usage threshold through an attribute of the `MemoryMXBean`. For more information, see [Memory Thresholds](#).

Monitoring Thread Use

You can use the Threads tab of the JConsole interface to monitor thread usage.

Figure 2–8 The Threads Tab of the JConsole Interface

The chart on the Threads tab displays the number of live threads versus time, with a particular color representing a particular type of thread:

- Magenta signifies total number of threads
- Red signifies peak number of threads
- Blue signifies number of live threads

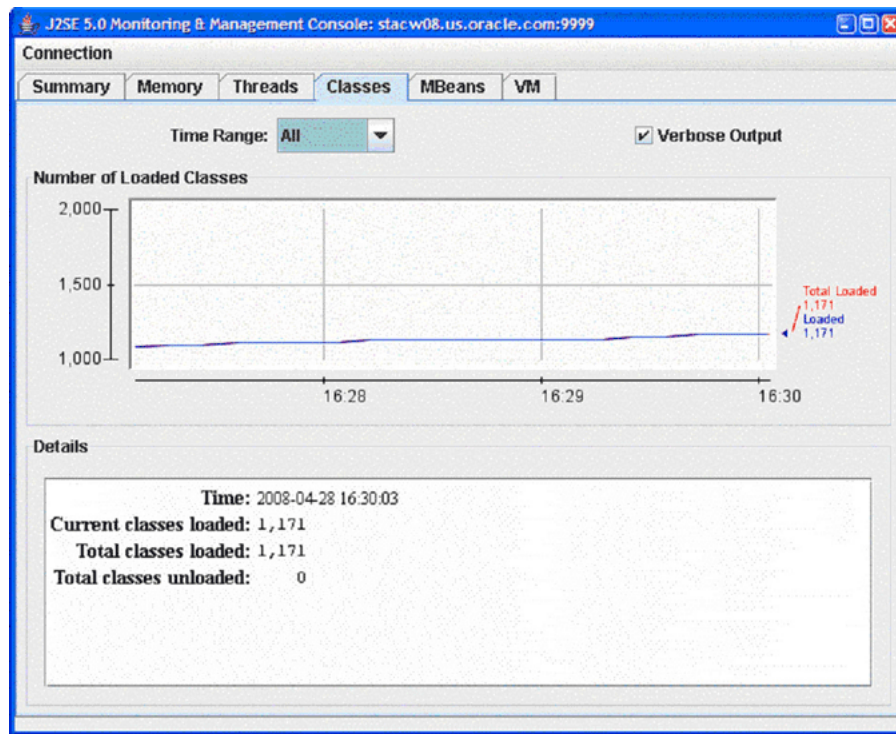
The list of threads on this tab displays the active threads. Select a thread in the list to display information about that thread on the right pane. This information includes name, state, and stack trace of the thread.

The Filter field helps to narrow the threads.

Monitoring Class Loading

You can use the Classes tab of the JConsole interface to monitor class loading. The chart on this tab plots the number of classes loaded versus time.

Figure 2–9 The Classes tab of the JConsole interface

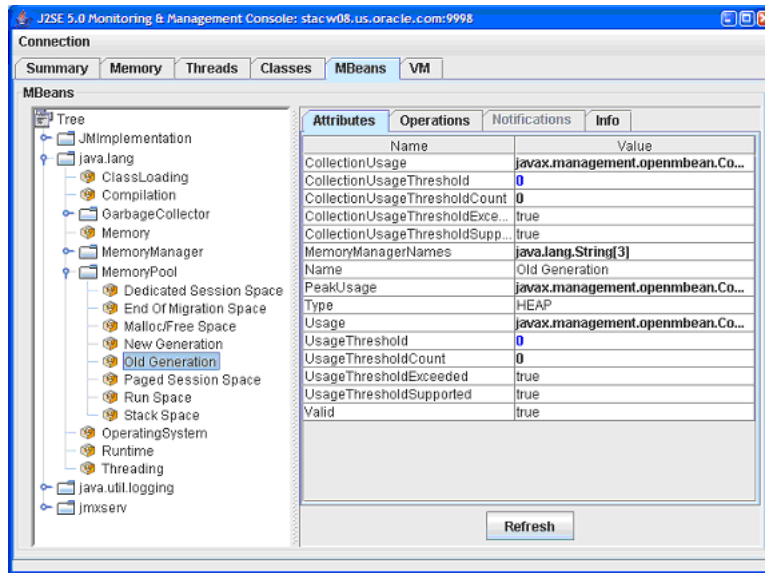


Monitoring and Managing MBeans

You can use the MBeans to monitor and manage MBeans. This tab displays information on all the MBeans registered with the platform MBean server.

The tree on the left pane of the MBean tab is called the MBean tree and it shows all the MBeans, organized according to their object Names. When you select an MBean in the MBean tree, then its attributes, operations, notifications, and other information are displayed on the right pane. For example, in [Figure 2–10](#), we have selected the Old Generation MemoryPool MBean in the MBean tree on the left and the attributes of the Old Generation MemoryPool MBean are displayed on the right.

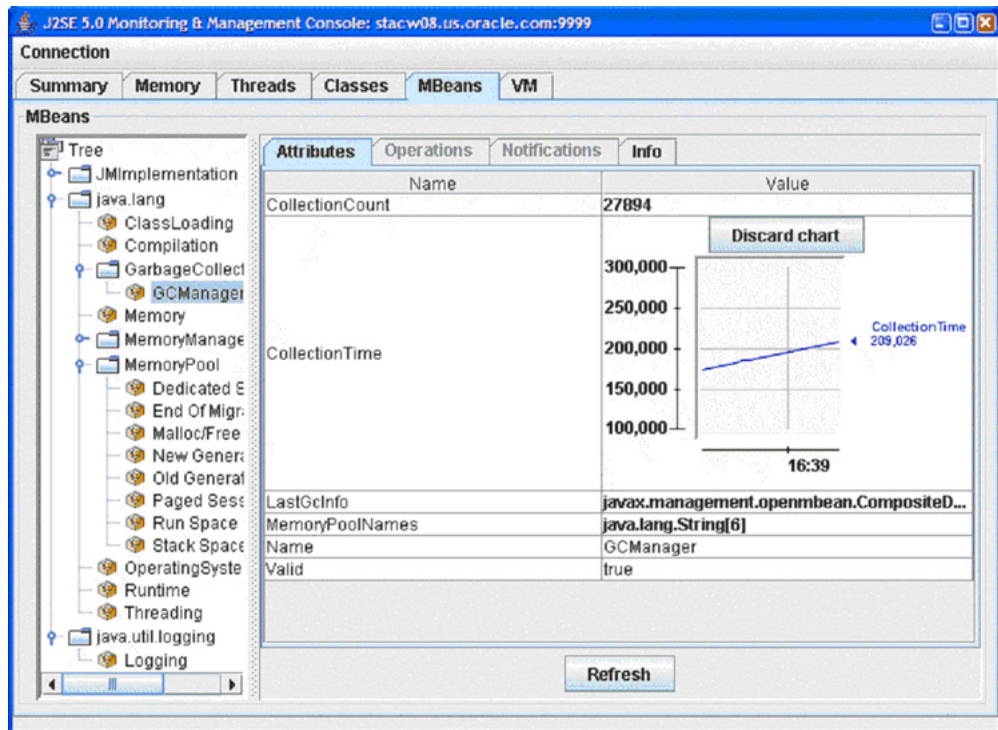
Figure 2–10 *Displaying the Attributes of an MBean*



You can set the values of an attribute, if it is writeable. The writeable values are displayed in blue color. For example, in [Figure 2–10](#), the attributes `CollectionUsageThreshold` and `UsageThreshold` are writeable.

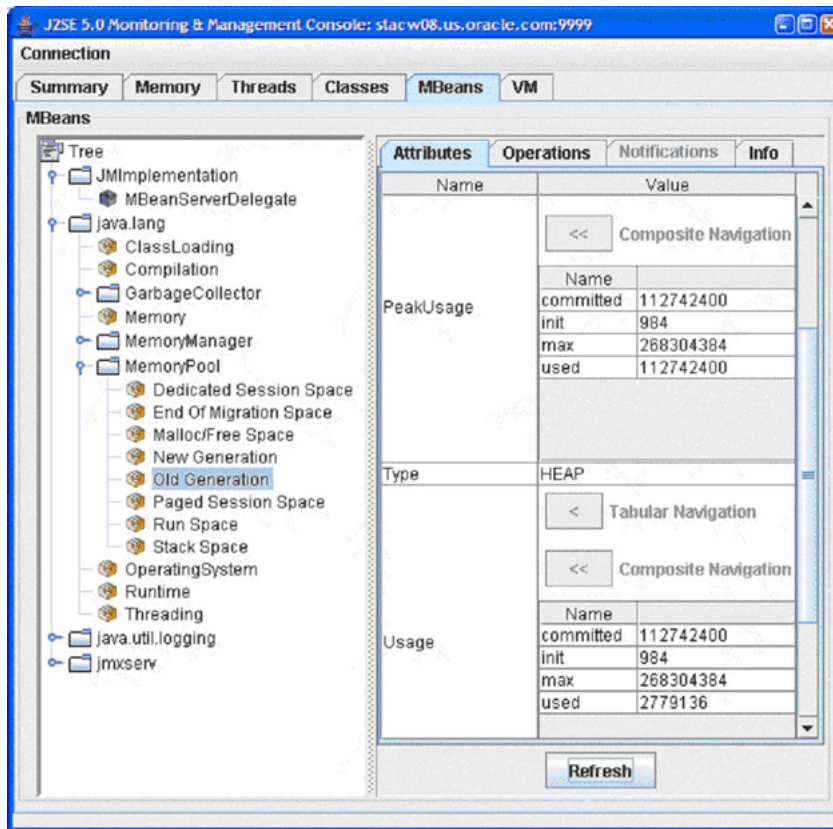
You can also display a chart of the values of an attribute versus time, by double-clicking on the attribute value. For example, if you click on the value of the `CollectionTime` property of the `GCManger` MBean, then you will see a chart similar to [Figure 2–11](#):

Figure 2–11 *Displaying a Chart of the Values of an Attribute*



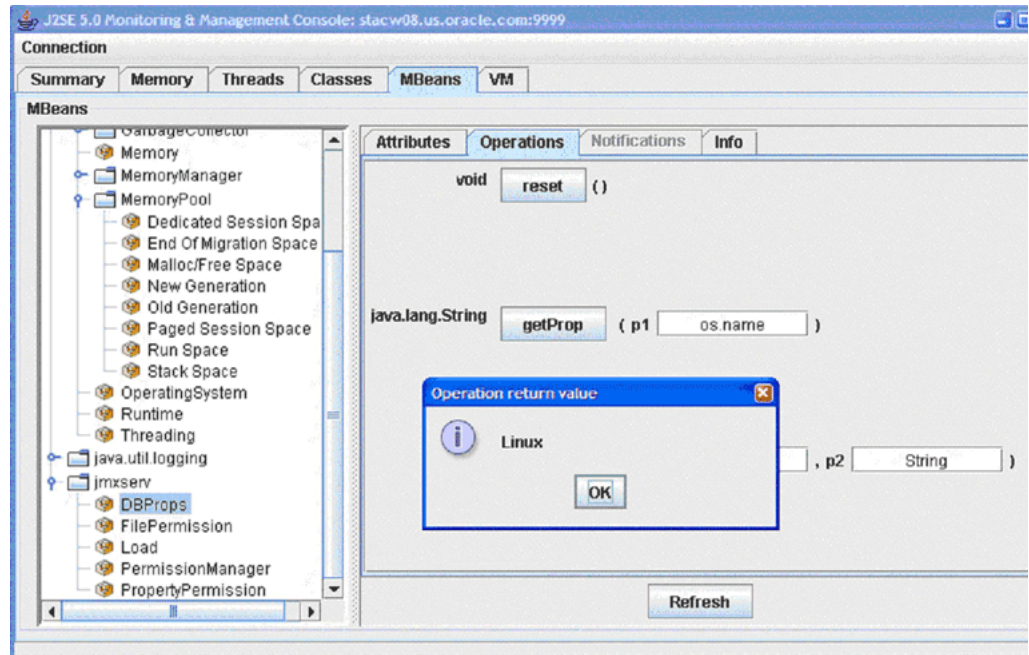
You can display the details of a complex attribute by clicking on the attribute. For example, you can display the details of Usage and PeakUsage attributes of the Memory Pools as shown in [Figure 2–12](#):

Figure 2–12 *Displaying Details of a Complex Attribute in the MBeans Tab*



The Operations tab of an MBean provides manageability interface. For example, garbage collection can be invoked on a Memory Pool or Memory Manager by clicking **Perform Garbage Collection**. The JMX demo of Oracle JVM, namely, `javavm/demo/jmx/`, provides several additional custom MBeans that are loaded into Oracle JVM. Here is an example shows the result of the `getProp` operation of the `DBProps` Mbean:

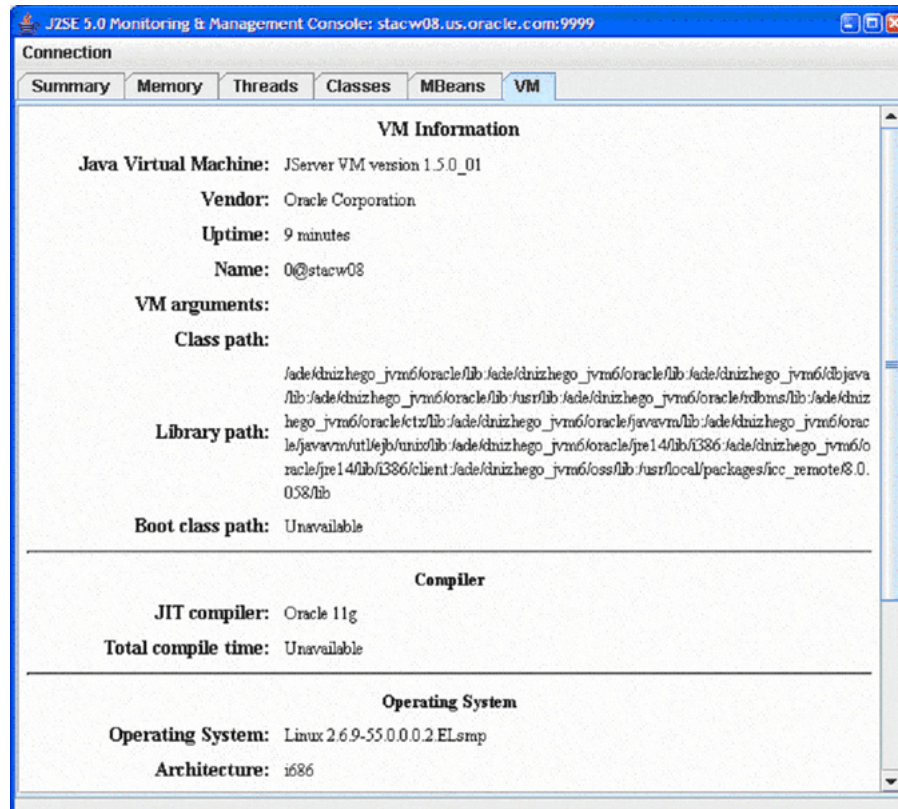
Figure 2–13 Operations Tab of the MBeans Tab of the JConsole Interface



Viewing VM Information

You can use the VM tab of the JConsole interface to view VM information.

Figure 2–14 The VM Tab of the JConsole Interface

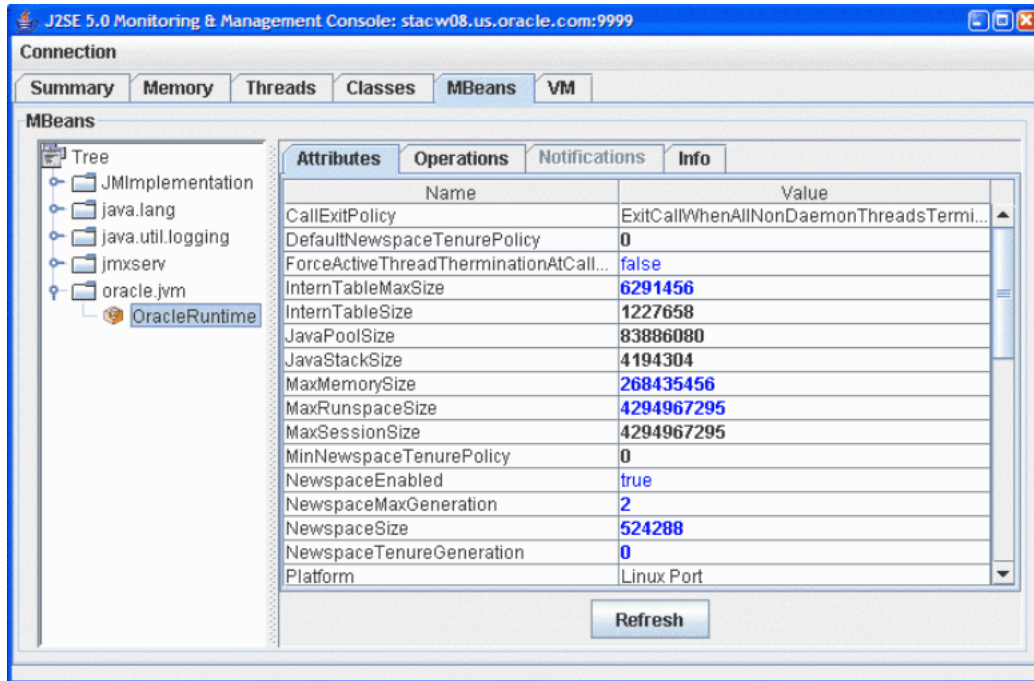


The OracleRuntime MBean

Starting from Oracle Database 11g Release 2 (11.2), a new MBean, `OracleRuntime` is added to the list of Oracle JVM platform MBeans, when the `dbms_java.start_jmx_agent` procedure is called. This MBean is specific to Oracle JVM.

The Attributes Tab of the `OracleRuntime` MBean exposes most of the parameters manipulated by the `oracle.aurora.vm.OracleRuntime` class. Figure 2–15 shows the Attributes tab of the `OracleRuntime` MBean.

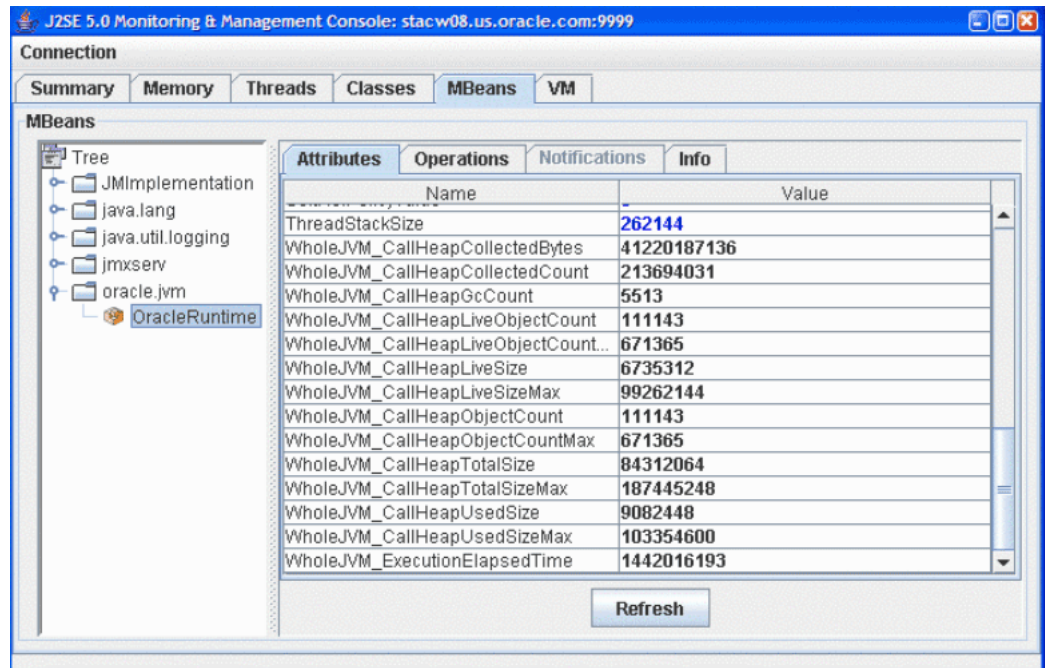
Figure 2–15 Attributes Tab of the OracleRuntime MBean



The parameters displayed in black color are read-only and cannot be modified. For example, `JavaPoolSize`, `Platform`, and so on. Values in blue color are read/write, which means you can modify them. Most of the attributes of the `OracleRuntime` MBean are local to the current session.

The `WholeJVM_` attributes of the `OracleRuntime` MBean are global. These attributes reflect the totals of Oracle JVM memory usage statistics across all Java-enabled sessions in the Database instance, as gathered from the `v$session` and `v$sesstat` performance views. Figure 2–16 displays the `WholeJVM_` attributes of the `OracleRuntime` MBean.

Figure 2–16 OracleRuntime MBean



The Operations Tab of the OracleRuntime MBean exposes many of the operations of the oracle.aurora.vm.OracleRuntime class.

In addition, individual memory consumption statistics of a specific Java-active Database session can be monitored using the sessionsRunningJava and sessionDetailsBySID operations as shown in Figure 2–17 and Figure 2–18.

Figure 2–17 Operation sessionsRunningJava

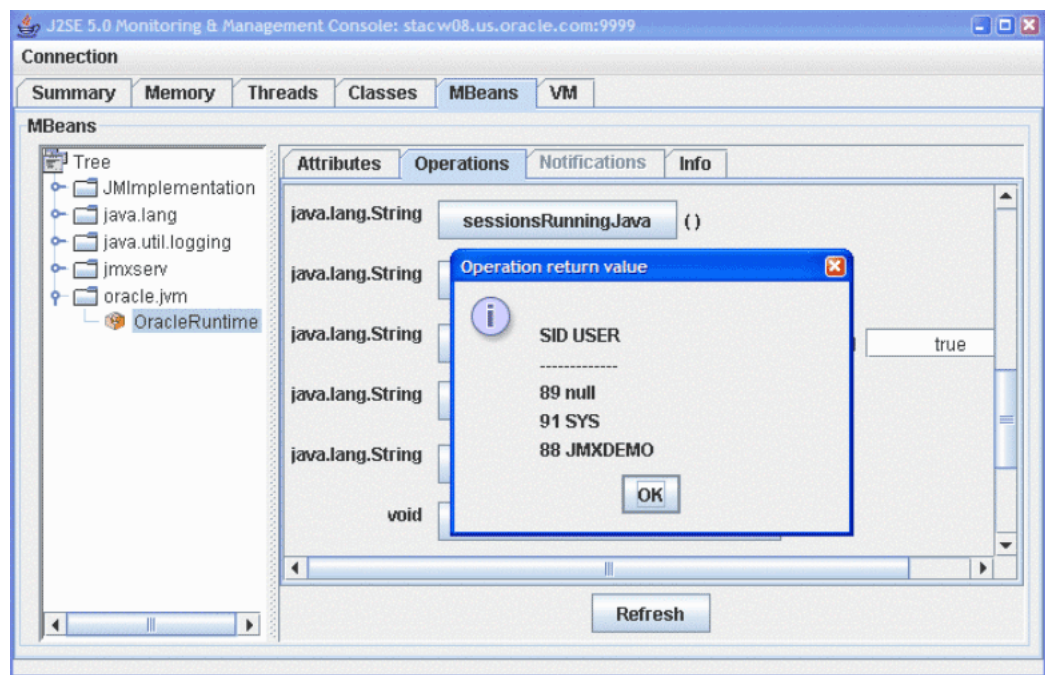
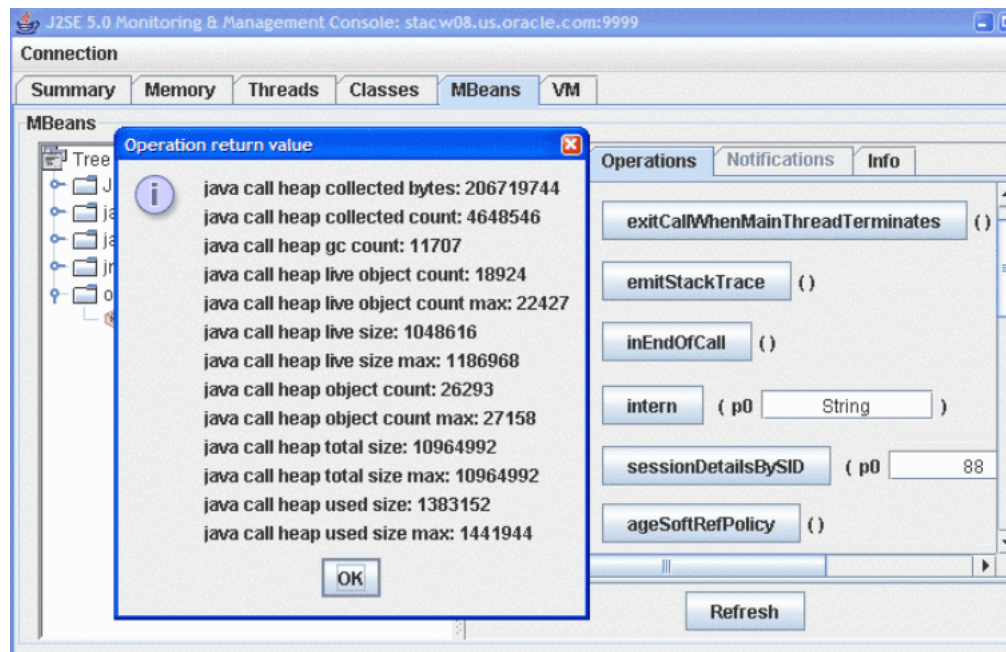


Figure 2–18 Operation sessionDetailsBySID

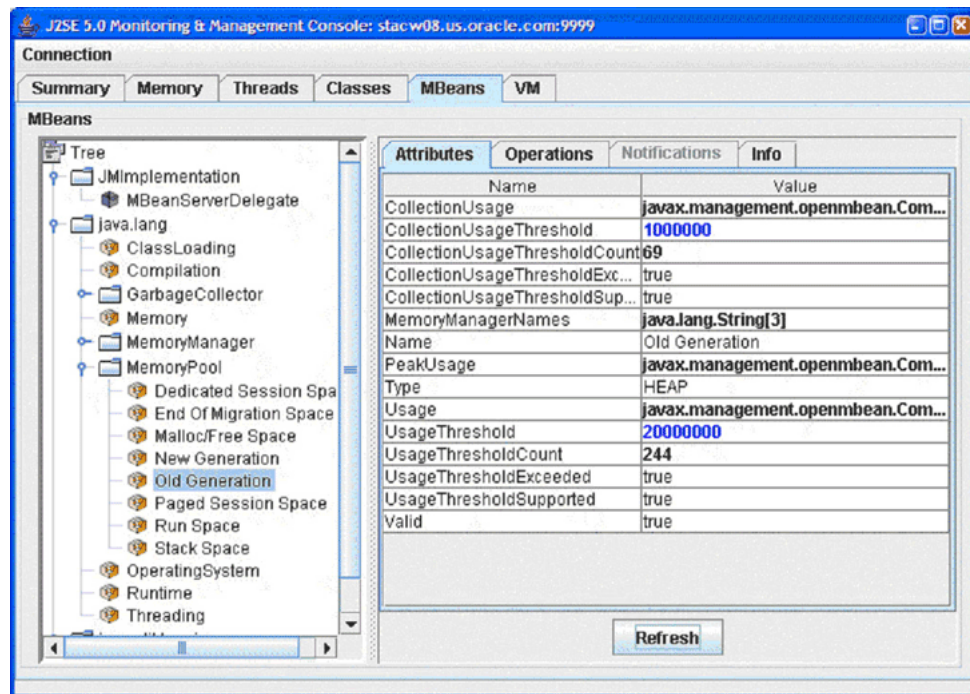


Memory Thresholds

The usage threshold is a manageable attribute of the memory pools. Collection usage threshold is a manageable attribute of some of the garbage-collected memory pools. You can set each of these to a positive value to enable corresponding threshold checking for a pool. Setting a threshold to zero disables the threshold checking for the memory pool. By default, threshold checking for all Oracle JVM pools is disabled.

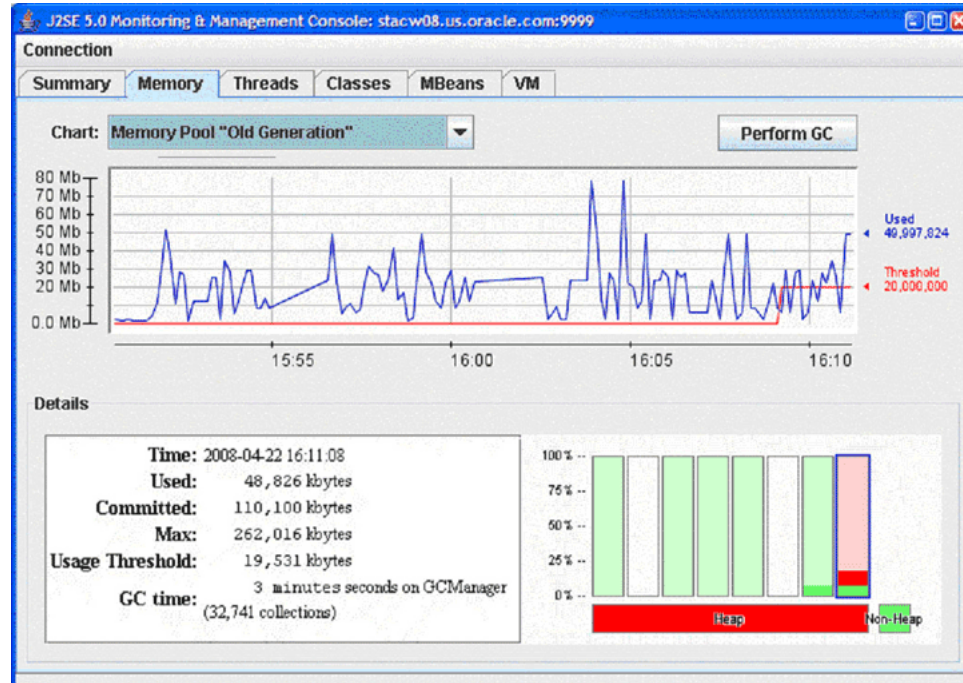
The usage threshold and the collection usage threshold are set in the MBeans tab. For example, if you select the Old Generation memory pool from the tree on the left pane, and set the usage threshold of this memory pool to 20 megabytes and the collection threshold to 1 megabyte, then after a while, the threshold counts will show the number of threshold crossing events as shown in [Figure 2–19](#):

Figure 2–19 Setting the Usage Threshold and Collection Usage Threshold in the MBeans Tab



When the memory usage of the Old Generation memory pool exceeds 20 megabytes, then part of the bar representing the Old Generation memory pool in the JConsole interface turns red. The red portion indicates the portion of used memory that exceeds the usage threshold. The bar representing the heap memory also turns red as shown in Figure 2–20:

Figure 2–20 Memory Tab of the JConsole Interface When Used Memory Exceeds the Usage Threshold



Important Security Notes

By starting the remote listener with disabled SSL and authentication you violate the general security guidelines and hence make server vulnerable to attacks. Therefore, it is always advisable not to use such mode in production environment. This mode is supported for compatibility with JDK and for development; any production use of JMX in OJVM must use secure JMX connections.

When supplying security-related property values to `dbms_java.set_property`, `System.setProperty`, or `dbms_java.start_jmx_agent`, use a non-echo listener or invoke these through an encrypted JDBC connection from a secure application layer, such as Oracle Application Server. Do not store passwords in clear-text files. Use Oracle Wallet to create and manage certificates. Use client certificates for SSL authentication for better security.

See Also: *Oracle Database Security Guide* for more information about Oracle database security features

Threading in Oracle Database

Oracle JVM is based on the database session model, which is a single-client, nonpreemptive threading model. Although Java in Oracle Database allows running threaded programs, it is single-threaded at the execution level. In this model, JVM runs all Java threads associated with a database session on a single operating system thread. Once dispatched, a thread continues execution until it explicitly yields by calling `Thread.yield()`, blocks by calling `Socket.read()`, or is preempted by the execution engine. Once a thread yields, blocks or is preempted, JVM dispatches another thread.

Note: Starting with 11g release 1 (11.1), Oracle JVM supports thread preemption. Thread preemption is not mandated by the Java specification, but is needed to support the new `java.util.concurrent` API, present in JDK1.5, properly.

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

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

Thread Life Cycle

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

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

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

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

See Also: ["Operating System Resources Affected Across Calls"](#) on page 2-54.

Prior to 11g release 1 (11.1), when a call ended because of a call to `System.exit()` or `oracle.aurora.vm.OracleRuntime.exitCall()`, Oracle JVM ended the call abruptly and terminated all threads, in both the dedicated and shared server modes. Since 11g release 1 (11.1), this is addressed by the addition of the following PL/SQL functions to the `DBMS_JAVA` package:

- `FUNCTION endsession RETURN VARCHAR2;`
- `FUNCTION endsession_and_related_state RETURN VARCHAR2;`

See Also: ["Two-Tier Duration for Java Session State"](#) on page 4-4.

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

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

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

The behavior of `OracleRuntime.exitCall()` varies depending on `OracleRuntime.threadTerminationPolicy()`. This method returns a boolean value. If this value is `true`, then any active thread should be terminated, rather than left quiescent, at the end of a database call.

- In a shared server process, `threadTerminationPolicy()` is always `true`.
- In a shadow (dedicated) process, the default value is `false`. You can change the value by calling `OracleRuntime.setThreadTerminationPolicy()`.
 - If you set the value to `false`, that is the default value, all threads are left quiescent but receive a `ThreadDeath` exception for graceful termination.
 - If the value is `true`, all threads are terminated abruptly.

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

- `OracleRuntime.EXIT_CALL_WHEN_MAIN_THREAD_TERMINATES`

If set to this value, then as soon as the main thread returns or an uncaught exception occurs on the main thread, all remaining threads, both daemon and non-daemon are:

 - Killed, if `threadTerminationPolicy()` is `true`, always in shared server mode.
 - Left quiescent, if `threadTerminationPolicy()` is `false`.
- `OracleRuntime.EXIT_CALL_WHEN_ALL_NON_DAEMON_THREADS_TERMINATE`

This is the default value. If this value is set, then the call ends when only daemon threads are left running. At this point:

 - If the `threadTerminationPolicy()` is `true`, always in shared server mode, then the daemon threads are killed.

- If the `threadTerminationPolicy()` is `false`, then the daemon threads are left quiescent until the next call. This is the default setting for shadow (dedicated) server mode.
- `OracleRuntime.EXIT_CALL_WHEN_ALL_THREADS_TERMINATE`
If set to this value, then the call ends only when all threads have either returned or ended due to an uncaught exception. At this point, the call ends regardless of the value of `threadTerminationPolicy()`.

Note: In Oracle database 9.x and earlier database releases, JVM behaves as if the `callExitPolicy()` were `OracleRuntime.EXIT_CALL_WHEN_ALL_NON_DAEMON_THREADS_TERMINATE` and the `threadTerminationPolicy()` were `true` for both shared and dedicated server processes. This means kill the daemon threads at this point. Also, if `exitCall()` were executed, then all threads are killed before the call is ended, in both shared and dedicated server processes.

Shared Servers Considerations

Note: Oracle recommends dedicated servers for performance reasons. But it is also possible to have a class of applications that rely on threads and sockets staying open across calls.

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

This section covers the following topics:

- [End-of-Call Migration](#)
- [Oracle-Specific Support for End-of-Call Optimization](#)
- [The `EndOfCallRegistry.registerCallback\(\)` Method](#)
- [The `EndOfCallRegistry.runCallbacks\(\)` Method](#)
- [The Callback Interface](#)
- [The `Callback.act\(\)` method](#)
- [Operating System Resources Affected Across Calls](#)

End-of-Call Migration

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

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

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

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

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

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

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

Oracle-Specific Support for End-of-Call Optimization

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

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

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

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

Static fields

Use `EndOfCallRegistry` to clear state associated with an entire class. In this case, the `Callback` object should be held in a `private static` field. Any code that requires access to the cached data that was dropped between calls must call a method that lazily creates, or re-creates, the cached data.

Consider the following example:

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

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

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

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

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

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

The preceding example does the following:

1. Creates a `Callback` object within a static field, `thunk`.
2. Registers this `Callback` object for end-of-call migration.
3. Implements the `Callback.act()` method to free up all static variables, including the `Callback` object itself.
4. Provides a method, `createCachedField()`, for lazily re-creating the cache.

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

Instance fields

Use `EndOfCallRegistry` to clear state in data structures held in instance fields. For example, when a state is associated with each instance of a class, each instance has a field that holds the cached state for the instance and fills in the cached field as necessary. You can access the cached field with a method that ensures the state is cached.

Consider the following example:

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

class Example2 implements Callback
{
    private Object cachedField = null;

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

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

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

```
}

```

The preceding example does the following:

1. Implements the instance as a `Callback` object.
2. Implements the `Callback.act()` method to free up the instance fields.
3. When you request a cache, the `Callback` object registers itself for the end-of-call migration.
4. Provides a method, `createCachedField()`, for lazily re-creating the cache.

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

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

The `EndOfCallRegistry.registerCallback()` Method

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

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

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

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

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

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

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

The `EndOfCallRegistry.runCallbacks()` Method

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

```
static void runCallbacks()
```

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

Note: Do not call this method in your code.

The Callback Interface

The interface is declared as follows:

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

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

The `Callback.act()` method

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

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

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

Operating System Resources Affected Across Calls

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

Resource	Lifetime
Files	The system closes all files left open when a database call ends.
Threads	All threads are terminated when a call ends.
Sockets	<ul style="list-style-type: none">Client sockets can exist across calls.Server sockets terminate when the call ends.
Objects that depend on operating system resources	Regardless of the usable lifetime of the object, the Java object can be valid for the duration of the session. This can occur, for example, if the Java object is stored in a <code>static</code> class variable, or a class variable references it directly or indirectly. If you attempt to use one of these Java objects after its usable lifetime is over, then Oracle Database will throw an exception. This is true for the following examples: <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, then a <code>java.io.IOException</code> is raised.<code>java.lang.Thread.isAlive()</code> is <code>false</code> for any <code>Thread</code> object running in a previous call and still accessible in a subsequent call.

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

Files

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

In [Example 2-5](#), the `Concat` class enables multiple files to be written into a single file, `outFile`. On the first call, `outFile` is created. The first input file is opened, read, written to `outFile`, and the call ends. Because `outFile` is defined as a `static` variable, it is moved into session space between call invocations. However, the file handle is closed at the end of the call. The next time you call `addFile()`, you will get an exception.

Example 2-5 Compromising Your Operating System Resources

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

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

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

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

[Example 2-6](#) shows how you can perform concatenation, as in [Example 2-5](#), without compromising your operating system resources. The `addFile()` method opens the `outme.txt` file within each call, ensuring that anything written into the file is appended to the end. At the end of each call, the file is closed. Two things occur:

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

Example 2-6 Correctly Managing Your Operating System Resources

```
public class Concat
{

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

```

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

/*close the output file between calls*/
out.close();
}
}

```

Sockets

Sockets are used in setting up a connection between a client and a server. For each database connection, sockets are used at either end of the connection. Your application does not set up the connection. The connection is set up by the underlying networking protocol, TTC or IIOP of Oracle Net.

See Also: ["Configuring Oracle JVM"](#) on page 4-2 for information about how to configure your connection.

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

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

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

The following table lists the socket types and their description:

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

Threads

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

As a specific example, the RMI Server, which Sun Microsystems supplies, functions in the shared server mode. However, it is useful only within the context of a single call. This is because the RMI Server forks daemon threads, which are in the shared server mode, are killed at the end of call, that is, the daemon thread are killed when all non-daemon threads return. If the RMI server session is reentered in a subsequent call,

then these daemon threads are not restarted and the RMI server fails to function properly. If RMI daemon threads are started to run JMX in a shared server session, then these daemons will be re-created automatically, together with other JMX-related daemon threads. The daemon threads will be re-created at the beginning of every Java call following the call starting the JMX agent and terminated at the end of the Java call.

Calling Java Methods in Oracle Database

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

- [Invoking Java Methods](#)
- [Debugging Server Applications](#)
- [How To Tell You Are Running on the Server](#)
- [Redirecting Output on the Server](#)
- [Calling Java in the Database Directly](#)

Invoking Java Methods

The type of the Java application determines how the client calls a Java method. The following sections discuss each of the Java application programming interfaces (APIs) available for calling a Java method:

- [Using PL/SQL Wrappers](#)
- [JNI Support](#)
- [Utilizing SQLJ and JDBC with Java in the Database](#)
- [Using Command-Line Interface](#)
- [Using the Client-Side Stub](#)

Using PL/SQL Wrappers

You can run Java stored procedures in the same way as PL/SQL stored procedures. In Oracle Database, Java is usually invoked through PL/SQL interface.

To call a Java stored procedure, you must publish it through a call specification. The following example shows how to create, resolve, load, and publish a simple Java stored procedure that returns a string:

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

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

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

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

```
javac Hello.java
```

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

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

- As individual files in one or more directories, where the directories are specified in the `CLASSPATH`
 - Within `.jar` or `.zip` files, where the directories containing these files are specified in the `CLASSPATH`
3. Decide on the resolver for the `Hello` class.

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

See Also: ["Resolving Class Dependencies"](#) on page 2-9 and [Chapter 11, "Schema Objects and Oracle JVM Utilities"](#)

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

```
loadjava -user scott Hello.class
Password: password
```

5. Publish the stored procedure through a call specification.

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

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

```
SQL> CONNECT scott
Enter password: password
connected
SQL> CREATE OR REPLACE FUNCTION helloworld RETURN VARCHAR2 AS
  2  LANGUAGE JAVA NAME 'Hello.world () return java.lang.String';
  3  /
Function created.
```

6. Call the stored procedure, as follows:

```
SQL> VARIABLE myString VARCHAR2(20);
```

```
SQL> CALL helloworld() INTO :myString;
Call completed.
SQL> PRINT myString;
```

```
MYSTRING
```

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

```
SQL>
```

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

See Also: [Chapter 5, "Developing Java Stored Procedures"](#)

JNI Support

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

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

Utilizing SQLJ and JDBC with Java in the Database

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

API	Description
JDBC	Use this API for more complex or dynamic SQL queries. JDBC requires you to establish the session, construct the query, and so on.
SQLJ	Use this API for easy SQL queries, both static and dynamic. SQLJ typically runs against a known table with known column names.

This section covers the following topics:

- [JDBC](#)
- [SQLJ](#)
- [Example Comparing JDBC and SQLJ](#)
- [Complete SQLJ Example](#)
- [SQLJ Strong Typing Paradigm](#)
- [Translating a SQLJ Program](#)

- [Running a SQLJ Program in the Server](#)
- [Converting a Client Application to Run on the Server](#)
- [Interacting with PL/SQL](#)

JDBC

JDBC is an industry-standard API developed by Sun Microsystems that lets you embed SQL statements as Java method arguments. JDBC is based on the X/Open SQL Call Level Interface (CLI) and complies with the Entry Level of SQL-92 standard. Each vendor, such as Oracle, creates its JDBC implementation by implementing the interfaces of the standard `java.sql` package. Oracle provides the following JDBC drivers that implement these standard interfaces:

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

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

1. Obtaining a connection handle
2. Creating a statement object of some type for your desired SQL operation
3. Assigning any local variables that you want to bind to the SQL operation
4. Carrying out the operation
5. Optionally retrieving the result sets

This process is sufficient for many applications, but becomes cumbersome for any complicated statements. Dynamic SQL operations, where the operations are not known until run time, require JDBC. However, in typical applications, this represents a minority of the SQL operations.

SQLJ

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

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

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

Example Comparing JDBC and SQLJ

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

JDBC:

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

// Set up JDBC prepared statement.
PreparedStatement pstmt = conn.prepareStatement
("SELECT ename FROM emp WHERE empno=? AND sal>?");
pstmt.setInt(1, id);
pstmt.setFloat(2, salary);

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

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

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

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

You can use a prepared statement whenever values in the SQL statement must be dynamically set. You can use the same prepared statement repeatedly with different variable values. The question marks (?) in the prepared statement are placeholders for Java variables. In the preceding example, these variables are assigned values using the `pstmt.setInt()` and `pstmt.setFloat()` methods. The first ? refers to the `int` variable `id` and is set to a value of 37115. The second ? refers to the `float` variable `salary` and is set to a value of 20000.

3. Run the query and return the data into a `ResultSet` object.
4. Retrieve the data of interest from the `ResultSet` object and display it. In this case, the `ename` column. A result set usually contains multiple rows of data, although this example has only one row.

SQLJ:

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

In addition to allowing SQL statements to be directly embedded in Java code, SQLJ supports Java host expressions, also known as bind expressions, to be used directly in

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

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

Complete SQLJ Example

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

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

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

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

In the preceding example, you do the following:

1. Declare your iterators.

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

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

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

2. Connect to the database.

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

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

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

- b. Opens a database connection for the specified schema, in this case, user `scott` with password `tiger`, at the specified URL. In this case, the URL points to host `oow11`, port `5521`, and SID `so12`.
 - c. Establishes this connection as the default connection for the SQLJ statements. Although each JDBC statement must explicitly specify a connection object, a SQLJ statement can either implicitly use a default connection or optionally specify a different connection.
3. Process a SQL statement. The following is accomplished:

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

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

- b. Instantiate and populate the iterator:

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

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

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

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

SQLJ Strong Typing Paradigm

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

Translating a SQLJ Program

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

```
%sqlj MyExample.sqlj
```

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

and specify the user name, password, and URL of your schema, as in the following example:

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

Running a SQLJ Program in the Server

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

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

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

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

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

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

Converting a Client Application to Run on the Server

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

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

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

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

Interacting with PL/SQL

All Oracle JDBC drivers communicate seamlessly with Oracle SQL and PL/SQL, and it is important to note that SQLJ interoperates with PL/SQL. You can start using SQLJ without having to rewrite any PL/SQL stored procedures. Oracle SQLJ includes

syntax for calling PL/SQL stored procedures and also lets you embed anonymous PL/SQL blocks in SQLJ statements.

Using Command-Line Interface

The command-line interface to Oracle JVM is analogous to using the JDK or JRE shell commands. You can:

- Use the standard `-classpath` syntax to indicate where to find the classes to load
- Set the system properties by using the standard `-D` syntax

The interface is a PL/SQL function that takes a string (`VARCHAR2`) argument, parses it as a command-line input and if it is properly formed, runs the indicated Java method in Oracle JVM. To do this, PL/SQL package `DBMS_JAVA` provides the following functions:

- [runjava](#)
- [runjava_in_current_session](#)

runjava

This function takes the Java command line as its only argument and runs it in Oracle JVM. The return value is null on successful completion, otherwise an error message. The format of the command line is the same as that taken by the JDK shell command, that is:

```
[option switches] name_of_class_to_execute [arg1 arg2 ... argn]
```

You can use the option switches `-classpath`, `-D`, `-Xbootclasspath`, and `-jar`. This function differs from the `runjava_in_current_session` function in that it clears any Java state remaining from previous use of Java in the session, prior to running the current command. This is necessary, in particular, to guarantee that static variable values derived at class initialization time from `-classpath` and `-D` arguments reflect the values of those switches in the current command line.

```
FUNCTION runjava(cmdline VARCHAR2) RETURN VARCHAR2;
```

runjava_in_current_session

This function is the same as the `runjava` function, except that it does not clear Java state remaining from previous use of Java in the session, prior to executing the current command line.

```
FUNCTION runjava_in_current_session(cmdline VARCHAR2) RETURN VARCHAR2;
```

Syntax

The syntax of the command line is of the following form:

```
[-options] classname [arguments...]  
[-options] -jar jarfile [arguments...]
```

Options

```
-classpath  
-D  
-Xbootclasspath  
-Xbootclasspath/a  
-Xbootclasspath/p
```

-cp

Note: The effect of the first form is to run the main method of the class identified by `classname` with the arguments. The effect of the second form is to run the main method of the class identified by the `Main-Class` attribute in the manifest of the JAR file identified by JAR. This is analogous to how the JDK/JRE interprets this syntax.

Argument Summary

Table 3–1 summarizes the command-line arguments.

Table 3–1 *Command Line Argument Summary*

Argument	Description
classpath	Accepts a colon (:) separated list of directories, JAR archives, and ZIP archives to search for class files. In general, the value of <code>-classpath</code> or similar arguments refer to file system locations as they would in a standard Java runtime. You also have an extension to this syntax to allow for terms that refer to database resident Java objects and sets of bytes. Refer to Appendix B, "Classpath Extensions and User Classloaded Metadata" for more information about <code>-classpath</code> extensions.
D	Establishes values for system properties that you use only when the Java session is first initialized. The default behavior of the command line interface is to terminate any existing Java session prior to running the new command, so that session initialization occurs each time the command line interface is used.
Xbootclasspath	Accepts a colon (:) separated list of directories, JAR archives, and ZIP archives. This option is used to set search path for bootstrap classes and resources.
Xbootclasspath/a	Accepts a colon (:) separated list of directories, JAR archives, and ZIP archives. This is appended to the end of bootstrap class path.
Xbootclasspath/p	Accepts a colon (:) separated list of directories, JAR archives, and ZIP archives. This is added in front of bootstrap class path.
cp	Acts as a synonym of <code>-classpath</code> .

Note: System classes created by `create java system` are always used before using any file or folder that are found using the `-Xbootclasspath` option.

Using the Client-Side Stub

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

In previous releases, calling Java stored procedures and functions from a database client required Java Database Connectivity (JDBC) calls to the associated PL/SQL wrappers. Each wrapper had to be manually published with a SQL signature and a Java implementation. This had the following disadvantages:

- The signatures permitted only Java types that had direct SQL equivalents
- Exceptions issued in Java were not properly returned

The `JPublisher -java` option provides functionality to overcome these disadvantages. To remedy the deficiencies of JDBC calls to associated PL/SQL wrappers, the `-java` option uses an API for direct invocation of `static` Java methods. This functionality is also useful for Web services.

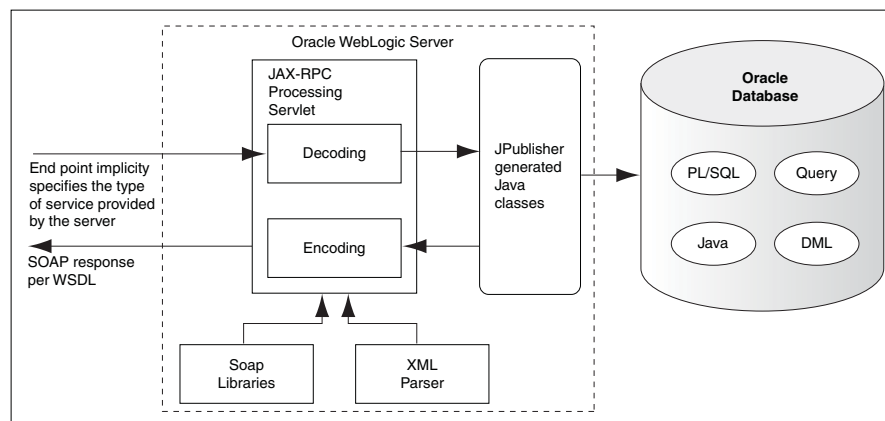
The functionality of the `-java` option mirrors that of the `-sql` option, creating a client-side Java stub class to access a server-side Java class, as opposed to creating a client-side Java class to access a server-side SQL object or PL/SQL package. The client-side stub class uses `JPublisher` code that mirrors the server-side class and includes the following features:

- Methods corresponding to the `public static` methods of the server class
- Two constructors, one that takes a JDBC connection and one that takes the `JPublisher` default connection context instance

At run time, the stub class is instantiated with a JDBC connection. Calls to the methods of the stub class result in calls to the corresponding methods of the server-side class. Any Java types used in these published methods must be primitive or serializable.

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

Figure 3–1 Client-Side Stub Interface



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

```
-java=className
```

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

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

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

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

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

```
% jpub -sql=scott -java=oracle.sqlj.checker.JdbcVersion:JdbcVersion Client
Enter scott password: password
```

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

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

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

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

Using the Default Service Feature

Starting from Oracle Database 11g release 1 (11.1), Oracle Database client provides a new default connection feature. If you install Oracle Database client, then you need not specify all the details of the database server in the connection URL. Under certain conditions, Oracle Database connection adapter requires only the host name of the computer where the database is installed.

For example, in the JDBC connection URL syntax, that is:

```
jdbc:oracle:driver_type:[username/password]@[//]host_name[:port][:ORCL]
```

,the following have become optional:

- `//` is optional.
- `:port` is optional.
You must specify a port only if the default Oracle Net listener port (1521) is not used.
- `:ORCL` or the service name is optional.
The connection adapter for Oracle Database Client connects to the default service on the host. On the host, this is set to `ORCL` in the `listener.ora` file.

Testing the Default Service with a Basic Configuration

The following code snippet shows a basic configuration of the `listener.ora` file, where the default service is defined.

Example 3–1 Basic Configuration of `listener.ora` with the Default Service Defined

```
MYLISTENER = (ADDRESS_LIST=(ADDRESS=(PROTOCOL=tcp) (HOST=testserver1) (PORT=1521)))
DEFAULT_SERVICE_MYLISTENER=dbjf.app.myserver.com
SID_LIST_MYLISTENER = (SID_LIST=(SID_DESC=(SID_NAME=dbjf)
(GLOBAL_DBNAME=dbjf.app.myserver.com) (ORACLE_HOME=/test/oracle))
```

)

After defining the `listener.ora` file, restart the listener with the following command:

```
lsnrctl start mylistener
```

Now, any of the following URLs should work with this configuration of the `listener.ora` file:

- `jdbc:oracle:thin:@//testserver1.myserver.com.com`
- `jdbc:oracle:thin:@//testserver1.myserver.com:1521`
- `jdbc:oracle:thin:@testserver1.myserver.com`
- `jdbc:oracle:thin:@testserver1.myserver.com:1521`
- `jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(HOST=testserver1.myserver.com)(PORT=1521)))`
- `jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(HOST=testserver1.myserver.com)))`
- `jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(HOST=testserver1.myserver.com)(PORT=1521))(CONNECT_DATA=(SERVICE_NAME=)))`

Note: Default service is a feature since Oracle Database 11g Release 1 (11.1). If you use any version prior to Oracle Database 11g Client to connect to the database, then you must specify the `SID`.

Debugging Server Applications

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

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

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

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

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

How To Tell You Are Running on the Server

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

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

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

The `getProperty()` method returns the following:

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

Redirecting Output on the Server

You can pass Java output to SQL statements to provide more extensive control over the destination of output from Oracle JVM. The PL/SQL package `DBMS_JAVA` has been enhanced by adding the following new functions, which provide extended functionality to what was previously available only with the `DBMS_JAVA.SET_OUTPUT` procedure:

- [set_output_to_sql](#)
- [remove_output_to_sql](#)
- [enable_output_to_sql](#)
- [disable_output_to_sql](#)
- [query_output_to_sql](#)

set_output_to_sql

`set_output_to_sql` defines a named output specification that constitutes an instruction for executing a SQL statement, whenever output to the default `System.out` and `System.err` streams occurs. The specification is defined either for the duration of the current session, or till the `remove_output_to_sql` function is called with its ID. The SQL actions prescribed by the specification occur whenever there is Java output. This can be stopped and started by calling the `disable_output_to_sql` and `enable_output_to_sql` functions respectively. The return value of this function is `null` on success, otherwise an error message.

```
FUNCTION set_output_to_sql (id VARCHAR2,
stmt VARCHAR2,
bindings VARCHAR2,
no_newline_stmt VARCHAR2 default null,
no_newline_bindings VARCHAR2 default null,
newline_only_stmt VARCHAR2 default null,
newline_only_bindings VARCHAR2 default null,
maximum_line_segment_length NUMBER default 0,
allow_replace NUMBER default 1,
from_stdout NUMBER default 1,
from_stderr NUMBER default 1,
include_newlines NUMBER default 0,
eager NUMBER default 0) return VARCHAR2;
```

Table 3–2 describes the arguments the `set_output_to_sql` function takes.

Table 3–2 *set_output_to_sql* Argument Summary

Argument	Description
id	The name of the specification. Multiple specifications may exist in the same session, but each must have a distinct ID. The ID is used to identify the specification in the functions <code>remove_output_to_sql</code> , <code>enable_output_to_sql</code> , <code>disable_output_to_sql</code> , and <code>query_output_to_sql</code> .
stmt	The default SQL statement to execute when Java output occurs.
bindings	A string containing tokens from the set <i>ID</i> , <i>TEXT</i> , <i>LENGTH</i> , <i>LINENO</i> , <i>SEGNO</i> , <i>NL</i> , and <i>ERRROUT</i> . This string defines how the SQL statement <code>stmt</code> is bound. The position of a token in the bindings string corresponds to the bind position in the SQL statement. The meanings of the tokens are: <ul style="list-style-type: none"> ■ <i>ID</i> is the ID of the specification. It is bound as a <code>VARCHAR2</code>. ■ <i>TEXT</i> is the text being output. It is bound as a <code>VARCHAR2</code>. ■ <i>LENGTH</i> is the length of the text. It is bound as a <code>NUMBER</code>. ■ <i>LINENO</i> is the line number since the beginning of session output. It is bound as a <code>NUMBER</code>. ■ <i>SEGNO</i> is the segment number within a line that is being output in more than one piece. It is bound as a <code>NUMBER</code>. ■ <i>NL</i> is a boolean indicating whether the text is to be regarded as newline terminated. It is bound as a <code>NUMBER</code>. The newline may or may not actually be included in the text, depending on the value of the <code>include_newlines</code> argument. ■ <i>ERRROUT</i> is a boolean indicating whether the output came from <code>System.out</code> or <code>System.err</code>. It is bound as a <code>NUMBER</code>. The value is 0, if the output came from <code>System.out</code>.
no_newline_stmt	An optional alternate SQL statement to execute, when the output is not newline terminated.
no_newline_bindings	A string with the same syntax as for the bindings argument discussed previously, describing how the <code>no_newline_stmt</code> is bound.
newline_only_stmt	An optional alternate SQL statement to execute when the output is a single newline.
newline_only_bindings	A string with the same syntax as for the bindings argument discussed previously, describing how the <code>newline_only_stmt</code> is bound.
maximum_line_segment_length	The maximum number of characters that is bound in a given execution of the SQL statement. Longer output sequences are broken up into separate calls with distinct <i>SEGNO</i> values. A value of 0 means no maximum.
allow_replace	Controls behavior when a previously defined specification with the same ID exists. A value of 1 means replacing the old specification. 0 means returning an error message without modifying the old specification.
from_stdout	Controls whether output from <code>System.out</code> causes execution of the SQL statement prescribed by the specification. A value of 0 means that if the output came from <code>System.out</code> , then the statement is not executed, even if the specification is otherwise enabled.
from_stderr	Controls whether output from <code>System.err</code> causes execution of the SQL statement prescribed by the specification. A value of 0 means that if the output came from <code>System.err</code> , then the statement is not executed, even if the specification is otherwise enabled.

Table 3–2 (Cont.) set_output_to_sql Argument Summary

Argument	Description
include_newlines	Controls whether newline characters are left in the output when they are bound to text. A value of 0 means new lines are not included. But the presence of the newline is still indicated by the NL binding and the use of no_newline_stmt.
eager	Controls whether output not terminated by a newline causes execution of the SQL statement every time it is received, or accumulates such output until a newline is received. A value of 0 means that unterminated output is accumulated.

remove_output_to_sql

remove_output_to_sql deletes a specification created by set_output_to_sql . If no such specification exists, an error message is returned.

```
FUNCTION remove_output_to_sql (id VARCHAR2) return VARCHAR2;
```

enable_output_to_sql

enable_output_to_sql reenables a specification created by set_output_to_sql and subsequently disabled by disable_output_to_sql . If no such specification exists, an error message is returned. If the specification is not currently disabled, there is no change.

```
FUNCTION enable_output_to_sql (id VARCHAR2) return VARCHAR2;
```

disable_output_to_sql

disable_output_to_sql disables a specification created by set_output_to_sql . You can enable the specification by calling enable_output_to_sql . While disabled, the SQL statement prescribed by the specification is not executed. If no such specification exists, an error message is returned. If the specification is already disabled, there is no change.

```
FUNCTION disable_output_to_sql (id VARCHAR2) return VARCHAR2;
```

query_output_to_sql

query_output_to_sql returns a message describing a specification created by set_output_to_sql . If no such specification exists, then an error message is returned. Passing null to this function causes all existing specifications to be displayed.

```
FUNCTION query_output_to_sql (id VARCHAR2) return VARCHAR2;
```

Another way of achieving control over the destination of output from Oracle JVM is to pass your Java output to an autonomous Java session. This provides a very general mechanism for propagating the output to various kinds of targets, such as disk files, sockets, and URLs. But, you must keep in mind that the Java session that processes the output is logically distinct from the main session, so that there are no other, unwanted interactions between them. To do this, PL/SQL package DBMS_JAVA provides the following functions:

- [set_output_to_java](#)
- [remove_output_to_java](#)

- [enable_output_to_java](#)
- [disable_output_to_java](#)
- [query_output_to_java](#)
- [set_output_to_file](#)
- [remove_output_to_file](#)
- [enable_output_to_file](#)
- [disable_output_to_file](#)
- [query_output_to_file](#)

set_output_to_java

`set_output_to_java` defines a named output specification that gives an instruction for executing a Java method whenever output to the default `System.out` and `System.err` streams occurs. The Java method prescribed by the specification is executed in a separate VM context with separate Java session state from the rest of the session.

```
FUNCTION set_output_to_java (id VARCHAR2,
class_name VARCHAR2,
class_schema VARCHAR2,
method VARCHAR2,
bindings VARCHAR2,
no_newline_method VARCHAR2 default null,
no_newline_bindings VARCHAR2 default null,
newline_only_method VARCHAR2 default null,
newline_only_bindings VARCHAR2 default null,
maximum_line_segment_length NUMBER default 0,
allow_replace NUMBER default 1,
from_stdout NUMBER default 1,
from_stderr NUMBER default 1,
include_newlines NUMBER default 0,
eager NUMBER default 0,
initialization_statement VARCHAR2 default null,
finalization_statement VARCHAR2 default null)return VARCHAR2;
```

Table 3–3 describes the arguments the `set_output_to_java` method takes.

Table 3–3 *set_output_to_java* Argument Summary

Argument	Description
<code>class_name</code>	The name of the class defining one or more methods.
<code>class_schema</code>	The schema in which the class is defined. A null value means the class is defined in the current schema, or PUBLIC.
<code>method</code>	The name of the method.
<code>bindings</code>	A string that defines how the arguments to the method are bound. This is a string of tokens with the same syntax as <code>set_output_to_sql</code> . The position of a token in the string determines the position of the argument it describes. All arguments must be of type INT, except for those corresponding to the tokens ID or TEXT, which must be of type <code>java.lang.String</code> .
<code>no_newline_method</code>	An optional alternate method to execute when the output is not newline terminated.
<code>newline_only_method</code>	An optional alternate method to execute when the output is a single newline.

Table 3–3 (Cont.) *set_output_to_java* Argument Summary

Argument	Description
<code>initialization_statement</code>	An optional SQL statement that is executed once per Java session prior to the first time the methods that receive output are executed. This statement is executed in same Java VM context as the output methods are executed. Typically such a statement is used to run a Java stored procedure that initializes conditions in the separate VM context so that the methods that receive output can function as intended. For example, such a procedure might open a stream that the output methods write to.
<code>finalization_statement</code>	An optional SQL statement that is executed once when the output specification is about to be removed or the session is ending. Like the <code>initialization_statement</code> , this runs in the same JVM context as the methods that receive output. It runs only if the initialization method has run, or if there is no initialization method.

remove_output_to_java

`remove_output_to_java` deletes a specification created by `set_output_to_java`. If no such specification exists, an error message is returned

```
FUNCTION remove_output_to_java (id VARCHAR2) return VARCHAR2;
```

enable_output_to_java

`enable_output_to_java` reenables a specification created by `set_output_to_java` and subsequently disabled by `disable_output_to_java`. If no such specification exists, an error message is returned. If the specification is not currently disabled, there is no change.

```
FUNCTION enable_output_to_java (id VARCHAR2) return VARCHAR2;
```

disable_output_to_java

`disable_output_to_java` disables a specification created by `set_output_to_java`. The specification may be reenabled by `enable_output_to_java`. While disabled, the SQL statement prescribed by the specification is not executed. If no such specification exists, an error message is returned. If the specification is already disabled, there is no change.

```
FUNCTION disable_output_to_java (id VARCHAR2) return VARCHAR2;
```

query_output_to_java

`query_output_to_java` returns a message describing a specification created by `set_output_to_java`. If no such specification exists, an error message is returned. Passing `null` to this function causes all existing specifications to be displayed.

```
FUNCTION query_output_to_java (id VARCHAR2) return VARCHAR2;
```

set_output_to_file

`set_output_to_file` defines a named output specification that constitutes an instruction to capture any output sent to the default `System.out` and `System.err` streams and append it to a specified file. This is implemented using a special case of `set_output_to_java`. The argument `file_path` specifies the path to the file to which to append the output. The arguments `allow_replace`, `from_stdout`, and

`from_stderr` are all analogous to the arguments having the same name as in `set_output_to_sql`.

```
FUNCTION set_output_to_file (id VARCHAR2,
file_path VARCHAR2,
allow_replace NUMBER default 1,
from_stdout NUMBER default 1,
from_stderr NUMBER default 1) return VARCHAR2;
```

remove_output_to_file

This function is analogous to `remove_output_to_java`.

```
FUNCTION remove_output_to_file (id VARCHAR2) return VARCHAR2;
```

enable_output_to_file

This function is analogous to `enable_output_to_java`.

```
FUNCTION enable_output_to_file (id VARCHAR2) return VARCHAR2;
```

disable_output_to_file

This function is analogous to `disable_output_to_java`.

```
FUNCTION disable_output_to_file (id VARCHAR2) return VARCHAR2;
```

query_output_to_file

This function is analogous to `query_output_to_java`.

```
FUNCTION query_output_to_file (id VARCHAR2) return VARCHAR2;
```

The following DBMS_JAVA functions control whether Java output appears in the `.trc` file:

- PROCEDURE `enable_output_to_trc`;
- PROCEDURE `disable_output_to_trc`;
- FUNCTION `query_output_to_trc` return VARCHAR2;

Note: Prior to 11g release 1 (11.1), the fact that Java output appeared in the `.trc` file was not modifiable.

Redirecting the output to SQL*Plus Text Buffer

As in previous releases, you can use the DBMS_JAVA package procedure `SET_OUTPUT` to redirect output to the SQL*Plus text buffer:

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

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

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

The output is displayed at the end of the call.

Calling Java in the Database Directly

In releases prior to Oracle Database 10g, calling Java stored procedures and functions from a database client required JDBC calls to the associated PL/SQL wrappers. Each wrapper had to be manually published with a SQL signature and a Java implementation. This had the following disadvantages:

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

To remedy these deficiencies, a simple API has been implemented since Oracle Database 11g to directly call `static` Java stored procedures and functions. This new functionality is useful for general applications, but is particularly useful for Web services.

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

The following is the Java interface for the API:

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

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

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

You can call the method, as follows:

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

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

Note the following:

- Any serializable type, such as `int[]` and `String[]`, can be passed as an argument.
- As an optimization, parameter values can be represented as `String`:

```
String sig = oracle.jpub.reflect.Client.getSignature(new Class[] {});
...
Client.invoke(conn, "oracle.sqlj.checker.JdbcVersion", "to_string", sig, new
Object[] {});
```
- The semantics of this API are different from the semantics for calling stored procedures or functions through a PL/SQL wrapper, in the following ways:
 - Arguments cannot be `OUT` or `IN OUT`. Returned values must all be part of the function result.

- Exceptions are properly returned.
- The method invocation uses invoker's rights. There is no tuning to obtain the definer's rights.

Java Installation and Configuration

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

- [Initializing a Java-Enabled Database](#)
- [Configuring Oracle JVM](#)
- [Using The DBMS_JAVA Package](#)
- [Enabling the Java Client](#)
- [Two-Tier Duration for Java Session State](#)
- [Setting System Properties](#)

Initializing a Java-Enabled Database

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

This section contains the following topics:

- [Configuring with Oracle Database Template](#)
- [Modifying an Existing Oracle Database to Include Oracle JVM](#)

Configuring with Oracle Database Template

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

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

Modifying an Existing Oracle Database to Include Oracle JVM

If you have already installed Oracle Database without Oracle JVM, then you can add Java to your database through the modify mode of the Database Configuration Assistant of Oracle Database 11g. The modify mode enables you to choose the features, such as Oracle JVM, that you would like to install on top of an existing Oracle Database instance.

Configuring Oracle JVM

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

- Java memory requirements

You must have at least 50 MB of `JAVA_POOL_SIZE` and 96 MB of `SHARED_POOL_SIZE`.

See Also: ["Java Memory Usage"](#) on page 9-4

- Database processes

You must decide whether to use dedicated server processes or shared server processes for your database server.

Using The DBMS_JAVA Package

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

See Also: [Appendix A, "DBMS_JAVA Package"](#)

Enabling the Java Client

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

1. [Install J2SE on the Client](#)
2. [Set Up Environment Variables](#)
3. [Test Install with Samples](#)

Install J2SE on the Client

The client requires Java Development Kit (JDK) 1.5.2 or later. To confirm the version of JDK you are using, run the following commands on the command line:

```
$ which java
/usr/local/j2se1.5.2/bin/java

$ which javac
/usr/local/j2se1.5.2/bin/javac

$ java -version
java version "1.5.2"
```

Set Up Environment Variables

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

- `$JAVA_HOME`

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

- `$PATH`
This variable must include `$JAVA_HOME/bin`.
- `$LD_LIBRARY_PATH`
This variable must include `$JAVA_HOME/lib`.

JAR Files Necessary for Java2 Clients

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

Note: Specifics of `CLASSPATH` requirements may vary for Oracle JVMs running on different platforms. You must ensure that all elements of `CLASSPATH`, as defined in the script for Oracle JVM utilities, are present.

- For JDK 1.5.2, include `$JAVA_HOME/lib/dt.jar`
- For JRE 1.5.2, include `$JAVA_HOME/lib/rt.jar`
- For any interaction with JDBC, include `$ORACLE_HOME/jdbc/lib/ojdbc5.jar`
- For any client that uses SSL, include `$ORACLE_HOME/jlib/jssl-1_2.jar` and `$ORACLE_HOME/jlib/javax-ssl-1_2.jar`
- For any client that uses the Java Transaction API (JTA) functionality, include `$ORACLE_HOME/jlib/jta.jar`
- For any client that uses the Java Naming and Directory Interface (JNDI) functionality, include `$ORACLE_HOME/jlib/jndi.jar`
- If you are using the accelerator for native compilation, include `$JAVA_HOME/lib/tools.jar`

JAR Files Included for Clients that Use SQLJ

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

In addition to this file, add the appropriate `runtimeX.zip` file, as follows:

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

Server Application Development on the Client

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

Test Install with Samples

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

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

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

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

Two-Tier Duration for Java Session State

Prior to 11g release 1 (11.1), Java session state was single-tier, which included all values associated with running Java, such as System property values and static variable values, the set of classes loaded during the session and so on. The duration of this state used to start with the first invocation of a Java method in the RDBMS session and it used to last till JVM exited, either due to a call to `java.lang.System.exit` or similar `OracleRuntime` methods, an uncaught exception, a fatal error, or the end of the RDBMS session. This required modifying Java code and also failed to fully complete the termination of the session before the end of the RDBMS call. This made it impossible to start a new Java session within the same call.

Starting with 11g release 1 (11.1), Java session state is split into two tiers. One tier has a longer duration and it encompasses the duration of the other tier. The duration of the shorter tier is the same as before, that is, it starts when a Java method is invoked and ends when JVM exits. The duration of the longer tier starts when a Java method is invoked in the RDBMS session for the first time. This session lasts until the RDBMS session ends or the session is explicitly terminated by a call to the function `dbms_java.endsession_and_related_state`. This is addressed by the addition of the following two PL/SQL functions to the `DBMS_JAVA` package, which account for the two kinds of Java session duration:

- `FUNCTION endsession RETURN VARCHAR2;`

This function clears any Java session state remaining from previous execution of Java in the current RDBMS session. The return value is a message indicating the action taken.

- `FUNCTION endsession_and_related_state RETURN VARCHAR2;`

This function clears any Java session state remaining from previous execution of Java in the current RDBMS session and all supporting data related to running Java, such as property settings and output specifications. The return value is a message indicating the action taken.

Most of the values associated with running Java remain in the shorter tier. The values that can be useful for multiple invocations of JVM have been moved to the longer tier. For example, the system property values established by `dbms_java.set_property` and the output redirection specifications.

See Also: ["Setting System Properties"](#) on page 4-5 and ["Redirecting Output on the Server"](#) on page 3-14.

Setting System Properties

Within an RDBMS session you can maintain a set of values that are added to the system properties whenever a Java session is started in the RDBMS session. This set of values remains valid for the duration of the longer tier of Java session state, which is typically the same as the duration the RDBMS session.

See Also: ["Two-Tier Duration for Java Session State"](#) on page 4-4

There is a set of PL/SQL functions in the `DBMS_JAVA` package for setting, retrieving, removing and displaying key value pairs in an internal, RDBMS session duration table, where both elements of a pair are strings (`VARCHAR2`) and there is at most one pair for a given key. These functions are as follows:

- [set_property](#)
- [get_property](#)
- [remove_property](#)
- [show_property](#)

set_property

This function establishes a value for a system property that is then used for the duration of the current RDBMS session, whenever a Java session is initialized. The first argument is the name of the property and the second is the value to be established for it. The return value for `set_property` is null unless there is some error. For example, if an attempt is made to set a value for a prescribed property, then an error message is returned.

```
FUNCTION set_property(name VARCHAR2, value VARCHAR2) RETURN VARCHAR2;
```

get_property

This function returns any value previously established by `set_property`. It returns null if there is no such value.

```
FUNCTION get_property(name VARCHAR2) RETURN VARCHAR2;
```

remove_property

This function removes any value previously established by `set_property`. The return value is null unless an error occurred, in which case an error message is returned.

```
FUNCTION remove_property(name VARCHAR2) RETURN VARCHAR2;
```

show_property

This function displays a message of the form `name = value` for the input name, or for all established property bindings, if name is null. The return value for this function is null on successful completion, otherwise it is an error message. The output is displayed to wherever you have currently directed your Java output.

```
FUNCTION show_property(name VARCHAR2) RETURN VARCHAR2;
```

Before initializing the Java session, the values from this table are added to the set of default system property values already maintained by Oracle JVM. When you run a Java method by using the command-line interface, the values determined by the `-D`

option, if present, override the values set in the table. As soon as you terminate the Java session, the values established by the `-D` option become obsolete and the keys are set to the original values as present in the table.

Developing Java Stored Procedures

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

This chapter contains the following sections:

- [Stored Procedures and Run-Time Contexts](#)
- [Advantages of Stored Procedures](#)
- [Java Stored Procedures Steps](#)

Stored Procedures and Run-Time Contexts

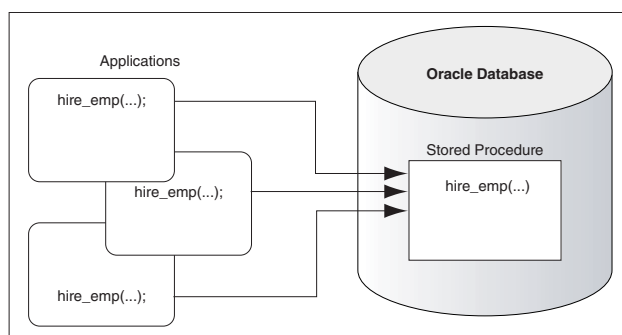
Stored procedures are Java methods published to SQL and stored in the database for general use. To publish Java methods, you write call specifications, which map Java method names, parameter types, and return types to their SQL counterparts.

Unlike a wrapper, which adds another layer of execution, a call specification publishes the existence of a Java method. As a result, when you call the method through its call specification, the run-time system dispatches the call with minimal overhead.

When called by client applications, a stored procedure can accept arguments, reference Java classes, and return Java result values.

[Figure 5-1](#) shows a stored procedure being called by various applications.

Figure 5-1 Calling a Stored Procedure



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

- [Functions and Procedures](#)

- [Database Triggers](#)
- [Object-Relational Methods](#)

Functions and Procedures

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

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

Only top-level and package-level PL/SQL functions and procedures can be used as call specifications. When you define them using the SQL `CREATE FUNCTION`, `CREATE PROCEDURE`, or `CREATE PACKAGE` statement, they are stored in the database, where they are available for general use.

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

- SQL data manipulation language (DML) statements
- SQL `CALL` statements
- PL/SQL blocks, subprograms, and packages

Database Triggers

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

A trigger has the following parts:

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

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

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

Object-Relational Methods

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

You can create an abstract template for some real-world object as a SQL object type. The template specifies only those attributes and methods that the object will need in the application environment. At run time, when you fill the data structure with values, you create an instance of the object type. You can create as many instances as required.

Typically, an object type corresponds to some business entity, such as a purchase order. To accommodate a variable number of items, object types can use a `VARRAY`, a nested table, or both.

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

Advantages of Stored Procedures

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

- [Performance](#)
- [Productivity and Ease of Use](#)
- [Scalability](#)
- [Maintainability](#)
- [Interoperability](#)
- [Replication](#)
- [Security](#)

Performance

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

By grouping SQL statements, a stored procedure allows the statements to be processed with a single call. This reduces network traffic and improves round-trip response time.

Additionally, stored procedures enable you to take advantage of the computing resources of the server. For example, you can move computation-bound procedures from client to server, where they will run faster. Stored functions enhance performance by running application logic within the server.

Productivity and Ease of Use

By designing applications around a common set of stored procedures, you can avoid redundant coding and increase the productivity. Moreover, stored procedures let you extend the functionality of the database.

You can use the Java integrated development environment (IDE) of your choice to create stored procedures. They can be called by standard Java interfaces, such as Java Database Connectivity (JDBC), and by programmatic interfaces and development tools, such as SQLJ, Oracle Call Interface (OCI), Pro*C/C++, and JDeveloper.

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

Scalability

Java in the database inherits the scalable session model of Oracle Database. Stored procedures increase scalability by isolating application processing on the server. In

addition, automatic dependency tracking for stored procedures helps in developing scalable applications.

Maintainability

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

Interoperability

Java in Oracle Database fully conforms to the Java Language Specification (JLS) and furnishes all the advantages of a general-purpose, object-oriented programming language. Also, as with PL/SQL, Java provides full access to Oracle data. As a result, any procedure that is written in PL/SQL can also be written in Java.

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

Oracle Database allows a high degree of interoperability between Java and PL/SQL. Java applications can call PL/SQL stored procedures using an embedded JDBC driver. Conversely, PL/SQL applications can call Java stored procedures directly.

Replication

With Oracle Advanced Replication, you can replicate stored procedures from one Oracle Database instance to another. This enables you to use stored procedures to implement a central set of business rules. Once you write the procedures, you can replicate and distribute them to work groups and branch offices throughout the company. In this way, you can revise policies on a central server rather than on individual servers.

Security

Security is a large arena that includes:

- Network security for the connection
- Access and execution control of operating system resources or of JVM and user-defined classes
- Bytecode verification of JAR files imported from an external source.

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

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

See Also: [Chapter 10, "Security for Oracle Database Java Applications"](#)

Java Stored Procedures Steps

You can run Java stored procedures in the same way as PL/SQL stored procedures. Usually, a call to a Java stored procedure is a result of database manipulation, because it is usually the result of a trigger or SQL DML call. To call a Java stored procedure, you must publish it through a call specification.

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

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

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

- [Step 1: Create or Reuse the Java Classes](#)
- [Step 2: Load and Resolve the Java Classes](#)
- [Step 3: Publish the Java Classes](#)
- [Step 4: Call the Stored Procedures](#)
- [Step 5: Debug the Stored Procedures, if Necessary](#)

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

Step 1: Create or Reuse the Java Classes

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

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

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

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

```
javac Oscar.java
```

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

Step 2: Load and Resolve the Java Classes

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

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

```
$ loadjava -user scott Oscar.class
Password: password
```

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

Step 3: Publish the Java Classes

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

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

```
SQL> connect scott
Enter password: password

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

See Also: [Chapter 6, "Publishing Java Classes With Call Specifications"](#)

Step 4: Call the Stored Procedures

You can call Java stored procedures from JDBC, SQLJ, and all third party languages that can access the call specification. Using the SQL `CALL` statement, you can also call the stored procedures from the top level, for example, from SQL*Plus. Stored procedures can also be called from database triggers.

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

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

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

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

```
SQL> PRINT theQuote;
```

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

See Also: [Chapter 7, "Calling Stored Procedures"](#)

You can also call the Java class using the `ojvmjava` tool.

See Also: ["The ojvmjava Tool"](#) on page 11-21

Step 5: Debug the Stored Procedures, if Necessary

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

See Also: ["Debugging Server Applications"](#) on page 3-13

Publishing Java Classes With Call Specifications

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

This chapter contains the following sections:

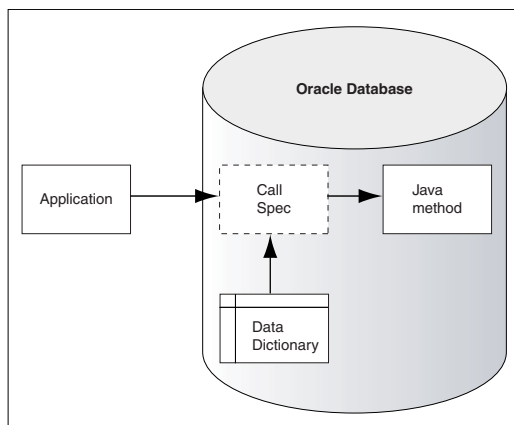
- [Understanding Call Specifications](#)
- [Defining Call Specifications](#)
- [Writing Top-Level Call Specifications](#)
- [Writing Packaged Call Specifications](#)
- [Writing Object Type Call Specifications](#)

Understanding Call Specifications

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

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

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

Figure 6–1 Calling a Java Method

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

See Also: ["Using the Client-Side Stub"](#) on page 3-10

Defining Call Specifications

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

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

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

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

This section covers the following topics:

- [Setting Parameter Modes](#)
- [Mapping Data Types](#)
- [Using the Server-Side Internal JDBC Driver](#)

Setting Parameter Modes

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

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

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

Mapping Data Types

In a call specification, the corresponding SQL and Java parameters and function results must have compatible data types.

Table 6–1 lists the legal data type mappings. Oracle Database converts between the SQL types and Java classes automatically.

Table 6–1 Legal Data Type Mappings

SQL Type	Java Class
CHAR, VARCHAR2, LONG	java.lang.String oracle.sql.CHAR
NUMBER	boolean char byte short int long float double java.lang.Byte java.lang.Short java.lang.Integer java.lang.Long java.lang.Float java.lang.Double java.math.BigDecimal oracle.sql.NUMBER
BINARY_INTEGER	boolean char byte short int long
BINARY_FLOAT	oracle.sql.BINARY_FLOAT
BINARY_DOUBLE	oracle.sql.BINARY_DOUBLE
DATE	oracle.sql.DATE

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

SQL Type	Java Class
RAW	oracle.sql.RAW
BLOB	oracle.sql.BLOB
CLOB	oracle.sql.CLOB
BFILE	oracle.sql.BFILE
ROWID	oracle.sql.ROWID
TIMESTAMP	oracle.sql.TIMESTAMP
TIMESTAMP WITH TIME ZONE	oracle.sql.TIMESTAMPTZ
TIMESTAMP WITH LOCAL TIME ZONE	oracle.sql.TIMESTAMPLTZ
ref cursor	java.sql.ResultSet sqlj.runtime.ResultSetIterator
user defined named types, ADTs	oracle.sql.STRUCT
opaque named types	oracle.sql.OPAQUE
nested tables and VARRAY named types	oracle.sql.ARRAY
references to named types	oracle.sql.REF

You also must consider the following:

- The last four SQL types are collectively referred to as named types.
- All SQL types except BLOB, CLOB, BFILE, REF CURSOR, and the named types can be mapped to the Java type `byte[]`, which is a Java byte array. In this case, the argument conversion means copying the raw binary representation of the SQL value to or from the Java byte array.
- Java classes that implement the `ORADData` interface and related methods, or Java classes that are subclasses of the `oracle.sql` classes appearing in the table, can be mapped from SQL types other than `BINARY_INTEGER` and `REF CURSOR`. This is generally used for mapping named types to corresponding Java classes generated by `JPublisher`.
- The `UROWID` type and the `NUMBER` subtypes, such as `INTEGER` and `REAL`, are not supported.
- A value larger than 32 KB cannot be retrieved from a `LONG` or `LONG RAW` column into a Java stored procedure.

Using the Server-Side Internal JDBC Driver

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

However, the server-side internal JDBC driver runs within a default session and a default transaction context. As a result, you are already connected to the database, and all your SQL operations are part of the default transaction. You need not register the driver because it comes preregistered. To get a `Connection` object, run the following line of code:

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

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

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

Use the `PreparedStatement` class for SQL statements that take `IN` parameters or are run more than once. The SQL statement, which can contain one or more parameter placeholders, is precompiled. A question mark (?) serves as a placeholder. When called on a `Connection` object, the `prepareStatement()` method returns a new `PreparedStatement` object, which contains the precompiled SQL statement. For example:

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

A `ResultSet` object contains SQL query results, that is, the rows that meet the search condition. You can use the `next()` method to move to the next row, which then becomes the current row. You can use the `getXXX()` methods to retrieve column values from the current row. For example:

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

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

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

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

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

Important Points

When developing JDBC applications that access stored procedures, you must consider the following:

- Each Oracle JVM session has a single implicit native connection to the Database session in which it exists. This connection is conceptual and is not a Java object. It is an inherent aspect of the session and cannot be opened or closed from within the JVM.
- The server-side internal JDBC driver runs within a default transaction context. You are already connected to the database, and all your SQL operations are part of the default transaction. Note that this transaction is a local transaction and not part of a global transaction, such as that implemented by Java Transaction API (JTA) or Java Transaction Service (JTS).
- Statements and result sets persist across calls and their finalizers do not release database cursors. To avoid running out of cursors, close all statements and result sets after you have finished using them. Alternatively, you can ask your DBA to raise the limit set by the initialization parameter, `OPEN_CURSORS`.
- The server-side internal JDBC driver does not support auto-commits. As a result, your application must explicitly commit or roll back database changes.
- You cannot connect to a remote database using the server-side internal JDBC driver. You can connect only to the server running your Java program. For server-to-server connections, use the server-side JDBC Thin driver. For client/server connections, use the client-side JDBC Thin or JDBC Oracle Call Interface (OCI) driver.
- Typically, you should not close the default connection instance because it is a single instance that can be stored in multiple places, and if you close the instance, each would become unusable. If it is closed, a later call to the `OracleDriver.defaultConnection` method gets a new, open instance. The `OracleDataSource.getConnection` method returns a new object every time you call it, but, it does not create a new database connection every time. They all utilize the same implicit native connection and share the same session state, in particular, the local transaction.

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

Writing Top-Level Call Specifications

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

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

where `param` is represented by the following syntax:

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

The `AUTHID` clause determines the following:

- Whether a stored procedure runs with the privileges of its definer (`AUTHID DEFINER`) or invoker (`AUTHID CURRENT_USER`)

- Whether its unqualified references to schema objects are resolved in the schema of the definer or invoker

If you do not specify the `AUTHID`, then the default behavior is `DEFINER`, that is, the stored procedure runs with the privileges of its definer. You can override the default behavior by specifying the `AUTHID` as `CURRENT_USER`. However, you cannot override the `loadjava` option `-definer` by specifying `CURRENT_USER`.

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

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

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

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

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

This section provides the following examples:

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

Example 6–1 Publishing a Simple JDBC Stored Procedure

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

```
import java.sql.*;  
import java.io.*;  
import oracle.jdbc.*;  
  
public class GenericDrop  
{  
    public static void dropIt(String object_type, String object_name)  
                                throws SQLException  
    {  
        // Connect to Oracle using JDBC driver  
        Connection conn = DriverManager.getConnection("jdbc:default:connection:");  
        // Build SQL statement  
        String sql = "DROP " + object_type + " " + object_name;  
        try  
        {
```

```

        Statement stmt = conn.createStatement();
        stmt.executeUpdate(sql);
        stmt.close();
    }
    catch (SQLException e)
    {
        System.err.println(e.getMessage());
    }
}
}

```

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

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

```

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

```

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

Example 6–2 Publishing the `main()` Method

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

```

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

```

To publish `main()`, write the following call specification:

```

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

```

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

```

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

```

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

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

```

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

```

```

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

```

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

```

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

```

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

Consider the swap() method in the following Swapper class, which switches the values of its arguments:

```

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

```

The call specification publishes the swap() method as a call specification, swap(). The call specification declares IN OUT formal parameters, because values must be passed in and out. All call specification OUT and IN OUT parameters must map to Java array parameters.

```

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

```

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

Writing Packaged Call Specifications

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

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

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

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

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

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

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

[Example 6-5](#) provides an example of packaged call specification.

Example 6-5 Packaged Call Specification

Consider a Java class, `DeptManager`, which consists of methods for adding a new department, dropping a department, and changing the location of a department. Note that the `addDept()` method uses a database sequence to get the next department number. The three methods are logically related, and therefore, you may want to group their call specifications in a PL/SQL package.

```
import java.sql.*;
```



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

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

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

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

```

        pstmt.close();
    }
    catch (SQLException e)
    {
        System.err.println(e.getMessage());
    }
}
}

```

Suppose you want to package the methods `addDept()`, `dropDept()`, and `changeLoc()`. First, you must create the package specification, as follows:

```

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

```

Then, you must create the package body by writing the call specifications for the Java methods, as follows:

```

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

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

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

```

To reference the stored procedures in the `dept_mgmt` package, use the dot notation, as follows:

```

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

```

Writing Object Type Call Specifications

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

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

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

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

```

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

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

```

The `AUTHID` clause determines whether all member methods of the type run with the privileges of their definer (`AUTHID DEFINER`), which is the default, or invoker (`AUTHID CURRENT_USER`). It also determines whether unqualified references to schema objects are resolved in the schema of the definer or invoker.

This section covers the following topics:

- [Declaring Attributes](#)
- [Declaring Methods](#)

Declaring Attributes

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

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

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

Declaring Methods

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

```
instance_expression.method()
```

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

```
object_type_name.method()
```

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

This section contains the following topics:

- [Map and Order Methods](#)
- [Constructor Methods](#)
- [Examples](#)

Map and Order Methods

The values of a SQL scalar data type, such as `CHAR`, have a predefined order and, therefore, can be compared with other values. However, instances of an object type have no predefined order. To put them in order, SQL calls a user-defined `map` method.

SQL uses the ordering to evaluate boolean expressions, such as `x > y`, and to make comparisons implied by the `DISTINCT`, `GROUP BY`, and `ORDER BY` clauses. A `map` method returns the relative position of an object in the ordering of all such objects. An object type can contain only one `map` method, which must be a function without any parameters and with one of the following return types: `DATE`, `NUMBER`, or `VARCHAR2`.

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

You can declare a `map` method or an `order` method, but not both. If you declare either of these methods, then you can compare objects in SQL and PL/SQL. However, if you do not declare both methods, then you can compare objects only in SQL and solely for equality or inequality.

Note: Two objects of the same type are equal if the values of their corresponding attributes are equal.

Constructor Methods

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

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

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

Examples

In this section, each example builds on the previous one. To begin, you create two SQL object types to represent departments and employees. First, you write the specification for the object type `Department`. The body is not required, because the specification declares only attributes. The specification is as follows:

```
CREATE TYPE Department AS OBJECT (
```

```
deptno NUMBER(2),
dname VARCHAR2(14),
loc VARCHAR2(13)
);
```

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

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

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

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

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

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

Selecting a `REF` returns a handle to an object. It does not materialize the object itself. To do that, you can use methods in the `oracle.sql.REF` class, which supports Oracle object references. This class, which is a subclass of `oracle.sql.Datum`, extends the standard JDBC interface, `oracle.jdbc2.Ref`.

Using Class `oracle.sql.STRUCT`

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

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

```

import oracle.jdbc.*;
import oracle.oracore.*;
import oracle.jdbc2.*;
import java.math.*;

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

```

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

```

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

```

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

Implementing the `SQLData` Interface

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

```

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

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

    public static BigDecimal wages(Paymaster e)

```

```

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

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

    // Implement SQLData interface.

    private String sql_type;

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

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

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

```

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

```

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

```

Because the new method, `raiseSal()`, is `void`, write a procedure call specification for it, as follows:

```

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

```

Again, this is a top-level call specification.

Implementing Object Type Methods

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

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

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

```
import java.sql.*;  
import java.io.*;  
import oracle.sql.*;  
import oracle.jdbc.*;  
import oracle.oracore.*;  
import oracle.jdbc2.*;  
import java.math.*;  
  
public class Paymaster implements SQLData  
{  
  // Implement the attributes and operations for this type.  
  private BigDecimal empno;  
  private String ename;  
  private String job;  
  private BigDecimal mgr;  
  private Date hiredate;  
  private BigDecimal sal;  
  private BigDecimal comm;  
  private Ref dept;  
  
  public BigDecimal wages()  
  {  
    BigDecimal pay = sal;  
    if (comm != null)  
      pay = pay.add(comm);  
    return pay;  
  }  
}
```



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

// Implement SQLData interface.

String sql_type;

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

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

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

Calling Stored Procedures

After you load and publish a Java stored procedure, you can call it. This chapter describes the procedure for calling Java stored procedures in various contexts. It also describes how Oracle JVM handles SQL exceptions.

This chapter contains the following sections:

- [Calling Java from the Top Level](#)
- [Calling Java from Database Triggers](#)
- [Calling Java from SQL DML](#)
- [Calling Java from PL/SQL](#)
- [Calling PL/SQL from Java](#)
- [How Oracle JVM Handles Exceptions](#)

Calling Java from the Top Level

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

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

where `param` is represented by the following syntax:

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

Host variables are variables that are declared in a host environment. They must be prefixed with a colon. The following examples show that a host variable cannot appear twice in the same `CALL` statement and that a subprogram without parameters must be called with an empty parameter list:

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

This section covers the following topics:

- [Redirecting Output](#)
- [Examples of Calling Java Stored Procedures From the Top Level](#)

Redirecting Output

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

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

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

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

The output is displayed when the stored procedure exits.

Examples of Calling Java Stored Procedures From the Top Level

This section provides the following examples

- [Example 7-1, "A Simple JDBC Stored Procedure"](#)
- [Example 7-2, "Fibonacci Sequence"](#)

Example 7-1 A Simple JDBC Stored Procedure

In the following example, the `main()` method accepts the name of a database table, such as `emp`, and an optional `WHERE` clause specifying a condition, such as `sal > 1500`. If you omit the condition, then the method deletes all rows from the table, else it deletes only those rows that meet the condition.

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

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

The `main()` method can take either one or two arguments. Usually, the `DEFAULT` clause is used to vary the number of arguments passed to a PL/SQL subprogram.

However, this clause is not allowed in a call specification. As a result, you must overload two packaged procedures, as follows:

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

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

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

Now, you can call the `delete_rows` procedure, as follows:

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

Call completed.

```
SQL> SELECT ename, sal FROM emp;
```

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

7 rows selected.

Note: You cannot overload top-level procedures.

Example 7-2 Fibonacci Sequence

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

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

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

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

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

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

PL/SQL procedure successfully completed.

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

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

Call completed.

```
SQL> PRINT f
```

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

Calling Java from Database Triggers

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

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

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

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

- [Example 7-3, "Calling Java Stored Procedure from Database Trigger - I"](#)
- [Example 7-4, "Calling Java Stored Procedure from Database Trigger - II"](#)

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

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

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

public class DBTrigger
{
    public static void logSal (int empID, float oldSal, float newSal)
                                throws SQLException
    {
        Connection conn = DriverManager.getConnection("jdbc:default:connection:");
        String sql = "INSERT INTO sal_audit VALUES (?, ?, ?)";
```

```

try
{
    PreparedStatement pstmt = conn.prepareStatement(sql);
    pstmt.setInt(1, empID);
    pstmt.setFloat(2, oldSal);
    pstmt.setFloat(3, newSal);
    pstmt.executeUpdate();
    pstmt.close();
}
catch (SQLException e)
{
    System.err.println(e.getMessage());
}
}
}

```

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

```

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

```

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

```

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

```

Finally, create the database trigger, which fires when a salary increase exceeds 20 percent:

```

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

```

When you run the following `UPDATE` statement, it updates all rows in the `emp` table:

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

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

```
SQL> SELECT * FROM sal_audit;
```

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

6 rows selected.

Example 7-4 Calling Java Stored Procedure from Database Trigger - II

Assume you want to create a trigger that inserts rows into a database view, which is defined as follows:

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

The `sales` and `mktg` database tables are defined as:

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

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

First, add the following Java method to the `DBTrigger` class, which is defined in [Example 7-3](#):

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

The `addEmp()` method inserts a row into the `sales` or `mktg` table depending on the value of the `deptName` parameter. Write the call specification for this method, as follows:

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

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

```
CREATE OR REPLACE TRIGGER emps_trig
INSTEAD OF INSERT ON emps
```



```
FOR EACH ROW
CALL add_emp(:new.empno, :new.ename, :new.dname);
```

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

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

```
SQL> SELECT * FROM sales;
```

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

```
SQL> SELECT * FROM mktg;
```

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

```
SQL> SELECT * FROM emps;
```

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

Calling Java from SQL DML

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

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

```
}

```

The `Formatter` class has the `formatEmp()` method, which returns a formatted string containing a staffer's name and job status. Write the call specification for this method, as follows:

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

Now, call the `format_emp` function to format a list of employees:

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

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

Restrictions

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

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

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

Calling Java from PL/SQL

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

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

```

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

```

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

```

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

```

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

```

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

```

In the following example, you call the `row_count` function, which defined in [Example 6-3](#) on page 6-8, from a standalone PL/SQL stored procedure:

```

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

```

In the following example, you call the `raise_sal` method of the `Employee` object type, which is defined in ["Implementing Object Type Methods"](#) on page 6-18, from an anonymous PL/SQL block:

```

DECLARE
emp_id NUMBER(4);
v emp_type;
BEGIN
-- assign a value to emp_id

```

```
SELECT VALUE(e) INTO v FROM emps e WHERE empno = emp_id;
v.raise_sal(500);
UPDATE emps e SET e = v WHERE empno = emp_id;
...
END;
```

Calling PL/SQL from Java

Java Database Connectivity (JDBC) and SQLJ enable you to call PL/SQL stored functions and procedures. For example, you want to call the following stored function, which returns the balance of a specified bank account:

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

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

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

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

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

How Oracle JVM Handles Exceptions

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

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

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

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

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

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

Java Stored Procedures Application Example

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

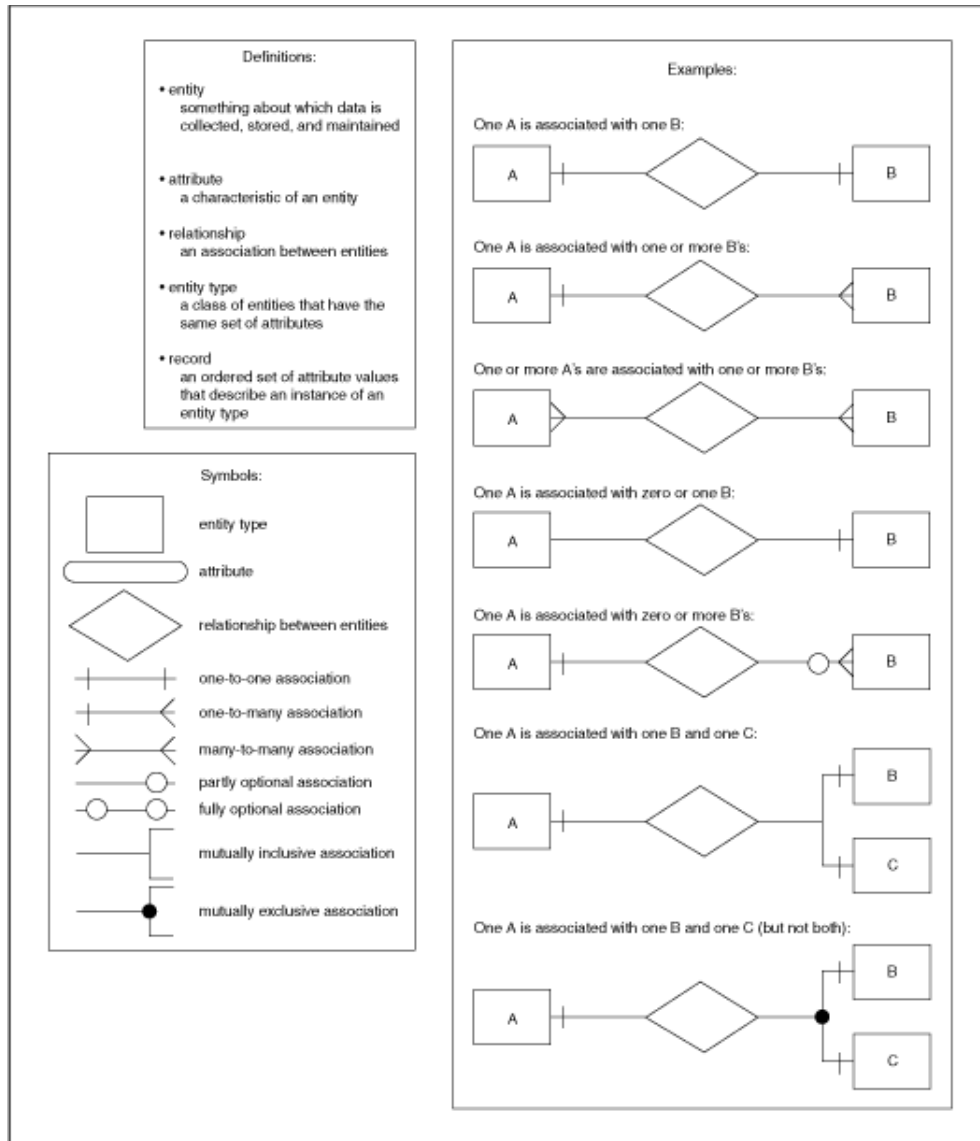
This chapter contains the followings sections:

- [Drawing the Entity-Relationship Diagram](#)
- [Planning the Database Schema](#)
- [Creating the Database Tables](#)
- [Writing the Java Classes](#)
- [Loading the Java Classes](#)
- [Publishing the Java Classes](#)
- [Calling the Java Stored Procedures](#)

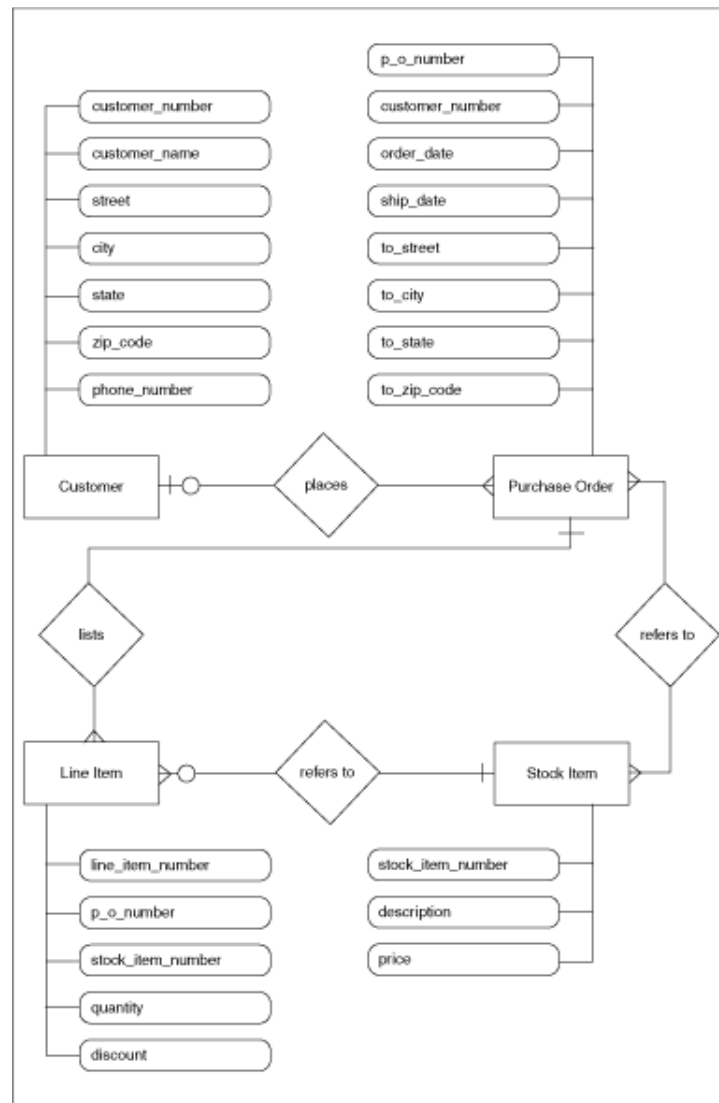
Drawing the Entity-Relationship Diagram

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

Figure 8–1 Rule for Drawing an E-R Diagram



As [Figure 8–2](#) illustrates, the basic entities in this example are customers, purchase orders, line items, and stock items.

Figure 8–2 E-R Diagram for Purchase Order Application

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

Purchase Order has a many-to-many relationship with Stock Item because a purchase order can refer to many stock items, and a stock item can be referred to by many purchase orders. However, you do not know which purchase orders refer to which stock items. As a result, you introduce the notion of a line item. Purchase Order has a one-to-many relationship with Line Item because a purchase order can list many line items, but a given line item can be listed by only one purchase order.

Line Item has a many-to-one relationship with Stock Item because a line item can refer to only one stock item, but a given stock item can be referred to by many line items. The relationship is optional because zero line items may refer to a given stock item.

Planning the Database Schema

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

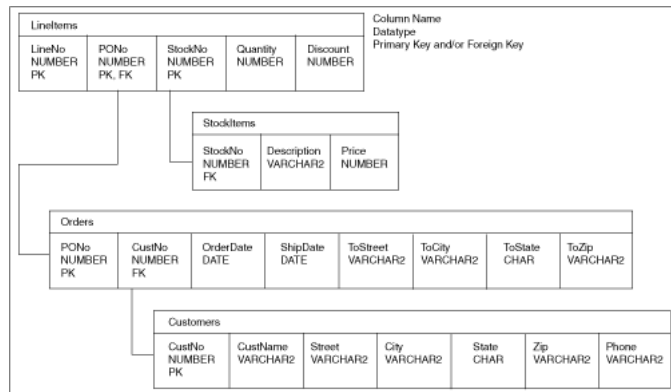
- Customers
- Orders
- LineItems
- StockItems

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

Figure 8–3 depicts the relationships between tables. The E-R diagram showed that a line item has a relationship with a purchase order and with a stock item. In the schema plan, you establish these relationships using primary and foreign keys.

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

Figure 8–3 Schema Plan for Purchase Order Application



Creating the Database Tables

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

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


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

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

The E-R diagram in [Figure 8-2](#) showed that line items have a relationship with purchase orders and stock items. The `LineItems` table manages these relationships using foreign keys. For example, the `StockNo` foreign key column in `LineItems` references the `StockNo` primary key column in `StockItems`, which you can define as follows:

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

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

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

Writing the Java Classes

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

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

public class POManager
{
```

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

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

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

```

        pstmt.setString(8, toZipCode);
        pstmt.executeUpdate();
        pstmt.close();
    }
    catch (SQLException e)
    {
        System.err.println(e.getMessage());
    }
}

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

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

static void printResults (ResultSet rset) throws SQLException
{
    String buffer = "";
    try
    {
        ResultSetMetaData meta = rset.getMetaData();
        int cols = meta.getColumnCount(), rows = 0;

```

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

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

public static void changeQuantity (int newQty, int orderNo, int stockNo)
```

```

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

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

```

Loading the Java Classes

After writing the Java classes, use the `loadjava` tool to upload your Java stored procedures into Oracle Database, as follows:

```

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

```

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

Publishing the Java Classes

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

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

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

Then, create the package body by writing call specifications for the Java methods, as follows:

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

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

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

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

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

PROCEDURE check_stock_item (stock_no NUMBER)
```

```

AS LANGUAGE JAVA
NAME 'POManager.checkStockItem(int)';

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

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

```

Calling the Java Stored Procedures

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

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

```

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

```

Register your customers, as follows:

```

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

```

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

```

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

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

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

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

```

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

```

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

```

Call completed.

Oracle Database Java Application Performance

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

- [Oracle JVM Just-in-Time Compiler \(JIT\)](#)
- [Java Memory Usage](#)

Oracle JVM Just-in-Time Compiler (JIT)

This section describes the just-in-time (JIT) compiler that has been introduced since Oracle Database 11g release 1 (11.1). This section covers the following topics:

- [Overview of Oracle JVM JIT](#)
- [Advantages of JIT Compilation](#)
- [Methods Introduced in Oracle Database 11g](#)

Note: The JIT compiler is a replacement of the compilers that were used in the earlier versions of Oracle Database.

Overview of Oracle JVM JIT

Starting with Oracle 11g release 1 (11.1), there is a just-in-time (JIT) compiler for Oracle JVM environment. A JIT compiler for Oracle JVM enables much faster execution because, it manages the invalidation, recompilation, and storage of code without an external mechanism. Based on dynamically gathered profiling data, this compiler transparently selects Java methods to compile the native machine code and dynamically makes them available to running Java sessions. Additionally, the compiler can take advantage of the class resolution model of Oracle JVM to optionally persist compiled Java methods across database calls, sessions, or instances. Such persistence avoids the overhead of unnecessary recompilations across sessions or instances, when it is known that semantically the Java code has not changed.

The JIT compiler is controlled by a new boolean-valued initialization parameter called `java_jit_enabled`. When running heavily used Java methods with `java_jit_enabled` parameter value as `true`, the Java methods are automatically compiled to native code by the JIT compiler and made available for use by all sessions in the instance. Setting the `java_jit_enabled` parameter to `true` also causes further JIT compilation to cease, and reverts any already compiled methods to be interpreted. The VM automatically recompiles native code for Java methods when necessary, such as following resolution of the containing Java class.

The JIT compiler runs as an MMON slave, in a single background process for the instance. So, while the JIT compiler is running and actively compiling methods, you may see this background process consuming CPU and memory resources on par with an active user Java session.

Advantages of JIT Compilation

The following are the advantages of using JIT compilation over the compilation techniques used in earlier versions of Oracle database:

- JIT compilation works transparently
- JIT compilation speeds up the performance of Java classes
- JIT stored compiled code avoids recompilation of Java programs across sessions or instances when it is known that semantically the Java code has not changed.
- JIT compilation does not require a C compiler
- JIT compilation eliminates some of the array bounds checking
- JIT compilation eliminates common sub-expressions within blocks
- JIT compilation eliminates empty methods
- JIT compilation defines the region for register allocation of local variables
- JIT compilation eliminates the need of flow analysis
- JIT compilation limits inline code

Methods Introduced in Oracle Database 11g

Since 11g release 1 (11.1), the `DBMS_JAVA` package has been enhanced with the following new public methods to provide Java entrypoints for controlling synchronous method compilation and reverting to interpreted method execution:

set_native_compiler_option

This procedure sets a native-compiler option to the specified value for the current schema. If the option given by *optionName* is not allowed to have duplicate values, then the value is ignored.

```
PROCEDURE set_native_compiler_option(optionName VARCHAR2,  
value VARCHAR2);
```

unset_native_compiler_option

This procedure unsets a native-compiler option/value pair for the current schema. If the option given by *optionName* is not allowed to have duplicate values, then the value is ignored.

```
PROCEDURE unset_native_compiler_option(optionName VARCHAR2,  
value VARCHAR2);
```

compile_class

This function compiles all methods defined by the class that is identified by *classname* in the current schema. It returns the number of methods successfully compiled. If the class does not exist, then an `ORA-29532 (Uncaught Java exception)` occurs.

```
FUNCTION compile_class(classname VARCHAR2) return NUMBER;
```

compile_class

This function compiles all methods defined by the class that is identified by *classname* in the supplied schema. It returns the number of methods successfully compiled. If the class does not exist in the schema or the schema does not exist, then an ORA-29532 (Uncaught Java exception) occurs.

```
FUNCTION compile_class(schema VARCHAR2,
classname VARCHAR2) return NUMBER;
```

uncompile_class

This function uncompiles all methods defined by the class that is identified by *classname* in the current schema. It returns the number of methods successfully uncompiled. If the value of the argument *permanentp* is nonzero, then mark these methods as permanently dynamically uncompileable. Otherwise, they are eligible for future dynamic recompilation. If the class does not exist, then an ORA-29532 (Uncaught Java exception) occurs.

```
FUNCTION uncompile_class(classname VARCHAR2,
permanentp NUMBER default 0) return NUMBER;
```

uncompile_class

This function uncompiles all methods defined by the class that is identified by *classname* in the supplied schema. It returns the number of methods successfully uncompiled. If the value of the argument *permanentp* is nonzero, then mark these methods as permanently dynamically uncompileable. Otherwise, they are eligible for future dynamic recompilation. If the class does not exist in the schema or the schema does not exist, then an ORA-29532 (Uncaught Java exception) occurs.

```
FUNCTION uncompile_class(schema VARCHAR2,
classname VARCHAR2,
permanentp NUMBER default 0) return NUMBER;
```

compile_method

This function compiles the method specified by *name* and *Java type* signatures defined by the class, which is identified by *classname* in the current schema. It returns the number of methods successfully compiled. If the class does not exist, then an ORA-29532 (Uncaught Java exception) occurs.

```
FUNCTION compile_method(classname VARCHAR2,
methodname VARCHAR2,
methodsig VARCHAR2) return NUMBER;
```

compile_method

This function compiles the method specified by *name* and *Java type* signatures defined by the class that is identified by *classname* in the supplied schema. It returns the number of methods successfully compiled. If the class does not exist in the schema or the schema does not exist, then an ORA-29532 (Uncaught Java exception) occurs.

```
FUNCTION compile_method(schema VARCHAR2,
classname VARCHAR2,
methodname VARCHAR2,
```

```
methodsig VARCHAR2) return NUMBER;
```

uncompile_method

This function uncompiler the method specified by the *name* and *Java type* signatures defined by the class that is identified by *classname* in the current schema. It returns the number of methods successfully uncompiled. If the value of the argument *permanenttp* is nonzero, then mark the method as permanently dynamically uncompiler. Otherwise, it is eligible for future dynamic recompilation. If the class does not exist, then an ORA-29532 (Uncaught Java exception) occurs.

```
FUNCTION uncompile_method(classname VARCHAR2,  
methodname VARCHAR2,  
methodsig VARCHAR2,  
permanenttp NUMBER default 0) return NUMBER;
```

uncompile_method

This function uncompiler the method specified by the *name* and *Java type* signatures defined by the class that is identified by *classname* in the supplied schema. It returns the number of methods successfully uncompiled. If the value of the argument *permanenttp* is nonzero, then mark the method as permanently dynamically uncompiler. Otherwise, it is eligible for future dynamic recompilation. If the class does not exist in the schema or the schema does not exist, then an ORA-29532 (Uncaught Java exception) occurs.

```
FUNCTION uncompile_method(schema VARCHAR2,  
classname VARCHAR2,  
methodname VARCHAR2,  
methodsig VARCHAR2,  
permanenttp NUMBER default 0) return NUMBER;
```

Java Memory Usage

The typical and custom database installation process furnishes a database that has been configured for reasonable Java usage during development. However, run-time use of Java should be determined by the usage of system resources for a given deployed application. Resources you use during development can vary widely, depending on your activity. The following sections describe how you can configure memory, how to tell how much System Global Area (SGA) memory you are using, and what errors denote a Java memory issue:

- [Configuring Memory Initialization Parameters](#)
- [Java Pool Memory](#)
- [Displaying Used Amounts of Java Pool Memory](#)
- [Correcting Out of Memory Errors](#)
- [Displaying Java Call and Session Heap Statistics](#)

Configuring Memory Initialization Parameters

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

- SHARED_POOL_SIZE

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

The memory specified in `SHARED_POOL_SIZE` is consumed transiently when you use the `loadjava` tool. The database initialization process requires `SHARED_POOL_SIZE` to be set to 96 MB because it loads the Java binaries for approximately 8,000 classes and resolves them. The `SHARED_POOL_SIZE` resource is also consumed when you create call specifications and as the system tracks dynamically loaded Java classes at run time.

- `JAVA_POOL_SIZE`

Oracle JVM memory manager uses `JAVA_POOL_SIZE` mainly for in-memory representation of Java method and class definitions, and static Java states that are migrated to session space at end-of-call in shared server mode. In the first case, you will be sharing the memory cost with all Java users. In the second case, the value of `JAVA_POOL_SIZE` varies according to the actual amount of state held in static variables for each session. But, Oracle recommends the minimum value as 50 MB.

- `JAVA_SOFT_SESSIONSPACE_LIMIT`

This parameter lets you specify a soft limit on Java memory usage in a session, which will warn you if you must increase your Java memory limits. Every time memory is allocated, the total memory allocated is checked against this limit.

When a user's session Java state exceeds this size, Oracle JVM generates a warning that is written into the trace files. Although this warning is an informational message and has no impact on your application, you should understand and manage the memory requirements of your deployed classes, especially as they relate to usage of session space.

Note: This parameter is applicable only to a shared-server environment.

- `JAVA_MAX_SESSIONSPACE_SIZE`

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

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

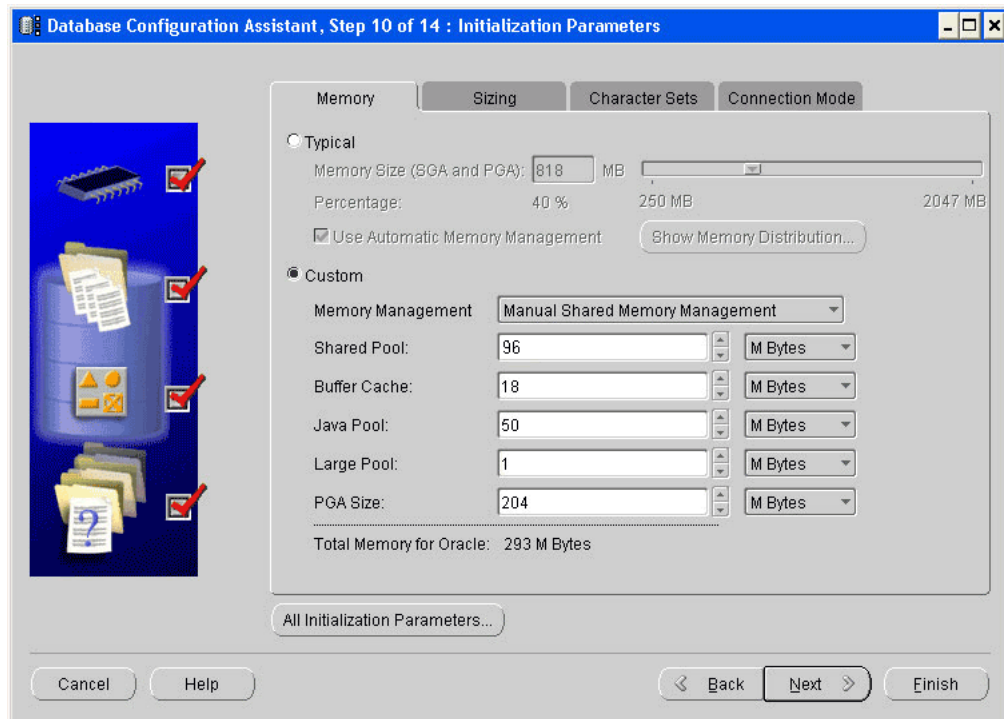
Note: This parameter is applicable only to a shared-server environment.

Initializing Pool Sizes within Database Templates

You can set the defaults for `JAVA_POOL_SIZE` and `SHARED_POOL_SIZE` in the database installation template.

Figure 9–1 illustrates how the Database Configuration Assistant enables you to modify these values in the Memory section.

Figure 9–1 Configuring Oracle JVM Memory Parameters



Java Pool Memory

Java pool memory is a subset of SGA, which is used exclusively by Java for memory that must be aligned pagewise. This includes the majority, but, not all of the memory used for the shared definitions of Java classes. Other uses of Java pool memory depend on the mode in which the Oracle Database server runs.

Java Pool Memory Used within a Dedicated Server

The following is what constitutes the Java pool memory used within a dedicated server:

- Most of the shared part of each Java class in use.
This includes read-only memory, such as code vectors, and methods. In total, this can average about 4 KB to 8 KB for each class.
- None of the per-session Java state of each session.
For a dedicated server, this is stored in the User Global Area (UGA) within the Program Global Area (PGA), and not within the SGA.

Under dedicated servers, the total required Java pool memory depends on the applications running and usually ranges between 10 and 50 MB.

Java Pool Memory Used within a Shared Server

The following is what constitutes the Java pool memory used within a shared server:

- Most of the shared part of each Java class in use.

This includes read-only memory, such as vectors, and methods. In total, this usually averages about 4 KB to 8 KB for each class.

- Some of the UGA for per session memory is allocated from Java pool memory. In particular, the memory for objects that remain in use across Database calls is always allocated from Java pool.

Because the Java pool memory size is limited, you must estimate the total requirement for your applications and multiply by the number of concurrent sessions the applications want to create, to calculate the total amount of necessary Java pool memory. Each UGA grows and shrinks as necessary. However, all UGAs combined must be able to fit within the entire fixed Java pool space.

Under shared servers, Java pool could be large. Java-intensive, multiuser applications could require more than 100 MB.

Note: If you are compiling code on the server, rather than compiling on the client and loading to the server, then you might need a bigger JAVA_POOL_SIZE than the default 20 MB.

Displaying Used Amounts of Java Pool Memory

You can find out how much of Java pool memory is being used by viewing the V\$SGASTAT table. Its rows include pool, name, and bytes. Specifically, the last two rows show the amount of Java pool memory used and how much is free. The total of these two items equals the number of bytes that you configured in the database initialization file.

```
SVRMGR> select * from v$sgastat;
```

POOL	NAME	BYTES
	fixed_sga	69424
	db_block_buffers	2048000
	log_buffer	524288
shared pool	free memory	22887532
shared pool	miscellaneous	559420
shared pool	character set object	64080
shared pool	State objects	98504
shared pool	message pool freequeue	231152
shared pool	PL/SQL DIANA	2275264
shared pool	db_files	72496
shared pool	session heap	59492
shared pool	joxlod: init P	7108
shared pool	PLS non-lib hp	2096
shared pool	joxlod: in ehe	4367524
shared pool	VIRTUAL CIRCUITS	162576
shared pool	joxlod: in phe	2726452
shared pool	long op statistics array	44000
shared pool	table definiti	160
shared pool	KGK heap	4372
shared pool	table columns	148336
shared pool	db_block_hash_buckets	48792
shared pool	dictionary cache	1948756
shared pool	fixed allocation callback	320
shared pool	SYSTEM PARAMETERS	63392
shared pool	joxlod: init s	7020
shared pool	KQLS heap	1570992
shared pool	library cache	6201988

```

shared pool trigger inform          32876
shared pool sql area                7015432
shared pool sessions                211200
shared pool KGFF heap               1320
shared pool joxs heap init          4248
shared pool PL/SQL MPCODE           405388
shared pool event statistics per sess 339200
shared pool db_block_buffers        136000
java pool free memory             30261248
java pool memory in use          19742720
37 rows selected.

```

Correcting Out of Memory Errors

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

- Verify that the class was actually included in the set you are loading to the server.
- Use the `loadjava -force` option to force the new class being loaded to replace the class already resident in the server.
- Use the `loadjava -resolve` option to attempt resolution of a class during the load process. This enables you to catch missing classes at load time, rather than at run time.
- Double check the status of the newly loaded class by connecting to the database in the schema containing the class, and run the following:

```
SELECT * FROM user_objects WHERE object_name = dbms_java.shortname('');
```

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

Displaying Java Call and Session Heap Statistics

Database performance view `v$sesstat` records a number of Java memory usage statistics. These statistics are updated often during Java calls. The following example shows the Java call return and session heap statistics for the database session with `SID=102`.

```
SQL> select s.sid, n.name p_name, st.value from v$session s, v$sesstat st,
v$statname n where s.sid=102
and s.sid=st.sid and n.statistic# = st.statistic# and n.name like 'java%';
```

SID	P_NAME	VALUE
102	java call heap total size	6815744
102	java call heap total size max	6815744
102	java call heap used size	668904
102	java call heap used size max	846920
102	java call heap live size	667112
102	java call heap live size max	704312
102	java call heap object count	13959
102	java call heap object count max	17173
102	java call heap live object count	13907

102	java call heap live object count max	14916
102	java call heap gc count	432433
102	java call heap collected count	123196423
102	java call heap collected bytes	5425972216
102	java session heap used size	444416
102	java session heap used size max	444416
102	java session heap live size	444416
102	java session heap live size max	444416
102	java session heap object count	0
102	java session heap object count max	0
102	java session heap live object count	0
102	java session heap live object count max	0
102	java session heap gc count	0
102	java session heap collected count	0
102	java session heap collected bytes	0

24 rows selected.

Security for Oracle Database Java Applications

Security is a large arena that includes network security for the connection, access, and execution control of operating system resources or of Java virtual machine (JVM)-defined and user-defined classes. Security also includes bytecode verification of Java Archive (JAR) files imported from an external source. The following sections describe the various security support available for Java applications within Oracle Database:

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

Network Connection Security

The two major aspects to network security are authentication and data confidentiality. The type of authentication and data confidentiality is dependent on how you connect to the database, either through Oracle Net or Java Database Connectivity (JDBC) connection. The following table provides the security description for Oracle Net and JDBC connections:

Connection Security	Description
Oracle Net	<p>The database can require both authentication and authorization before allowing a user to connect to it. Oracle Net database connection security can require one or more of the following:</p> <ul style="list-style-type: none"> ■ A user name and password for client verification. For each connection request, a user name and password configured within Oracle Net has to be provided. ■ Advanced Networking Option for encryption, kerberos, or secureId. ■ SSL for certificate authentication.
JDBC	<p>The JDBC connection security that is required is similar to the constraints required on an Oracle Net database connection.</p>

See Also:

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

Database Contents and Oracle JVM Security

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

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

These resources can be protected by the following methods:

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

Note: The Oracle JVM is shipped with strong but limited encryption as included in JDK1.5 and JDK 6. If you want to have unlimited encryption strength in your application, then you must download and install the appropriate version-specific files from the following Sun Microsystems Web site

<http://developers.sun.com/downloads/>

This section covers the following topics:

- [Java2 Security](#)
- [Setting Permissions](#)
- [Debugging Permissions](#)
- [Permission for Loading Classes](#)

See Also:

- *Oracle Database Advanced Application Developer's Guide*
- [Chapter 11, "Schema Objects and Oracle JVM Utilities"](#)

Java2 Security

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

Java2 security provides a flexible and configurable security for Java applications. With Java2 security, you can define exactly what permissions on each loaded object that a schema or role will have. In Oracle8i Database release 8.1.5, the following secure roles are available:

- `JAVAUSERPRIV`
Few permissions, including examining properties
- `JAVASYSPRIV`
Major permissions, including updating Oracle JVM-protected packages

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

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

Java2 security was created for the non-database world. When you apply the Java2 security model within the database, certain differences manifest themselves. For example, Java2 security defines that all applets are implicitly untrusted and all classes within the `CLASSPATH` are trusted. In Oracle Database, all classes are loaded within a secure database. As a result, no classes are trusted.

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

Java2 Security Standard	Oracle Database Security 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 using the <code>-usepolicy</code> flag on the <code>java</code> command.	You must specify the policy within <code>PolicyTable</code> .
You can write your own <code>SecurityManager</code> or use the Launcher.	You can write your own <code>SecurityManager</code> . However, Oracle recommends that you use only Oracle Database <code>SecurityManager</code> or that you extend it. If you want to modify the behavior, then you should not define a <code>SecurityManager</code> . Instead, you should extend <code>oracle.aurora.rdbms.SecurityManagerImpl</code> and override specific methods.
<code>SecurityManager</code> is not initialized by default. You must initialize <code>SecurityManager</code> .	Oracle JVM always initializes <code>SecurityManager</code> at startup.
Permissions are determined by the location or the URL, where the application or applet is loaded, or keycode, that is, signed code.	Permissions are determined by the schema in which the class is loaded. Oracle Database does not support signed code.
The security policy is defined in a file.	The <code>PolicyTable</code> definition is contained in a secure database table.

Java2 Security Standard	Oracle Database Security Implementation
You can update the security policy file using a text editor or a tool, if you have the appropriate permissions.	You can update <code>PolicyTable</code> through <code>DBMS_JAVA</code> procedures. After initialization, only <code>JAVA_ADMIN</code> has permission to modify <code>PolicyTable</code> . <code>JAVA_ADMIN</code> must grant you the right to modify <code>PolicyTable</code> so that you can grant permissions to others.
Permissions are assigned to a protection domain, which classes can belong to.	All classes within the same schema are in the same protection domain.
You can use the <code>CodeSource</code> class for identifying code.	You can use the <code>CodeSource</code> class for identifying schema.
<ul style="list-style-type: none"> ▪ The <code>equals()</code> method returns <code>true</code> if the URL and certificates are equal. ▪ The <code>implies()</code> method returns <code>true</code> 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 <code>true</code> if the schemas are the same. ▪ The <code>implies()</code> method returns <code>true</code> if the schemas are the same.
Supports positive permissions only, that is, <code>grant</code> .	Supports both positive and limitation permissions, that is, <code>grant</code> and <code>restrict</code> .

Setting Permissions

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

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

Table Column	Description
Kind	<code>GRANT</code> or <code>RESTRICT</code> . Shows whether this permission is a positive or a limitation permission.
Grantee	The name of the user, schema, or role to which the <code>Permission</code> object is assigned.
Permission_schema	The schema in which the <code>Permission</code> object is loaded.
Permission_type	The <code>Permission</code> class type, which is designated by a string containing the full class name, such as, <code>java.io.FilePermission</code> .
Permission_name	The target attribute of the <code>Permission</code> object. You use this when defining the permission. When defining the target for a <code>Permission</code> object of type <code>PolicyTablePermission</code> , the name can become quite complicated. See Also: " Acquiring Administrative Permission to Update Policy Table " on page 10-8
Permission_action	The action attribute of the <code>Permission</code> object. Many permissions expect a null value if no action is appropriate for the permission.
Status	<code>ENABLED</code> and <code>DISABLED</code> . After creating a row for a <code>Permission</code> object, you can disable or reenale it. This column shows whether the permission is enabled or disabled.

Table Column	Description
Key	Sequence number you use to identify this row. This number should be supplied when disabling, enabling, or deleting a permission.

There are two ways to set permissions:

- [Fine-Grain Definition for Each Permission](#)
- [General Permission Definition Assigned to Roles](#)

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

Fine-Grain Definition for Each Permission

Using fine-grain definition, you can grant each permission individually to specific users or roles. If you do not grant a permission for access, then the schema will be denied access. To set individual permissions within the policy table, you must provide the following information:

Parameter	Description
Grantee	The name of the user, schema, or role to which you want the grant to apply. <code>PUBLIC</code> specifies that the row applies to all users.
Permission type	The <code>Permission</code> class on which you are granting permission. For example, if you were 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> , then 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 meaning of the target attribute as defined by the <code>Permission</code> class. Examine the appropriate <code>Permission</code> class for the relevant name.
Permission action	The type of action that you can specify. This can vary according to the permission type. For example, <code>FilePermission</code> can have the action, read or write.
Key	Number returned from grant or limit to use on enable, disable, or delete methods.

You can grant permissions using either SQL or Java. Each version returns a row key identifier that identifies the row within the permission table. In the Java version of `DBMS_JAVA`, each method returns the row key identifier, either as a returned parameter or as an `OUT` variable in the parameter list. In the PL/SQL `DBMS_JAVA` package, the row key is returned only in the procedure that defines the `key OUT` parameter. This key is used to enable and disable specific permissions.

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

Note: If you are granting `FilePermission`, then you must provide the physical name of the directory or file, such as `/private/oracle`. You cannot provide either an environment variable, such as `$ORACLE_HOME`, or a symbolic link. To denote all files within a directory, provide the `*` symbol, as follows:

```
/private/oracle/*
```

To denote all directories and files within a directory, provide the `-` symbol, as follows:

```
/private/oracle/-
```

You can grant permissions using the `DBMS_JAVA` package, as follows:

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

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

You can grant permissions using Java, as follows:

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

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

You can limit permissions using the `DBMS_JAVA` package, as follows:

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

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

You can limit permissions using Java, as follows:

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

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


[Example 10–1](#) shows how to use the `grant_permission()` method to grant permissions. [Example 10–2](#) shows how to limit permissions using the `restrict()` method.

Example 10–1 Granting Permissions

Assuming that you have appropriate permissions to modify the policy table, you can use the `grant_permission()` method, which is in the `DBMS_JAVA` package, to modify `PolicyTable` to allow user access to the indicated file. In this example, the user, Larry, has modification permission on `PolicyTable`. Within a SQL package, Larry can grant permission to Dave to read and write a file, as follows:

```
connect larry
Enter password: password

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

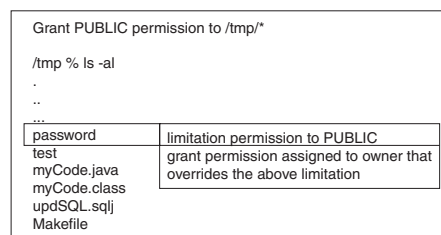
REM commit the changes to PolicyTable
commit;
```

Example 10–2 Limiting Permissions

You can use the `restrict()` method to specify a limitation or exception to general rules. A general rule is a rule where, in most cases, the permission is true or granted. However, there may be exceptions to this rule. For these exceptions, you specify a limitation permission.

If you have defined a general rule that no one can read or write an entire directory, then you can define a limitation on an aspect of this rule through the `restrict()` method. For example, if you want to allow access to all files within the `/tmp` directory, except for your password file that exists in that directory, then you would grant permission for read and write to all files within `/tmp` and limit read and write access to the password file.

If you want to specify an exception to the limitation, then you must create an explicit grant permission to override the limitation permission. In the previously mentioned scenario, if you want the file owner to still be able to modify the password file, then you can grant a more explicit permission to allow access to one user, which will override the limitation. Oracle JVM security combines all rules to understand who really has access to the password file. This is demonstrated in the following diagram:



The explicit rule is as follows:

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

The following code implements this example:

```
connect larry
```

Enter password: *password*

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

REM Limit permission to all users (PUBLIC) from reading or writing the
REM password file in /tmp.
call dbms_java.restrict_permission('PUBLIC', 'java.io.FilePermission',
'/tmp/password', 'read,write');

REM By providing a more specific rule that overrides the limitation,
REM Larry can read and write /tmp/password.
call dbms_java.grant_permission('LARRY', 'java.io.FilePermission',
'/tmp/password', 'read,write');

commit;
```

The preceding code performs the following actions:

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

Acquiring Administrative Permission to Update Policy Table

All permissions are rows in `PolicyTable`. Because it is a table in the database, you need appropriate permissions to modify it. Specifically, the `PolicyTablePermission` object is required to modify the table. After initializing Oracle JVM, only a single role, `JAVA_ADMIN`, is granted `PolicyTablePermission` to modify `PolicyTable`. The `JAVA_ADMIN` role is immediately assigned to the database administrator (DBA). Therefore, if you are assigned to the DBA group, then you will automatically take on all `JAVA_ADMIN` permissions.

If you need to add permissions as rows to this table, `JAVA_ADMIN` must grant your schema update rights using `PolicyTablePermission`. This permission defines that your schema can add rows to the table. Each `PolicyTablePermission` is for a specific type of permission. For example, to add a permission that controls access to a file, you must have `PolicyTablePermission` that lets you grant or limit a permission on `FilePermission`. Once this occurs, you have administrative permission for `FilePermission`.

An administrator can grant and limit `PolicyTablePermission` in the same manner as other permissions, but the syntax is complicated. For ease of use, you can use the `grant_policy_permission()` or `grantPolicyPermission()` method to grant administrative permissions.

You can grant policy table administrative permission using `DBMS_JAVA`, as follows:

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

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

You can grant policy table administrative permission using Java, as follows:

```
long oracle.aurora.rdbms.security.PolicyTableManager.grantPolicyPermission (
java.lang.String grantee,
```

```

java.lang.String permission_type, java.lang.String permission_name);

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

```

Parameter	Description
Grantee	The name of the user, schema, or role to which you want the grant to apply. PUBLIC specifies that the row applies to all users.
Permission_schema	The schema where the Permission class is loaded.
Permission_type	The Permission class on which you are granting permission. For example, if you were defining access to a file, the permission type would be FilePermission. This parameter requires a fully-qualified name of a class that extends java.lang.security.Permission. If the class is not within SYS, the name should be prefixed by schema:. For example, mySchema:myPackage.MyPermission is a valid name for a user-generated permission.
Permission_name	The meaning of the target attribute as defined by the Permission class. Examine the appropriate Permission class for the relevant name.
Row_number	Number returned from grant or limitation to use on enable, disable, or delete methods.

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

[Example 10–3](#) shows how you can modify PolicyTable.

Example 10–3 Granting PolicyTable Permission

This example shows SYS, which has the JAVA_ADMIN role assigned, giving Larry permission to update PolicyTable for FilePermission. Once this permission is granted, Larry can grant permissions to other users for reading, writing, and deleting files.

```

REM Connect as SYS, which is assigned JAVA_ADMIN role, to give Larry permission
REM to modify the PolicyTable
connect SYS as SYSDBA
Enter password: password

REM SYS grants Larry the right to administer permissions for
REM FilePermission
call dbms_java.grant_policy_permission('LARRY', 'SYS', 'java.io.FilePermission',
'*');

```

Creating Permissions

You can create your own permission type by performing the following steps:

1. Create and load the user permission

Create your own permission by extending the `java.security.Permission` class. Any user-defined permission must extend `Permission`. The following example creates `MyPermission`, which extends `BasicPermission`, which, in turn, extends `Permission`.

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

public class MyPermission extends BasicPermission
{
    public MyPermission(String name)
    {
        super(name);
    }

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

2. Grant administrative and action permissions to specified users

When you create a permission, you are designated as the owner of that permission. The owner is implicitly granted administrative permission. This means that the owner can be an administrator for this permission and can run `grant_policy_permission()`. Administrative permission enable the user to update the policy table for the user-defined permission.

For example, if `LARRY` creates a permission, `MyPermission`, then only he can call `grant_policy_permission()` for himself or another user. This method updates `PolicyTable` on who can grant rights to `MyPermission`. The following code demonstrates this:

```
REM Since Larry is the user that owns MyPermission, Larry connects to
REW the database to assign permissions for MyPermission.
connect larry
Enter password: password
```

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

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

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

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

```

REM Once able to modify PolicyTable for MyPermission, Larry grants himself
REM full permission for MyPermission. Notice that the Permission is prefixed
REM with its owner schema.
call dbms_java.grant_permission( 'LARRY', 'LARRY:test.larry.MyPermission', '*',
null);

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

REM commit the changes to PolicyTable
commit;

```

3. Implement security checks using the permission

Once you have created, loaded, and assigned permissions for `MyPermission`, you must implement the call to `SecurityManager` to have the permission checked. There are four methods in the following example: `sensitive()`, `act()`, `print()`, and `hello()`. Because of the permissions granted using SQL in the preceding steps, the following users can run methods within the example class:

- LARRY can run any of the methods.
- DAVE is given permission to run only the `act()` method.
- Anyone can run the `print()` and `hello()` methods. The `print()` method does not check any permissions. As a result, anyone can run it. The `hello()` method runs `AccessController.doPrivileged()`, which means that the method runs with the permissions assigned to LARRY. This is referred to as the definer's rights.

```

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

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

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

public class Larry {

    private static String secret = "Larry's secret";
    MyPermission sensitivePermission = new MyPermission("sensitive");

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

```

```
    }

    /**
    * Will display a message from Larry. You must 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);
    }

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

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

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

Enabling or Disabling Permissions

Once you have created a row that defines a permission, you can disable it so that it no longer applies. However, if you decide that you want the row action again, then you can enable the row. You can delete the row from the table if you believe that it will never be used again. To delete, you must first disable the row. If you do not disable the row, then the deletion will not occur.

To disable rows, you can use either of the following methods:

- `revoke_permission()`

This method accepts parameters similar to the `grant()` and `restrict()` methods. It searches the entire policy table for all rows that match the parameters provided.

- `disable_permission()`

This method disables only a single row within the policy table. To do this, it accepts the policy table key as parameter. This key is also necessary to enable or delete a permission. To retrieve the permission key number, perform one of the following:

- Save the key when it is returned on the `grant` or `limit` calls. If you do not foresee a need to ever enable or disable the permission, then you can use the `grant` and `limit` calls that do not return the permission number.
- Look up `DBA_JAVA_POLICY` or `USER_JAVA_POLICY` for the appropriate permission key number.

You can disable permissions using `DBMS_JAVA`, as follows:

```
procedure revoke_permission (permission_schema varchar2, permission_type varchar2,
permission_name varchar2,
permission_action varchar2)
procedure disable_permission (key number)
```

You can disable permissions using Java, as follows:

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

You can enable permissions using `DBMS_JAVA`, as follows:

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

You can enable permissions using Java, as follows:

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

You can delete permissions using `DBMS_JAVA`, as follows:

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

You can delete permissions using Java, as follows:

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

Permission Types

Whenever you want to grant or limit a permission, you must provide the permission type. The permission types with which you control access are the following:

- Java2 permission types
- Oracle-specific permission types
- User-defined permission types that extend `java.security.Permission`

[Table 10-1](#) lists the installed permission types.

Table 10–1 Predefined Permissions

Type	Permissions
Java2	<ul style="list-style-type: none"> ■ <code>java.util.PropertyPermission</code> ■ <code>java.io.SerializablePermission</code> ■ <code>java.io.FilePermission</code> ■ <code>java.net.NetPermission</code> ■ <code>java.net.SocketPermission</code> ■ <code>java.lang.RuntimePermission</code> ■ <code>java.lang.reflect.ReflectPermission</code> ■ <code>java.security.SecurityPermission</code>
Oracle specific	<ul style="list-style-type: none"> ■ <code>oracle.aurora.rdbms.security.PolicyTablePermission</code> ■ <code>oracle.aurora.security.JServerPermission</code>

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

The Oracle-specific permissions are:

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

This permission controls who can update the policy table. Once granted the right to update the policy table for a certain permission type, you can control the access to few resources.

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

To grant policy table updates, you can use the `grant_policy_permission()` method, which is in the `DBMS_JAVA` package. Once you have updated the table, you can view either the `DBA_JAVA_POLICY` or `USER_JAVA_POLICY` view to see who has been granted permissions.

- `oracle.aurora.security.JServerPermission`

This permission is used to grant and limit access to Oracle JVM resources. The `JServerPermission` extends `BasicPermission`. The following table lists the permission names for which `JServerPermission` grants access:

Permission Name	Description
<code>LoadClassInPackage.package_name</code>	Grants the ability to load a class within the specified package
<code>Verifier</code>	Grants the ability to turn the bytecode verifier on or off
<code>Debug</code>	Grants the ability for debuggers to connect to a session
<code>JRIExtensions</code>	Grants the use of MEMSTAT

Permission Name	Description
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 Oracle JVM, several roles are populated with certain permission grants. The following tables show these roles and their initial Permissions:

- [Table 10-2, "SYS Initial Permissions"](#)
- [Table 10-3, "PUBLIC Default Permissions"](#)
- [Table 10-4, "JAVAUSERPRIV Permissions"](#)
- [Table 10-5, "JAVASYSPRIV Permissions"](#)
- [Table 10-6, "JVADEBUGPRIV Permissions"](#)

The `JAVA_ADMIN` role is given access to modify the policy table for all permissions. All DBAs, including `SYS`, are granted `JAVA_ADMIN`. Full administrative rights to update the policy table are granted for the permissions listed in [Table 10-1](#). In addition to the `JAVA_ADMIN` permissions, `SYS` is granted some additional permissions.

[Table 10-2](#) lists the additional permissions granted to `SYS`.

Table 10-2 *SYS Initial Permissions*

Permission Type	Permission Name	Action
<code>oracle.aurora.rdbms.security.PolicyTablePermission</code>	*	Administrative rights to modify the policy table
<code>oracle.aurora.security.JServerPermission</code>	*	null
<code>java.net.NetPermission</code>	*	null
<code>java.security.SecurityPermission</code>	*	null
<code>java.util.PropertyPermission</code>	*	write
<code>java.lang.reflect.ReflectPermission</code>	*	null
<code>java.lang.RuntimePermission</code>	*	null
	<code>loadLibrary.xaNative</code>	null
	<code>loadLibrary.corejava</code>	null
	<code>loadLibrary.corejava_d</code>	null

[Table 10-3](#) lists permissions initially granted to all users.

Table 10–3 PUBLIC Default Permissions

Permission Type	Permission Name	Action
oracle.aurora.rdbms.security. PolicyTablePermission	java.lang.RuntimePermission.loadLibrary.*	null
java.util.PropertyPermission	*	read
	user.language	write
java.lang.RuntimePermission	_	null
	exitVM	null
	createSecurityManager	null
	modifyThread	null
	modifyThreadGroup	null
oracle.aurora.security. JServerPermission	loadClassInPackage.* except for loadClassInPackage.java.*, loadClassInPackage.oracle.aurora.*, and loadClassInPackage.jdbc.*	null

Table 10–4 lists permissions initially granted to the JAVAUSERPRIV role.

Table 10–4 JAVAUSERPRIV Permissions

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

Table 10–5 lists permissions initially granted to the JAVASYSPRIV role.

Table 10–5 JAVASYSPRIV Permissions

Permission Type	Permission Name	Action
java.io.SerializablePermission	*	no applicable action
java.io.FilePermission	<<ALL FILES>>	read, write, execute, delete
java.net.SocketPermission	*	accept, connect, listen, resolve
java.lang.RuntimePermission	createClassLoader	null
	getClassLoader	null
	setContextClassLoader	null
	setFactory	null
	setIO	null
	setFileDescriptor	null
	readFileDescriptor	null
	writeFileDescriptor	null

Table 10–6 lists permissions initially granted to the JVADEBUGPRIV role.

Table 10–6 *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 Oracle8i Database release 8.1.5, Oracle JVM security was controlled by granting the JAVASYSPRIV, JAVAUSERPRIV, or JVADEBUGPRIV role to schemas. In Oracle Database 11g, these roles still exist as permission groups. You can set up and define your own collection of permissions. Once defined, you can grant any collection of permissions to any user or role. That user will then have the same permissions that exist within the role. In addition, if you need additional permissions, then you can add individual permissions to either your specified user or role. Permissions defined within the policy table have a cumulative effect.

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

The following example gives Larry and Dave the following permissions:

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

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

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

```
commit;
```

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

See Also: ["Fine-Grain Definition for Each Permission"](#) on page 10-5.

Debugging Permissions

A debug role, JVADEBUGPRIV, was created to grant permissions for running the debugger. The permissions assigned to this role are listed in [Table 10–6](#). To receive permission to call the debug agent, the caller must have been granted JVADEBUGPRIV or the debug JServerPermission as follows:

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

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

Although a debugger provides extensive access to both code and data on the server, its use should be limited to development environments.

See Also: ["Debugging Server Applications"](#) on page 3-13

Permission for Loading Classes

To load classes, you must have the following permission:

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

where, *class_name* is the fully qualified name of the class that you are loading.

This excludes loading into System packages or replacing any System classes. Even if you are granted permission to load a System class, Oracle Database prevents you from performing the load. System classes are classes that are installed by Oracle Database using the `CREATE JAVA SYSTEM` statement. The following error is thrown if you try to replace a System class:

```
ORA-01031 "Insufficient privileges"
```

The following describes what each user can do after database installation:

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

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

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

Database Authentication Mechanisms

The following database authentication mechanisms are available:

- Password authentication
- Strong authentication
- Proxy authentication
- Single sign-on

Schema Objects and Oracle JVM Utilities

This chapter describes the schema objects that you use in Oracle Database Java environment and Oracle JVM utilities. You run these utilities from a UNIX shell or from the Microsoft Windows DOS prompt.

Note: All names supplied to these tools are case-sensitive. As a result, the schema, user name, and password should not be changed to uppercase.

This chapter contains the following sections:

- [Overview of Schema Objects](#)
- [What and When to Load](#)
- [Resolution of Schema Objects](#)
- [Compilation of Schema Objects](#)
- [The ojvmtc Tool](#)
- [The loadjava Tool](#)
- [The dropjava Tool](#)
- [The ojvmjava Tool](#)

Overview of Schema Objects

Unlike conventional Java virtual machine (JVM), which compiles and loads Java files, Oracle JVM compiles and loads schema objects. The following kinds of Java schema objects are loaded:

- Java class schema objects, which correspond to Java class files.
- Java source schema objects, which correspond to Java source files.
- Java resource schema objects, which correspond to Java resource files.

To ensure that a class file can be run by Oracle JVM, you must use the `loadjava` tool to create a Java class schema object from the class file or the source file and load it into a schema. To make a resource file accessible to Oracle JVM, you must use the `loadjava` tool to create and load a Java resource schema object from the resource file.

The `dropjava` tool deletes schema objects that correspond to Java files. You should always use the `dropjava` tool to delete a Java schema object that was created with the `loadjava` tool. Dropping schema objects using SQL data definition language (DDL)

commands will not update auxiliary data maintained by the `loadjava` tool and the `dropjava` tool.

What and When to Load

You must load resource files using the `loadjava` tool. If you create `.class` files outside the database with a conventional compiler, then you must load them with the `loadjava` tool. The alternative to loading class files is to load source files and let Oracle Database compile and manage the resulting class schema objects. In Oracle Database 11g, the most productive approach is to compile and debug most of your code outside the database, and then load the `.class` files. For a particular Java class, you can load either its `.class` file or the corresponding `.java` file, but not both.

The `loadjava` tool accepts Java Archive (JAR) files that contain either source and resource files or class and resource files. When you pass a JAR or ZIP file to the `loadjava` tool, it opens the archive and loads its members individually. There are no JAR or ZIP schema objects.

Note: When you load the contents of a JAR into the database, you have the option of creating a database object representing the JAR itself. For more information, refer to "[Database Resident JARs](#)" on page 2-15.

A file, whose content has not changed since the last time it was loaded, is not reloaded. As a result, there is little performance penalty for loading JAR files. Loading JAR files is a simple, fool-proof way to use the `loadjava` tool.

It is illegal for two schema objects in the same schema to define the same class. For example, assume that `a.java` defines class `x` and you want to move the definition of `x` to `b.java`. If `a.java` has already been loaded, then the `loadjava` tool will reject an attempt to load `b.java`. Instead, do either of the following:

- Drop `a.java`, load `b.java`, and then load the new `a.java`, which does not define `x`.
- Load the new `a.java`, which does not define `x`, and then load `b.java`.

Resolution of Schema Objects

All Java classes contain references to other classes. A conventional JVM searches for classes in the directories, ZIP files, and JAR files named in the `CLASSPATH`. In contrast, Oracle JVM searches schemas for class schema objects. Each class in the database has a resolver specification, which is Oracle Database counterpart to `CLASSPATH`. For example, the resolver specification of a class, `alpha`, lists the schemas to search for classes that `alpha` uses. Notice that resolver specifications are per-class, whereas in a classic JVM, `CLASSPATH` is global to all classes.

In addition to a resolver specification, each class schema object has a list of interclass reference bindings. Each reference list item contains a reference to another class and one of the following:

- The name of the class schema object to call when the class uses the reference
- A code indicating whether the reference is unsatisfied, that is, whether the referent schema object is known

Oracle Database facility known as **resolver** maintains reference lists. For each interclass reference in a class, the resolver searches the schemas specified by the resolver specification of the class for a valid class schema object that satisfies the reference. If all references are resolved, then the resolver marks the class valid. A class that has never been resolved, or has been resolved unsuccessfully, is marked invalid. A class that depends on a schema object that becomes invalid is also marked invalid at the time the first class is marked invalid. In other words, invalidation cascades upward from a class to the classes that use it and the classes that use these classes, and so on. When resolving a class that depends on an invalid class, the resolver first tries to resolve the referenced class, because it may be marked invalid only because it has never been resolved. The resolver does not resolve classes that are marked valid.

A developer can direct the `loadjava` tool 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 `ClassNotFoundException` exception. Furthermore, run-time resolution can fail for the following reasons:

- Lack of database resources, if the tree of classes is very large
- Deadlocks due to circular dependencies

The `loadjava` tool has two resolution modes:

- Load-and-resolve

The `-resolve` option loads all classes you specify on the command line, marks them invalid, and then resolves them. Use this mode when initially loading classes that refer to each other, and, in general, when reloading isolated classes as well. By loading all classes and then resolving them, this mode avoids the error message that occurs if a class refers to a class that will be loaded later while the command is being carried out.

- Load-then-resolve

This mode resolves each class at run time. The `-resolve` option is not specified.

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

If you can, defer resolution until all classes have been loaded. This avoids a situation in which the resolver marks a class invalid because a class it uses has not yet been loaded.

Schema Object Digest Table

The schema object digest table is an optimization that is usually invisible to developers. The digest table enables the `loadjava` tool to skip files that have not changed since they were last loaded. This feature improves the performance of makefiles and scripts that call the `loadjava` tool for collections of files, some of which must be reloaded. A reloaded archive file might also contain some files that have changed since they were last loaded and some that have not.

The `loadjava` tool detects unchanged files by maintaining a digest table in each schema. The digest table relates a file name to a digest, which is a shorthand representation or a hash, of the content of the file. Comparing digests computed for the same file at different times is a fast way to detect a change in the content of the file.

This is much faster than comparing every byte in the file. For each file it processes, the `loadjava` tool computes a digest of the content of the file and then looks up the file name in the digest table. If the digest table contains an entry for the file name that has an identical digest, then the `loadjava` tool does not load the file, because a corresponding schema object exists and is up to date. If you call the `loadjava` tool with the `-verbose` option, then it will show you the results of its digest table lookups.

Typically, the digest table is invisible to developers, because the `loadjava` tool and the `dropjava` tool keep the table synchronized with schema object additions, changes, and deletions. For this reason, always use the `dropjava` tool to delete a schema object that was created with the `loadjava` tool, even if you know how to drop a schema object using DDL.

Compilation of Schema Objects

Loading a source file creates or updates a Java source schema object and invalidates the class schema objects previously derived from the source. If the class schema objects do not exist, then the `loadjava` tool creates them. The `loadjava` tool invalidates the old class schema objects because they were not compiled from the newly loaded source. Compilation of a newly loaded source, for example, class A, is automatically triggered by any of the following conditions:

- The resolver, while working on class B, finds that class B refers to class A, but class A is invalid.
- The compiler, while compiling the source of class B, finds that class B refers to class A, but class A is invalid.
- The class loader, while trying to load class A for running it, finds that class A is invalid.

To force compilation when you load a source file, use the `loadjava -resolve` option.

The compiler writes error messages to the predefined `USER_ERRORS` view. The `loadjava` tool retrieves and displays the messages produced by its compiler invocations.

The compiler recognizes some options. There are two ways to specify options to the compiler. If you run the `loadjava` tool with the `-resolve` option, then you can specify compiler options on the command line. You can additionally specify persistent compiler options in a per-schema database table, `JAVA$OPTIONS`. You can use the `JAVA$OPTIONS` table for default compiler options, which you can override selectively using a `loadjava` tool option.

Note: A command-line option overrides and clears the matching entry in the `JAVA$OPTIONS` table.

A `JAVA$OPTIONS` row contains the names of source schema objects to which an option setting applies. You can use multiple rows to set the options differently for different source schema objects. The compiler looks up options in `JAVA$OPTIONS` when it has been called by the class loader or when called from the command line without specifying any options. When compiling a source schema object for which there is neither a `JAVA$OPTIONS` entry nor a command-line value for an option, the compiler assumes a default value, as follows:

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

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

See Also: ["Compiler Options Specified in a Database Table"](#) on page 2-7

The ojvmtc Tool

The `ojvmtc` tool enables you to resolve all external references, prior to running the `loadjava` tool. The `ojvmtc` tool allows the specification of a classpath that specifies the JARs, classes, or directories to be used to resolve class references. When an external reference cannot be resolved, this tool either produces a list of unresolved references or generated stub classes to allow resolution of the references, depending on the options specified. Generated stub classes throw a `java.lang.ClassNotFoundException` if it is referenced at runtime.

The syntax is:

```
ojvmtc [-help ] [-bootclasspath] [-server connect_string] [-jar jar_name] [-list]
-classpath jar1:path2:jar2
jars,...,classes
```

For example:

```
ojvmtc -bootclasspath $JAVA_HOME/jre/lib/rt.jar -classpath
classdir/lib1.jar:classdir/lib2.jar -jar set.jar app.jar
```

The preceding example uses `rt.jar`, `classdir/lib1.jar`, and `classdir/lib2.jar` to resolve references in `app.jar`. All the classes examined are added to `set.jar`, except for those found in `rt.jar`.

Another example is:

```
ojvmtc -server thin:scott/@localhost:5521:orcl -classpath jar1:jar2 -list app2.jar
Password:password
```

The preceding example uses classes found in the server specified by the connection string as well as `jar1` and `jar2` to resolve `app2.jar`. Any missing references are displayed to `stdout`.

[Table 11–1](#) summarizes the arguments of this command.

Table 11–1 *ojvmtc Argument Summary*

Argument	Description
<code>-classpath</code>	Uses the specified JARs and classes for the closure set.
<code>-bootclasspath</code>	Uses the specified classes for closure, but does not include them in the closure set.
<code>-server connect_string</code>	Connects to the server using visible classes in the same manner as <code>-bootclasspath</code> .

Table 11–1 (Cont.) ojvmtc Argument Summary

Argument	Description
connect_string thin OCI	<p>Connects to the server using thin or Oracle Call Interface (OCI) specific driver.</p> <p>If you use thin driver, the syntax is as follows:</p> <pre>thin:user/passwd@host:port:sid</pre> <p>If you use OCI driver, the syntax is as follows:</p> <pre>oci:user/passwd@host:port:sid oci:user/passwd@tnsname oci:user/passwd@(connect descriptor)</pre>
-jar jar_name	Writes each class of the closure set to a JAR and generates stubs for missing classes
-list	Lists the missing classes.

The loadjava Tool

The `loadjava` tool creates schema objects from files and loads them into a schema. Schema objects can be created from Java source, class, and data files. The `loadjava` tool can also create schema objects from SQLJ files.

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

- `CREATE PROCEDURE` and `CREATE TABLE` privileges to load into your schema.
- `CREATE ANY PROCEDURE` and `CREATE ANY TABLE` privileges to load into another schema.
- `oracle.aurora.security.JServerPermission.loadLibraryInClass.className`.

You can run the `loadjava` tool either from the command line or by using the `loadjava` method contained in the `DBMS_JAVA` class. To run the tool from within your Java application, do the following:

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

The options are the same as those that can be specified on the command line with the `loadjava` tool. Separate each option with a space. Do not separate the options with a comma. The only exception for this is the `-resolver` option, which contains spaces. For `-resolver`, specify all other options in the first input parameter and the `-resolver` options in the second parameter, as follows:

```
call dbms_java.loadjava('..options...', 'resolver_options');
```

Do not specify the `-thin`, `-oci`, `-user`, and `-password` options, because they relate to the database connection for the `loadjava` command-line tool. The output is directed to `stderr`. Set `serveroutput` on, and call `dbms_java.set_output`, as appropriate.

Note: The `loadjava` tool is located in the `bin` directory under `$ORACLE_HOME`.

Just before the `loadjava` tool exits, it checks whether the processing was successful. All failures are summarized preceded by the following header:

The following operations failed

Some conditions, such as losing the connection to the database, cause the `loadjava` tool to terminate prematurely. These errors are displayed with the following syntax:

exiting: *error_reason*

This section covers the following:

- [Syntax](#)
- [Argument Summary](#)
- [Argument Details](#)

Syntax

The syntax of the `loadjava` tool command is as follows:

```
loadjava {-user | -u} user/[password][@database] [options]
file.java | file.class | file.jar | file.zip |
file.sqlj | resourcefile | URL...
[-casesensitivepub]
[-cleargrants]
[-debug]
[-d | -definer]
[-dirprefix prefix]
[-e | -encoding encoding_scheme]
[-fileout file]
[-f | -force]
[-genmissing]
[-genmissingjar jar_file]
[-g | -grant user [, user]...]
[-help]
[-jarasresource]
[-noaction]
[-norecursivejars]
[-nosynonym]
[-nousage]
[-noverify]
[-o | -oci | oci8]
[-optionfile file]
[-optiontable table_name]
[-publish package]
[-pubmain number]
[-recursivejars]
[-r | -resolve]
[-R | -resolver "resolver_spec"]
[-resolveonly]
[-S | -schema schema]
[-stdout]
[-stoponerror]
[-s | -synonym]
[-tableschema schema]
[-t | -thin]
[-unresolvedok]
[-v | -verbose]
[-jarsasdbobjects]
[-prependjarnames]
```

[-nativecompile]

Argument Summary

Table 11–2 summarizes the `loadjava` tool command arguments. If you run the `loadjava` tool multiple times specifying the same files and different options, then the options specified in the most recent invocation hold. However, there are two exceptions to this, as follows:

- If the `loadjava` tool does not load a file because it matches a digest table entry, then most options on the command line have no effect on the schema object. The exceptions are `-grant` and `-resolve`, which always take effect. You must use the `-force` option to direct the `loadjava` tool to skip the digest table lookup.
- The `-grant` option is cumulative. Every user specified in every invocation of the `loadjava` tool for a given class in a given schema has the `EXECUTE` privilege.

Table 11–2 *loadjava Argument Summary*

Argument	Description
<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 name arguments.
<code>-proxy host:port</code>	<p>If you do not have physical access to the server host or the <code>loadjava</code> client for loading classes, resources, and Java source, then you can use an HTTP URL with the <code>loadjava</code> tool to specify the JAR, class file, or resource and load the class from a remote server. <code>host</code> is the host name or address and <code>port</code> is the port the proxy server is using. The URL implementation must be such that the <code>loadjava</code> tool can determine the type of file to load, that is, JAR, class, resource, or Java source. For example:</p> <pre>loadjava -u scott -r -v -proxy proxy_server:1020 http://my.server.com/this/is /the/path/my.jar Password: password</pre> <p>When the URL support is used inside the server, you should have proper Java permissions to access to the remote source. The URL support also includes <code>ftp:</code> and <code>file:</code> URLs.</p>
<code>-casesensitivepub</code>	Publishing will create case-sensitive names. Unless the names are already all uppercase, it will usually require quoting the names in PL/SQL.
<code>-cleargrants</code>	<p>The <code>-grant</code> option causes the <code>loadjava</code> tool to grant <code>EXECUTE</code> privileges to classes, sources, and resources. However, it does not cause it to revoke any privileges. If <code>-cleargrants</code> is specified, then the <code>loadjava</code> tool will revoke any existing grants of <code>execute</code> privilege before it grants <code>execute</code> privilege to the users and roles specified by the <code>-grant</code> operand. For example, if the intent is to have <code>execute</code> privilege granted to only <code>SCOTT</code>, then the proper options are:</p> <pre>-grant SCOTT -cleargrants</pre>
<code>-debug</code>	Turns on SQL logging.
<code>-definer</code>	By default, class schema objects run with the privileges of their invoker. This option confers <code>definer</code> privileges upon classes instead. This option is conceptually similar to the UNIX <code>setuid</code> facility.

Table 11–2 (Cont.) loadjava Argument Summary

Argument	Description
<code>-dirprefix <i>prefix</i></code>	For any files or JAR entries that start with <i>prefix</i> , this <i>prefix</i> will be deleted from the name before the name of the schema object is determined. For classes and sources, the name of the schema object is determined by their contents. Therefore, this option will only have an effect for resources.
<code>-encoding</code>	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 the <code>JAVA\$OPTIONS</code> table, then the encoding is assumed to be the value returned by: <code>System.getProperty("file.encoding");</code> This option is relevant only when loading a source file.
<code>-fileout <i>file</i></code>	Displays all message to the designated file.
<code>-force</code>	Forces files to be loaded, even if they match digest table entries.
<code>-genmissing</code>	Determines what classes and methods are referred to by the classes that the <code>loadjava</code> tool is asked to process. Any classes not found in the database or file arguments are called missing classes. This option generates dummy definitions for missing classes containing all the referred methods. It then loads the generated classes into the database. This processing happens before the class resolution. Because detecting references from source is more difficult than detecting references from class files, and because source is not generally used for distributing libraries, the <code>loadjava</code> tool will not attempt to do this processing for source files. The schema in which the missing classes are loaded will be the one specified by the <code>-user</code> option, even when referring classes are created in some other schema. The created classes will be flagged so that tools can recognize them. In particular, this is needed, so that the verifier can recognize the generated classes.
<code>-genmissingjar <i>jar_file</i></code>	This option performs the same actions as <code>-genmissing</code> . In addition, it creates a JAR file, <i>jar_file</i> , that contains the definitions of any generated classes.
<code>-grant</code>	Grants the <code>EXECUTE</code> privilege on loaded classes to the listed users. Any number and combination of user names can be specified, separated by commas, but not spaces. Granting the <code>EXECUTE</code> privilege on an object in another schema requires that the original <code>CREATE PROCEDURE</code> privilege was granted with the <code>WITH GRANT</code> options. Note: <ul style="list-style-type: none"> ▪ <code>-grant</code> is a cumulative option. Users are added to the list of those with the <code>EXECUTE</code> privilege. To remove privileges, use the <code>-cleargrants</code> option. ▪ The schema name should be used in uppercase.
<code>-help</code>	Displays usage message on how to use the <code>loadjava</code> tool and its options.
<code>-jarasresource</code>	Instead of unpacking the JAR file and loading each class within it, loads the whole JAR file into the schema as a resource.

Table 11–2 (Cont.) loadjava Argument Summary

Argument	Description
<code>-noaction</code>	Take no action on the files. Actions include creating the schema objects, granting execute permissions, and so on. The typical use is within an option file to suppress creation of specific classes in a JAR. When used on the command line, unless overridden in the option file, it will cause the <code>loadjava</code> tool to ignore all files. Except that JAR files will still be examined to determine if they contain a <code>META-INF/loadjava-options</code> entry. If so, then the option file is processed. The <code>-action</code> option in the option file will override the <code>-noaction</code> option specified on the command line.
<code>-norecursivejars</code>	Treat JAR files contained in other JAR files as resources. This is the default behavior. This option is used to override the <code>-recursivejars</code> option.
<code>-nosynonym</code>	Do not create a public synonym for the classes. This is the default behavior. This overrides the <code>-synonym</code> option.
<code>-nousage</code>	Suppresses the usage message that is given if either no option is specified or if the <code>-help</code> option is specified.
<code>-noverify</code>	Causes the classes to be loaded without bytecode verification. <code>oracle.aurora.security.JServerPermission(Verifier)</code> must be granted to use this option. To be effective, this option must be used in conjunction with <code>-resolve</code> .
<code>-oci</code> <code>-oci8</code>	Directs the <code>loadjava</code> tool to communicate with the database using the JDBC Oracle Call Interface (OCI) driver. <code>-oci</code> and <code>-thin</code> are mutually exclusive. If neither is specified, then <code>-oci</code> is used by default. Choosing <code>-oci</code> implies the syntax of the <code>-user</code> value. You do not need to provide the URL.
<code>-optionfile file</code>	A file can be provided with <code>loadjava</code> options.
<code>-optiontable tablename</code>	This option works like <code>-optionfile</code> , except that the source for the patterns and options is a SQL table rather than a file.
<code>-publish package</code>	The <code>package</code> is created or replaced by the <code>loadjava</code> tool. Wrappers for the eligible methods will be defined in this package. Through the use of option files, a single invocation of the <code>loadjava</code> tool can be instructed to create more than one package. Each package will undergo the same name transformations as the methods.
<code>-pubmain number</code>	A special case applied to methods with a single argument, which is of type <code>java.lang.String[]</code> . Multiple variants of the SQL procedure or function will be created, each of which takes a different number of arguments of type <code>VARCHAR</code> . In particular, variants are created taking all arguments up to and including <code>number</code> . The default value is 3. This option applies to <code>main</code> , as well as any method that has exactly one argument of type <code>java.lang.String[]</code> .
<code>-recursivejars</code>	Usually, if the <code>loadjava</code> tool encounters an entry in a JAR with a <code>.jar</code> extension, it will load the entry as a resource. If this option is specified, then the <code>loadjava</code> tool will process contained JAR files as if they were top-level JAR files. That is, it will read their entries and load classes, sources, and resources.
<code>-resolve</code>	Compiles, if necessary, and resolves external references in classes after all classes on the command line have been loaded. If you do not specify the <code>-resolve</code> option, the <code>loadjava</code> tool loads files, but does not compile or resolve them.

Table 11–2 (Cont.) loadjava Argument Summary

Argument	Description
<code>-resolver</code>	Specifies an explicit resolver specification, which is bound to the newly loaded classes. If <code>-resolver</code> is not specified, then the default resolver specification, which includes current user's schema and <code>PUBLIC</code> , is used.
<code>-resolveonly</code>	Causes the <code>loadjava</code> tool to skip the initial creation step. It will still perform grants, resolves, create synonyms, and so on.
<code>-schema</code>	Designates the schema where schema objects are created. If not specified, then the <code>-user</code> schema is used. To create a schema object in a schema that is not your own, you must have the following privileges: <ul style="list-style-type: none"> ■ <code>CREATE TABLE</code> or <code>CREATE ANY TABLE</code> ■ <code>CREATE INDEX</code> or <code>CREATE ANY INDEX</code> ■ <code>SELECT ANY TABLE</code> ■ <code>UPDATE ANY TABLE</code> ■ <code>INSERT ANY TABLE</code> ■ <code>DELETE ANY TABLE</code> ■ <code>CREATE PROCEDURE</code> or <code>CREATE ANY PROCEDURE</code> ■ <code>ALTER ANY PROCEDURE</code> Finally, you must have the <code>JServerPermission loadLibraryInClass</code> for the class. <p>Note: The above-mentioned privileges allow the grantee to create and manipulate tables in any schema except the <code>SYS</code> schema. For security reasons, Oracle recommends that you use these settings only with great caution.</p>
<code>-stdout</code>	Causes the output to be directed to <code>stdout</code> , rather than to <code>stderr</code> .
<code>-stoponerror</code>	Usually, if an error occurs while the <code>loadjava</code> tool is processing files, it will issue a message and continue to process other classes. This option stops when an error occurs. In addition, it reports all errors that apply to Java objects and are contained in the <code>USER_ERROR</code> table of the schema in which classes are being loaded. Except that it does not report <code>ORA-29524</code> errors. These are errors that are generated when a class cannot be resolved, because a referred class could not be resolved. Therefore, these errors are a secondary effect of whatever caused a referred class to be unresolved.
<code>-synonym</code>	Creates a <code>PUBLIC</code> synonym for loaded classes making them accessible outside the schema into which they are loaded. To specify this option, you must have the <code>CREATE PUBLIC SYNONYM</code> privilege. If <code>-synonym</code> is specified for source files, then the classes compiled from the source files are treated as if they had been loaded with <code>-synonym</code> .
<code>-tableschemaschema</code>	Creates the <code>loadjava</code> tool internal tables within the specified schema, rather than in the Java file destination schema.
<code>-thin</code>	Directs the <code>loadjava</code> tool to communicate with the database using the JDBC Thin driver. 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.
<code>-unresolvedok</code>	When combined with <code>-resolve</code> , will ignore unresolved errors.
<code>-user</code>	Specifies a user name, password, and database connection string. The files will be loaded into this database instance.

Table 11–2 (Cont.) loadjava Argument Summary

Argument	Description
<code>-verbose</code>	Directs the <code>loadjava</code> tool to display detailed status messages while running. Use the <code>-verbose</code> option to learn when the <code>loadjava</code> tool does not load a file, because it matches a digest table entry.
<code>-jarsasdbobjects</code>	Indicates that JARs processed by the current <code>loadjava</code> tool command are to be stored in the database as database resident JARs.
<code>-prependjarnames</code>	Is used with the <code>-jarsasdbobjects</code> option. This option enables classes with the same names coming from different JARs to coexist in the same schema. It does this by prefixing a version of the name of the JAR to the class name to produce a unique name for the database object.

Argument Details

This section describes the details of some of the `loadjava` tool arguments whose behavior is more complex than the summary descriptions contained in [Table 11–2](#).

File Names

You can specify as many `.class`, `.java`, `.sqlj`, `.jar`, `.zip`, and resource files as you want and in any order. If you specify a JAR or ZIP file, then the `loadjava` tool processes the files in the JAR or ZIP. There is no JAR or ZIP schema object. If a JAR or ZIP contains another JAR or ZIP, the `loadjava` tool does not process them.

The best way to load files is to put them in a JAR or ZIP and then load the archive. Loading archives avoids the resource schema object naming complications. If you have a JAR or ZIP that works with the Java Development Kit (JDK), then you can be sure that loading it with the `loadjava` tool will also work, without having to learn anything about resource schema object naming.

Schema object names are different from file names, and the `loadjava` tool names different types of schema objects differently. Because class files are self-identifying, the mapping of class file names to schema object names done by the `loadjava` tool is invisible to developers. Source file name mapping is also invisible to developers. The `loadjava` tool 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. The `loadjava` tool generates Java resource schema object names from the literal names you supply as arguments. Because classes use resource schema objects and the correct specification of resources is not always intuitive, it is important that you specify resource file names correctly on the command line.

The perfect way to load individual resource files correctly is to run the `loadjava` tool from the top of the package tree and specify resource file names relative to that directory.

Note: The top of the package tree is the directory you would name in a CLASSPATH.

If you do not want to follow this rule, then observe the details of resource file naming that follow. When you load a resource file, the `loadjava` tool generates the resource schema object name from the resource file name, as literally specified on the command line. For example, if you type:

```
% cd /home/scott/javastuff
```



```
% loadjava options alpha/beta/x.properties
% loadjava options /home/scott/javastuff/alpha/beta/x.properties
```

Although you have specified the same file with a relative and an absolute path name, the `loadjava` tool creates two schema objects, `alpha/beta/x.properties` and `ROOT/home/scott/javastuff/alpha/beta/x.properties`. The name of the resource schema object is generated from the file name as entered.

Classes can refer to resource files relatively or absolutely. To ensure that the `loadjava` tool and the class loader use the same name for a schema object, enter the name on the command line, which the class passes to `getResource()` or `getResourceAsString()`.

Instead of remembering whether classes use relative or absolute resource names and changing directories so that you can enter the correct name on the command line, you can load resource files in a JAR, as follows:

```
% cd /home/scott/javastuff
% jar -cf alpharesources.jar alpha/*.properties
% loadjava options alpharesources.jar
```

To simplify the process further, place both the class and resource files in a JAR, which makes the following invocations equivalent:

```
% loadjava options alpha.jar
% loadjava options /home/scott/javastuff/alpha.jar
```

The preceding `loadjava` tool commands imply that you can use any path name to load the contents of a JAR file. Even if you run the redundant commands, the `loadjava` tool would realize from the digest table that it need not load the files twice. This implies that reloading JAR files is not as time-consuming as it might seem, even when few files have changed between the different invocations of the `loadjava` tool.

definer

```
{-definer | -d}
```

This option is identical to the `definer` rights in stored procedures and is conceptually similar to the UNIX `setuid` facility. However, you can apply the `-definer` option to individual classes, in contrast to `setuid`, which applies to a complete program. Moreover, different definers may have different privileges. Because an application can consist of many classes, you must apply `-definer` with care to achieve the desired results. That is, classes run with the privileges they need, but no more.

See Also: ["Controlling the Current User"](#) on page 2-16

noverify

```
[-noverify]
```

This option causes the classes to be loaded without bytecode verification. `oracle.aurora.security.JServerPermission(Verifier)` must be granted to run this option. Also, this option must be used in conjunction with `-resolve`.

The verifier ensures that incorrectly formed Java binaries cannot be loaded for running on the server. If you know that the JAR or classes you are loading are valid, then the use of this option will speed up the process associated with the `loadjava` tool. Some Oracle Database-specific optimizations for interpreted performance are put in place during the verification process. Therefore, the interpreted performance of your application may be adversely affected by using this option.

optionfile

```
[-optionfile <file>]
```

This option enables you to specify a file with different options that you can specify with the `loadjava` tool. This file is read and processed by the `loadjava` tool before any other `loadjava` tool options are processed. The file can contain one or more lines, each of which contains a pattern and a sequence of options. Each line must be terminated by a newline character (`\n`).

For each file or JAR entry that is processed by the `loadjava` tool, the long name of the schema object that is going to be created is checked against the patterns. Patterns can end in a wildcard (`*`) to indicate an arbitrary sequence of characters, or they must match the name exactly.

Options to be applied to matching Java schema objects are supplied on the rest of the line. Options are appended to the command-line options, they do not replace them. In case more than one line matches a name, the matching rows are sorted by length of pattern, with the shortest first, and the options from each row are appended. In general, the `loadjava` tool options are not cumulative. Rather, later options override earlier ones. This means that an option specified on a line with a longer pattern will override a line with a shorter pattern.

This file is parsed by a `java.io.StreamTokenizer`.

You can use Java comments in this file. A line comment begins with a `#`. Empty lines are ignored. The quote character is a double quote (`"`). That is, options containing spaces should be surrounded by double quotes. Certain options, such as `-user` and `-verbose`, affect the overall processing of the `loadjava` tool and not the actions performed for individual Java schema objects. Such options are ignored if they appear in an option file.

To help package applications, the `loadjava` tool looks for the `META-INF/loadjava-options` entry in each JAR it processes. If it finds such an entry, then it treats it as an options file that is applied for all other entries in the JAR file. However, the `loadjava` tool does some processing on entries in the order in which they occur in the JAR.

If the `loadjava` tool has partially processed entities before it processes `META-INF/loadjava-options`, then it attempts to patch up the schema object to conform to the applicable options. For example, the `loadjava` tool alters classes that were created with invoker rights when they should have been created with definer rights. The fix for `-noaction` is to drop the created schema object. This yields the correct effect, except that if a schema object existed before the `loadjava` tool started, then it would have been dropped.

optiontable

```
[-optiontable table_name]
```

This option enables you to specify the properties of classes persistently. No mechanism is provided for loading the table. The table name must contain three character columns, `PATTERN`, `OPTION`, and `VALUE`. The value of `PATTERN` is interpreted in the same way as a pattern in an option file. The other two columns are the same as the corresponding command-line options and take an operand. Suppose, you create a table `FOO` with the following command:

```
create table foo (pattern varchar2(2000), option varchar2(2000), value
varchar2(2000));
```

Then, you can use the `optiontable` option in the following way:

```
loadjava -optionable foo myjar.jar
```

For options that do not take an operand, the VALUE column should be NULL. The rows are processed in the same way as the lines of an option file are processed. To determine the options for a given schema object, the rows are examined and for any match the option is appended to the list of options. If two rows have the same pattern and contradictory options, such as `-synonym` and `-nosynonym`, then it is unspecified which will prevail. If two rows have the same pattern and option columns, then it is unspecified which VALUE will prevail.

publish

```
[-publish <package>]
[-pubmain <number>]
```

The publishing options cause the `loadjava` tool to create PL/SQL wrappers for methods contained in the processed classes. Typically, a user wants to publish wrappers for only a few classes in a JAR. These options are most useful when specified in an option file.

To be eligible for publication, the method must satisfy the following:

- It must be a member of a `public` class.
- It must be declared `public` and `static`.
- The method signature should satisfy the following rules so that it can be mapped:
 - Java arithmetic types for arguments and return values are mapped to `NUMBER`.
 - `char` as an argument and return type is mapped to `VARCHAR`.
 - `java.lang.String` as an argument and return type is mapped to `VARCHAR`.
 - If the only argument of the method has type `java.lang.String`, special rules apply, as listed in the `-pubmain` option description.
 - If the return type is `void`, then a procedure is created.
 - If the return type is an arithmetic, `char`, or `java.lang.String` type, then a function is created.

Methods that take arguments or return types that are not covered by the preceding rules are not eligible. No provision is made for `OUT` and `IN OUT` SQL arguments, `OBJECT` types, and many other SQL features.

resolve

```
{-resolve | -r}
```

Use `-resolve` to force the `loadjava` tool to compile and resolve a class that has previously been loaded. It is not necessary to specify `-force`, because resolution is performed after, and independent of, loading.

resolver

```
{-resolver | -R} resolver_specification
```

This option associates an explicit resolver specification with the class schema objects that the `loadjava` tool creates or replaces.

A resolver specification consists of one or more items, each of which consists of a name specification and a schema specification expressed in the following syntax:

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

A name specification is similar to a name in an `import` statement. It can be a fully qualified Java class name or a package name whose final element is the wildcard character asterisk (*) or simply an asterisk (*). However, the elements of a name specification must be separated by slashes (/), not periods (.). For example, the name specification `a/b/*` matches all classes whose names begin with `a.b`. The special name `*` matches all class names.

A schema specification can be a schema name or the wildcard character dash (-). The wildcard does not identify a schema, but directs the resolve operation not to mark a class invalid, because a reference to a matching name cannot be resolved. Use dash (-) when you must test a class that refers to a class you cannot or do not want to load. For example, GUI classes that a class refers to but does not call, because when run in the server there is no GUI.

When looking for a schema object whose name matches the name specification, the resolution operation looks in the schema named by the partner schema specification.

The resolution operation searches schemas in the order in which the resolver specification lists them. For example,

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

This implies that search for any reference first in `SCOTT` and then in `PUBLIC`. If a reference is not resolved, then mark the referring class invalid and display an error message.

Consider the following example:

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

This implies that search for any reference first in `SCOTT` and then in `PUBLIC`. If the reference is to a class in the package `my.gui` and is not found, then mark the referring class valid and do not display an error. If the reference is not to a class in `my.gui` and is not found, then mark the referring class invalid and produce an error message.

user

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

By default, the `loadjava` tool loads into the logged in schema specified by the `-user` option. You use the `-schema` option to specify a different schema to load into. This does not require you to log in to that schema, but does require that you have sufficient permissions to alter the schema.

The permissible forms of `@database_url` depend on whether you specify `-oci` or `-thin`, as described:

- `-oci:@database_url` is optional. If you do not specify, then the `loadjava` tool uses the user's default database. If specified, `database_url` can be a TNS name or an Oracle Net Services name-value list.
- `-thin:@database_url` is required. The format is `host: lport: SID`.

where:

- `host` is the name of the computer running the database.
- `lport` is the listener port that has been configured to listen for Oracle Net Services connections. In a default installation, it is 5521.
- `SID` is the database instance identifier. In a default installation, it is `ORCL`.

The following are examples of the `loadjava` tool commands:

- Connect to the default database with the default OCI driver, load the files in a JAR into the `TEST` schema, and then resolve them:

```
loadjava -u joe -resolve -schema TEST ServerObjects.jar
Password: password
```

- Connect with the JDBC Thin driver, load a class and a resource file, and resolve each class:

```
loadjava -thin -u SCOTT@dbhost:5521:orcl \
  -resolve alpha.class beta.props
Password: password
```

- Add Betty and Bob to the users who can run `alpha.class`:

```
loadjava -thin -schema test -u SCOTT@localhost:5521:orcl \
  -grant BETTY,BOB alpha.class
Password: password
```

jarsasdbobjects

This option indicates that JARs processed by the current `loadjava` tool are to be stored in the database along with the classes they contain, and knowledge of the association between the classes and the JAR is to be retained in the database. In other words, this argument indicates that the JARs processed by the current `loadjava` tool are to be stored in the database as database resident JARs.

prependjarnames

This option is used with the `-jarsasdbobjects` option. This option enables classes with the same names coming from different JARs to coexist in the same schema.

The dropjava Tool

The `dropjava` tool is the converse of the `loadjava` tool. It transforms command-line file names and JAR or ZIP file contents to schema object names, drops the schema objects, and deletes their corresponding digest table rows. You can enter `.java`, `.class`, `.sqlj`, `.ser`, `.zip`, `.jar`, and resource file names on the command line and in any order.

Alternatively, you can specify a schema object name directly to the `dropjava` tool. A command-line argument that does not end in `.jar`, `.zip`, `.class`, `.java`, or `.sqlj` is presumed to be a schema object name. If you specify a schema object name that applies to multiple schema objects, then all will be removed.

Dropping a class invalidates classes that depend on it, recursively cascading upwards. Dropping a source drops classes derived from it.

Note: You must remove Java schema objects in the same way that you first loaded them. If you load a `.sqlj` source file and translate it in the server, then you must run the `dropjava` tool on the same source file. If you translate on a client and load classes and resources directly, then run the `dropjava` tool on the same classes and resources.

You can run the `dropjava` tool either from the command line or by using the `dropjava` method in the `DBMS_JAVA` class. To run the `dropjava` tool from within your Java application, use the following command:

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

The options are the same as specified on the command line. Separate each option with a space. Do not separate the options using commas. The only exception to this is the `-resolver` option. The connection is always made to the current session. Therefore, you cannot specify another user name through the `-user` option.

For `-resolver`, you should specify all other options first, a comma (,), then the `-resolver` option with its definition. Do not specify the `-thin`, `-oci`, `-user`, and `-password` options, because they relate to the database connection for the `loadjava` tool. The output is directed to `stderr`. Set `serveroutput` on and call `dbms_java.set_output`, as appropriate.

This section covers the following topics:

- [Syntax](#)
- [Argument Summary](#)
- [Argument Details](#)
- [List Based Deletion](#)
- [Dropping Resources](#)

Syntax

The syntax of the `dropjava` tool command is:

```
dropjava [options] {file.java | file.class | file.sqlj |
file.jar | file.zip | resourcefile} ...
  -u | -user user/[password] [@database]
  [-genmissingjar JARfile]
  [-jarasresource]
  [-o | -oci | -oci8]
  [-optionfile file]
  [-optiontable table_name]
  [-S | -schema schema]
  [-stdout]
  [-s | -synonym]
  [-t | -thin]
  [-v | -verbose]
```

Argument Summary

Table 11–3 summarizes the `dropjava` tool arguments.

Table 11–3 *dropjava* Argument Summary

Argument	Description
<code>-user</code>	Specifies a user name, password, and optional database connection string. The files will be dropped from this database instance.
<i>filenames</i>	Specifies 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.
<code>-genmissingjar</code> <i>JARfile</i>	Treats the operand of this option as a file to be processed.

Table 11-3 (Cont.) dropjava Argument Summary

Argument	Description
<code>-jarasresource</code>	Drops the whole JAR file, which was previously loaded as a resource.
<code>-oci</code> <code>-oci8</code>	Directs the <code>dropjava</code> tool to connect with the database using the OCI JDBC driver. The <code>-oci</code> and the <code>-thin</code> options are mutually exclusive. If neither is specified, then the <code>-oci</code> option is used by default. Choosing the <code>-oci</code> option implies the form of the <code>-user</code> value.
<code>-optionfile file</code>	Has the same usage as for the <code>loadjava</code> tool.
<code>-optiontable table_name</code>	Has the same usage as for <code>loadjava</code> .
<code>-schema schema</code>	Designates the schema from which schema objects are dropped. If not specified, then the <code>logon</code> 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.
<code>-stdout</code>	Causes the output to be directed to <code>stdout</code> , rather than to <code>stderr</code> .
<code>-synonym</code>	Drops a <code>PUBLIC</code> synonym that was created with the <code>loadjava</code> tool.
<code>-thin</code>	Directs the <code>dropjava</code> tool to communicate with the database using the JDBC Thin driver. Choosing the <code>-thin</code> option implies the form of the <code>-user</code> value.
<code>-verbose</code>	Directs the <code>dropjava</code> tool to emit detailed status messages while running.
<code>-list</code>	Drops the classes, Java source, or resources listed on the command line without them being present on the client machine or server machine.
<code>-listfile</code>	Reads a file and drops the classes, Java source, or resources listed in the file without them being present on the client machine or server machine. The file contains the internal representation of the complete class, Java source, or resource name one per line.

Argument Details

This section describes a few of the `dropjava` tool arguments, which are complex.

File Names

The `dropjava` tool interprets most file names as the `loadjava` tool does:

- `.class` files
 - Finds the class name in the file and drops the corresponding schema object.
- `.java` and `.sqlj` files
 - Finds the first class name in the file and drops the corresponding schema object.
- `.jar` and `.zip` files
 - Processes the archived file names as if they had been entered on the command line.

If a file name has another extension or no extension, then the `dropjava` tool interprets the file name as a schema object name and drops all source, class, and resource objects that match the name.

If the `dropjava` tool encounters a file name that does not match a schema object, then it displays a message and processes the remaining file names.

user

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

The permissible forms of `@database` depend on whether you specify `-oci` or `-thin`:

- `-oci:@database` is optional. If you do not specify, then the `dropjava` tool uses the user's default database. If specified, then `database` can be a TNS name or an Oracle Net Services name-value list.
- `-thin:@database` is required. The format is `host:listener:SID`.

where:

- `host` is the name of the computer running the database.
- `listener` is the listener port that has been configured to listen for Oracle Net Services connections. In a default installation, it is 5521.
- `SID` is the database instance identifier. In a default installation, it is ORCL.

The following are examples of the `dropjava` tool command:

- Drop all schema objects in the `TEST` schema in the default database that were loaded from `ServerObjects.jar`:

```
dropjava -u SCOTT -schema TEST ServerObjects.jar  
Password: password
```

- Connect with the JDBC Thin driver, then drop a class and a resource file from the user's schema:

```
dropjava -thin -u SCOTT@dbhost:5521:orcl alpha.class beta.props  
Password: password
```

List Based Deletion

Earlier versions of the `dropjava` tool required that the classes, JARs, source, and resources be present on the machine, where the client or server side utility is running. The current version of `dropjava` has an option that enables you to drop classes, resources, or sources based on a list of classes, which may not exist on the client machine or the server machine. This list can be either on the command line or in a text file. For example:

```
dropjava -list -u scott -v this.is.my.class this.is.your.class  
Password: password
```

The preceding command drops the classes `this.is.my.class` and `this.is.your.class` listed on the command line without them being present on the client machine or server machine.

```
dropjava -listfile my.list -u scott -s -v  
Password: password
```

The preceding command drops classes, resources, or sources and their synonyms based on a list of classes listed in `my.list` and displays verbosely.

Dropping Resources

Care must be taken if you are removing a resource that was loaded directly into the server. This includes profiles, if you translated them on the client without using the `-ser2class` option. When dropping source or class schema objects or resource schema objects that were generated by the server-side SQLJ translator, the schema objects will be found according to the package specification in the applicable `.sqlj` source file. However, the fully qualified schema object name of a resource that was generated on the client and loaded directly into the server depends on path information in the `.jar` file or that specified on the command line at the time you loaded it. If you use a `.jar` file to load resources and use the same `.jar` file to remove resources, then there will be no problem. However, if you use the command line to load resources, then you must be careful to specify the same path information when you run the `dropjava` tool to remove the resources.

The ojvmjava Tool

The `ojvmjava` tool is an interactive interface to the session namespace of a database instance. You specify database connection arguments when you start the `ojvmjava` tool. It then presents you with a prompt to indicate that it is ready for commands.

The shell can launch an executable, that is, a class with a `static main()` method. This is done either by using the command-line interface or by calling a database resident class. If you call a database resident class, the executable must be loaded with the `loadjava` tool.

This section covers the following topics:

- [Syntax](#)
- [Argument Summary](#)
- [Example](#)
- [Functionality](#)

Syntax

The syntax of the `ojvmjava` tool command is:

```
ojvmjava {-user user[/password@database ] [options]
          [@filename]
          [-batch]
          [-c | -command command args]
          [-debug]
          [-d | -database conn_string]
          [-fileout filename]
          [-o | -oci | -oci8]
          [-oschema schema]
          [-t | -thin]
          [-version | -v]
          -runjava [server_file_system]
          -jdpw port [host]
          -verbose
```

Argument Summary

[Table 11–4](#) summarizes the `ojvmjava` tool arguments.

Table 11–4 *ojvmjava Argument Summary*

Argument	Description
-user -u	Specifies user name for connecting to the database. This name is not case-sensitive. The name will always be converted to uppercase. If you provide the database information, then the default syntax used is OCI. You can also specify the default database.
-password -p	Specifies the password for connecting to the database.
@ <i>filename</i>	Specifies a script file that contains the <code>ojvmjava</code> tool commands to be run.
-batch	Disables all messages displayed to the screen. No help messages or prompts will be displayed. Only responses to commands entered are displayed.
-command	Runs the desired command. If you do not want to run the <code>ojvmjava</code> tool in interpretive mode, but only want to run a single command, then run it with this option followed by a string that contains the command and the arguments. Once the command runs, the <code>ojvmjava</code> tool exits.
-debug	Displays debugging information.
-d -database <i>conn_string</i>	Provides a database connection string.
-fileout <i>file</i>	Redirects output to the provided file.
-o -oci -oci8	Uses the JDBC OCI driver. The OCI driver is the default. This flag specifies the syntax used in either the <code>@database</code> or <code>-database</code> option.
-o schema <i>schema</i>	Uses this schema for class lookup.
-t -thin	Specifies that the database syntax used is for the JDBC Thin driver. The database connection string must be of the form <code>host:port:SID</code> or an Oracle Net Services name-value list.
-verbose	Displays the connection information.
-version	Shows the version.
-runjava	Uses <code>DBMS_JAVA.runjava</code> when executing Java commands. With no argument, interprets <code>-classpath</code> as referring to the client file system. With argument <code>server_file_system</code> interprets <code>-classpath</code> as referring to the file system on which Oracle server is running, as <code>DBMS_JAVA.runjava</code> typically does.
-jdpw	Makes the connection listen for a debugger connection on the indicated port and host. The default value of host is <code>localhost</code> .

Example

Open a shell on the session namespace of the database `orcl` on listener port 2481 on the host `dbserver`, as follows:

```
ojvmjava -thin -user SCOTT@dbserver:2481:orcl
Password: password
```

Functionality

The `ojvmjava` tool commands span several different types of functionality, which are grouped as follows:

- [ojvmjava Tool Command-Line Options](#)
- [Shell Commands](#)

ojvmjava Tool Command-Line Options

This section describes the options for the `ojvmjava` tool command.

The ojvmjava Tool Output Redirection

You can direct any output generated by the `ojvmjava` tool to a file by appending `&>filename` at the end of the command options. The following command directs all output to the `listDir` file:

```
ls -lR &>/tmp/listDir
```

Scripting the ojvmjava Tool Commands in the @filename Option

This `@filename` option designates a script file that contains one or more `ojvmjava` tool commands. The script file specified is located on the client. The `ojvmjava` tool reads the file and runs all commands on the designated server. In addition, because the script file is run on the server, any interaction with the operating system in the script file, such as redirecting output to a file or running another script, will occur on the server. If you direct the `ojvmjava` tool to run another script file, then this file must exist in `$ORACLE_HOME` on the server.

Enter the `ojvmjava` tool command followed by any options and any expected input arguments.

The script file contains the `ojvmjava` tool command followed by options and input parameters. The input parameters can be passed to the `ojvmjava` tool on the command line. The `ojvmjava` tool processes all known options and passes on any other options and arguments to the script file.

To access arguments within the commands in the script file, use `&1 . . . &n` to denote the arguments. If all input parameters are passed to a single command, then you can type `&*` to denote that all input parameters are to be passed to this command.

The following shows the contents of the script file, `execShell`:

```
chmod +x SCOTT nancy /alpha/beta/gamma
chown SCOTT /alpha/beta/gamma
java hello.World &*
```

Because only two input arguments are expected, you can implement the Java command input parameters, as follows:

```
java hello.World &1 &2
```

Note: You can also supply arguments to the `-command` option in the same manner. The following shows an example:

```
ojvmjava ... -command "cd &1" contexts
```

After processing all other options, the `ojvmjava` tool passes `contexts` as argument to the `cd` command.

To run this file, do the following:

```
ojvmjava -user SCOTT -thin -database dbserver:2481:orcl \  
  @execShell alpha beta  
Password: password
```

The `ojvmjava` tool processes all options that it knows about and passes along any other input parameters to be used by the commands that exist within the script file. In this example, the parameters, `alpha` and `beta`, are passed to the `java` command in the script file. The actual command is run as follows:

```
java hello.World alpha beta
```

You can add any comments in your script file using hash (`#`). Comments are ignored by the `ojvmjava` tool. For example:

```
#this whole line is ignored by ojvmjava
```

-runjava

This option controls whether or not the `ojvmjava` tool shell command Java runs executable classes using the command-line interface or database resident classes. When the `-runjava` option is present the command-line interface is used. Otherwise, the executable must be a database resident class that was previously loaded with the `loadjava` tool. Using the optional argument `server_file_system` means that the `-classpath` terms are on the file system of the machine running Oracle server. Otherwise, they are interpreted as being on the file system of the machine running the `ojvmjava` tool.

See Also: ["Using Command-Line Interface"](#) on page 3-9

-jdwp

This option specifies a debugger connection to listen for when the shell command `java` is used to run an executable. This allows for debugging the executable. The arguments specify the port and host. The default value of the host argument is `localhost`. These are used to execute a call to `DBMS_DEBUG_JDWP.CONNECT_TCP` from the RDBMS session, in which the executable is run.

Running sess_sh Within Applications

You can run `sess_sh` commands from within a Java or PL/SQL application using the following commands:

Application Type	Command and Description
PL/SQL applications	<p><code>dbms_namespace.shell(in command VARCHAR2)</code></p> <p>Your application can run individual commands on a unique session instance. The state of the shell is preserved between different calls of <code>sess_sh</code> within the same session. The following examples run the <code>cd</code> command of the <code>sess_sh</code> tool within a PL/SQL application:</p> <pre>dbms_namespace.shell('cd /webdomains');</pre> <pre>dbms_namespace.shell('ls &> /tmp/test');</pre> <p>To reset the state of the shell instance, run the <code>exit</code> command, as follows:</p> <pre>dbms_namespace.shell('exit');</pre>
Java applications	<p>Instantiates <code>oracle.aurora.namespace.shell.Shell</code> within a Java server object. After creation, you must initialize the <code>Shell</code> object using its <code>initialize</code> method. Once initialized, you can run <code>sess_sh</code> commands, as follows:</p> <pre>String commands="cd /webdomains\nls -l"; StringReader commandReader = new StringReader(commands); Shell sh = new oracle.aurora.namespace.shell.Shell(); try { sh.initialize(); sh.invoke(new BufferedReader(commandReader), false); } catch (ToolsException te) { //Error executing the commands }</pre>

Shell Commands

This section describes the following commands available within the `ojvmjava` shell:

- [echo](#)
- [exit](#)
- [help](#)
- [java](#)
- [version](#)
- [whoami](#)
- [connect](#)
- [runjava](#)
- [jdwp](#)

Note: An error is reported if you enter an unsupported command.

Table 11–5 summarizes the commands that share one or more common options, which are summarized in Table 11–5:

Table 11–5 *ojvmjava Command Common Options*

Option	Description
-describe -d	Summarizes the operation of the tool.
-help -h	Summarizes the syntax of the tool.
-version	Shows the version.

echo

This command displays to `stdout` exactly what is indicated. This is used mostly in script files.

The syntax is as follows:

```
echo [echo_string] [args]
```

echo_string is a string that contains the text you want written to the screen during the shell script invocation and *args* are input arguments from the user. For example, the following command displays out a notification:

```
echo "Adding an owner to the schema" &1
```

If the input argument is `SCOTT`, then the output would be:

```
Adding an owner to the schema SCOTT
```

exit

This command terminates `ojvmjava`. The syntax is as follows:

```
exit
```

For example, to leave a shell, use the following command:

```
$ exit
%
```

help

This command summarizes the syntax of the shell commands. You can also use the `help` command to summarize the options for a particular command. The syntax is as follows:

```
help [command]
```

java

This command is analogous to the JDK `java` command. It calls the static `main()` method of a class. It does this either by using the command-line interface or using a database resident class, depending on the setting of the `runjava` mode. In the latter case, the class must have been previously loaded with the `loadjava` tool. The command provides a convenient way to test Java code that runs in the database. In particular, the command catches exceptions and redirects the standard output and standard error of the class to the shell, which displays them as with any other command output. The destination of standard out and standard error for Java classes

that run in the database is one or more database server process trace files, which are inconvenient and may require DBA privileges to read.

See Also: ["Using Command-Line Interface"](#) on page 3-9

The syntax of the command with `runjava` mode off is:

```
java [-schema schema] class [arg1 ... argn]
```

The syntax of the command with `runjava` mode on is:

```
java [command-line options] class [arg1 ... argn]
```

where, command-line options can be any of those mentioned in [Table 3-1](#).

[Table 11-6](#) summarizes the arguments of this command.

Table 11-6 *java Argument Summary*

Argument	Description
<i>class</i>	Names the Java class schema object that is to be run.
<code>-schema</code>	Names the schema containing the class to be run. The default is the invoker's schema. The schema name is case-sensitive.
<i>arg1 ... argn</i>	Arguments to the <code>static main()</code> method of the class.

Consider the following Java file, `World.java`:

```
package hello;
public class World
{
    public World()
    {
        super();
    }

    public static void main(String[] argv)
    {
        System.out.println("Hello from Oracle Database");
        if (argv.length != 0)
            System.out.println("You supplied " + argv.length + " arguments: ");
        for (int i = 0; i < argv.length; i++)
            System.out.println(" arg[" + i + "] : " + argv[i]);
    }
}
```

You can compile, load, publish, and run the class, as follows:

```
% javac hello/World.java

% loadjava -r -user SCOTT@localhost:2481:orcl hello/World.class
Password: password
% ojvmjava -user SCOTT -database localhost:2481:orcl
Password: password
$ java hello.World alpha beta
Hello from Oracle Database
You supplied 2 arguments:
arg[0] : alpha
arg[1] : beta
```

version

This command shows the version of the `ojvmjava` tool. You can also show the version of a specified command. The syntax of this command is:

```
version [options] [command]
```

For example, you can display the version of the shell, as follows:

```
$ version
1.0
```

whoami

This command displays the user name of the user who logged in to the current session. The syntax of the command is:

```
whoami
```

connect

This command enables the client to drop the current connection and connect to different databases without having to reinvoke the `ojvmjava` tool with a different connection description.

The syntax of this command is:

```
connect [-service service] [-user user][-password password]
```

You can use this command as shown in the following examples:

```
connect -s thin@localhost:5521:orcl -u scott/tiger
connect -s oci@localhost:5521:orcl -u scott -p tiger
```

[Table 11–7](#) summarizes the arguments of this command.

Table 11–7 *connect Argument Summary*

Argument	Description
-service -s	Any valid JDBC driver URLs, namely, <code>oci @<connection descriptor></code> and <code>thin@<host:port:db></code>
-user -u	User to connect as
-password -p	Password to connect with

runjava

This command queries or modifies the `runjava` mode. The `runjava` mode determines whether or not the `java` command uses the command-line interface to run executables. The `java` command:

- Uses the command-like interface when `runjava` mode is `on`
- Uses database resident executables when `runjava` mode is `off`

See Also: ["Using Command-Line Interface"](#) on page 3-9

Using the `runjava` command with no arguments displays the current setting of `runjava` mode.

[Table 11–8](#) summarizes the arguments of this command.

Table 11–8 *runjava Argument Summary*

Argument	Description
off	Turns runjava mode off.
on	Turns runjava mode on.
server_file_system	Turns runjava mode on. Using this option means that <code>-classpath</code> terms are on the file system of the machine running Oracle server. Otherwise, they are interpreted as being on the file system of the machine running the <code>ojvmjava</code> tool.

jdwp

This command queries or modifies whether and how a debugger connection is listened for when an executable is run by the Java command.

Note: The RDBMS session, prior to starting the executable, executes a `DBMS_DEBUG_JDWP.CONNECT_TCP` call with the specified port and host. This is called **Listening**.

Using this command with no arguments displays the current setting.

[Table 11–9](#) summarizes the arguments of this command.

Table 11–9 *jdwp Argument Summary*

Argument	Description
off	Stops listening in future executables.
port	Enables listening and specifies the port to be used.
host	Enables listening and specifies the host to be used. The default value for this argument is <code>localhost</code> .

Database Web Services

This chapter provides an overview of database Web services and discusses how to call existing Web services. This chapter contains the following sections:

- [Overview of Database Web Services](#)
- [Using Oracle Database as Web Services Provider](#)
- [Using Oracle Database as Web Services Consumer](#)

Overview of Database Web Services

Web services enable application-to-application interaction over the Web, regardless of platform, language, or data formats. The key ingredients, including Extensible Markup Language (XML), Simple Object Access Protocol (SOAP), Web Services Description Language (WSDL), and Universal Description, Discovery, and Integration (UDDI), have been adopted across the entire software industry. Web services usually refer to services implemented and deployed in middle-tier application servers. However, in heterogeneous and disconnected environments, there is an increasing need to access stored procedures, as well as data and metadata, through Web services interfaces.

The Database Web services technology is a database approach to Web services. It works in the following two directions:

- Accessing database resources as a Web service
- Consuming external Web services from the database

Oracle Database can access Web services through PL/SQL packages and Java classes deployed within the database. Turning Oracle Database into a Web service provider leverages investment in Java stored procedures, PL/SQL packages, predefined SQL queries, and data manipulation language (DML). Conversely, consuming external Web services from the database, together with integration with the SQL engine, enables Enterprise Information Integration.

Using Oracle Database as Web Services Provider

Web Services use industry-standard mechanisms to provide easy access to remote content and applications, regardless of the platform and location of the provider and implementation and data format. Client applications can query and retrieve data from Oracle Database and call stored procedures using standard Web service protocols. There is no dependency on Oracle-specific database connectivity protocols. This approach is highly beneficial in heterogeneous, distributed, and disconnected environments.

You can call into the database from a Web service, using the database as a service provider. This enables you to leverage existing or new SQL, PL/SQL, Java stored procedures, or Java classes within Oracle Database. You can access and manipulate database tables from a Web service client.

This section covers the following topics:

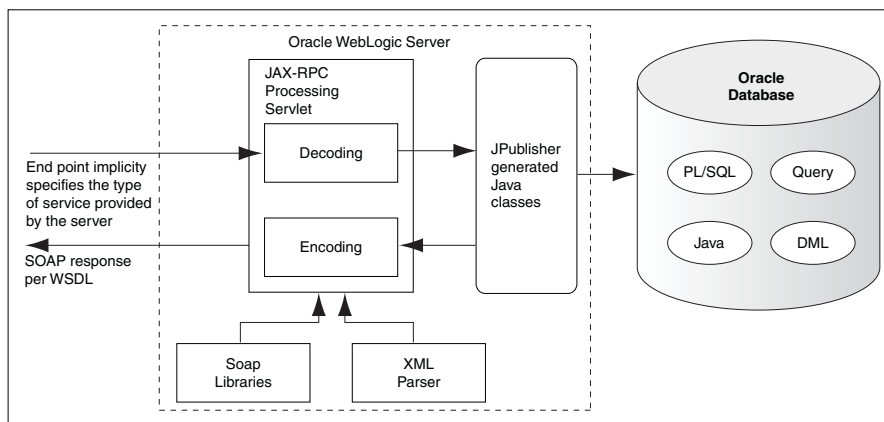
- [How to Use JPublisher for Web Services Call-Ins](#)
- [Features of Oracle Database as a Web Service Provider](#)
- [JPublisher Support for Web Services Call-Ins to Oracle Database](#)

How to Use JPublisher for Web Services Call-Ins

You can use JPublisher to generate Java wrappers that correspond to Database operations and deploy the wrappers as Web services in Oracle Application Server.

Figure 12-1 illustrates how you use JPublisher to publish PL/SQL packages, SQL objects, collections, and packages as Java classes. Once published, these classes can be accessed by any Web service through a WebLogic Server (WLS) Web services servlet.

Figure 12-1 Web Services Call-In to the Database



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

Features of Oracle Database as a Web Service Provider

Using Oracle Database as a Web service provider offers the following features:

- Enhances PL/SQL Web services
 - Improves PL/SQL Web services by extending the Web services support for additional PL/SQL types, including CLOB, BLOB, XMLTYPE, ref cursor, and PL/SQL records and tables. This enables you to use most of your existing PL/SQL packages as Web services.
- Exposes Java in the database as Web services
 - Exposes existing Java classes deployed in Oracle Database as Web services. Java classes implementing data-related services can be migrated between the middle tier and the database. Java portability results in database independence.
- Provides SQL query Web services
 - Leverages warehousing or business intelligence queries, data monitoring queries, and any predefined SQL statements as Web services.

- Enables DML Web services
 - Offers secure, persistent, transactional, and scalable logging, auditing, and tracking operations implemented through SQL DML, as Web services. DML Web services are implemented as atomic or group, or batch, INSERT, UPDATE, and DELETE operations.

JPublisher Support for Web Services Call-Ins to Oracle Database

The following JPublisher features support Web services call-ins to the code running in Oracle Database:

- Generation of Java interfaces
- JPublisher styles and style files
- Ref cursor returning and result set mapping
- Options to filter what JPublisher publishes
- Support for calling Java classes in the database without PL/SQL call specifications
- Support for publishing SQL queries or DML statements
- Support for unique method names
- Support for Oracle Streams AQ

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

Using Oracle Database as Web Services Consumer

You can extend the storage, indexing, and searching capabilities of a relational database to include semistructured and nonstructured data, including Web services, in addition to enabling federated data. By calling Web services, the database can track, aggregate, refresh, and query dynamic data produced on-demand, such as stock prices, currency exchange rates, and weather information.

An example of using Oracle Database as a service consumer would be to call external Web services from a predefined database job to retrieve inventory information from multiple suppliers, and then update your local inventory database. Another example is that of a Web crawler, where a database job can be scheduled to collate product and price information from a number of sources.

This section covers the following topics:

- [How to Use Oracle Database for Web Services Call-Outs](#)
- [Web Service Data Sources \(Virtual Table Support\)](#)
- [Features of Oracle Database as a Web Service Consumer](#)
- [Overview of JPublisher Generation](#)
- [Adjusting the Mapping of SQL Types](#)

How to Use Oracle Database for Web Services Call-Outs

The Web services client code is written in SQL, PL/SQL, or Java to run inside Oracle Database, which then calls the external Web service. You can call a Web service from a Java client within the database, using one of the following methods:

- SQL and PL/SQL call specifications

Start a Web service through a user-defined function call, which is generated through JPublisher, either directly within a SQL statement or view or through a variable.

- Pure Java static proxy class

Use JPublisher to generate a client proxy class, which uses Java API for XML-based remote procedure call (JAX-RPC). This method simplifies the Web service invocation because the location of the service is already known without needing to look up the service in the UDDI registry. The client proxy class does all the work required to construct the SOAP request, including marshalling and unmarshalling parameters.

- Pure Java using dynamic invocation interface (DII) over JAX-RPC

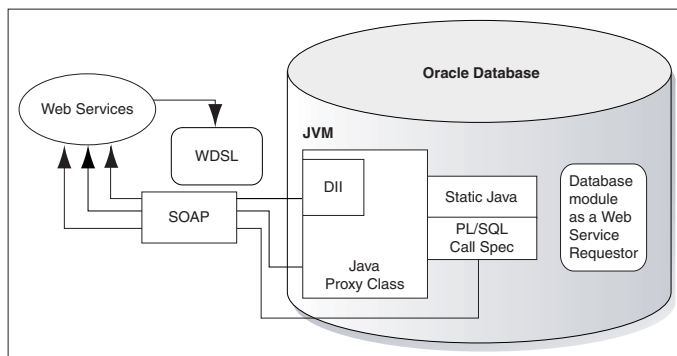
Dynamic invocation provides the ability to construct the SOAP request and access the service without the client proxy.

For Web services call-outs using PL/SQL, use the UTL_DBWS PL/SQL package. This package essentially uses the same application programming interfaces (APIs) as the DII classes.

You can use a Web services data source to process the results from any Web service request.

Figure 12–2 illustrates how you can call a Web service from a Java client within the database.

Figure 12–2 Calling Web Services From Within the Database



Web Service Data Sources (Virtual Table Support)

To access data that is returned from single or multiple Web service invocations, create a virtual table using a Web service data source. This table lets you query a set of returned rows as though it were a table.

The client calls a Web service and the results are stored in a virtual table in the database. You can pass result sets from function to function. This enables you to set up a sequence of transformation without a table holding intermediate results. To reduce memory usage, you can return the result set rows, a few at a time, within a function.

By using Web services with the table function, you can manipulate a range of input values from single or multiple Web services as a real table. In the following example, the inner `SELECT` statement creates rows whose columns are used as arguments for calling the `CALL_WS` Web service call-out.

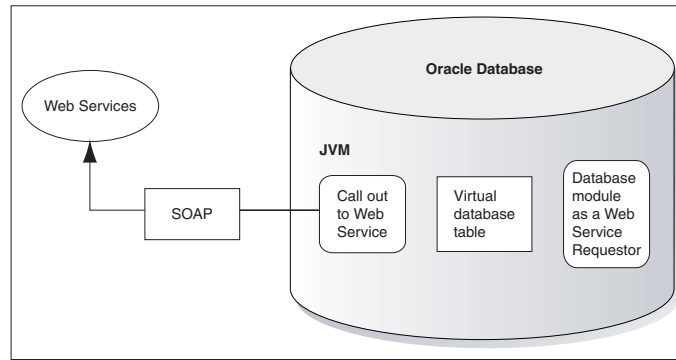
```
SELECT column1, cloumn2, ...
FROM TABLE(WS_TABFUN(CURSOR(SELECT s FROM table_name)))
```

WHERE ...

The table expression in the preceding example can be used in other SQL queries, for constructing views, and so on.

Figure 12–3 illustrates the support for virtual table.

Figure 12–3 Storing Results from Request in a Virtual Table



Features of Oracle Database as a Web Service Consumer

Using Oracle Database as a Web service consumer provides the following features:

- Consuming Web services from Java

Provides an easy-to-use interface for Web services call-outs, thereby insulating developers from low-level SOAP programming. Java classes running in the database can directly call external Web services by using the previously loaded Java proxy class or through dynamic invocation.
- Consuming Web services from SQL and PL/SQL

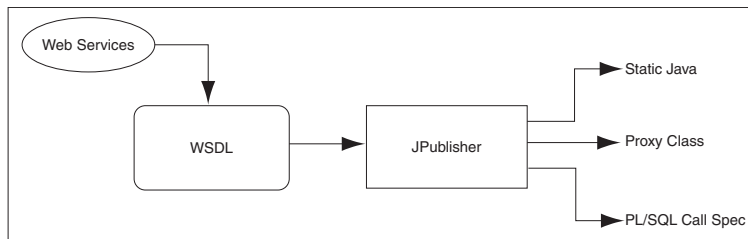
Enables any SQL-enabled tool or application to transparently and easily consume dynamic data from external Web services. After exposing Web services methods as Java stored procedures, a PL/SQL wrapper on top of a Java stored procedure hides all Java and SOAP programming details from the SQL client.
- Using Web services data source

Enables application and data integration by turning external Web service into a SQL data source, making the external Web service appear as regular SQL table. This table function represents the output of calling external Web services and can be used in a SQL query.

Overview of JPublisher Generation

JPublisher can receive the WSDL file from a Web service and create the static Java, proxy class, or PL/SQL call specification.

Figure 12–4 illustrates creating Web Services Call-Out Stubs.

Figure 12–4 Creating Web Services Call-Out Stubs

This support is created through the following JPublisher key options:

- `-proxywsdl=url`

Use this option to generate JAX-RPC static client proxies, given the WSDL document at the specified URL. This option generates additional wrapper classes to expose instance methods as `static` methods and generates PL/SQL wrappers. It performs the following steps:

1. Generates JAX-RPC client proxy classes.
2. Generates wrapper classes to publish instance methods as `static` methods.
3. Generates PL/SQL wrappers for classes that must be accessible from PL/SQL.
4. Loads generated code into the database.

Note: The `-proxywsdl` option uses the `-proxyclasses` option behind the scenes for steps 2 and 3, and takes the `-proxyopts` setting as input.

Once generated, your database client can access the Web service through PL/SQL using the call specifications or through the JAX-RPC client proxy classes. The PL/SQL wrappers use the `static` methods. A client would *not* usually access any Web service using the `static` method directly.

- `-httpproxy=proxy_url`

Where WSDL is accessed through a firewall, use this option to specify a proxy URL to use in resolving the URL of the WSDL document.

- `-proxyclasses=class_list`

For Web services, this option is used behind the scenes by the `-proxywsdl` option and is set automatically, as appropriate. In addition, you can use this option directly, for general purposes, any time you want to create PL/SQL wrappers for Java classes with `static` methods, and optionally to produce wrapper classes to expose instance methods as `static` methods.

The `-proxyclasses` option accepts the `-proxyopts` setting as input.

- `-proxyopts=wrapper_specifications`

This option specifies JPublisher behavior in generating wrapper classes and PL/SQL wrappers, usually, but not necessarily, for Web services. For typical usage of the `-proxywsdl` option, the `-proxyopts` default setting is sufficient. In situations where you use the `-proxyclasses` option directly, you may want to use the special `-proxyopts` settings.

- `-endpoint=Web_services_endpoint`

Use this option in conjunction with the `-proxywsdl` option to specify the Web services endpoint.

- `-sysuser=superuser_name/superuser_password`

Use this option to specify the name and password for the superuser account used to grant permissions for the client proxy classes to access Web services using HTTP.

Adjusting the Mapping of SQL Types

Although Oracle Application Server does not currently support LOB types, XMLTYPE, ref cursors, and OUT and IN OUT arguments, you can use an alternative approach to expose PL/SQL methods and SQL types as Web services.

You can change the default action of JPublisher to generate code that uses a user-provided subclass. For example, if you have a PL/SQL method that returns a REF CURSOR, then JPublisher automatically maps the return type to `java.sql.ResultSet`. However, this `ResultSet` type cannot be published as a Web service. To solve this, create a new method that can return the result set in a Web service-supported format, as follows:

```
public String [] readRefCursorArray(String arg1, Integer arg2)
{
    java.sql.ResultSet rs = getRefCursor(arg1,arg2);
    ...
    //create a String[] from rs and return it
    ...
}
```

After creating a method, create an interface that contains the exact methods to publish. You can use JPublisher to easily accomplish this mapping by using the following command:

```
jpub -sql=MYAPP:MyAppBase:MyApp#MyAppInterf...
```

In the preceding command:

- `MyApp` contains the method to return the result set.
- `MyAppInterf` is the interface that contains the method to publish.

After translating the code for your application, archive all the class files into a single Java Archive (JAR) file and use the Web Services Assembler to create a deployable Web service Enterprise Archive (EAR) file.

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

DBMS_JAVA Package

This chapter provides a description of the DBMS_JAVA package. The functions and procedures in this package provide an entry point for accessing RDBMS functionality from Java.

longname

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

The function returns the fully qualified name of the specified Java schema object. Because Java classes and methods can have names exceeding the maximum SQL identifier length, Oracle JVM uses abbreviated names internally for SQL access. This function returns the original Java name for any truncated name. An example of this function is to display 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';
```

See Also: ["Shortened Class Names"](#) on page 2-21

shortname

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

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

See Also: ["Shortened Class Names"](#) on page 2-21

get_compiler_option

```
FUNCTION get_compiler_option(name VARCHAR2, optionName VARCHAR2) RETURN VARCHAR2
```

This function returns the value of the option specified through the `optionName` parameter. It is one of the functions used to control the options of the Java and SQLJ compiler supplied with Oracle Database.

See Also: ["Compiler Options Specified in a Database Table"](#) on page 2-7

set_compiler_option

```
PROCEDURE set_compiler_option(name VARCHAR2, optionName VARCHAR2, value VARCHAR2)
```

This procedure is used to set the options of the Java and SQLJ compiler supplied with Oracle Database.

See Also: ["Compiler Options Specified in a Database Table"](#) on page 2-7

reset_compiler_option

```
PROCEDURE reset_compiler_option(name VARCHAR2, optionName VARCHAR2)
```

This procedure is used to reset the specified compiler option to the default value.

See Also: ["Compiler Options Specified in a Database Table"](#) on page 2-7

resolver

```
FUNCTION resolver (name VARCHAR2, owner VARCHAR2, type VARCHAR2) RETURN VARCHAR2
```

This function returns the resolver specification for the object specified in *name* and in the schema specified in *owner*, where the object is of the type specified in *type*. The caller must have EXECUTE privilege and have access to the given object to use this function.

The *name* parameter is the short name of the object.

The value of *type* can be either SOURCE or CLASS.

If there is an error, then NULL is returned. If the underlying object has changed, then `ObjectTypeChangedException` is thrown.

You can call this function as follows:

```
SELECT dbms_java.resolver('tst', 'SCOTT', 'CLASS') FROM DUAL;
```

This would return:

```
DBMS_JAVA.RESOLVER('TST','SCOTT','CLASS')
-----
((* SCOTT)(* PUBLIC))
```

derivedFrom

```
FUNCTION derivedFrom (name VARCHAR2, owner VARCHAR2, type VARCHAR2) RETURN VARCHAR2
```

This function returns the source name of the object specified in *name* of the type specified in *type* and in the schema specified in *owner*. The caller must have EXECUTE privilege and have access to the given object to use this function.

The *name* parameter, as well as the returned source name, is the short name of the object.

The value of *type* can be either SOURCE or CLASS.

If there is an error, then NULL is returned. If the underlying object has changed, then `ObjectTypeChangedException` is thrown.

The returned value will be NULL if the object was not compiled in Oracle JVM.

You can call this function as follows:

```
SELECT dbms_java.derivedFrom('tst', 'SCOTT', 'CLASS') FROM DUAL;
```

This would return:

```
DBMS_JAVA.DERIVEDFROM('TST', 'SCOTT', 'CLASS')
-----
tst
```

fixed_in_instance

```
FUNCTION fixed_in_instance (name VARCHAR2, owner VARCHAR2, type VARCHAR2) RETURN
NUMBER
```

This function returns the permanently kept status for object specified in `name` of the type specified in `type` and in the schema specified in `owner`. The caller must have EXECUTE privilege and have access to the given object to use this function.

The `name` parameter is the short name for the object.

The value of `type` can be either of RESOURCE, SOURCE, CLASS, or SHARED_DATA.

The number returned is either 0, indicating the status is not kept, or 1, indicating the status is kept.

You can call this function as follows:

```
SELECT dbms_java.fixed_in_instance('tst', 'SCOTT', 'CLASS') FROM DUAL;
```

This would return:

```
DBMS_JAVA.FIXED_IN_INSTANCE('TST', 'SCOTT', 'CLASS')
-----
0
```

Consider the following statement:

```
SELECT dbms_java.fixed_in_instance('java/lang/String', 'SYS', 'CLASS') FROM DUAL;
```

This would return:

```
DBMS_JAVA.FIXED_IN_INSTANCE('JAVA/LANG/STRING', 'SYS', 'CLASS')
-----
1
```

set_output

```
PROCEDURE set_output (buffersize NUMBER)
```

This procedure redirects the output of Java stored procedures and triggers to the DBMS_OUTPUT package.

See Also: ["Redirecting Output"](#) on page 7-2

export_source

```
PROCEDURE export_source(name VARCHAR2, schema VARCHAR2, blob BLOB)
```

```
PROCEDURE export_source(name VARCHAR2, blob BLOB)
```

```
PROCEDURE export_source(name VARCHAR2, clob CLOB)
```

These procedures are used to export the Java source as a Java source schema object to Oracle Database. The source is specified through the name parameter. The source can be exported into a BLOB or CLOB object. The internal representation of the source uses the UTF8 format, so that format is used to store the source in the BLOB as well. The source schema object is created in the specified schema. If the schema is not specified then the current schema is used.

export_class

```
PROCEDURE export_class(name VARCHAR2, schema VARCHAR2, blob BLOB)
```

```
PROCEDURE export_class(name VARCHAR2, blob BLOB)
```

These procedures are used to export Java classes specified through the name parameter as Java class schema objects to Oracle Database. You cannot export a class into a CLOB object, only into a BLOB object. If the schema is specified, then the class schema object is created in this schema, else in the current schema.

export_resource

```
PROCEDURE export_resource(name VARCHAR2, schema VARCHAR2, blob BLOB)
```

```
PROCEDURE export_resource(name VARCHAR2, blob BLOB)
```

```
PROCEDURE export_resource(name VARCHAR2, schema VARCHAR2, clob CLOB)
```

```
PROCEDURE export_resource(name VARCHAR2, clob CLOB)
```

The resource specified through the name parameter is exported to Oracle Database as a resource schema object in the schema specified through the schema parameter. If the schema is not specified then the current schema is used. The resource can be exported into either a CLOB object or BLOB object.

loadjava

```
PROCEDURE loadjava(options VARCHAR2)
```

```
PROCEDURE loadjava(options VARCHAR2, resolver VARCHAR2)
```

These procedures enable you to load classes in to the database using a call, rather than through the loadjava command-line tool. You can call this procedure within your Java application as follows:

```
CALL dbms_java.loadjava('... options...');
```

The options are identical to those specified on the command line. Each option should be separated by a space. Do not separate the options with a comma. The only exception to this is the loadjava -resolver option, which contains spaces. For -resolver, specify all other options first, separate these options by a comma, and then specify the -resolver options, as follows:

```
CALL dbms_java.loadjava('... options...', 'resolver_options');
```

Do not specify the `-thin`, `-oci`, `-user`, and `-password` options, because they relate to the database connection. The output is directed to `System.err`. The output typically goes to a trace file, but can be redirected.

See Also: ["The loadjava Tool"](#) on page 11-6

dropjava

```
PROCEDURE dropjava(options VARCHAR2)
```

This procedure enables you to drop classes within the database using a call, rather than through the `dropjava` command-line tool. You can call this procedure within your Java application as follows:

```
CALL dbms_java.dropjava('... options...');
```

See Also: ["The dropjava Tool"](#) on page 11-17

grant_permission

```
PROCEDURE grant_permission(grantee VARCHAR2, permission_type VARCHAR2,
permission_name VARCHAR2,
permission_action VARCHAR2)
```

This method is used to grant permission to specific users or roles.

See Also: ["Fine-Grain Definition for Each Permission"](#) on page 10-5

restrict_permission

```
PROCEDURE restrict_permission(grantee VARCHAR2, permission_type VARCHAR2,
permission_name VARCHAR2,
permission_action VARCHAR2)
```

This method is used to specify limitations or exceptions to general rules.

See Also: ["Fine-Grain Definition for Each Permission"](#) on page 10-5

grant_policy_permission

```
PROCEDURE grant_policy_permission(grantee VARCHAR2, permission_schema VARCHAR2,
permission_type VARCHAR2,
permission_name VARCHAR2)
```

This method is used to grant and limit `PolicyTablePermission`.

See Also: ["Acquiring Administrative Permission to Update Policy Table"](#) on page 10-8

revoke_permission

```
PROCEDURE revoke_permission(permission_schema VARCHAR2, permission_type VARCHAR2,
permission_name VARCHAR2,
permission_action VARCHAR2)
```

This method is used to disable a granted permission.

See Also: ["Enabling or Disabling Permissions"](#) on page 10-12

disable_permission

```
PROCEDURE disable_permission(key NUMBER)
```

This method is used to disable a granted permission.

See Also: ["Enabling or Disabling Permissions"](#) on page 10-12

enable_permission

```
PROCEDURE enable_permission(key NUMBER)
```

This method is used to enable a permission.

See Also: ["Enabling or Disabling Permissions"](#) on page 10-12

delete_permission

```
PROCEDURE delete_permission(key NUMBER)
```

This method is used to delete a granted permission.

See Also: ["Enabling or Disabling Permissions"](#) on page 10-12

set_preference

```
procedure set_preference(user VARCHAR2, type VARCHAR2, abspath VARCHAR2, key  
VARCHAR2, value VARCHAR2)
```

This procedure inserts or updates a row in the `SYS:java$prefs$` table as follows:

```
CALL dbms_java.set_preference('SCOTT', 'U', '/my/package/method/three',  
'windowsize', '22:32');
```

The `user` parameter specifies the name of the schema to which the preference should be attached. If the logged in schema is not `SYS`, then `user` must specify the current logged in schema or the `INSERT` will fail. The `type` parameter can take either the value `U`, indicating user preference, or `S`, indicating system preference. The `abspath` parameter specifies the absolute path for the preference. `key` is the preference key used for the lookup, and `value` is the value of the preference key.

runjava

```
FUNCTION runjava(cmdline VARCHAR2) RETURN VARCHAR2;
```

This function takes the Java command line as its only argument and runs it in Oracle JVM.

See Also: ["Using Command-Line Interface"](#) on page 3-9

runjava_in_current_session

```
FUNCTION runjava_in_current_session(cmdline VARCHAR2) RETURN VARCHAR2;
```

This function is the same as the `runjava` function, except that it does not clear Java state remaining from previous use of Java in the session, prior to executing the current command line.

See Also: ["Using Command-Line Interface"](#) on page 3-9

set_property

```
FUNCTION set_property(name VARCHAR2, value VARCHAR2) RETURN VARCHAR2;
```

This function establishes a value for a system property that is then used for the duration of the current RDBMS session, whenever a Java session is initialized.

Note: In order to execute the `SET_PROPERTY` function, a user must have write permission on `SYS:java.util.PropertyPermission` for the property name. You can grant this permission using the following command:

```
call dbms_java.grant_permission( '<user_name>',  
'SYS:java.util.PropertyPermission', '<property_name>', 'write' );
```

See Also: ["Setting System Properties"](#) on page 4-5

get_property

```
FUNCTION get_property(name VARCHAR2) RETURN VARCHAR2;
```

This function returns any value previously established by `set_property`.

See Also: ["Setting System Properties"](#) on page 4-5

remove_property

```
FUNCTION remove_property(name VARCHAR2) RETURN VARCHAR2;
```

This function removes any value previously established by `set_property`.

Note: In order to execute the `remove_property` function, a user must have write permission on `SYS:java.util.PropertyPermission` for the property name. You can grant this permission using the following command:

```
call dbms_java.grant_permission( '<user_name>',  
'SYS:java.util.PropertyPermission', '<property_name>', 'write' );
```

See Also: ["Setting System Properties"](#) on page 4-5

show_property

```
FUNCTION show_property(name VARCHAR2) RETURN VARCHAR2;
```

This function displays a message of the form `name = value` for the input name, or for all established property bindings, if name is null.

See Also: ["Setting System Properties"](#) on page 4-5

set_output_to_sql

```
FUNCTION set_output_to_sql (id VARCHAR2,  
    stmt VARCHAR2,  
    bindings VARCHAR2,  
    no_newline_stmt VARCHAR2 default null,  
    no_newline_bindings VARCHAR2 default null,  
    newline_only_stmt VARCHAR2 default null,  
    newline_only_bindings VARCHAR2 default null,  
    maximum_line_segment_length NUMBER default 0,  
    allow_replace NUMBER default 1,  
    from_stdout NUMBER default 1,  
    from_stderr NUMBER default 1,  
    include_newlines NUMBER default 0,  
    eager NUMBER default 0) return VARCHAR2;
```

`set_output_to_sql` defines a named output specification that constitutes an instruction for executing a SQL statement, whenever output to the default

`System.out` and `System.err` streams occurs.

See Also: ["Redirecting Output on the Server"](#) on page 3-14

remove_output_to_sql

```
FUNCTION remove_output_to_sql (id VARCHAR2) return VARCHAR2;
```

`remove_output_to_sql` deletes a specification created by `set_output_to_sql`.

See Also: ["Redirecting Output on the Server"](#) on page 3-14

enable_output_to_sql

```
FUNCTION enable_output_to_sql (id VARCHAR2) return VARCHAR2;
```

`enable_output_to_sql` reenables a specification created by `set_output_to_sql` and subsequently disabled by `disable_output_to_sql`.

See Also: ["Redirecting Output on the Server"](#) on page 3-14

disable_output_to_sql

```
FUNCTION disable_output_to_sql (id VARCHAR2) return VARCHAR2;
```

`disable_output_to_sql` disables a specification created by `set_output_to_sql`.

See Also: ["Redirecting Output on the Server"](#) on page 3-14

query_output_to_sql

```
FUNCTION query_output_to_sql (id VARCHAR2) return VARCHAR2;
```

query_output_to_sql returns a message describing a specification created by set_output_to_sql.

See Also: ["Redirecting Output on the Server"](#) on page 3-14

set_output_to_java

```
FUNCTION set_output_to_java (id VARCHAR2,
class_name VARCHAR2,
class_schema VARCHAR2,
method VARCHAR2,
bindings VARCHAR2,
no_newline_method VARCHAR2 default null,
no_newline_bindings VARCHAR2 default null,
newline_only_method VARCHAR2 default null,
newline_only_bindings VARCHAR2 default null,
maximum_line_segment_length NUMBER default 0,
allow_replace NUMBER default 1,
from_stdout NUMBER default 1,
from_stderr NUMBER default 1,
include_newlines NUMBER default 0,
eager NUMBER default 0,
initialization_statement VARCHAR2 default null,
finalization_statement VARCHAR2 default null)return VARCHAR2;
```

set_output_to_java defines a named output specification that constitutes an instruction for executing a Java method whenever output to the default System.out and System.err streams occurs.

See Also: ["Redirecting Output on the Server"](#) on page 3-14

remove_output_to_java

```
FUNCTION remove_output_to_java (id VARCHAR2) return VARCHAR2;
```

remove_output_to_java deletes a specification created by set_output_to_java.

See Also: ["Redirecting Output on the Server"](#) on page 3-14

enable_output_to_java

```
FUNCTION enable_output_to_java (id VARCHAR2) return VARCHAR2;
```

enable_output_to_java reenables a specification created by set_output_to_java and subsequently disabled by disable_output_to_java.

See Also: ["Redirecting Output on the Server"](#) on page 3-14

disable_output_to_java

```
FUNCTION disable_output_to_java (id VARCHAR2) return VARCHAR2;
```

disable_output_to_java disables a specification created by set_output_to_java.

See Also: ["Redirecting Output on the Server"](#) on page 3-14

query_output_to_java

```
FUNCTION query_output_to_java (id VARCHAR2) return VARCHAR2;
```

`query_output_to_java` returns a message describing a specification created by `set_output_to_java`.

See Also: ["Redirecting Output on the Server"](#) on page 3-14

set_output_to_file

```
FUNCTION set_output_to_file (id VARCHAR2,  
file_path VARCHAR2,  
allow_replace NUMBER default 1,  
from_stdout NUMBER default 1,  
from_stderr NUMBER default 1) return VARCHAR2;
```

`set_output_to_file` defines a named output specification that constitutes an instruction to capture any output sent to the default `System.out` and

`System.err` streams and append it to a specified file.

See Also: ["Redirecting Output on the Server"](#) on page 3-14

remove_output_to_file

```
FUNCTION remove_output_to_file (id VARCHAR2) return VARCHAR2;
```

`remove_output_to_file` deletes a specification created by `set_output_to_file`.

See Also: ["Redirecting Output on the Server"](#) on page 3-14

enable_output_to_file

```
FUNCTION enable_output_to_file (id VARCHAR2) return VARCHAR2;
```

`enable_output_to_file` reenables a specification created by `set_output_to_file` and subsequently disabled by `disable_output_to_file`.

See Also: ["Redirecting Output on the Server"](#) on page 3-14

disable_output_to_file

```
FUNCTION disable_output_to_file (id VARCHAR2) return VARCHAR2;
```

`disable_output_to_file` disables a specification created by `set_output_to_file`.

See Also: ["Redirecting Output on the Server"](#) on page 3-14

query_output_to_file

```
FUNCTION query_output_to_file (id VARCHAR2) return VARCHAR2;
```

`query_output_to_file` returns a message describing a specification created by `set_output_to_file`.

See Also: ["Redirecting Output on the Server"](#) on page 3-14

enable_output_to_trc

```
PROCEDURE enable_output_to_trc;
```

This procedure reenables printing the output to `System.out` and `System.err` in the `.trc` file that was disabled by the `disable_output_to_trc` procedure.

See Also: ["Redirecting Output on the Server"](#) on page 3-14

disable_output_to_trc

```
PROCEDURE disable_output_to_trc;
```

This procedure prevents output to `System.out` and `System.err` from appearing in the `.trc` file.

See Also: ["Redirecting Output on the Server"](#) on page 3-14

query_output_to_trc

```
FUNCTION query_output_to_trc return VARCHAR2;
```

This function returns a value indicating whether printing output to `System.out` and `System.err` in the `.trc` file is currently enabled.

See Also: ["Redirecting Output on the Server"](#) on page 3-14

endsession

```
FUNCTION endsession RETURN VARCHAR2;
```

This function clears any Java session state remaining from previous execution of Java in the current RDBMS session.

See Also: ["Two-Tier Duration for Java Session State"](#) on page 4-4

endsession_and_related_state

```
FUNCTION endsession_and_related_state RETURN VARCHAR2;
```

This function clears any Java session state remaining from previous execution of Java in the current RDBMS session and all supporting data related to running Java.

See Also: ["Two-Tier Duration for Java Session State"](#) on page 4-4

set_native_compiler_option

```
PROCEDURE set_native_compiler_option(optionName VARCHAR2,  
value VARCHAR2);
```

This procedure sets a native-compiler option to the specified value for the current schema.

See Also: ["Oracle JVM Just-in-Time Compiler \(JIT\)"](#) on page 9-1

unset_native_compiler_option

```
PROCEDURE unset_native_compiler_option(optionName VARCHAR2,  
value VARCHAR2);
```

This procedure unsets a native-compiler option/value pair for the current schema.

See Also: ["Oracle JVM Just-in-Time Compiler \(JIT\)"](#) on page 9-1

compile_class

```
FUNCTION compile_class(classname VARCHAR2) return NUMBER;
```

This function compiles all methods defined by the class that is identified by *classname* in the current schema.

See Also: ["Oracle JVM Just-in-Time Compiler \(JIT\)"](#) on page 9-1

compile_class

```
FUNCTION compile_class(schema VARCHAR2,  
classname VARCHAR2) return NUMBER;
```

This function compiles all methods defined by the class that is identified by *classname* in the supplied schema.

See Also: ["Oracle JVM Just-in-Time Compiler \(JIT\)"](#) on page 9-1

uncompile_class

```
FUNCTION uncompile_class(classname VARCHAR2,  
permanentp NUMBER default 0) return NUMBER;
```

This function uncompiles all methods defined by the class that is identified by *classname* in the current schema.

See Also: ["Oracle JVM Just-in-Time Compiler \(JIT\)"](#) on page 9-1

uncompile_class

```
FUNCTION uncompile_class(schema VARCHAR2,  
classname VARCHAR2,  
permanentp NUMBER default 0) return NUMBER;
```

This function uncompiles all methods defined by the class that is identified by *classname* in the supplied schema.

See Also: ["Oracle JVM Just-in-Time Compiler \(JIT\)"](#) on page 9-1

compile_method

```
FUNCTION compile_method(classname VARCHAR2,  
methodname VARCHAR2,  
methodsig VARCHAR2) return NUMBER;
```

This function compiles the method specified by *name* and *Java type* signatures defined by the class, which is identified by *classname* in the current schema.

See Also: ["Oracle JVM Just-in-Time Compiler \(JIT\)"](#) on page 9-1

compile_method

```
FUNCTION compile_method(schema VARCHAR2,
  classname VARCHAR2,
  methodname VARCHAR2,
  methodsig VARCHAR2) return NUMBER;
```

This function compiles the method specified by *name* and *Java type* signatures defined by the class that is identified by *classname* in the supplied schema.

See Also: ["Oracle JVM Just-in-Time Compiler \(JIT\)"](#) on page 9-1

uncompile_method

```
FUNCTION uncompile_method(classname VARCHAR2,
  methodname VARCHAR2,
  methodsig VARCHAR2,
  permanentp NUMBER default 0) return NUMBER;
```

This function uncompiles the method specified by the *name* and *Java type* signatures defined by the class that is identified by *classname* in the current schema.

See Also: ["Oracle JVM Just-in-Time Compiler \(JIT\)"](#) on page 9-1

uncompile_method

```
FUNCTION uncompile_method(schema VARCHAR2,
  classname VARCHAR2,
  methodname VARCHAR2,
  methodsig VARCHAR2,
  permanentp NUMBER default 0) return NUMBER;
```

This function uncompiles the method specified by the *name* and *Java type* signatures defined by the class that is identified by *classname* in the supplied schema.

See Also: ["Oracle JVM Just-in-Time Compiler \(JIT\)"](#) on page 9-1

start_jmx_agent

```
procedure start_jmx_agent(port VARCHAR2 default NULL,
  ssl VARCHAR2 default NULL,
  auth VARCHAR2 default NULL);
```

This procedure starts the JMX agent in a specific session. Generally, the agent remains active for the duration of the session.

See Also: ["Managing Your Applications Using JMX"](#) on page 2-28

Classpath Extensions and User Classloaded Metadata

This section provides a description of the extensions to the `-classpath` search path and User Classloaded Metadata.

B.1 Classpath Extensions

This section provides a description of the extensions to the `-classpath` search path and `jserver` URL protocol syntaxes that allow specification of database resident objects and byte sets in search paths used by the command-line interface.

B.1.1 `jserverQuotedClassPathTermPrefix`

When a classpath term begins with the `jserverQuotedClassPathTermPrefix` string, it extends through the next occurrence of the string, regardless of the `File.pathSeparator` characters it may contain. The actual value of this string is given by the system property `jserver.quoted.classpath.term.prefix`. If this property is not defined, the default value is `||`.

B.1.2 `jserverURLPrefix`

When a dequoted classpath term begins with the `jserverURLPrefix` string, the rest of the term is treated as a URL. The value of this string is given by the system property `jserver.url.in.classpath.prefix`. If this value is `null`, any quoted term that does not begin with one of the following three prefixes, is treated as a URL:

- `jserverSpecialTokenPrefix`, if the value is set
- `JSERVER_CP`
- `JSERVER_SCHEMAC`

Note: A quoted term is one that begins and ends with the string that is the value of `jserverQuotedClassPathTermPrefix`. A dequoted term is either the whole original term if it is not quoted, or the part of a quoted term between the beginning and ending occurrences of `jserverQuotedClassPathTermPrefix`.

B.1.3 `jserverSpecialTokenPrefix`

The value of the `jserverSpecialTokenPrefix` string is given by the system property `jserver.specialtoken.in.classpath.prefix`. If this value is not

null, then the prefixes `JSERVER_CP` and `JSERVER_SCHEMAC` are recognized only when preceded by this string.

See Also: ["JSERVER_CP"](#) on page B-2 and ["JSERVER_SCHEMAC"](#) on page B-2

B.1.4 JSERVER_CP

A classpath term beginning with the literal substring `"JSERVER_CP"` is converted to a URL by replacing `JSERVER_CP` with `jserver:/CP`.

B.1.5 JSERVER_SCHEMAC

A classpath term beginning with the literal substring `"JSERVER_SCHEMAC"` is converted to a URL by replacing `JSERVER_SCHEMAC` with `jserver:/CPcSCHEMAC`. Here `c` can be any character, but is typically `/`. This means that a term of the form `JSERVER_SCHEMAC + <remaining string>` is treated as a prescription for looking for shared System classloaded classes and resources in the schema identified by `<remaining string>`. For example, the term `JSERVER_SCHEMA/SCOTT` is equivalent to `jserver:/CP/SCOTT` and it instructs to look for shared classes and resources in the schema named `SCOTT`.

B.1.6 jserver:/CP general syntax

A URL beginning with `jserver:/CP` is meaningful only as a classpath term. The first character following `jserver:/CP` is used as the token separator for the remainder of the string. This is typically the character `/`. The subsequent tokens are the following:

- The possible values of the first token are `JAR`, `RESOURCE`, or `SHARED_DATA`, where `RESOURCE` indicates a Java resource object, `SHARED_DATA` indicates a Java shared data object, and `JAR` indicates a database resident JAR object. This token is optional and all of the values are case-insensitive. If one of these is present, the URL is called a JAR specifier. Otherwise, it is called a SCHEMA specifier.
- The value of the second token is `PRIVATE`. This is an optional token and is case-insensitive.
- The value of the third token is `SCHEMA`. This is a required token and is case-insensitive.
- The fourth token is a required token, which is interpreted as a schema name.
- The fifth token is required for a JAR specifier and prohibited for a SCHEMA specifier. It is interpreted as the name of an object in the schema identified by the fourth token, if present.

Functionally, a classpath term is used to look for an object that matches a class or resource name that is being searched for. In the case of a SCHEMA specifier, the object is looked for in the indicated schema. In the case of a JAR specifier, a particular object in the schema is identified by the fifth token in the URL. This object is treated as a JAR and the searched for object is looked for by name, within that JAR. In the case of looking for a class within a database resident JAR, this may mean finding the class as a class object in the schema. Otherwise, it means finding the search object in the actual bytes of the JAR object.

The searched for object is a database Java class object, if it meets the following conditions:

- The search name ends in `.class`

- the URL is either a SCHEMA specifier or a JAR specifier for a database resident JAR

Such a class object may be:

- Loaded as a shared system classloaded class. This is done if the optional second token PRIVATE is not present.
- Interpreted as a set of bytecodes and loaded by the `defineClass` method as a private user classloaded class. This is done if the optional second token PRIVATE is present.

Note: Classes loaded from classpath terms not beginning with the `jserver` URL marker are always private, user classloaded classes.

B.2 User Classloaded Metadata

Starting from 11g release 1 (11.1), there is a new system table created in the following manner in `javavm/install/initjvma.sql` during database creation and upgrade:

```
create table java$jvm$runtime$parameters (owner# number not null, flags number);
create unique index java$jvm$runtime$parameters$i on
java$jvm$runtime$parameters(owner#);
```

This table is removed during downgrade by `javavm/install/rmjvm.sql`. If you want to share private class metadata and have DBA privileges, then you can populate this table manually. The rule is that if there is a row matching your owner ID, then the flag value of this row is bitwise anded with the flag value from the row with `owner# = -1`, if any. If none of these previously mentioned rows exist, then the bit set in the result is `-1`, that is, all bits set. If bit 0 (`1<<0`) is set in the result, then your session attempts to share existing shared private metadata. If bit 1 (`1<<1`) is set in the result, then the session creates shared metadata when existing shared metadata is not found. By default, there is no row in the table. So, all sessions both use and create shared private metadata.

A

act method, 2-50, 2-51
application
 compiling, 2-5
 developing, 8-1
 development, 2-1
 executing in a session, 2-1
 execution control, 2-3
 execution rights, 2-15
 invoking, 3-1
 running on the server, 3-13
 threading, 2-46
attributes, 5-2, 6-12
 declaring, 6-13
 definition, 1-2
 types of, 1-2
authentication, 10-1
 mechanisms, 10-18
AUTHID clause, 6-6, 6-10, 6-13

B

BasicPermission, 10-9
body
 package, 6-10
 SQL object type, 6-12
bytecode
 defined, 1-6
 definition, 1-18
 verification, 2-11
 verifier, 2-11

C

call
 definition, 1-14
 managing resources across calls, 2-54
 static fields, 2-2
call memory, 2-3
call specification, 3-1
 defining, 6-2
 mapping data types, 6-3
 object type, 6-12
 packaged, 6-10
 top-level, 6-6

 understanding, 6-1
Callback class
 act method, 2-50
calss
 loader, 1-18
class
 attributes, 1-2
 auditing, 2-19
 definition, 1-2
 dynamic loading, 1-13
 execution, 2-3
 hierarchy, 1-7
 inheritance, 1-7
 interpretation, 2-3
 loading, 2-3, 2-12
 loading permission, 10-18
 marking valid, 2-9
 methods, 1-2
 name, 2-21
 publish, 2-19, 6-1
 resolving dependencies, 2-9
 resolving reference, 2-9
 schema object, 2-3, 2-9, 2-12, 2-13
 shortened name, 2-21
 single inheritance, 1-3
.class files, 2-4, 2-12, 2-13
Class interface
 forName method, 2-22
class schema object, 2-12, 2-13, 11-1, 11-3
Class.forName
 lookupClass method, 2-22, 2-24
class.forNameAndSchema method, 2-24
ClassNotFoundException, 2-22
CLASSPATH, 2-3, 2-22
client
 install JDK, 4-2
 set up environment variables, 4-2
 setup, 4-2
CodeSource class, 10-4
 equals method, 10-4
 implies method, 10-4
compiling, 1-18, 2-5
 error messages, 2-6, 11-4
 options, 2-6, 11-4
 run time, 2-5, 2-6
configuration, 4-1

- JVM, 4-2
 - performance, 9-4
- connection
 - security, 10-1
- constructor methods, 6-14
- context
 - run-time, 5-1
 - stored procedures, 5-1
- CREATE JAVA statement, 5-5

D

- data confidentiality, 10-1
- data types
 - mapping, 6-3
- database
 - privileges, 10-2
 - schema plan, 8-4
 - trigger, 7-4
 - triggers, 5-2
- database triggers
 - calling Java, 7-4
- DBA_JAVA_POLICY view, 10-4, 10-13, 10-14
- DBMS_JAVA package, 4-2, 7-2
 - compile_class, A-12
 - compile_method, A-12, A-13
 - delete_permission method, 10-13, A-6
 - derivedFrom method, A-2
 - disable_output_to_file, A-10
 - disable_output_to_java, A-9
 - disable_output_to_sql, A-8
 - disable_output_to_trc, A-11
 - disable_permission method, 10-13, A-6
 - dropjava method, A-5
 - enable_output_to_file, A-10
 - enable_output_to_java, A-9
 - enable_output_to_sql, A-8
 - enable_output_to_trc, A-11
 - enable_permission method, 10-13, A-6
 - endsession, A-11
 - endsession_and_related_state, A-11
 - export_class method, A-4
 - export_resource method, A-4
 - export_source method, A-4
 - fixed_in_instance method, A-3
 - get property, A-7
 - get_compiler_option method, A-1
 - grant_permission method, 10-6, A-5
 - grant_policy_permission method, 10-8, 10-10, 10-14, A-5
 - loadjava method, A-4
 - longname method, 2-18, 2-21, A-1
 - modifying permissions, 10-13
 - modifying PolicyTable permissions, 10-7, 10-8
 - query_output_to_file, A-10
 - query_output_to_sql, A-8
 - query_output_to_trc, A-11
 - remove property, A-7
 - remove_output_to_file, A-10
 - remove_output_to_java, A-9

- remove_output_to_sql, A-8
- reset_compiler_option method, A-2
- resolver method, A-2
- restrict_permission method, 10-6, A-5
- revoke_permission method, 10-13, A-5
- runjava, A-6
- runjava_in_current_session, A-6
- set property, A-7
- set_compiler_option method, A-2
- set_native_compiler_option, A-11
- set_output method, A-3
- set_output_to_file, A-10
- set_output_to_java, A-9
- set_output_to_sql, A-8
- set_preference method, A-6
- setting permissions, 10-4
- shortname method, 2-18, 2-22, A-1
- show property, A-7
- start_jmx_agent, A-13
- uncompile_class, A-12
- uncompile_method, A-13
- unset_native_compiler_option, A-12
- DBMS_JAVA query_output_to_java, A-10
- DBMS_OUTPUT package, A-3
- DbmsJava class *see* DBMS_JAVA package
- DbmsObjectInputStream class, 2-25
- DbmsObjectOutputStream class, 2-25
- deadlock, 2-47
- DeadlockError exception, 2-47
- debug
 - compiler option, 2-7, 11-5
 - stored procedures, 5-7
- debugging
 - Java applications, 3-13
 - permissions, 10-17
- definer rights, 2-16
- delete method, 10-13
- delete_permission method, A-6
- derivedFrom method, A-2
- DETERMINISTIC hint, 6-7
- digest table, 11-3
- disable method, 10-13
- disable_permission method, A-6
- dropjava method, A-5
- dropjava tool, 2-13, 11-17

E

- ease of use, 5-3
- echo command, 11-26
- enable method, 10-13
- enable_permission method, A-6
- encapsulation, 1-3
- encoding
 - compiler option, 2-6, 11-4
- end-of-call Migration, 2-49
- EndOfCallRegistry class, 2-50
 - registerCallback method, 2-50
- entity-relationship (ER) diagram, 8-1
 - drawing, 8-1

- example, 8-3
- equals method, 10-4
- errors
 - compilation, 2-6
- exception
 - ClassNotFoundException, 2-22
 - DeadlockError, 2-47
 - how Oracle JVM handles, 7-10
 - LimboError, 2-47
- exit command, 11-26
- exitCall method, 2-47
- export_class method, A-4
- export_resource method, A-4
- export_source method, A-4

F

- file names
 - dropjava tool, 11-19
 - loadjava tool, 11-12
- FilePermission, 10-4, 10-6, 10-8, 10-14, 10-16
- files, 2-26
 - lifetime, 2-54
- finalizers, 2-28
- fixed_in_instance method, A-3
- footprint, 1-11, 2-2
- foreign key, 8-4
- forName method, 2-22
- full name, Java, 2-5
- functions, 5-2

G

- garbage collection, 2-2
 - managing resources, 2-26
 - misuse, 2-27
- garbage collectioin
 - purpose, 2-27
- get_compiler_option method, 2-7, A-1
- getCallerClass method, 2-23
- getClassLoader method, 2-23
- getProperty method, 3-14
- grant method, 10-6
- grant_permission method, 10-6, A-5
- grant_policy_permission method, 10-8, 10-14, A-5
- granting permissions, 10-4
- grantPolicyPermission method, 10-8
- graphical user interface *see* GUI
- GUI, 1-15, 2-20

H

- help command, 11-26

I

- implies method, 10-4
- inheritance, 1-3
- installation, 4-1
- integrated development environment (IDE), 1-15
- integrity, 10-1

- interface, 1-3
- internal JDBC driver, 1-19
- interoperability, 5-4
- interpreter, 1-18
- IOException, 2-54

J

- J2SE
 - install, 4-2
- Java
 - applications, 2-1, 2-12
 - attributes, 1-2
 - calling from database triggers, 7-4
 - calling from PL/SQL, 7-8
 - calling from SQL DML, 7-7
 - calling from the top level, 7-1
 - calling restrictions, 7-8
 - checking loaded classes, 2-18
 - classes, 1-2
 - client setup, 4-2
 - compiling, 2-5
 - development environment, 2-3
 - development tools, 1-22
 - execution control, 2-3
 - execution in database, 5-1
 - execution rights, 2-15
 - full name, 2-5
 - in the database, 1-8, 2-1
 - invoking, 3-1
 - key features, 1-5
 - loading applications, 2-12
 - loading classes, 2-3
 - methods, 1-2
 - overview, 1-1
 - polymorphism, 1-4
 - publishing, 2-3, 6-1
 - resolving classes, 2-9
 - short name, 2-5
 - stored procedures *see* stored procedures
- Java audit
 - object level, 2-20
 - statement level, 2-19
- java command, 11-26
- Java Compatibility Kit *see* JCK
- Java Database Connectivity *see* JDBC
- Java Development Kit *see* JDK
- .java files, 2-4, 2-12, 2-13
- java interpreter, 2-3
- Java Language Specification *see* JLS
- Java Naming and Directory Interface *see* JNDI
- Java Native Interface *see* JNI
- Java stored procedures *see* stored procedures
- Java virtual machine *see* JVM
- JAVA\$OPTIONS table, 2-6
- JAVA_ADMIN
 - granting permission, 10-2, 10-4, 10-8, 10-14
- JAVA_ADMIN example, 10-9
- JAVA_ADMIN role
 - assigned permissions, 10-15

- java_jit_enabled, 9-1
- JAVA_MAX_SESSIONSPACE_SIZE parameter, 9-5
- JAVA_POOL_SIZE parameter, 9-5
 - default, 4-2
- JAVA_SOFT_SESSIONSPACE_LIMIT
 - parameter, 9-5
- Java2
 - security, 10-2
- Java2 Platform, Standard Edition *see* J2SE
- JAVADEBUGPRIV role, 10-17
- JAVASYSPRIV role, 10-3, 10-16, 10-17
- JVAUSERPRIV role, 10-3, 10-16, 10-17
- JCK, 1-7
- JDBC, 1-6, 9-1
 - accessing SQL, 1-21
 - defined, 3-3
 - driver, 1-17
 - driver types, 1-21, 3-4
 - drivers, 1-21
 - example, 3-5
 - security, 10-1
 - server-side internal driver, 1-19
- JDeveloper
 - development environment, 1-22
- JDK
 - install, 4-2
- JIT
 - advantages, 9-2
 - overview, 9-1
- JLS, 1-2, 1-7
- JNDI, 1-5
- JNI
 - support, 3-3
- JPublisher
 - overview, 12-5
- JServerPermission, 10-14, 10-15, 10-17
- Just-in-Time Compiler *see* JIT
- JVM, 1-6
 - bytecode, 1-6
 - configure, 4-1
 - defined, 1-1
 - garbage collection, 1-9, 1-10
 - install, 4-1
 - multithreading, 1-9
 - responsibilities, 2-2

K

- key
 - foreign, 8-4
 - primary, 8-4

L

- library manager, 1-18
- LimboError exception, 2-47
- loader, class, 1-18
- loading, 2-12
 - checking results, 2-13, 2-18
 - class, 1-13, 2-3, 2-5, 2-12

- compilation option, 2-5
- granting execution, 2-15
- JAR or ZIP Files, 2-15
- necessary privileges and permission, 2-14
- reloading classes, 2-15
- restrictions, 2-14
- loadjava method, A-4
- loadjava tool, 2-12 to 2-15, 11-6 to 11-17
 - compiling source, 2-6, 9-8
 - example, 3-2, 5-6
 - execution rights, 2-15, 10-2
 - loading class, 2-12
 - resolution modes, 11-3
 - using memory, 9-5
- logging, 2-6
- longname method, 2-18, 2-22, A-1
- lookupClass method, 2-24

M

- main method, 1-15, 2-3
- maintainability, 5-4
- map methods, 6-14
- memory
 - across calls, 2-27
 - call, 2-3
 - Java pool, 9-6
 - leaks, 2-27
 - lifetime, 2-26, 2-54
 - performance configuration, 9-4
 - session, 2-2
- methods, 1-2, 5-2, 6-12
 - constructor, 6-14
 - declaring, 6-13
 - map and order, 6-14
 - object-relational, 5-2
- missing classes, 11-9
- modes
 - parameter, 6-2
- multiple inheritance, 1-3
- multithreading, 1-9

N

- NAME clause, 6-7
- namespace, 11-21
- native compilation
 - methods, 1-12
 - performance, 1-12
- native Java interface, 3-10
- NetPermission, 10-14, 10-15

O

- object, 1-2
 - full to short name conversion, 2-18
 - lifetime, 2-54
 - schema, 2-3
 - serialization, 2-24
 - SQL type, 5-2
 - table, 6-14

- writing call specifications, 6-12
- ObjectInputStream class, 2-25
- ObjectOutputStream class, 2-25
- object-relational methods, 5-2
- ojvmjava tool, 11-21 to 11-28
- online
 - compiler option, 2-6, 11-5
- operating system
 - resources, 2-26
- operating system resources
 - access, 2-27
 - across calls, 2-54
 - closing, 2-27
 - garbage collection, 2-27
 - lifetime, 2-27
 - managing, 2-26
 - overview, 2-26
 - performance, 9-4
- Oracle, 9-1
- Oracle JVM
 - class loader, 1-18
 - compiler, 1-18
 - configuration requirements, 4-2
 - interpreter, 1-18
 - JDBC internal driver, 1-18, 1-19
 - library manager, 1-18
 - main components, 1-17
 - SQLJ translator, 1-19
 - verifier, 1-18
- Oracle Net Services Connection Manager, 1-9
- OracleRuntime class
 - exitCall method, 2-47
 - getCallerClass method, 2-23
 - getClassLoader method, 2-23
- order methods, 6-14
- output
 - redirecting, 3-14, 7-2

P

- package DBMS_JAVA, 4-2, 7-2
- packaged call specifications, writing, 6-10
- PARALLEL_ENABLE option, 6-7
- parameter modes, 6-2
- performance, 5-3, 9-1
 - native compilation, 1-12
- permission
 - Oracle-specific, 10-14
 - types, 10-13
- Permission class, 10-4, 10-5, 10-9, 10-10, 10-13
- permissions, 10-1 to 10-18
 - administrating, 10-8
 - assigning, 10-3, 10-4
 - creating, 10-9
 - debugging, 10-17
 - deleting, 10-13
 - disabling, 10-12
 - enabling, 10-12
 - granting, 10-4, 10-6
 - granting policy, 10-8

- granting, example, 10-7
- grouped into roles, 10-17
- JAVA_ADMIN role, 10-15
- JAVADEBUGPRIV role, 10-17
- JAVASYSPRIV role, 10-16
- JAVAUSERPRIV role, 10-16
- limiting, 10-4, 10-6
- limiting, example, 10-7
- PUBLIC, 10-16
- restricting, 10-4, 10-6
- restricting, example, 10-7
- specifying policy, 10-3
- SYS permission, 10-15
- types, 10-13
- PL/SQL
 - calling Java from, 7-8
 - packages, 6-10
- policy table
 - managing, 10-8
 - modifying, 10-4
 - setting permissions, 10-4
 - viewing, 10-4
- PolicyTable class
 - specifying policy, 10-3
 - updating, 10-3, 10-10
- PolicyTableManager class
 - delete method, 10-13
 - disable method, 10-13
 - enable method, 10-13
- PolicyTablePermission, 10-4, 10-8, 10-14, 10-15
- polymorphism, 1-4
- primary key, 8-4
- privileges
 - database, 10-2
- procedures, 5-2
- productivity, 5-3
- .properties files, 2-4, 2-12, 2-13
- PropertyPermission, 10-14, 10-15, 10-17
- PUBLIC permissions, 10-16
- publishing, 2-3, 2-5, 2-19
 - example, 3-2, 5-6

R

- redirecting output, 3-14, 7-2
- REF, 6-15
- ReflectPermission, 10-14, 10-15
- registerCallback method, 2-50
- replication, 5-4
- reset_compiler_option method, 2-7, A-2
- resolver, 2-9 to 2-12, 11-2, 11-3
 - default, 2-10
 - defined, 2-3, 2-5, 2-22
 - example, 3-2, 5-6
 - ignoring non-existent references, 2-10, 2-11
- resolver method, A-2
- resolver specification
 - definition, 2-9
- resource schema object, 2-12, 2-13, 11-1
- restrict method, 10-6

- restrict_permission method, 10-6, 10-7, A-5
- revoke method, 10-13
- revoke_permission method, 10-13, A-5
- row trigger, 7-4
- run-time contexts, stored procedures, 5-1
- RuntimePermission, 10-14, 10-15, 10-16

S

- scalability, 5-3
- scavenging, 1-11
- security, 5-4
- SecurityPermission, 10-14, 10-15
- .ser files, 2-4, 2-12, 2-13
- SerializablePermission, 10-14, 10-16
- session, 1-14
 - namespace, 11-21
- session memory, 2-2
- set_compiler_option method, 2-7, A-2
- set_output method, A-3
- set_preference method, A-6
- SHARED_POOL_SIZE parameter, 9-4
 - default, 4-2
- shell commands, 11-25
- short name, Java, 2-5
- shortname method, A-1
- single inheritance, 1-3
- SocketPermission, 10-14, 10-16
- source schema object, 2-12, 2-13, 11-1
- SQLJ
 - server-side translator, 1-19
 - translator, 1-19
- .sqlj files, 2-4, 2-12, 2-13
- start_jmx_agent, A-13
- statement trigger, 7-4
- stored procedures
 - advantages, 5-3
 - calling, 7-1
 - defined, 1-20
 - developing, 5-1, 8-1
 - example, 8-1
 - introduction, 5-1
 - invoking, 3-1
 - publishing, 6-1
 - steps, 5-5

T

- threading
 - model, 1-9, 2-46
- threads
 - across calls, 2-56
 - life cycle, 2-47
 - Oracle JVM, 2-26
 - threading in Oracle Database, 2-46
- top-level call specifications, writing, 6-6
- triggers
 - calling Java from, 7-4
 - database, 5-2, 7-4
 - row, 7-4

- statement, 7-4
- using Java stored procedures, 3-1, 5-5

U

- user interface, 2-20
- USER_ERRORS, 2-6
- USER_JAVA_POLICY view, 10-4, 10-13, 10-14
- USER_OBJECTS, 2-13, 2-18, A-1
 - accessing, example, 2-19

V

- V\$SGASTAT table, 9-7
- variables
 - static, 2-2
- verifier, 1-18
- version
 - retrieving, 3-14
- version command, 11-28

W

- Web services
 - call-ins to database, 12-2
 - call-outs from database, 12-3, 12-4
 - JPublisher support for call-ins, 12-3
 - Oracle Database features, 12-2, 12-5
 - overview, 12-1
 - service consumer, 12-3
 - service provider, 12-1
 - using JPublisher for call-ins, 12-2
 - working directions, 12-1
- whoami command, 11-28